

Notes du cours de développement (DEV2)

Nathan Furnal

13 juin 2021

Table des matières

1	Introduction à l'orienté objet	3
1.1	Motivation	3
1.2	Vocabulaire	3
1.3	Classe	3
1.4	Constructeur & Instanciation	4
1.5	Membres	4
1.5.1	Appel de méthode	4
1.5.2	Membre privé	5
1.6	Accesseur et Mutateur	5
1.7	Visibilité	5
1.8	Attribut et constructeur par défaut	6
1.9	Surcharge	6
1.10	<code>this</code>	6
1.11	<code>static</code>	7
1.11.1	<code>import static</code>	7
1.12	Encapsulation	7
2	Orienté objet (étude de cas)	7
3	Grammaire	9
3.1	Principe	9
3.2	Fonctionnement	9
3.3	Lexicale	10
3.4	Syntaxe	10
4	Expression-instruction et instructions	10
4.1	Instruction	10
4.1.1	Instruction étiquetée	11
4.1.2	Instructions de rupture	11
4.2	Expression-instruction	12
4.2.1	Assignment	12
4.2.2	Incrémentation et décrémentation	12
4.2.3	Appel de méthode	13
4.2.4	Instanciation	13
5	Tableaux	13
5.1	Tableau de tableau	13
5.2	Création en donnant des valeurs	14
5.3	Création en donnant des tailles	14
5.4	Parcours	14
5.5	La classe <code>Arrays</code>	15
5.6	Copie de tableaux	15
5.6.1	Copie superficielle	16
5.6.2	Copie avec <code>java.util.Arrays.copyOf()</code>	16
5.6.3	Copie en profondeur avec <code>java.util.Arrays.copyOf()</code>	17
5.6.4	Copie avec des boucles	17
5.6.5	Copie profonde défensive	17
6	Les collections	17
6.1	La <code>List</code>	17
6.2	L' <code>interface</code>	18

6.3	L'ArrayList	18
6.4	La LinkedList	18
6.5	Polymorphisme	19
6.6	Wrappers	19
6.7	La classe Collections	20
7	Héritage	22
7.1	Le mot-clé <code>super</code>	22
7.2	Le polymorphisme	22
7.3	La classe Object	23
7.4	La classe Objects	23
8	Énumérations et exceptions	24
8.1	Énumérations	24
8.1.1	Notions avancées	24
8.2	Exceptions	25
8.2.1	Hiérarchie et exception contrôlée	25
8.2.2	Créer sa propre exception	26
8.2.3	Précisions sur l'utilisation de <code>catch</code>	26
8.3	Le mot-clé <code>var</code>	27
9	Le codage des fichiers	27
9.1	Binaire vs. texte	28
9.2	UTF-8	28
9.3	Trouver son chemin	29
9.3.1	Paths	29
9.3.2	Varargs	29
9.3.3	La classe Path	30
9.4	La classe Files	30
9.5	Les entrées-sorties de bas niveau	30
9.5.1	Binaire	31
9.5.2	Texte	31
10	Entrées et sorties de haut niveau	31
10.1	Les flux englobants	32
10.2	Les expressions régulières	32
10.3	Retour sur Scanner et flux standards	33
10.4	La classe Console	34
10.5	Sérialisation	34
11	Programmation fonctionnelle	34
11.1	Les fonctions lambda λ - Introduction	34
11.1.1	Application	35
11.1.2	Référence de méthode	36
11.2	Retour sur les tris avec Comparable	36
11.3	Les tris et le fonctionnel avec Comparator	36
11.4	Comparable vs. Comparator	37
11.5	Les fonctions lambda - Explications	38
11.5.1	Approche par classe nommée	38
11.5.2	Approche par classe anonyme	38
11.5.3	Approche par fonction λ	38
11.6	Les streams - Introduction	39
11.7	Les streams - Détails et exemples	39
11.7.1	Création de Stream	39
11.7.2	Opérations intermédiaires (non-terminales)	39
11.7.3	Opérations terminales	39
11.7.4	Précisions	40
11.7.5	Parallélisme	41
12	Concepts importants	41
	Lexique	41
13	Crédits	42

1 Introduction à l'orienté objet

Au début, il n'existait que le langage machine, il est proche de la machine et du matériel et donc performant, mais il est difficile à lire et à manipuler. Avec la complexification des problèmes, les demandes faites aux programmes augmentent aussi. De ce fait, on peut soit améliorer les processus via l'analyse ou bien on peut inventer de nouveaux langages de plus "haut niveau", s'éloignant du langage machine et qui permettent une écriture plus simple et compréhensible.

Pour résoudre de nouveaux problèmes, que ce soit limiter le nombre de lignes et de bugs ou bien répondre à des nouvelles demandes (business, intelligence artificielle); on crée donc de nouveaux langages, comme C, FORTRAN, LISP, etc. Puis, dans le passé plus proche on voit apparaître d'autres langages comme Java ou Python fournissant à la fois une écriture plus moderne et une manière de fonctionner différente.

1.1 Motivation

Conceptuellement, un programme est composé de code et de données. Le code a vu énormément de progression à travers le temps mais la manière de manipuler les données, beaucoup moins. L'apport de l'orienté-objet c'est de fournir une abstraction qui permet de manipuler des données. Grâce à l'orienté-objet, on peut définir des nouvelles structures de données et des nouveaux types, plus compacts et plus lisibles.

Voici un exemple où on définit un **objet** élève qui peut être réutilisé autant de fois qu'on veut pour créer de nouveaux étudiants avec un nom.

```
1 // On peut définir un âge pour un élève et une année
2
3 public class Eleve {
4     private String name;
5
6     public Eleve(String name){
7         this.name = name;
8     }
9
10    public static void main(String[] args){
11        Eleve etudiant = new Eleve("Mon nom");
12    }
13 };
```

1.2 Vocabulaire

Un **objet** représente un élément de problème à résoudre, un concept de la vie réelle ou un concept de données. Comme un client, un étudiant, une machine. Le programme crée et manipule les objets.

Un type de donnée orienté objet est défini par une **classe** et un objet est une **instance** de classe. On peut voir la **classe** comme un plan, une idée générale et un **objet** comme la réalisation de ce plan.

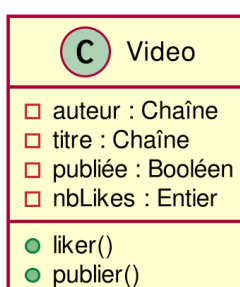
Un **objet** se caractérise par :

- son **état** : Ce sont les **attributs**, les données propres à l'objet qui sont stockées dans des variables.
- son **comportement** : Ce sont les **méthodes**, c'est-à-dire des fonctions qui peuvent agir sur l'objet ou faire réagir l'objet.

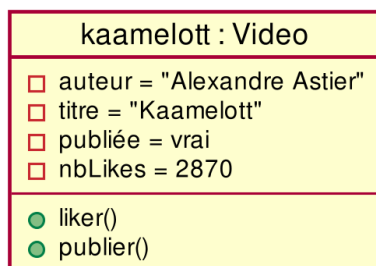
L'ensemble des attributs et des **méthodes** porte le nom de **membres**.

1.3 Classe

Avant tout, une **classe** est de **type référence**. On peut aussi donner un exemple via un diagramme de classe.



Sur le diagramme, on peut voir les attributs privés avec leur type ainsi que les **méthodes** publiques, le comportement de la classe. Quand une **classe** est instanciée, on peut aussi créer un diagramme pour l'instance de la classe. Dans ce cas, il faut donner les valeurs des attributs, l'**état** de l'instance.



Mais ce ne sont que des représentations graphiques, pour définir la **classe**, on utilise le langage Java.

```

1 public class Video {
2     // Déclaration d'attributs dans la classe
3     private String auteur;
4     private String titre;
5     private boolean publiée;
6     private int nbLikes;
7
8     // Déclaration de méthodes non-statiques
9     public void liker(){
10         nbLikes++;
11     }
12
13     public void publier(){
14         publiée = true;
15     }
16 }

```

Maintenant, on peut utiliser la **classe** Video pour créer des **objets** de type Video.

1.4 Constructeur & Instanciation

Quand on instancie un objet, on fait **référence** à sa définition de classe et on lui attribue une 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.

1.5 Membres

1.5.1 Appel de méthode

Pour appeler une méthode, on va utiliser la notation pointée (opérateur **.**). C'est-à-dire qu'on écrit le nom de l'objet qu'on utilise, puis un point et en fin le nom de la méthode. Cette méthode ne peut être utilisée que par les instances de classes de Video (sauf dans le cas de l'**héritage**, expliqué plus tard).

```

1 Video v = new Video("Omar Sy", "Lupin", true, 0);
2 v.liker(); // Incrémente le nbr de likes de un

```

1.5.2 Membre privé

Souvent, on voudra lire ou accéder aux attributs privés d'un objet mais on ne peut pas le faire directement, c'est une forme de protection des données liées à l'objet.

```
1 public static void main(String [] args) {
2     Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
3     System.out.println(kaamelott.titre); // ne compile : attribut privé
4     kaamelott.nbLikes = 1_000_000; // idem
5 }
```

On résout ce problème en utilisant des méthodes publiques pour manipuler ces données : les **accesseurs** et **mutateurs**.

