**Cours Programmation Orientée Objet**

**Chapitre 2 : L’encapsulation**

# Introduction

Dans ce chapitre, nous allons voir le concept ‘encapsulation’, à quoi il sert et pourquoi il est si important dans la programmation orientée objet.

L’encapsulation est une notion fondamentale en programmation orientée objet. Elle permet de regrouper les données (attributs) et les méthodes en une seule unité et garantir un accès restreint à ces données.

Tout d’abord, nous commençons par présenter les niveaux de visibilités des membres d’une classe. Ensuite, nous définissons l’encapsulation des attributs et des méthodes. Enfin, nous allons voir l’encapsulation en Java.

# Niveaux de visibilité

L’encapsulation définit des niveaux de visibilité des membres (attributs et méthodes) de la classe. Ces niveaux de visibilité déterminent les droits d’accès aux données. Le tableau ci-dessous présente les différents niveaux de visibilité :

|  |  |  |
| --- | --- | --- |
| Modificateur | Mot clé en Java | Rôle |
| Publique « + » | **public** | Une variable, méthode ou classe déclarée public est visible par tous les autres objets. |
| Protégé « # » | **protected** | Une méthode ou un attribut déclaré protected, sont visible uniquement dans cette classe ou ses sous-classes. Une classe ne peut pas être déclarée *protected.* |
| Privé « - » | **private** | C'est le niveau de protection le plus fort. Les membres ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. |

Tableau 1: Niveaux de visibilité

# 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. Elle permet de restreindre l’accès direct aux données et empêche la modification de l’objet hors de ses méthodes. L’encapsulation garantira l’intégrité des données contenues dans l'objet.

Prenons l'exemple d'une voiture où sont disponibles plusieurs commandes (volant, vitesse, pédales ...) pour la conduire. Chacun de ces commandes constitue une action qu’on peut effectuer sur la voiture. Un chauffeur ne s’intéresse pas de qui est composé la voiture, son rôle est la conduire. Pour cela, il va se servir des commandes afin de conduire la voiture. Entre autres, le chauffeur ne doit pas se charger de modifier manuellement ces composants, sinon il risque de faire des erreurs.

Le principe est exactement le même pour la POO : l'utilisateur de la classe doit se contenter d'invoquer les méthodes en ignorant les attributs. Pour instaurer une telle contrainte, on dit que les attributs sont **privés**.

En effet, l’encapsulation introduit une manière pour sécuriser les membres d’une classe afin de garder une cohérence dans la gestion de l’objet.

L'encapsulation offre de nombreux avantages :

* Diminution des risques de mauvaise manipulation d'une classe ;
* Création de classes « boîtes noires » par masquage des détails internes ;
* Possibilité de modifier les détails internes d'une classe (la manière dont elle fonctionne) sans changer son comportement extérieur (ce qu'elle permet de faire).

## Encapsulation de données (attributs)

L’encapsulation des attributs permet d’interdire toute modification directe (accidentelle ou volontaire) des données d’une classe. En encapsulant un attribut il est possible de choisir si celui-ci doit être accéder en lecture ou en écriture. Dans le cas d’accès en écriture il faut définir les règles pour lesquelles les valeurs sont autorisés, par exemple l’attribut *âge* ne peut pas être négatif.

