Chapitre 2

Les concepts de la programmation

L'objectif de ce chapitre est d'apprendre les concepts de la programmation dont aussi un peu de programmation "objet" car Java est un langage orienté objet.

L'approfondissement des concepts de la programmation objet se fera dans le cours NFA 032.

1. Introduction

Dans le monde de l'informatique il existe deux grands types de programmation[1] :

- la programmation procédurale, et
- la programmation orienté objet.

La **programmation procédurale** a précédé la **programmation objet**. Elle est encore très utilisée dans de nombreux langages comme par exemple le langage C, php, ...

Ces deux types de programmation ne sont pas exclusifs car beaucoup de concepts de la programmation procédurale sont inclus dans la programmation objet :

- les notions de variables locales à un traitement ou de variables globales au programme
- les notions de traitement principal, de sous-programmes et de bibliothèques de programme
- les notions de passage des paramètres dans les sous-programmes
- les notions de structures de contrôles (boucles, conditions)
- les types de données et leurs structures
- …etc…

Avec un langage objet, il est souvent possible de faire une programmation uniquement procédurale avec une utilisation exclusive de méthodes statiques : à opposer aux méthodes objets. On verra cela plus loin.

Dans la mesure où la programmation objet est plus puissante que la programmation procédurale, il est dommage de se restreindre à une programmation procédurale quand cela n'est pas nécessaire.

Sans rentrer dans le détail, l'avantage de la programmation objet est son **potentiel** intrinsèque (réutilisabilité, évolutivité, maintenance). Il existe des cas où la programmation objet n'est pas adaptée. Mais ces cas sont rares.

Le danger est de mélanger les deux types de programmation par méconnaissance du langage ou par "paresse" car la programmation objet est plus difficile et complexe mais aussi plus couteuse à court terme.

De plus, dans les langages objet, certains traitements et propriétés prédéfinis ne sont accessibles qu'à travers l'utilisation d'objet donc inévitable.

[1] il existe aussi les types de programmation impérative, fonctionnelle, logique, parallèle, …

2. Le programme principal

Que l'on soit dans une programmation procédurale ou objet, un programme commence toujours par l'écriture d'un **programme principal** qui représente le début de l'exécution du programme.

On dit que l'on exécute (ou lance) le programme principal.

Lors du lancement, on peut passer en paramètre du programme des **paramètres d'entrée**. Cela est optionnel. Ces paramètres d'entrée permettent de lancer le programme dans des situations différentes.

Un programme gère toujours des données. Les origines de ces données sont multiples :

- les paramètres du programme
- les valeurs saisies par un opérateur humain soit au clavier dans la fenêtre de commande de lancement, soit à travers une interface graphique (IHM)
- les données issues de fichiers ou d'une base de données
- les données provenant d'une communication réseau.

Exemple 1 en JAVA : un programme Java qui prend en entrée deux entiers et affiche dans la *fenêtre de commande de lancement* le résultat de l'addition des deux entiers.

```
public class Exemple
{
   public static void main(String[] args)
   {
      int valeur1;
      int valeur2;
      int resultat;

      valeur1 = Integer.parseInt(args[0]);
      valeur2 = Integer.parseInt(args[1]);

      resultat = valeur1 + valeur2;

      System.out.println("Le resultat est " + resultat);
   }
 }
```

Exemple 2 en JAVA : Le même exemple en utilisant un sous-programme (méthode statique) :

```
public class Exemple
{
   public static void main(String[] args)
   {
      int valeur1;
      int valeur2;
      int resultat;

      valeur1 = Integer.parseInt(args[0]);
      valeur2 = Integer.parseInt(args[1]);

      resultat = additionner(valeur1,valeur2);

      System.out.println("Le resultat est " + resultat);

       int x = additionner(100,300);
       System.out.println("Valeur de x : " + x);
   }

    static int additionner(int val1, int val2)
    {
        int resultat;
        resultat = val1 + val2;
        return resultat;
    }

}
```

On voit ici l'avantage d'utiliser un sous-programme qui permet un paramétrage complet du traitement et donc sa ré-utilisation n'importe où dans le programme.

Ne pas écrire 2 fois le même code à quelques variantes près et essentiel dans l'action de programmer. On parle de factorisation du code.

Ainsi l'exécution d'un programme est une arborescence d'appels successifs de sous-programmes. Les sous-programmes du programme principal pouvant à leurs tours appeler d'autres sous-programmes.

3. Le sous-programme

Le sous-programme est un traitement que l'on isole dans une "structure de bloc d'instruction" que cela soit dans une programmation procédurale ou orientée objet.