1.6 Accesseur et Mutateur

Donc, un **accesseur** permet simplement d'accéder à la valeur d'un attribut. Tandis qu'un **mutateur** permettra de modifier la valeur d'un attribut. On parle, en anglais, de **getter** et de **setter**. Le **setter** permet de modifier l'état d'un objet puisqu'il modifie un attribut.

Quand on crée un **getter** :

```
1 public String getAuteur() {return auteur;}
2 public String getTitre () {return titre ;}
3 public int getNbLikes() {return nbLikes;}
4 public boolean isPubliée () {return publiée ;}
```

On l'utilise directement avec la notation *pointée* et on commence son nom par «get» suivi du nom de l'attribut et «is» pour les **boolean**.

```
1 public static void main(String [] args) {
2     Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
3     System.out.println(kaamelott.getNbLikes()); // 0
4     kaamelott.liker();
5     System.out.println(kaamelott.getNbLikes()); // 1
6 }
```

De même pour les **setter**. On utilise le préfixe «set».

```
1 public void setTitre ( String unTitre) { titre = unTitre;}
```

et un usage :

```
1 Video kaamelott = new Video("Alexandre Astier", "Kaamelott");
2 System.out.println(kaamelott.getTitre());
3 kaamelott.setTitre("Kaamelott - Livre 1 - Tome II");
4 System.out.println(kaamelott.getTitre());
```

1.7 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.

1.8 Attribut et constructeur par défaut

En Java, il existe un comportement par défaut même si un constructeur n'est pas explicitement créé. Ce constructeur **par défaut** ne prend aucuns paramètres et disparaît simplement quand un nouveau constructeur est créé.

Si on a fourni un constructeur mais qu'on oublie de l'initialiser avec les paramètres nécessaires. Java fournira alors des attributs par défaut : 0 pour les nombres, `null` pour les types références et `false` pour les booléens.

1.9 Surcharge

Pour revenir au constructeur en tant que tel, il est mentionné plus haut qu'on peut fournir plusieurs constructeurs. Le fait que le même appel de méthode ait plusieurs comportements différents s'appelle **surcharge**. Quand on utilise la **surcharge**, il **faut** que les différentes méthodes de même nom se différencient par le type des paramètres ou nombre des paramètres. La **surcharge** peut être appliquée aux constructeurs comme aux méthodes.

Voici un exemple avec la méthode `liker` de la classe `Video`.

```
1 public void liker() {
2     liker(1); // On utilise l'autre définition de liker
3 }
4
5 public void liker(int nbFois){
6     nbLikes = nbLikes*nbFois;
7 }
```

1.10 `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 :

- constructeur `this()` ;
- attribut `this.titre` ;
- 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 :

```
1 public class Example {
2     private String name;
3     private int value;
4     private boolean nice;
5
6     public Example(String aName, int aValue, boolean niceness){
7         // On assigne aux attributs les valeurs des paramètres
8         // this avec la notation pointée
9         this.name = aName;
10        this.value = aValue;
11        this.nice = niceness;
12    }
13
14    public Example(String aName){
15        // Utiliser this() comme fonction !!!
16        // Permet d'assigner des valeurs par défaut
17        // Doit être la première ligne
18        this(aName, 11, false);
19    }
20
21    public static void main(String[] args){
22        // construction classique
23        Example monExemple = new Example("hello", 10, true);
24        // construction avec moins de paramètres et des valeurs par défaut
25        Example other = new Example("hi hi");
```

26
27

```
}  
}
```

1.11 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 :

```
1 Video v = new Video("Omar Sy", "Lupin");  
2 v.liker(); // Méthode non statique qui modifie un attribut de l'instance  
3 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.

1.11.1 import static

Pour importer une méthode d'un paquet ou bien une autre classe d'un même paquet sans devoir réécrire l'entière 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 :

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

Peut être utilisé comme ça :

```
1 import static java.lang.Math.*;  
2 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.

1.12 Encapsulation

Un principe important de l'orienté objet est l'**encapsulation**, c'est-à-dire que tous les attributs d'une classe sont privés et qu'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.

2 Orienté objet (étude de cas)

Mettons en pratique ce que nous avons appris avec une étude de cas, créer un objet `Point` qui représente un point dans un plan à deux dimensions. Il doit avoir des attributs qui définissent sa position, un **constructeur** et des méthodes pour se déplacer.

 Point
<div> <div>□</div> x : Réel </div> <div> <div>□</div> y : Réel </div>
<div> <div>●</div> Point(x : Réel, y : Réel) </div> <div> <div>●</div> display() </div> <div> <div>●</div> getX() : double </div> <div> <div>●</div> getY() : double </div> <div> <div>●</div> move(deltaX : Réel, deltaY: Réel) </div> <div> <div>●</div> distance(other : Point) : Réel </div>

Voici une implémentation en Java.

```

1  public class Point {
2      private double x;
3      private double y;
4
5      public Point(double x, double y){
6          this.x = x;
7          this.y = y;
8      }
9      public Point(){
10         this(0, 0);
11     }
12     public void display(){
13         System.out.println("(" + this.x + ", " + this.y + ")");
14     }
15     public double getX(){
16         return this.x;
17     }
18     public double getY(){
19         return this.y;
20     }
21     public void move(double deltaX, double deltaY){
22         this.x += deltaX;
23         this.y += deltaY;
24     }
25     public double distance(Point other){
26         double dx = other.x - this.x;
27         double dy = other.y - this.y;
28         return Math.sqrt(dx*dx + dy*dy);
29     }
30     public double distance(){
31         // distance à l'origine
32         Point other = new Point();
33         return distance(other);
34     }
35     public static void main(String[] args){
36         Point p = new Point(4, 5);
37         Point o = new Point(1, 1);
38         p.move(-1, 1);
39         o.move(-2, -2);
40         System.out.println("Positions de chaque point : ");
41         p.display();
42         o.display();
43         System.out.println("Distance entre les points : " + p.distance(o));
44     }
45 }

```

```

Positions      de  chaque  point   :
(3.0,         6.0)
(-1.0,        -1.0)
Distance entre les points : 8.06225774829855

```

Ce qui conclut l'exemple.

3 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.

3.1 Principe

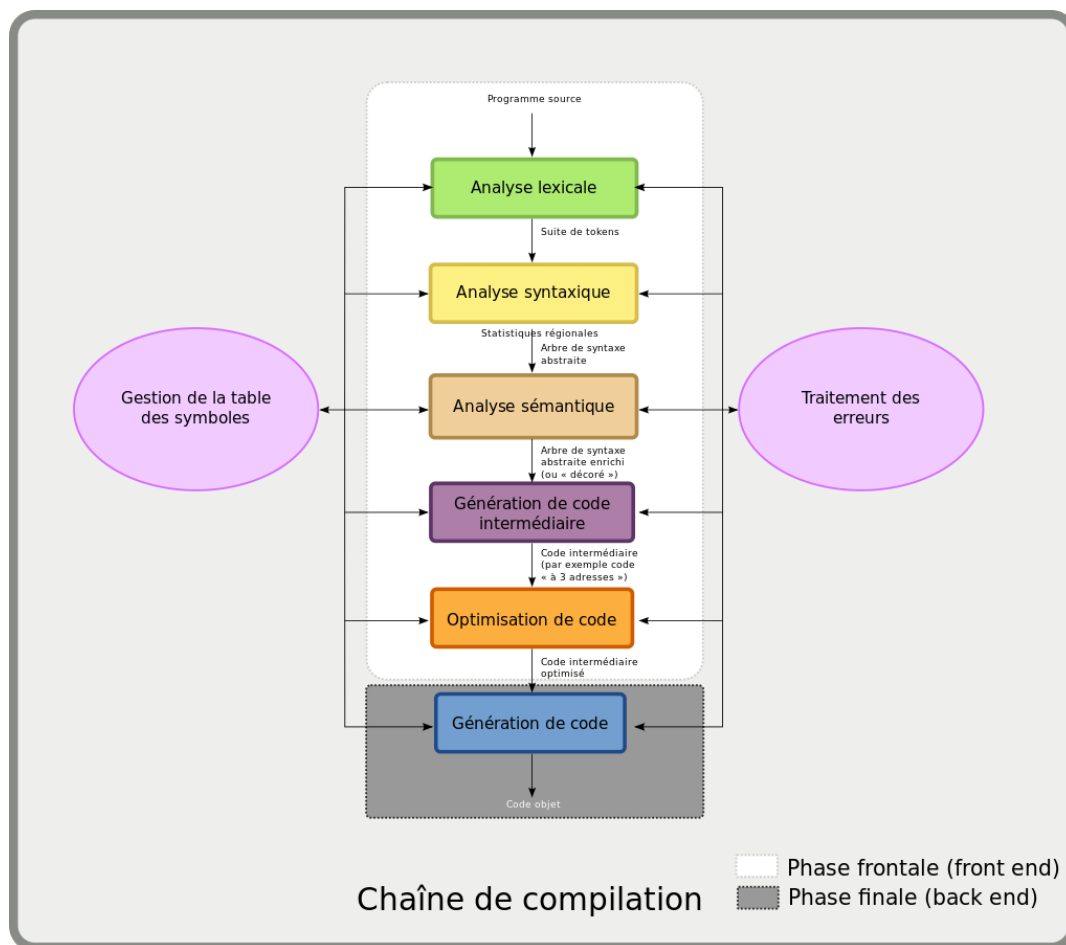
La nécessité de pouvoir écrire un **compilateur**, la nécessité de pouvoir décrire le langage imposent que les règles soient clairement décrites. C'est le rôle de la grammaire.