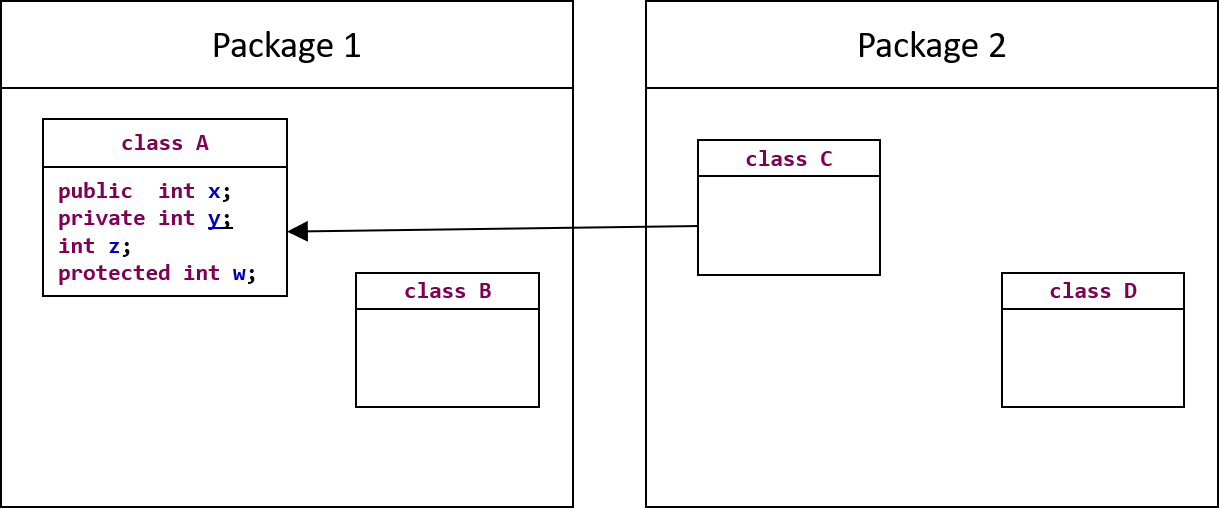


Figure 1 Encapsulation des attributs

En programmation orientée objet, on distingue différents types d’attributs : les **attributs publics**, les **attributs protégés** et les **attributs privés**. Un attribut privé n’est accessible *qu’à l’intérieur de la définition de la classe*. Par contre, les attributs publics sont accessibles partout et toujours.

Prenons l’exemple présenter sur la Figure 1, le tableau suivant résume la portée (visibilité) des attributs déclarés dans la classe A :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Visibilité à partir de : | Classe A | Package classe B | Classe fille C | Tout le reste classe D |
| Public x |  |  |  |  |
| Private y |  |  |  |  |
| Default z |  |  |  |  |
| Protected w |  |  |  |  |

Tableau 2 visibilité des attributs

## Encapsulation de code (méthodes)

Le même principe d’encapsulation s’applique aux méthodes : **on peut définir des méthodes privées, protégées ou publiques**. Les méthodes publiques sont toujours accessibles alors que les méthodes privées ne sont accessibles qu’à l’intérieur de la classe.

## Diagramme de classe

En utilise les symboles « + » « - », « # » ou « ~ » dans un diagramme de classe pour définir la visibilité d’un membre de la classe : *public, private, protected ou default. (voir* Figure 2*)*.

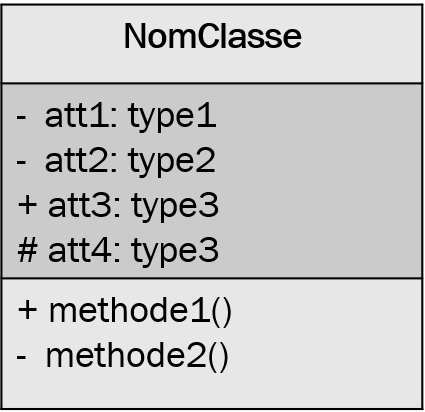


Figure 2 Diagramme de classe

## Encapsulation en Java

L'encapsulation en Java fait référence à l'intégration des données (variables) et du code (méthodes) en une seule unité. Dans l'encapsulation, les variables d'une classe sont cachées aux autres classes et ne sont accessibles que par les méthodes de la classe dans laquelle elles se trouvent. En ci-dessous la syntaxe à suivre pour implémenter l'encapsulation en Java.

*Classe* : < modificateur d’accès> class <Nom de la classe>

*Attribut* : < modificateur d’accès> <type> <nomAttribut>

*Méthode* : <modificateur> <type de retour> <nomMethode> (<paramètres>)

## Contrôle d’accès

Java fournit 4 contrôles d’accès à une classe et aux membres d’une classe : *default*, *public*, *private* et *protected*.