Dans les deux cas, les principes restent les mêmes :

- un sous-programme a un **nom** (obligatoire) qui permet de l'appeler dans un programme ou un autre sous-programme
- quand à un endroit du programme, le langage réalise l'appel d'un sous-programme, une fois le traitement du sous-programme réalisé, on revient juste après l'appel. On dit que les appels s'empilent dans une **pile d'exécution** (schéma en cours)
- un sous-programme peut prendre en entrée des valeurs : **paramètres d'entrée**
- un sous-programme peut contenir des **variables locales** utilisées par le traitement
- un sous-programme contient une séquence d'instruction (ou **code**) qui correspond au traitement réalisé
- un sous-programme peut avoir en sortie des résultats: **paramètres de sortie**
- un sous-programme peut aussi utiliser des **variables globales**

Une variable est dite globale quand cette même variable est accessible par tous les sous-programmes du programme ou par tout un *groupe* de sous-programme.
Cette notion de *groupe* de sous-programme a des formes différentes en fonction des langages : module, package, classe.

Le problème des variables globales est de maitriser leurs changements de valeur pendant tout le temps d'exécution du programme car on peut vite avoir des **effets de bord**.
Plus il existe de sous-programmes permettant de modifier et lire une variable globale, plus le risque est grand.

Schéma explicatif en cours.

Nous verrons qu'en programmation orientée objet, tous les **attributs** d'une classe sont des variables globales pour toutes les méthodes de la classe.
Il faut maitriser ce risque.
Pour le maitriser, il est indispensable de faire une "bonne" programmation objet.
C'est pourquoi il faut "penser objet". Inverser le paradigme de la programmation procédurale. Comme nous le verrons plus loin et surtout en NFA 032.

Dans presque tous les langages, et c'est le cas pour le langage Java, il existe deux types de sous-programme (que l'on appelle des **méthodes**) :

- les PROCEDURES, et
- les FONCTIONS

Dans le cas de la **Procédure**, les paramètres sont de deux types :

- les paramètres d'entrée
- les paramètres de sortie.

Dans le cas de la Fonction, il n'existe que des paramètres d'entrée et un seul paramètre de sortie qui passe par une interface de sortie dédiée appelée le **retour de la fonction**.

Ainsi, l'appel d'une Fonction se fait toujours dans une **expression fonctionnelle**.

Voir l'exemple précédent de l'addition de deux entiers.

Exemple :

```
int resultat = 10 + moyenne(30,40)*10 + carre(4);
System.out.println(resultat); // 376
```

avec

```
static int moyenne(int x,int y)
{
   return (x+y)/2;
}

static int carre(int x)
{
   return x*x;
}
```

Exemple 3 en JAVA : le même exemple que l'exemple 2 précédemment mais avec une procédure en Java :

```
public class Exemple
{
   public static void main(String[] args)
   {
      int valeur1;
      int valeur2;
      Resultat resultat;

      valeur1 = Integer.parseInt(args[0]);
      valeur2 = Integer.parseInt(args[1]);
      resultat = new Resultat();

      additionner(valeur1,valeur2, resultat);

      System.out.println("Le resultat est " + resultat.valeur);

       additionner(100,300, resultat);
       System.out.println("Valeur de resultat : " + resultat.valeur);
   }

    static void additionner(int val1, int val2, Resultat res)
    {
        res.valeur = val1 + val2;
    }

}

class Resultat
{
   int valeur;
}
```

4. Le passage des paramètres dans un sous-programme

Il existe deux modes de passage de paramètre en informatique :

- le passage **par valeur**, et
- le passage **par référence (ou passage par variable).**

Certains langages permettent les deux modes, d'autre qu'un des deux. Par exemple, Java ne fait que du passage par valeur, le C et C++ font les deux.

Le principe :
Lors de l'appel à une procédure, on passe en paramètre une variable.

Dans le cas où la variable est passée en paramètre par valeur, cela signifie que même si le paramètre est modifié dans la procédure, en retour de l'appel, la variable ne sera jamais modifiée.

Dans le cas où la variable est passée en paramètre par référence, cela signifie que si le paramètre est modifié dans la procédure, en retour de l'appel, la variable sera toujours modifiée.

Exemples :
Le langage Java ne fait que du passage par valeur. Voir l'exemple 3 précédent.

Exemple 4 en C++ : le même que l'exemple 3 avec un passage par référence :

```
  int main(int argc, char* argv[])
  {
     int valeur1;
     int valeur2;
     int resultat;

     valeur1 = atoi(argv[1]);
     valeur2 = atoi(argv[2]);

     additionner(valeur1,valeur2,resultat);
     cout << "Le resultat est " << resultat;

      additionner(100,300,resultat);
      cout << "Le resultat est " << resultat;
  }

  static void additionner(int val1, int val2, int &x)
  {
     x = val1 + val2;
  }
```

Par exemple en **Pascal** :

```
  Procedure additionner(val1 : integer, val2 : integer, var resultat : integer)
  {
     resultat := val1 + val2;
  }
```

5. Les Bibliothèques

A travers l'acte de programmation, on s'aperçoit qu'il existe de 3 familles de sous-programmes :

- les sous-programmes que l'on écrit soi-même
- les sous-programmes qui ont été écrits par quelqu'un d'autre et qui se trouve dans une **bibliothèque** de sous-programme
- les sous-programmes prédéfinis du langage qui sont, soit nativement dans le "noyau" du langage ou qui sont dans des bibliothèques prédéfinies, livrées avec le langage.