3.2 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.

3.3 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 **Identifiant**, ce sont les noms de variables, les chiffres, les lettres, etc. Mais ils ne peuvent pas être des mots-clés ou des littéraux.

3.4 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**.

Les instructions et expressions-instructions sont détaillées dans le chapitre suivant.

4 Expression-instruction et instructions

4.1 Instruction

Une **instruction** ou *statement*, représente un comportement à adopter pour le langage. Les instructions sont utilisées pour leur effet et n'ont **pas** de valeur. Elle peuvent inclure d'autres instructions (*nested if*) et des expressions. Par exemple, un **instruction** **if** contient une **expression**, par exemple **(3 > 2)** qui lui permettra d'avoir un effet ou non suivant la valeur de l'expression.

Voici quelques exemples : l'instruction vide, le bloc d'instruction, l'instruction d'arrêt (**break**), l'instruction **si...sinon** (**if{...} else{}**), l'instruction de choix **switch**, l'instruction de boucle (**for**), etc.

Dans la documentation de Java, on peut voir le fonctionnement des expressions de manière très générale. Dans le **manuel de spécification** de Java à la p. 454, on trouve l'explication générale du **if statement**.

L'ensemble de cette partie du cours caractérise le fonctionnement des **instructions**, les mots-clés nécessaires pour les utiliser et s'ils attendent des instructions ou expressions additionnelles.

Je ne les reprendrai pas à chaque fois ici, c'est tel quel dans les slides. Voici malgré tout quelques exemples expliqués.

```
EmptyStatement:  
    ;
```

Cette instruction ne fait rien. Par exemple `return 1;;` représente 3 instructions.

```
IfThenStatement:  
    if (Expression) Statement
```

```
IfThenElseStatement:  
    if (Expression) StatementNoShortIf else Statement
```

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

Ce qui signifie qu'il existe une **if** raccourci et le **if** classique. Dans le premier cas on ne doit pas forcément mettre d'accolades mais il ne permet pas d'utiliser de **else**.

De manière générale, les blocs de documentation décrivent le comportement de l'**instruction** et la ou les **expressions** attendues.

4.1.1 Instruction étiquetée

On peut étiqueter, ajouter un label à une instruction qui ne sera connu que dans le **scope** de l'instruction.

```
1  initialisation : int i = 1; // Le nom 'initialisation' n'existe que pour cette ligne  
2  test : if (i > 0){  
3      affichage : System.out.println(i);  
4  }
```

Cet usage est utile quand on veut quitter ou réitérer certaines instructions via **break** ou **continue**.

4.1.2 Instructions de rupture

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

```
BreakStatement:  
    break Identifieur(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 Identifieur(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.

```
1  for (int i = 0; i < 10; i++){  
2      if(i%2 == 0) continue;  
3      System.out.println(i);  
4  }
```

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 extérieure, là où un **continue** classique n'aura touché que la boucle englobante `bc1j`.

```

1 bcli : for(int i = 0; i < 10; i++){
2     bclj : for(int j = 0; i < 10; j++){
3         if((i*j)%2 == 0) continue bcli;
4         System.out.println(j);
5     }
6     System.out.println(i);
7 }

```

4.2 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.

4.2.1 Assignment

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 :

```

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

```

Voici quelques autres exemples :

```

1 élément = 1
2 éléments[i] = j
3 éléments[i] = j;
4 i = j = k = l = 0;
5 i = (j = i+j) + 1;
6 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.

```

1 i += 1; // équivaut i = i + 1

```

Voici des exemples plus complexes :

```

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

```

4.2.2 Incrémentation et décrémentation

En Java, il existe des expressions-instructions qu'on utilise pour augmenter ou diminuer la valeur de variables de 1 et on assigne la nouvelle valeur à la variable.

++ Permet d'incrémenter une variable.

-- Permet de décrémentation une variable.

Ces expressions sont *quasi* mais pas exactement comparables à l'opération $i = i + 1$ et $i = i - 1$ respectivement. Donc ++i et i++ sont des **expressions** avec une valeur mais des effets différents.

<code>i++</code>	<code>++i</code>
On regarde la valeur de <code>i</code> . Puis, <code>i</code> donne sa valeur à <code>i++</code> Après et seulement après, <code>i</code> est incrémenté.	D'abord <code>i</code> est incrémenté Puis seulement on regarde ce que vaut <code>i</code> Enfin, <code>i</code> donne sa valeur à <code>++i</code>

Voici un exemple de la différence, illustré en Java.

```
1  int i = 3;
2  i++; // Si on imprime cette ligne, elle vaut 3 ! Mais i vaut 4.
3  ++i; // i passe de 4 à 5 et ++i affiche 5 aussi !
```

Et un autre exemple plus compliqué, ici `i++` vaut 3 et `i` passe de 3 à 4. Ensuite, `++i` vaut directement 5 puisqu'on a incrémenté `i` de 1 et qu'on utilise cette valeur directement.

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

L'effet est exactement le même pour la soustraction. En pratique, il vaut mieux se contenter d'utiliser cette expression dans les cas simples et où il n'y a pas d'ambiguïté sur la valeur des variables.

4.2.3 Appel de méthode

L'appel de méthode est une **expression**, elle a un **type** et une **valeur**.

4.2.4 Instanciation

Créer une instance de classe est une expression, elle a un **type** et elle a une **valeur**.

5 Tableaux

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 :

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

5.1 Tableau de tableau

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.

```
1  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.

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

5.2 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.

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

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

5.3 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.

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

5.4 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.

```
1 int[] tab = {1, 2, 3, 4, 5};
2 // Parcours à une dimension
3 for(int i = 0; i < tab.length; i++){
4     System.out.println(tab[i]);
5 }
6
7 int[] [] tab = {{1, 2, 3, 4, 5}, {1, 3, 4, 8, 9}};
8 // Parcours à deux dimensions
9 for(int i = 0; i < tab.length; i++){
10     for(int j = 0; j < tab[0].length; j++){
11         System.out.println(tab[i][j]);
12     }
13 }
```

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 parcourir colonne par colonne puisqu'on a pas accès aux indices.

Voici un exemple :

```
1 int[] [] tab = {{1, 2, 3, 4}, {2, 1, 4, 5}};
2 // Foreach loop
3 for(int[] row : tab){
4     for(int val : row){
5         System.out.println(val);
6     }
7 }
8
```

```

9 // Accéder aux colonnes
10 // On fait l'hypothèse que toutes les lignes ont le même nombre de colonnes
11 for(int j = 0; j < tab[0].length; j++){
12     for(int i = 0; i < tab.length; i++){
13         System.out.println(tab[i][j]);
14     }

```

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

```

1 int[] tab = {1, 2, 3, 4, 5};
2
3 for(int i = 0; i < tab.length; i++){
4     tab[i] = 11;
5 }
6
7 // Le tableau vaut maintenant {11, 11, 11, 11, 11};
8
9 // Rien ne se passe dans cette boucle
10 // Chaque elem est une copie de la référence des éléments de tab, donc a rien
   ↪ modifié
11 for(int elem : tab){
12     elem = 22;
13 }

```

Ce qui clôture cette section sur le parcours.

5.5 La classe Arrays

Java fournit une classe `Arrays` qui permet de travailler avec les tableaux plus facilement, on peut l'importer avec `import java.util.Arrays`. Cette classe contient des méthodes statiques comme `Arrays.sort()` ou `Arrays.toString()`. Les méthodes dont le nom commence par `deep` sont des extensions d'autres méthodes pour les tableaux à plusieurs dimensions. Par exemple pour l'affichage (`deepToString`) ou l'égalité (`deepEquals`).

Voici quelques exemples :

```

1 import java.util.Arrays;
2
3 // Le mot-clé var permet d'inférer le type d'une variable
4 var tab = new String[] {"tableau", "de", "mots", "hello", "world"};
5 int[] vals = {3, 4, 5, 1, 2, 3, 9};
6
7 // Trier un tableau
8 Arrays.sort(vals);
9 // vals vaut maintenant {1, 2, 3, 3, 4, 5, 9};
10
11 // Convertir un tableau en String pour pouvoir l'afficher
12 System.out.println(Arrays.toString(vals));
13
14 // Faire une copie d'un tableau avec le nombre d'éléments à copier
15 // Si plus grand, les cases supplémentaires sont initialisées à null.
16 var phrase = Arrays.copyOf(tab, tab.length + 1);
17
18 // Recherche dans le tableau mais nécessite qu'il soit trié
19 // Renvoie une valeur négative pour les valeurs qui n'existe pas
20 Arrays.sort(tab);
21 Arrays.binarySearch(tab, "hello"); // Retourne 1, l'indice de hello dans le tableau
   ↪ trié

```

5.6 Copie de tableaux

Une autre question qui nous intéresse, c'est celle de la copie de tableau à deux dimensions, il y a plusieurs stratégies que nous allons explorer.

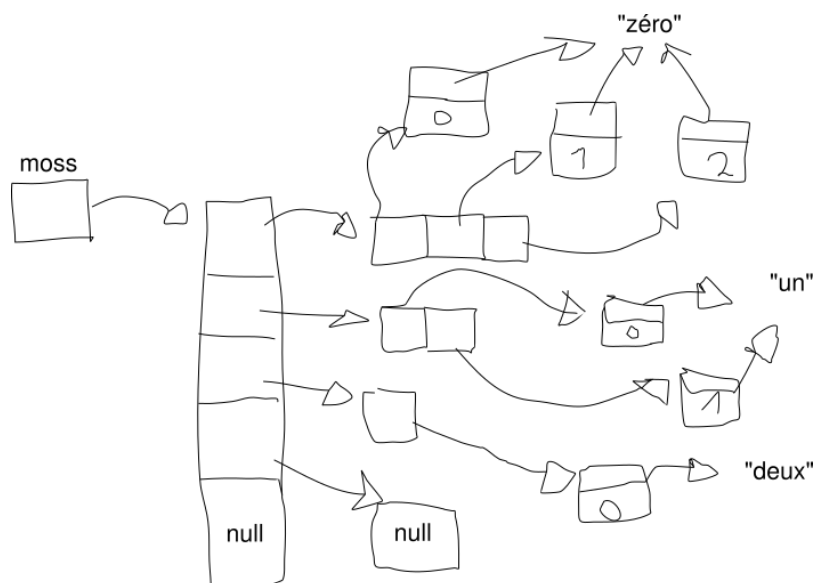
Dans les slides, on crée l'objet moss de cette façon :