* **Default :** Le niveau par défaut est lorsqu’on spécifie aucun modificateur. Le membre (classe, méthode ou attribut) est visible par tous les classes se trouvant dans le même package.
* **Private :** Le spécificateur d'accès privé donne accès aux données, et les méthodes se limitent à la classe elle-même.
* **Protected :** le membre est visible par les classes dans le même package et les sous classes.
* **Public :** le membre est accessible par tous les classes.

**Exemple :** modifier la classe Voiture (vu en chapitre 1), pour restreindre l’accès à ces attributs.

**public** **class** Voiture {

**private** **int** vitesse;

**private** **int** transmission;

**private** String marque;

**public** Voiture(**int** v, **int** t, String m) {

vitesse = v;

transmission = t;

marque = m;

}

**public** **void** accelerer(**int** v) {

vitesse +=v;

setTransmission(vitesse/10);

}

**public** **void** freiner(**int** v) {

vitesse -=v;

setTransmission(vitesse/10);

}

**private** **void** setTransmission(**int** t) {

**if** (t <= 6)

transmission = t;

**else**

transmission = 6;

}

**public** **int** getTransmission() {

**return** transmission;

}

**public** String toString() {

**return** "Voiture [vitesse=" + vitesse + ", transmission=" + transmission + ", marque=" + marque + "]";

}

## Accesseurs (getters et setters)

En java, les méthodes permettant d’accéder en lecture aux données sont des accesseurs (getters) et celles permettant de modifier les données sont des mutateurs (setters). Ces méthodes, généralement, sont publique pour qu’on puisse y’accéder depuis une autre classe.

Les accesseurs sont du même type que la variable qu’ils doivent retourner. La syntaxe Java d’une méthode accesseurs est la suivante :

**public** <type de l’attribut> getAttribut(){

**return** attribut ;

}

Par contre, les mutateurs sont de type *void.* Ces méthodes ne retournent aucune valeur, leurs rôles est de mettre ces valeurs à jour. La syntaxe java d’une méthode mutateur (setter) :

**public** void setAttribut(<type de l’attribut> nouvelleValeur){

attribut = nouvelleValeur ;

}

Exemple :

**public** **class** A {

**private** **int** a;

**public** **int** getA() {

**return** a;

}

**public** **void** setA(**int** a) {

a = a;

}

}

## Accès à l’instance (this)

En analysant le corps de la méthode mutateur *setA(int a)* de l’exemple précédent, on peut avoir une ambiguïté dans l’instruction ‘*a=a’*. Le paramètre et l’attribut ont un nom identique ‘*a’.* Dans ce cas on peut dire que l’attribut de l’objet est ombragé par un paramètre de méthode (ou de constructeur). Pour lever l’ambigüité on utilise le mot clé ‘*this’* pour qualifier l’attribut de l’objet et on écrit *this.a = a*. Cela signifie que la valeur de l’attribut ‘a’ est mis à jour par la nouvelle valeur du paramètre ‘a’.

Dans une méthode d'instance ou un constructeur, ‘*this’* est une référence à l'objet courant - l'objet dont la méthode ou le constructeur est appelé. On peut faire référence à n'importe quel membre de l'objet courant à partir d'une méthode d'instance ou d'un constructeur en utilisant ‘*this’.*

Exemple : modifier le constructeur de la classe voiture :

**public** **class** Voiture {

**private** **int** vitesse;

**private** **int** transmission;

**private** String marque;

**public** Voiture(**int** vitesse, **int** transmission) {

**this**.vitesse = vitesse;

**this**.transmission = transmission;

}

**public** Voiture(**int** vitesse, **int** transmission, String marque) {

**this**(vitesse, transmission);

**this**.marque = marque;

}

}

Dans cet exemple, des noms communs ont été choisis pour les attributs et les paramètres associés du constructeur. L’utilisation de *this* devient alors obligatoire pour lever l’ambigüité entre paramètres du constructeur et attributs.

En plus, *this* permet également de chaîner des constructeurs de même niveau et éviter ainsi la duplication de code. La première ligne du deuxième constructeur montre cette utilisation pour effectuer un chainage entre les deux constructeurs.

## Variables et méthodes de classe (static)

Dans le chapitre 1, nous avons vu les type primitif (int, float, …, etc) et le type référence « objet ». Java propose un 3éme type, les variables et les méthodes de *statique,* aussi nommés variables ou méthode de classe. En utilisant le mot clé *static,* avant un membre (variable ou méthode) signifie que ce dernier appartient à la classe et on peut y’accéder en utilisant le nom de la classe, par exemple *MaClasse.attributStatique*.

Une variable statique partage la même valeur entre tous les objets de la même classe. Une méthode, pour être de classe, ne doit pas manipuler, directement ou indirectement, un attribut non statiques de sa classe. En conséquence, si une méthode de classe utilise dans son code un attribut ou une méthode non statique de sa classe, une erreur est détectée à la compilation.

On trouve un certain nombre de méthodes statiques dans l'API. C'est le cas de toutes les méthodes de la classe java.lang.Math : on pourra ainsi utiliser Math.sin(...), Math.log(...), de même que la constante statique Math.PI.

Exemple :

**public class** Comparaisons {

**static int** seuil = 10;

**static int** min(**int** a, **int** b) {

**if** (a < b) **return** a;

**else return** b;

}

**static boolean** grand(**int** a) {

**return** (a > Comparaisons.seuil);

}

}

**class** EssaiComparaisons {

**public static void** main(String[] arg) {

**int** n = 12;

**int** p = 4;

System.out.println("Le plus petit : " + Comparaisons.min(n, p));

**if** (Comparaisons.grand(n)) System.out.println(n + " est grand");

**else** System.out.println(n + " est petit");

System.out.println("Le seuil vaut " + Comparaisons.seuil);

}

}

On obtient à l'exécution, pour les paramètres 12 et 4 :

Le plus petit : 4

12 est grand

Le seuil vaut 10

# Les collections en Java

Une collection - parfois appelée conteneur - est simplement un objet qui regroupe plusieurs éléments en une seule unité. Les collections sont utilisées pour stocker, récupérer, manipuler et communiquer des données agrégées.

## Framework Java Collections

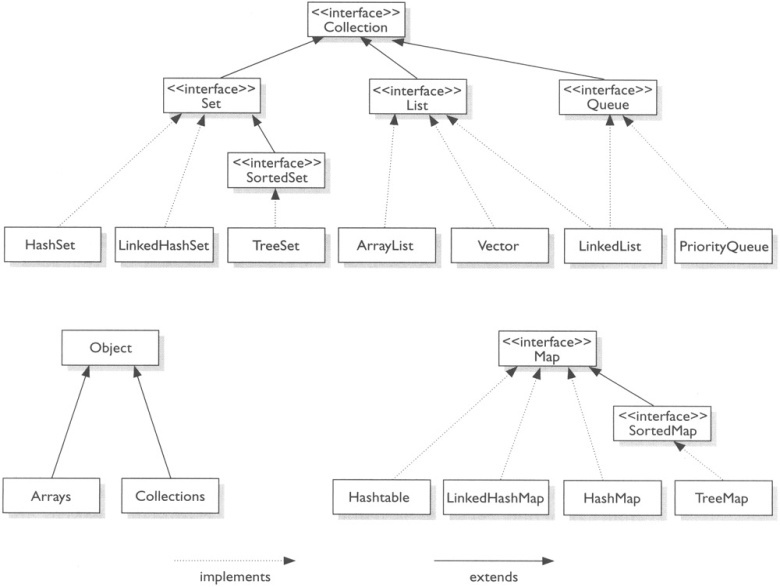


Figure 3Framework Java Collections

Le Framework Java collections est une architecture unifiée permettant de représenter et de manipuler des collections (voir Figure 1). Il contient les éléments suivants :