```
1 public class MyObject {
2     private String description;
3     private int number;
4     public MyObject(String description , int number) {
5         this.description = description;
6         this.number = number;
7     }
8     public static MyObject newInstance(MyObject mo) {
9         if (mo == null) {return null;}
10        return new MyObject(mo.description, mo.number);
11    }
12    public void bro1 () {number++;}
13 }
```

Avec la situation initiale suivante :

```
1 MyObject[] [] moss;
2 MyObject[] [] copy;
3 moss = new MyObject[] [] {
4     {new MyObject("zéro", 0), new MyObject("zéro", 1), new MyObject("zéro", 2)},
5     {new MyObject("un", 0), new MyObject("un", 1)},
6     {new MyObject("deux", 0)},
7     {null },
8     null
9 };
```

Pour laquelle on nous demande faire un schéma mémoire :



Ensuite, nous allons comparer le schéma mémoire avec chaque stratégie de copie.

5.6.1 Copie superficielle

Dans ce cas, on a simplement la variable copy qui fait référence à l'objet moss. Donc, copy reçoit la référence vers moss et référence exactement le même tableau que moss.

```
1 copy = moss;
```

5.6.2 Copie avec java.util.Arrays.copyOf()

Dans ce cas, le tableau copy fera référence à chaque sous-tableau de moss. Donc, chaque modification de tableau qui compose une ligne dans l'un, modifiera l'autre.


```

1 import java.util.Arrays;
2 copy = Arrays.copyOf(moss, moss.length);

```

5.6.3 Copie en profondeur avec `java.util.Arrays.copyOf()`

Dans ce cas, on déclare d'abord un nouveau tableau et puis on fera une copie, ligne par ligne. On peut le voir comme une avancée de un niveau en plus dans les références. Ici, `copy` est un tableau et chaque tableau en ligne est aussi en tableau. Enfin, les cellules de ce tableau font références à celle du tableau initial, `moss`. De ce fait, une modification d'une cellule (non plus d'une ligne) de l'un, affectera l'autre.

```

1 copy = new MyObject[moss.length] [];
2 for(int i = 0; i < moss.length; i++){
3     copy[i] = moss[i] == null
4         ? null
5         : Arrays.copyOf(moss[i], moss[i].length);
6 }

```

5.6.4 Copie avec des boucles

```

1 copy = new MyObject[moss.length] [];
2 for(int i = 0; i < moss.length; i++){
3     for(int j = 0; j < moss[i].length; j++){
4         copy[i][j] = moss[i][j];
5     }
6 }

```

5.6.5 Copie profonde défensive

Ici, on a la copie la plus défensive possible. On a créé deux tableaux entièrement distincts et donc les changements de l'un n'affectent pas l'autre. En clair, `copy` ne fait plus référence aux valeurs de `moss` et à bien des emplacements mémoire propres pour chaque tableau ainsi que les valeurs qu'ils contiennent et leurs références.

```

1 copy = new MyObject[moss.length] [];
2 for(int i = 0; i < moss.length; i++){
3     copy[i] = new MyObject[moss[i].length];
4     for(int j = 0; j < moss[i].length; j++){
5         copy[i][j] = MyObject.newInstance(moss[i][j]);
6     }
7 }

```

6 Les collections

Commençons par la définition de **collection**, c'est une structure de données qui permet de stocker plusieurs éléments.

Dans une **collection** tous les éléments sont de même **type**. Certaines collections sont triées, d'autres ordonnées. Ordonné signifie que chaque élément a une position, avec un index attaché à sa position. Mais certaines collections ne sont pas ordonnées. Certaines collections permettent des **doublons**, d'autres garantissent l'**unicité** des éléments.

6.1 La List

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. Elles sont ordonnées et pas nécessairement triées 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`

Une liste a des méthodes qui permettent d'ajouter ou supprimer des éléments par exemple. Attention, `List` n'est pas une classe mais une **interface**, que nous détaillons au point suivant.

6.2 L'interface

Tout d'abord, 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.

```
1 public interface MyInterface {
2     public void foo(int i);
3     public boolean isBar(char c);
4 }
5
6 public class MyClass implements MyInterface {
7     public void foo(int i){
8         // do something interesting
9     }
10
11     public boolean isBar(char c){
12         return true; // or something useful
13     }
14
15     // Possible to add other methods
16 }
```

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.

```
1 MyClass o = new MyClass();           // OK
2 MyClass o = new MyInterface();       // Non
3 MyInterface o = new MyClass();       // OK
4 MyInterface o = new MyInterface();   // Non
```

Pour revenir au point sur les listes, on ne peut donc pas directement en instancier car c'est une interface. On va devoir utiliser une classe qui implémente ses méthodes et instancier cette classe.

6.3 L'ArrayList

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

```
1 import java.util.ArrayList;
2 List<String> nombrils = new ArrayList<>();
3 nombrils.add("Vicky");
4 nombrils.add("Jenny");
5 nombrils.add(1, "Karine");
6 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 donc se trouver dans un `foreach`. Quand on l'implémente, on doit assurer que certains contrats sont remplis : savoir définir un élément suivant par exemple.

6.4 La LinkedList

La `LinkedList` (liste chaînée) est une autre implémentation de l'interface `List`. Voici un exemple :

```
1 import java.util.LinkedList;
2 List<String> nombrils = new LinkedList<>();
3 nombrils.add("Vicky");
```

```

4 nombrils.add("Jenny");
5 nombrils.add(1, "Karine");
6 System.out.println(nombrils); // ["Vicky", "Karine", "Jenny"]

```

Là où `ArrayList` implémente une liste avec un tableau classique tandis que la `LinkedList` utilise des listes et des nœuds. On voit que l'implémentation existe et est différente pour ces deux implémentations. L'accès aux membres et l'ajout ne se déroule donc pas de la même manière.

6.5 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 :

```

1 public static void display (List<String> liste){
2     for(int i = 0; i < liste.size(); i++){
3         System.out.println(i + ": " + liste.get(i));
4     }
5 }

```

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.

```

1 List<String> l1 = new ArrayList<>();
2 List<String> l2 = new LinkedList<>();
3 display(l1); // fonctionne
4 display(l2); // fonctionne

```

6.6 Wrappers

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
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>

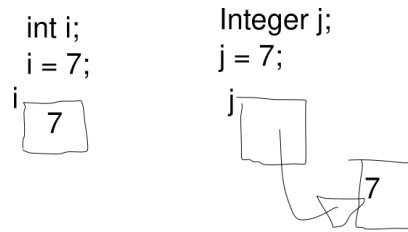
Par contre, lorsqu'on insère des types primitifs dans une liste, la conversion entre le type primitif et son *wrapper* est automatique. On ne doit pas préciser, comme le montre l'exemple suivant :

```

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

```

6.7 La classe Collections

Enfin, nous pouvons parler des Collections qui est un ensemble de services pour les listes. Par exemple pour obtenir les extrema, trier, inverser,... des listes.

Voici un exemple :

```

1 import java.util.ArrayList;
2 import java.util.Collections;
3 var arr = ArrayList<Integer>();
4 arr.add(3);
5 arr.add(2);
6 arr.add(1);
7 arr.add(11);
8 Collections.max(arr); // 11
9 Collections.min(arr); // 1
10 Collections.sort(arr); // [1, 2, 3, 11]
11 Collections.shuffle(arr); // [11, 1, 3, 2]

```

Mais attention, on ne peut pas utiliser les méthodes de Collections dans tous les cas. Par exemple, si on crée un tableau d'objets qu'on a défini nous-mêmes, il est possible que Collections ne sache pas comment les trier ou comment en trouver le maximum/minimum. En effet, un tableau de Video par exemple ne définit jamais comment une vidéo peut être plus grande ou plus petite qu'une autre.