* **Des interfaces :** Il s'agit de types de données abstraits qui représentent des collections. Les interfaces permettent de manipuler les collections indépendamment des détails de leur représentation. Dans les langages orientés objet, les interfaces forment généralement une hiérarchie.
* **Des implémentations :** ce sont les implémentations concrètes des interfaces de collection. Il s'agit essentiellement des classes réutilisables.
* **Des algorithmes :** Il s'agit des méthodes qui effectuent des calculs utiles, tels que la recherche et le tri, sur des objets qui implémente les interfaces de collection. Les algorithmes sont dits polymorphes, c'est-à-dire que la même méthode peut être utilisée sur de nombreuses implémentations différentes de l'interface de collection appropriée.

## Syntaxe java

La forme générale pour déclarer un objet de type collection est la suivante :

Interface<E> objet = new Implementation<E>()

La syntaxe <E> indique que l'interface est générique. Lorsque on déclare une instance de collection, on doit spécifier le type d'objet contenu dans la collection. La spécification du type permet au compilateur de vérifier (au moment de la compilation) que le type d'objet que on place dans la collection est correct, ce qui réduit les erreurs au moment de l'exécution.

**Par exemple** : List<Integer> listEntier = **new** ArrayList<Integer>();

‘listEntier’ est un objet qui permet de manipuler une liste (tableau) des entier.

List<Float> listFloat = **new** ArrayList<Float>();

‘listFloat’ est un objet qui permet de manipuler une liste (tableau) des flottants.

Le tableau 3 représente les interfaces existantes et leurs implémentations dans le Framework Java Collections :

|  |  |
| --- | --- |
| L’interface | Implémentation |
| List : une collection ordonnée qui permet de contrôler la position d’insertion d’un élément. Il est possible d'accéder aux éléments par leur index. | **ArrayList**  **LinkedList** |
| Set : une collection qui ne peut pas contenir d'éléments en double. | **HashSet**  **LinkedHashset** |
| Queue : Utilisée pour conserver plusieurs éléments avant de les traiter. En plus des opérations de base, elle permet d'effectuer des opérations supplémentaires d'insertion, d'extraction et d'inspection. | **LinkedList**  **PriorityQueue** |
| Map : un objet qui associe des clés à des valeurs. Une Map ne peut pas contenir de clés dupliquées ; chaque clé ne peut correspondre qu'à une seule valeur. | **HashMap**  **TreeMap** |

Tableau 3. Interfaces et implémentations

## Méthodes utiles

Le *Tableau 4* présente quelques méthodes utiles pour la manipulation des collections en Java.

|  |  |
| --- | --- |
| Méthode | Description |
| boolean add(E element) | Ajoute l’élément à la fin de la liste |
| void add(int index, E element) | Insère l'élément spécifié à la position spécifiée dans cette liste. Décale l'élément qui se trouve actuellement à cette position (le cas échéant) et tous les éléments suivants vers la droite |
| boolean addAll(Collection c) | Ajoute tous les éléments de la collection spécifiée à la fin de cette liste |
| E get(int index) | Renvoie l'élément à la position spécifiée dans cette liste. |
| int indexOf(E element) | Renvoie l'index de la première occurrence de l'élément spécifié dans cette liste, ou -1 si cette liste ne contient pas l'élément. |
| E set(int index, E element) | Remplace l'élément situé à la position spécifiée dans cette liste par l'élément spécifié |
| E remove(int index) | Supprime l'élément situé à la position spécifiée dans cette liste. Décale tous les éléments suivants vers la gauche |
| boolean remove(E element) | Supprime la première occurrence de l'élément spécifié de cette liste, s'il est présent. Si cette liste ne contient pas l'élément, elle reste inchangée. |
| void clear() | Supprime tous les éléments de cette liste. La liste sera vide après le retour de cet appel. |
| boolean isEmpty() | Renvoie vraie si la liste est vide |
| int size() | Renvoie la taille de la liste |