C'est à nous de d'explicitier via une classe anonyme ou en implémentant une méthode d'interfaces comme Comparable, Iterable. Voici un exemple avec des personnes que je veux trier par âge. Ici, je n'utilise pas la comparaison avec une classe anonyme comme dans le cours mais j'implémente l'interface nécessaire pour pouvoir comparer deux personnes sur base de leur âge. Voici la classe nécessaire et puis l'exemple.

```

1 public class Personne implements Comparable<Personne>{
2     private final int age;
3     private final String nom;
4
5     public Personne(int age, String nom) {
6         this.age = age;
7         this.nom = nom;
8     }
9
10    public Personne() {
11        this(0, "Nemo");
12    }
13
14    public int getAge() {

```

```

15     return age;
16 }
17
18 public String getNom() {
19     return nom;
20 }
21
22 @Override
23 public boolean equals(Object o) {
24     if (this == o) return true;
25     if (o == null || getClass() != o.getClass()) return false;
26
27     Personne personne = (Personne) o;
28
29     if (getAge() != personne.getAge()) return false;
30     return getNom().equals(personne.getNom());
31 }
32
33 @Override
34 public int hashCode() {
35     int result = getAge();
36     result = 31 * result + getNom().hashCode();
37     return result;
38 }
39
40 @Override
41 public int compareTo(Personne other) {
42     return getAge() - other.getAge();
43 }
44
45 @Override
46 public String toString() {
47     return getNom() + " " + getAge() + " " + "ans";
48 }
49 }

```

Maintenant que notre classe est créée, on peut l'insérer dans un exemple.

```

1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class Example {
5      public static void ComparePersonnes(){
6          var people = new ArrayList<Personne>();
7          people.add(new Personne(23, "Jack"));
8          people.add(new Personne(11, "Alice"));
9          people.add(new Personne(45, "Omar"));
10         people.add(new Personne());
11         people.add(new Personne(111, "Eve"));
12         for (Personne p : people){
13             System.out.println(p);
14         }
15         System.out.println("La personne la plus âgée est : " +
16             ↪ Collections.max(people));
17     }
18
19     public static void main(String[] args) {
20         // Imprime chaque personne et son âge, c.f la méthode toString
21         // Imprime l'objet de valeur maximum, basé sur l'âge. C'est "Eve 111 ans"
22         ComparePersonnes();
23     }
24 }

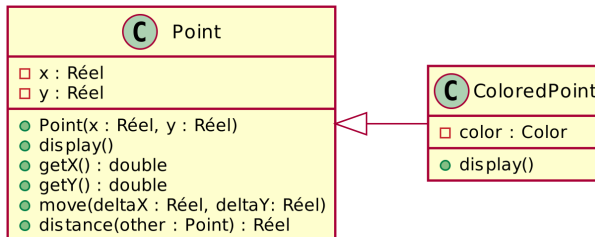
```

7 Héritage

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 visibilité 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.

```
1 import java.awt.Color;
2 public class ColoredPoint extends Point {
3     private Color color;
4     public ColoredPoint(double x, double y, Color c){
5         super(x, y); // Utilise le constructeur de Point
6         this.color = c;
7     }
8     public ColoredPoint(double x, double y){
9         this(x, y, Color.BLACK);
10    }
11 }
```

7.1 Le mot-clé **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.

```
1 public void display(){
2     System.out.println("(" + super.getX() + ", " + getY() + ") - " + color);
3 }
```

7.2 Le polymorphisme

Une autre définition du **polymorphisme** est que là où on attend une classe, on peut trouver une classe enfant. Le compilateur utilise le **type déclaré** et la machine virtuelle utilise le **type réel**. Voici un exemple :

```
1 Point p = new ColoredPoint(...); // OK (type déclaré = Point et type réel =
   ↳ ColoredPoint)
2 ColoredPoint cp = new Point(...); // NON, on instancie pas un enfant avec le parent
```

```

3 System.out.println(p.getColor()); // On ne peut pas utiliser une méthode de l'enfant
   ↳ dans le parent
4 p.display(); // méthode de ColoredPoint est utilisé, une fois que le programme
   ↳ compile

```

De manière générale, la classe parent n'a pas accès aux méthodes des classes enfants. Tandis que l'enfant à accès aux membres du parent.

Cependant, il faut se méfier de l'instanciation en mélangeant enfants et parents. Java utilise le type déclaré (de déclaration de variable) à la **compilation**. De ce fait, on ne peut pas utiliser les méthodes d'une classe **réelle** enfant avec un type **déclaré** parent, la compilation échoue. Si la compilation se déroule sans soucis alors le *runtime*, la machine virtuelle utilisera le type **réel**. Dans le cas du Point au-dessus, une fois qu'il est compilé, son type **réel** est ColoredPoint et donc la méthode display() utilise la définition de ColoredPoint.

7.3 La classe 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-implementer 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 préfère l'utilisation d'un constructeur par copie ou d'une méthode statique, décrite ici-bas.

```

1 public Point(Point point){
2     this(point.x, point.y);
3 }
4
5 // alternative avec une méthode statique
6
7 public static Point newInstanceOf(Point point){
8     return new Point(point.x, point.y);
9 }
10
11 // Exemple :
12
13 Point p = new Point(1, 3);
14 Point p2 = new Point(p);
15
16 Point p = new Point(3, 4);
17 Point p2 = Point.newInstanceOf(p);

```

7.4 La classe 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)
```

```

1 Object.equals(o1, o2);
2 // Remplace
3 o1 == o2 || o1 != null && o1.equals(o2);

```

Il y a aussi `static boolean deepEquals(Object o1, Object o2)` qui exécute la même opération mais pour les tableaux.

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

```
1 this.bar = Objects.requireNonNull(bar);
2 // Remplace
3 if(bar == null){
4     throw new NullPointerException("bar is null");
5 }
6 this.bar = bar;
```

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

```
1 @Override
2 public int hashCode(){
3     return Objects.hash(x,y);
4 }
```

Ceci clôt la section sur l'héritage.

8 Énumérations et exceptions

Cette section traite des énumérations, une ensemble de valeurs fixes et des exceptions, qui sont lancées à divers niveaux du code.

8.1 Énumérations

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...

En Java, on utilisera l'énumération. 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 :

```
1 public enum Saison {
2     PRINTEMPS, ÉTÉ, AUTOMNE, HIVER;
3 }
```

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.

8.1.1 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.


```

1  import java.time.LocalDate;
2  public enum Season {
3      SPRING(21, 3),
4      SUMMER(22, 6),
5      FALL(23, 9),
6      WINTER(21, 12);
7
8      private LocalDate begin;
9
10     private Season(int day, int month){
11         var now = LocalDate.now();
12         this.begin = LocalDate.of(now.getYear(), month, day);
13     }
14
15     public LocalDate getBegin(){
16         return this.begin;
17     }
18
19     public static Period when(Season s){
20         var now = LocalDate.now();
21         return now.until(s.begin);
22     }
23
24     public Season next(){
25         return Season.values()[this.ordinal() + 1] % 4];
26     }
27 }

```

8.2 Exceptions

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 :

```

1  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 améliore 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.

```

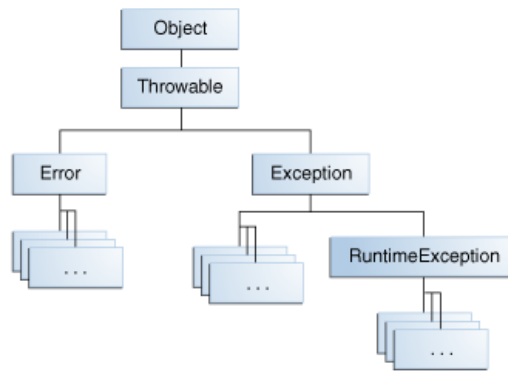
1  try {
2      // code pouvant lancer une exception
3  } catch (IllegalArgumentException e){
4      // gestion de l'erreur
5  }

```

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.

8.2.1 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.

```

1 public void myMethod() throws FileNotFoundException {
2     //...
3     throw new FileNotFoundException("Raison de l'exception");
4     //...
5 }

```

Une exception contrôlée doit être gérée :

```

1 try {
2     myMethod();
3 } catch (FileNotFoundException ex){
4     // gérer l'exception
5 }

```

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

```

1 public void otherMethod() throws FileNotFoundException {
2     myMethod();
3 }

```

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

8.2.2 Créer sa propre exception

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

```

1 public class MyException extends Exception {
2     public MyException(String s){
3         super(s);
4     }
5 }

```

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**.

8.2.3 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.

```
1 try {
2     // code
3 } catch (MyException e1){
4     // gestion de l'exception e1
5 } catch (MyException e2) {
6     // gestion de l'exception e2
7 }
```

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

```
1 try {
2     // code
3 } catch (MyException | IOException e){
4     // code pour n'importe laquelle des deux exceptions
5 }
```

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*.

```
1 String path = "cheminDuFichier";
2 try (BufferedReader br = new BufferedReader(new FileReader(path))){
3     return br.readLine();
4 } catch (NoSuchFileException e){
5     // traitement de l'absence du fichier
6 }
```

8.3 Le mot-clé `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`.

```
1 import java.util.ArrayList;
2 var i = 3; // Infère int
3 var s = "Hello"; // Infère String
4 var arr = new ArrayList<Integer>(); // Infère ArrayList<Integer>
5 //...
```

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.

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

9 Le codage des fichiers

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. Comme elle est très limitée au niveau des caractères disponibles, de nouvelles tables ont été créées, comme l'UTF-8 ou même UTF-16.

9.1 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	00000000	00000000	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.

9.2 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+FFFF	11100000 10xxxxxx 10xxxxxx	E0 (le 2 ^e octet est restreint de A0 à BF)	3 octets, codant 16 bits
U+1000 à U+1FFF	11100001 10xxxxxx 10xxxxxx	E1	
U+2000 à U+3FFF	1110001x 10xxxxxx 10xxxxxx	E2 à E3	
U+4000 à U+7FFF	111001xx 10xxxxxx 10xxxxxx	E4 à E7	
U+8000 à U+BFFF	111010xx 10xxxxxx 10xxxxxx	E8 à EB	
U+C000 à U+FFFF	1110110x 10xxxxxx 10xxxxxx	EC	
U+D000 à U+D7FF	11101101 10xxxxxx 10xxxxxx	ED (le 2 ^e octet est restreint de 80 à 9F)	
U+E000 à U+FFFF	1110111x 10xxxxxx 10xxxxxx	EE à EF	4 octets, codant 21 bits
U+10000 à U+1FFFF	11110000 10xxxxxx 10xxxxxx 10xxxxxx	F0 (le 2 ^e octet est restreint de 90 à BF)	
U+20000 à U+3FFFF	11110001 10xxxxxx 10xxxxxx 10xxxxxx	F1	
U+40000 à U+7FFFF	11110001 10xxxxxx 10xxxxxx 10xxxxxx	F2 à F3	
U+80000 à U+FFFFF	1111001x 10xxxxxx 10xxxxxx 10xxxxxx	F4	
U+100000 à U+10FFFF	1111010x 10xxxxxx 10xxxxxx 10xxxxxx	F4 (le 2 ^e 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	11000101 10010011
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.

```

1 hexdump -C euro.txt
2
3 # 00000000 e2 82 ac /.../
4 # 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.

9.3 Trouver son chemin

Un **fichier** est identifié par son chemin à travers le *filesystem*. On parle aussi de son nom complètement qualifié ou de son chemin, son *path*.

Par exemple :

- /home/nathan/file.pdf
- C:\Users\Nathan\dir\Hello.java

Attention, le séparateur est différent en fonction des OS. Aussi, un chemin peut être relatif ou absolu et certains systèmes permettent des liens **symboliques**.

9.3.1 Paths

L'interface Path en java représente un chemin et permet de le manipuler. On peut créer des *paths*, les convertir, les comparer. Comme d'habitude les noms de classe se terminant par s en Java donnent accès à des méthodes utilitaires. Ici, Paths est une classe utilitaire pour travailler avec des chemins.

```

1 Path path1 = Paths.get("/tmp/foo");
2 Path path2 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");

```

Aussi, le fichier que le chemin représente peut ne pas exister. On remarquera que Paths.get() reçoit un nombre différent d'arguments dans l'exemple du dessus. Cette notion de nombres d'arguments variables est appelée **varargs**. Elle est abordée au point suivant.

9.3.2 Varargs

La signature de la méthode utilisée plus haute est :

```

1 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,

```

1 public void foo(String... args){
2     System.out.println("Nb de params reçus : " + args.length);
3     for(String arg : args){
4         System.out.println(arg);
5     }
6 }

```

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

```
1 public class SumExample{
2     public static int sum(int... args){
3         int sum = 0;
4         for(int el : args){sum += el;}
5         return sum;
6     }
7     public static void main(String[] args){
8         int sum1 = sum(1, 2, 3, 4, 5); // 15
9         int sum2 = sum(1, 2); // 3
10        int sum3 = sum(2, 4, 6, 8, 10); // 30
11        System.out.println("sum1 : " + sum1 + ", sum2 : " + sum2 + ", sum3 : " +
12                               sum3);
13    }
```

sum1 : 15 sum2 : 3 sum3 : 30

9.3.3 La classe Path

Path fournit des méthodes pour interroger et travailler avec des chemins. Voici un exemple d'utilisation :

```
1 import java.nio.file.Path;
2 import java.nio.file.Paths;
3
4 Path path = Paths.get("/home/nathan/foo");
5 path.toString();
6 path.getFileName();
7 path.getParent();
8 path.getRoot();
9 path.toAbsolutePath();
10 path.resolve(Path);
11 //...
```

De manière générale, on pourra directement naviguer dans le système via cette classe et on opère sur les chemins avec la classe utilitaire Paths.

9.4 La classe Files

En plus des chemins, on peut travailler plus directement avec des fichiers via la classe utilitaire Files.

```
1 import java.nio.file.*;
2
3 public class WorkingWithFiles{
4     public static void main(String[] args){
5         Path path = Paths.get("/home/nathan/tmp/Hello.java");
6         System.out.println(Files.exists(path));
7         System.out.println(Files.isWritable(path));
8         System.out.println(Files.size(path));
9         /...
10    }
11 }
```

Ces méthodes donnent aussi accès aux méta-données des fichiers.

9.5 Les entrées-sorties de bas niveau

Cette section est consacrée aux entrées et sorties bas niveau, c'est-à-dire qu'on lit ou écrit les fichiers *byte* par *byte*. En Java, les fichiers sont des flux (*streams*). Ils sont **uniformes**, **non-structurés** et **binaire** ou **texte**.

Pour le format texte, on utilise les `FileReader` et `FileWriter`. Tandis que pour les fichiers binaires, on utilise les `InputStream` et `OutputStream`.

9.5.1 Binaire

Voici un exemple de lecture **binaire** d'un fichier :

```
1 import java.io.IOException;
2 import java.nio.*;
3 int b;
4 try(InputStream in = Files.newInputStream(Path.get("filename"),
5                                     StandardOpenOption.READ)){
6     b = in.read();
7     while(b != -1){
8         System.out.println(" " + b);
9         b = in.read();
10    }
11 } catch (IOException e){
12     System.err.println("Error : " + e.getMessage());
13 }
```

Il y a plusieurs choses importantes à voir. Premièrement, on lit des **byte** mais on donne le type **int**, codé sur 4 *bytes*, à la variable `b`. Quand on lit un fichier binaire, on lit le type primitif **byte**, codé sur un *byte*. Quand on lit un fichier texte, on lit le type primitif **char**, codé sur deux *bytes*. Pour différencier les lectures de fin de fichier de l'information, on doit coder l'information sur 4 *bytes*, d'où l'utilisation de **int**.

Secondement, on continue la lecture tant que la variable `b` est différente de -1. C'est parce que la valeur -1 indique la fin de fichier, c'est une valeur **sentinelle** qui n'a pas d'utilité en soi, à part indiquer la fin du fichier.

On utilise dans les exemples plusieurs classes utiles :

`InputStream` permet la lecture binaire de bas niveau.

`Files` classe utilitaire pour travailler avec les fichiers.

`newInputStream` retour un *input stream* sur le fichier concerné.

`StandardOpenOption` est une énumération.

L'`InputStream` est **closeable**, il sera fermé automatiquement par le *try with resources*.

`IOException` exceptions liées à l'input/output (I/O).

Maintenant passons à l'écriture **binaire** dans un fichier.

```
1 //... Même imports, classe qu'avant
2
3 try(OutputStream out = Files.newOutputStream(Paths.get("file2"),
4                                     StandardOpenOption.CREATE)){
5     out.write(64);
6 } catch (IOException e){
7     System.out.println("Erreur : " + e.getMessage());
8 }
```

9.5.2 Texte

L'utilisation est exactement la même dans le cas de fichiers textes, si ce n'est qu'on utilise des méthodes différentes dédiées au texte plutôt qu'au binaire.

`FileReader` pour lire un fichier et **int** `read()` lit un caractère (-1 si fin de fichier).

`FileWriter` pour écrire un fichier texte et **void** `write(int c)` écrit le caractère stocké dans la variable `c`.

`BufferedReader` et `BufferedWriter` qui lisent et écrivent aussi mais en utilisant un *buffer*.

10 Entrées et sorties de haut niveau

Après les lectures et écritures de bas niveau, l'API de Java propose aussi une série de **flux englobants** qui permettent de lire ligne par ligne plutôt que caractère par caractère ou byte par byte. Ces flux englobants se construisent à partir d'autres flux.

10.1 Les flux englobants

Pour écrire des données **primitives** dans un fichier **binaire**, on se base sur la classe `DataOutputStream`.

Par exemple :

```
1 import java.io.IOException;
2 import java.nio.*;
3 try(DataOutputStream out = new DataOutputStream(
4     Files.newOutputStream(Paths.get("file.data"),
5         StandardOpenOption.CREATE))) {
6     out.writeBoolean(true);
7     out.writeUTF("Hello");
8     out.writeDouble(2.5);
9 } catch(IOException e){
10     System.err.println("Error : " + e.getMessage());
11 }
```

Pour lire à partir d'un fichier **binaire**, on utilise la classe `DataInputStream`.

```
1 import java.io.IOException;
2 import java.nio.*;
3 try(DataInputStream in = new DataInputStream(
4     Files.newInputStream(Paths.get("file.data"),
5         StandardOpenOption.READ))) {
6     boolean b = in.readBoolean();
7     String s = in.readUTF();
8     double d = in.readDouble();
9     System.out.println("values : " + b + " " + s + " " + d);
10 } catch (IOException e) {
11     System.err.println("Error : " + e.getMessage());
12 }
```

Dans ce cas-ci, il n'y a pas de valeur sentinelle qui indique la fin de fichier. Le flux générera plutôt une `EOFException` s'il y a tentative de lecture au-delà de la fin de fichier. Du coup, on ajoute un `catch` :

```
1 catch(EOFException e){
2     // all data has already been read
3 }
```

Le `try with resources` s'occupe de fermer le fichier via `close`.