Tableau 4 méthodes utiles pour manipuler les listes

## Avantages du Framework Java collections :

Le Framework Java Collections offre les avantages suivants :

* Réduit l'effort de programmation
* Augmente la vitesse et la qualité des programmes
* Permet l'interopérabilité entre des API non apparentées
* Favorise la réutilisation des codes logiciels

# Les relations entre les classes

Dans une application Java, les classes peuvent être liées de façon plus ou moins forte. Ces relations sont représentées graphiquement par des diagrammes de classes. Chaque type de relation a un signe graphique différent. La connaissance des différentes notations utilisées est primordiale pour une transposition correcte en code Java. Dans cette section, on va voir les différentes relations qu'il est possible de définir entre différentes classes, en commençant par les relations association, agrégation et composition. On discutera la relation d’héritage dans le chapitre suivant.

## Association

En Java, une association est une connexion ou une relation entre deux classes distinctes, établie par l'intermédiaire de leurs objets. La relation d'association indique comment les objets se connaissent et comment ils utilisent leurs fonctionnalités respectives. Elle peut être de type **un à un**, **un à plusieurs**, **plusieurs à un** et **plusieurs à plusieurs**.

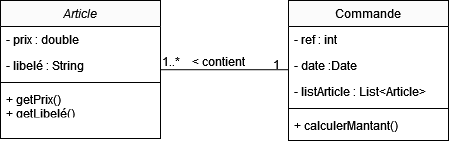


Figure 4: association entre classe Commande et Article

La Figure 4 illustre un exemple d’une association entre deux classe *Commande* et Article. La classe *Commande* contient une liste d’articles et ce n’est pas elle qui créé les *Articles*. Si la commande est détruite les articles ne le seront pas.

Une contrainte énumérative peut préciser les termes de l’association. Sur la Figure 4, on peut lire qu’une commande contient entre un et un nombre indéfini d’articles (1..\*). Dans l’autre sens, un article est attaché à une seule commande. On parle alors d’indice de cardinalité ou de multiplicité.

## Composition et Agrégation

La composition et l'agrégation sont les deux formes particulières d'association. Voyons-les à l'aide d'un exemple.

## Composition

Il s'agit d'une association de type "appartient à" ou "composer de”. Cela signifie simplement que l'un des objets est une structure logiquement plus grande, qui contient l'autre objet. En d'autres termes, il s'agit d'une partie ou d'un membre de l'objet le plus grand.

Par exemple, un bâtiment a une pièce ou, autrement dit, une pièce appartient à un bâtiment. La composition est un type de relation "composé de" forte car les cycles de vie des objets sont liés. Cela signifie que si nous détruisons l'objet propriétaire, ses membres seront également détruits avec lui. Par exemple, si le bâtiment est détruit, la pièce est également détruite dans notre exemple précédent. Cependant, cela ne signifie pas que l'objet contenant ne peut pas exister sans aucune de ses parties. Par exemple, si nous détruisons toutes les pièces à l'intérieur d'un bâtiment, le bâtiment existera toujours.

La composition est représentée par une flèche avec une losange (noire et pleine) reliant les deux classes. Le losange est du coté de la classe *conteneur (pére).*



Figure 5 Relation de composition

## Agrégation

L'agrégation est également une relation "composé de", mais ce qui la distingue de la composition, c'est que les cycles de vie des objets ne sont pas liés. Les deux objets peuvent survivre individuellement, ce qui signifie que la fin d'une entité n'affectera pas l'autre. Les deux entités peuvent exister indépendamment l'une de l'autre. C'est pourquoi on parle souvent d'association faible.



Figure 6 Relation d'agrégation

*Figure 6* montre un exemple d’une relation d’agrégation entre une classe *Joueur* et classe *Equipe*. Un joueur qui fait partie de l'équipe peut exister même si l'équipe cesse d'exister.