10.2 Les expressions régulières

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.

```
1 import java.util.regex.Pattern;
2
3 public class RegexExample{
4     public static void main(String[] args){
5         String s = "g12345";
6         s.matches("g\\d{5}"); // true
7         // Plus efficace si on compile l'expression avant de l'utiliser
8         Pattern pat = Pattern.compile("g\\d{5}");
9         pat.matcher(s).matches(); // true
10    }
11 }
```

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).

```
1 import java.util.regex.Pattern;
2
3 public class PatternMatching {
4     public static void main(String[] args){
5         Pattern cours = Pattern.compile("^([A-Z]{3}G?I?R?[1-6])$");
6         String[] tab = {"WEBG2", "DEV3", "DONGIR5", "WEBD1", "MIC2", "STA7",
7             ↪ "don3", "IMGI4", "SYSRG2"};
8         for(String s : tab)
9             System.out.print(cours.matcher(s).matches() + " ");
10    }
11 }
```

true true true false true false false true false

10.3 Retour sur Scanner et flux standards

On a déjà utilisé `System.in` et `System.out`. `System` est une classe et `[in|out]` sont des variables statiques. Ce `in` et `out` représente l'entrée standard et la sortie standard. En général, ce sont le clavier et l'écran.

Aussi, l'entrée standard est un `InputStream` et la sortie standard ainsi que la sortie d'erreur standard sont des `PrintStream`.

De ce fait, un objet `Scanner` peut recevoir des chaînes de caractères ou des fichiers par exemple et pas nécessairement des entrées standards.

Par exemple :

```
1 import java.util.*;
2
3 try(Scanner scanner = new Scanner(Paths.get("file"))){
4     int i = scanner.nextInt();
5     System.out.println("i : " + i);
6 } catch (IOException e){
7     System.err.println("Erreur : "+e.getMessage());
8 }
```

On peut aussi écrire des données primitives dans un fichier texte, via la classe `PrintWriter`, qui est un flux englobant.

```
1 import java.util.*;
2
3 try(PrintWriter out = new PrintWriter(Files.newBufferedWriter(Paths.get("file"),
4                                     StandardOpenOption.CREATE))){
5     out.println(10);
6     out.printf("%04d\n%4.2f", 12, 12.2);
7 } catch (IOException e){
8     System.err.println("Erreur : " + e.getMessage());
9 }
```

10.4 La classe Console

Pour des entrées et sorties *via* la console. La classe `Console` peut se substituer à `Scanner` et à des `System.out`.

Par exemple :

```
1 import java.util.*;
2
3 Console c = System.console();
4 // if console not null
5 String name = c.readLine("Enter name : ");
6 c.format("Your name is %s", name);
7 char[] password = c.readPassword("Password : ");
8 // some work
9 Arrays.fill(password, ' ');
```

10.5 Sérialisation

Dès lors qu'un objet est **sérialisable**, il pourra être transformé en une suite d'octets.

- Chaque attribut doit être sérialisable.
- `Serializable` est une interface de *tag*. Elle ne fait rien mais permet de déclarer qu'une classe est sérialisable lorsqu'elle l'implémente.
- La classe doit être la même à l'écriture et à la lecture, elle vérifie cette égalité par un identifiant `serialVersionUID`.

Voici un exemple :

```
1 import java.util.*;
2
3 try(ObjectOutput out = new ObjectOutputStream(
4     Files.newOutputStream(Paths.get("file.ser"),
5                             StandardOpenOption.CREATE))){
6     out.writeObject(new MyObject("Anonymous obj", 7));
7 } catch (IOException e){
8     System.err.println("Erreur : " + e.getMessage());
9 }
10 // extrait d'une lecture
11
12 MyObject mo = (MyObject) in.readObject();
```

11 Programmation fonctionnelle

À partir de Java 8, on peut utiliser la programmation **fonctionnelle** qui se concentre sur l'utilisation de fonctions.

11.1 Les fonctions lambda λ - Introduction

En Java, on utilise les fonctions λ comme des fonctions anonymes et compactes. Voici quelques exemples qui ne tournent pas mais qui donne une vision générale du fonctionnement.

```

1 x -> x*x; // lambda
2
3 // Équivalent non-fonctionnel
4 public int anonymous(int x){
5     return x*x;
6 }

```

Voici quelques exemples d'usage, qui seront normalement reçus comme **paramètre** d'une méthode comme `forEach`.

```

1 x -> x > 0 // true si positif
2 s -> System.out.println(s) // Affiche le paramètre
3 () -> Math.random() // Retourne un nombre aléatoire
4 (a, b) -> a > b ? a : b // Retourne le maximum

```

11.1.1 Application

Voici différentes manières d'imprimer une valeur de liste.

```

1 import java.util.List;
2
3 List<String> mots = List.of("ceci", "est", "une", "phrase");

```

On peut l'imprimer avec les boucles classiques :

```

1 for(int i = 0; i < mots.size(); i++){
2     System.out.println(i);
3 }
4
5 for(String s : mots){
6     System.out.println(s);
7 }

```

Mais on peut aussi utiliser l'interface fonctionnelle pour le faire, cette interface est valable pour tout ce qui est *Iterable* sauf les tableaux. On voit qu'on passe une méthode au `forEach` et que la méthode est utilisée sur chaque valeur de la liste.

```

1 mots.forEach(s -> System.out.println(s));

```

On peut même directement faire référence à la méthode, elle prend en argument les éléments de l'objet itérable. On en détaillera l'usage [plus tard](#).

```

1 mots.forEach(System.out::println);

```

Pour savoir si une méthode accepte une fonction λ , on peut aller le vérifier dans la [Javadoc](#). Par exemple, la signature de `forEach` est :

```

1 void forEach(Consumer<? super T> action)

```

et la *Javadoc* de `Consumer` donne l'entête.

```

1 @FunctionalInterface public interface Consumer<T>

```

Donc, un paramètre accepte une fonction λ si c'est une **interface** et si elle est annotée `@FunctionalInterface`. C'est une condition suffisante mais nécessaire.

Aussi, on ne doit pas forcément utiliser l'interface `Consumer`, cette dernière est utilisée pour les fonctions qui reçoivent un paramètre mais ne retournent rien. Il existe aussi les interfaces `Predicate` pour travailler avec les booléens par exemple. Il y a aussi des interfaces de fonctions.

Il faut noter que dans une fonction λ , on n'indique pas les types. Ils seront inférés, sur base des méthodes qu'on utilise car elles entrent en paramètre de fonction.

Néanmoins, il faudra parfois le préciser et on peut simplement utiliser le *type casting*.

```
1 (int) x -> x*x
```

11.1.2 Référence de méthode

Quand le code de la fonction λ est juste un appel de méthode, il existe une notation raccourcie, **la référence de méthode**.

Par exemple,

```
1 x -> System.out.println(x)
2
3 // Référence de méthode
4
5 System.out::println
```

et donc le `forEach` devient :

```
1 myList.forEach(System.out::println)
```

11.2 Retour sur les tris avec `Comparable`

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 :

```
1 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.

```
1 public int compareTo(Personne other){
2     return this.name.compareTo(other.name);
3 }
```

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.

11.3 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 certaines 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 :

```
1 import java.util.*;
2
3 List<String> mots = List.of("Ceci", "est", "une", "liste", "immuable", "de",
4   ↪ "String");
5 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](#).

```
1 Collections.sort(mots,
2   (m1, m2) -> m1.length() < m2.length() ?
3   -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`.

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

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

```
1 Collections.sort(mots,
2   Comparator.comparing(m -> m.length()));
3
4 // Ou encore par référence de méthode
5
6 Collections.sort(mots,
7   Comparator.comparing(String::length));
8
9 // À l'envers !
10
11 Collections.sort(mots,
12   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.

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

11.4 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.

Différence entre Comparable et Comparator

11.5 Les fonctions lambda - Explications

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. Nous allons donc voir les étapes qu'il fallait suivre à l'époque et comment elles ont évoluées.

11.5.1 Approche par classe nommée

```

1 public class SortedByLength implements Comparator<String> {
2     @Override
3     public int compare(String s1, String s2){
4         return Integer.compare(s1.length(), s2.length());
5     }
6 }
```

On voit qu'on a pu créer une classe qui implémente la méthode compare et qui fonctionne de manière attendue.

```

1 Collections.sort(mots, new SortedByLength());
```

11.5.2 Approche par classe anonyme

La méthode précédente nécessite de créer une classe alors qu'on ne va peut-être l'utiliser que peu de fois, on peut simplement créer une classe anonyme.

```

1 Collections.sort(mots, new Comparator<String>() {
2     @Override
3     public int compare(String s1, String s2){
4         return Integer.compare(s1.length(), s2.length());
5     }
6 });
```

11.5.3 Approche par fonction λ

On voit qu'on se rapproche de plus en plus de la notation λ , on peut écrire :

```

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

On peut voir qu'une fonction lambda est un objet et on peut donc la sauver dans une variable pour la réutiliser ou la rendre plus compréhensible.

```

1 Comparator<String> sortByLength = (m1, m2) -> Integer.compare(m1.length(),
2     ↪ m2.length());
2 Collections.sort(mots, sortByLength);
```

11.6 Les streams - Introduction

Attention, les *streams* ne sont pas les mêmes que ceux qu'on a vu pendant les entrées et sorties. Le *stream* ne fait pas partie intégrante du fonctionnel mais permet une écriture plus facile et compacte.

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.

Si on voulait mesurer le salaire moyen d'un objet *Employe* en ne conservant que les femmes, alors on peut simplement utiliser l'expression suivante :

```
1 var average = employees.stream()
2   .filter(p -> p.getGender().equals("F"))
3   .mapToInt(Person::getSalary)
4   .average();
```

C'est une approche très déclarative et séquencée où chaque sortie d'une action devient l'entrée de l'action suivante. Voici à quoi pourrait ressembler une opération équivalente de manière classique.

```
1 var sumSalaries = 0.0;
2 var womenCount = 0;
3 for(Employe p : employees){
4     if (p.getGender() == .Gender.FEMALE){
5         sumSalaries += p.getSalary();
6         womenCount++;
7     }
8 }
9 System.out.println(sumSalaries/womenCount); // calcul de la moyenne
```

11.7 Les streams - Détails et exemples

Ce n'est évidemment pas tout ce qu'on peut dire sur les *streams*, cette section aborde leur utilisation plus en détails.

11.7.1 Création de Stream

La première étape dans l'utilisation dans *stream* est de le créer, on doit indiquer quel type d'éléments vont être utilisées dans le *pipeline*. Voici différentes manières de faire :

- Via une collection, comme une liste : `myCollection.stream()`
- À partir d'un tableau (*array*) : `Arrays.stream(myArray)`
- Avec des nombres aléatoires : `new Random().ints(1, 100)`
 - Ce sont ici, des nombres aléatoires entre 1 et 100.
- Avec un **générateur** : `Stream.iterate(1, n->n+1)`
 - Le premier paramètre est la valeur de départ
 - Le second paramètre indique la logique à suivre pour itérer. Ici on incrémente les éléments par 1. Donc on aura 1, 2, 3, 4...

Attention, les deux derniers exemples ne précisent pas **combien** d'éléments sont concernés, on peut donc avoir des générateurs **infinis**. La limite viendra de la machine ou d'une autre opération.

11.7.2 Opérations intermédiaires (non-terminales)

Les opérations intermédiaires agissent sur les éléments du *stream* et peuvent être chaînées autant de fois qu'on le désire avant de rencontrer une opération terminale. On a par exemple, le `map`, `filter`, `limit`, `sorted`, etc.

Les opérations non terminales ne modifient pas directement la collection ou le tableau utilisé, si on filtre ou modifie des éléments, la collection (ou tableau) initiale n'est pas modifiée. En fait, chaque opération de *stream* retourne un *stream* et on devra utiliser une opération terminale pour avoir un résultat utile.

11.7.3 Opérations terminales

La dernière étape d'un *stream* est une opération terminale qui se place en fin de chaîne. Il ne peut y en avoir qu'une. On peut par exemple :

- Récouter les éléments :

- Dans une liste : `.collect(Collectors.toList())`
- Dans un tableau : `toArray()`
- Poser une question à la fin de chaîne :
 - `allMatch(n -> n > 12)` : Est-ce que tous les éléments sont plus grands que 12 ?
 - `anyMatch(n -> n > 12)` : Est-ce qu'au moins un élément est plus grand que 12 ?
- Agréger l'information :
 - `count()` : Compte le nombre d'éléments en bout de chaîne.
 - `count()` : Somme les éléments, si ce sont des nombres.
 - `count()` : Calcule la moyenne, si ce sont des nombres.
- Sélectionner une partie :
 - `max()` : donne le maximum.
 - `findFirst()` : donne le premier élément.
- Effectuer une opération sur les données :
 - `forEach(System.out::println)`
 - `forEach(p -> p.salaryIncrease(.15))`

Le document sur le fonctionnel en Java reprend des exemples d'utilisation d'opérations intermédiaires et terminales.

Voici malgré tout quelques exemples :

```
1 import java.util.ArrayList;
2 import java.util.Collectors;
3
4 var arr = new ArrayList<Integer>(List.of(1, 2, 3, 4, 5, 6, 7, 8));
5
6 arr.stream().map(x -> x+1).filter(x -> x > 5).collect(Collectors.toList());
7 // résultat : [6, 7, 8, 9], c'est une ArrayList
8 // arr n'a pas été modifié
```

On comprend deux choses importantes de cet exemple : les opérations sont effectuées de manière séquentielle et donc le filtre est appliqué sur les éléments qui ont été incrémenté de 1, pas sur la liste initiale. Aussi, on collecte le résultat dans une liste, **sans modifier** la liste initiale.

```
1 import java.util.ArrayList;
2 import java.util.Collectors;
3 import java.util.Comparator;
4
5 var arr = new ArrayList<Integer>(List.of(1, 2, 3, 4, 5, 6, 7, 8));
6
7 arr.stream().sorted(Comparator.reverseOrder()).skip(3).collect(Collectors.toList());
8 // résultat : [5, 4, 3, 2, 1]
9 // Trier du plus grand au plus petit, passer les trois premières valeurs et collecter
```

11.7.4 Précisions

Lorsqu'on crée un *stream*, on obtient un objet de la classe `Stream`. Par exemple :

```
1 Stream<Employe> stream = employees.stream();
```

Ce sont des *streams* génériques qui couvrent la plupart des collections. Mais il existe aussi des *streams* plus spécialisés qui ont des méthodes additionnelles. Par exemple le `IntStream`.

```
1 IntStream.rangeClosed(1, 50).sum() // somme des nombres de 1 à 50
```

Si on veut convertir à un *stream* en entiers, on peut utiliser `mapToInt` avec une méthode qui convertit un type.

```
1 var arr = new ArrayList<>(List.of('1', '2', '3', '4')); // caractères
2 arr.stream().mapToInt(Character::getNumericValue).sum(); // 10
```


11.7.5 Parallélisme

On peut aussi utiliser `.parallelStream()` pour obtenir un *stream* **parallèle**. Dans ce cas, Java va essayer d'utiliser les processeurs à bon escient pour accélérer les calculs. Il vaut mieux consulter la documentation pour voir les cas dans lesquels c'est utile et ceux dans lesquels ça ne l'est pas.

```
1 employees.parallelStream()  
2   .map(Employees::getSalary) // Un stream Stream<Integer>  
3   .mapToInt(Integer::valueOf) // Un IntStream
```

12 Concepts importants

Cette section reprend les concepts importants vus au cours et aux laboratoires, ils doivent être définis et expliqués correctement.

encapsulation décrite plus en détail [via Wikipédia](#).

equals décrite plus en détail dans la [documentation java](#).

toString décrite plus en détail dans la [documentation java](#).

liste décrite plus en détail dans la [documentation java](#).

ArrayList décrite plus en détail dans la [documentation java](#).

Lexique

ArrayList Une implémentation de l'interface liste via un tableau dont on peut changer la taille. Cette implémentation permet donc d'agrandir ou réduire le tableau qui la compose. [41](#)

accesseur les accesseurs permettent de récupérer la valeur de données membres privées sans y accéder directement de l'extérieur ; ils sécurisent donc l'attribut en restreignant sa modification. [5](#)

classe Une classe est une description des caractéristiques d'un ou de plusieurs objets. Chaque objet créé à partir de cette classe est une instance de la classe en question. [3, 4](#)

compilateur En informatique, un compilateur est un programme qui transforme un code source en un code objet. Généralement, le code source est écrit dans un langage de programmation (le langage source), il est de haut niveau d'abstraction, et facilement compréhensible par l'humain. [9, 10](#)

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. [4, 6, 7](#)

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. [7, 41](#)

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`. [41](#)

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. [10–13](#)

héritage Mécanisme typique de l'orienté-objet qui permet, lors de la déclaration d'une nouvelle classe (enfant), d'y inclure les caractéristiques d'une autre classe (parent). En Java, une classe hérite d'une autre si elle l'étend (`extends`). Toujours en Java, on ne peut hériter que d'une seule classe. [4, 5, 22](#)

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. [10–12](#)

liste Une interface qui permet d'implémenter des séquences (des collections ordonnées), ces implémentations utilisées dans le cours sont notamment `ArrayList` et `LinkedList`. [41](#)

mutateur les mutateurs permettent de modifier l'état de données membres tout en vérifiant si la valeur que l'on veut donner à la donnée membre respecte les normes de celle-ci ou diverses règles de cohérence. [5](#)

méthode Une méthode est une suite d'instructions qui manipule les caractéristiques et l'état d'un objet. 3, 4, 6, 24

objet un objet est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant un sujet, manipulés dans un programme. 3, 4, 26

surcharge 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. 6

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. 41

état L'état d'un objet est sa forme à un instant donné, telle que décrite par les valeurs de l'ensemble de ses propriétés. 3, 4

13 Crédits

Cours de Marco Codutti et Pierre Bettens à l'ESI, 2020-2021.