Un langage est toujours la réunion de ces deux composants : le noyau et les bibliothèques du langage (dont on n'a généralement pas les sources).

Souvent la puissance et la richesse d'un langage sont liées à la taille de ses bibliothèques (ce qui est le cas de Java).
Dans ces bibliothèques, on trouve, par exemple :

- les traitements d'entrée/sortie (lecture/écriture des fichiers, écran, clavier)
- les bibliothèques permettant de créer et gérer des interfaces graphiques
- les bibliothèques de calcul mathématique
- les bibliothèques de gestion des collections simples et complexes
- …

Exemple d'une bibliothèque en Java : la classe Terminal qui sera décrite et utilisée plus loin dans le cours.

6. Les variables d'un programme informatique

Dans tout programme informatique (procédural ou objet), une **variable** est une zone de la mémoire du programme, plus ou moins grande.

La taille de cette zone dépend du type de la variable.
Une variable est toujours typée : entier, double chaine, tableau, file, classe, …

Ainsi :

- une variable a toujours un nom. On appelle cela **l'identificateur** de la variable
- une variable a un type. On appelle cela le **type** de la variable. (ou type de donnée)
- une variable a toujours une **valeur**. La nature de cette valeur dépend directement de son type.

On peut faire deux choses sur une variable :

- "écrire" une valeur dans la variable. On parle **d'affectation** de la variable
- "lire" la valeur d'une variable. On parle **d'accès** à la variable.

Avant d'être utilisée, une variable doit être déclarée. On parle de **déclaration** de la variable.

Chaque langage a sa syntaxe de déclaration, et d'affectation.

La déclaration d'une variable

Avant d'être utilisée, une variable doit être déclarée. La déclaration d'une variable consiste à préciser deux éléments :

- son nom
- son type

Il existe de nombreux langages qui n'obligent pas la déclaration des variables avant leurs utilisations (exemple : javascript, php, …). Ce qui n'est pas le cas du langage Java, C, C++, ADA, Pascal, ...

L'affectation d'une variable

L'affectation d'une variable consiste à donner une valeur précise à cette variable.
Cette valeur est soit une constante, le résultat d'une expression arithmétique, le résultat d'un appel à une fonction.

L'opérateur le plus souvent utilisé est le "égal".

Ainsi, l'affectation est une expression basée sur un opérateur (le égale) qui possède deux opérandes : la variable (à gauche) et l'expression (à droite).

Cette expression peut être une expression très simple sous la forme d'une constante, une expression arithmétique plus ou moins complexe, ou l'appel à une fonction ou plusieurs fonctions dans une expression.

La règle fondamentale est que **les deux opérandes soient de même type**.

Dans la plupart des langages il existe certaines compatibilités entre les types qui se fait grâce à une **conversion implicite** des éléments utilisées dans les expressions.

Exemples d'expression mathématique, booléenne, relationnel, fonctionnel.

L'accès à une variable

La syntaxe d'accès à une variable est assez commune : il suffit d'utiliser le nom de la variable dans le code du programme.

Exemples.

7. Les blocs d'instruction

Un bloc d'instruction est composé de :

- déclarations de variables qui sont locales au bloc d'instruction (généralement en début du bloc)
- une suite **d'instruction** du langage qui s'exécute toujours dans l'ordre (de haut en bas)

Un bloc d'instruction est soit celui du programme principal, ou soit celui d'un sous-programme (procédure, fonction, méthodes).

Les instructions qu'il est possible de faire sont très, très nombreuses et dépendent complètement du langage utilisé.

En restant général et commun à tous les langages procéduraux et objet, on peut distinguer, les familles d'instructions suivantes :

- les instructions basées sur l'utilisation d'opérateur et d'opérandes. Leurs objectifs est de faire du "calcul"
- les instructions qui consistent à faire des appels à des procédures ou fonctions
- les instructions qui permettent de créer des données plus ou moins complexes (structure, tableau, objet)
- les instructions qui contrôlent le flot d'exécution : les **structures de contrôle**

Comme nous le verrons en détail plus loin dans les algorithmes et la syntaxe du langage Java, les structures de contrôle sont fondamentales.

Elles permettent notamment de faire des "**boucles**" et des "**conditions**".

8. La programmation objet

La Programmation Orientée Objet (P.O.O.) est une évolution majeure des techniques de programmation.

La POO est une programmation qui privilégie les données aux traitements.

Cette nouvelle approche se justifie par un premier constat : **les données sont plus stables que les traitements**. Dès lors, la vision de l'architecture du programme est plus stable dans le temps même si de nouveaux traitements sont créés au sein de la donnée.

L'objet représente un concept, une entité informatique, un élément de programmation qui possède une structure interne et des comportements.

9. L'objet (définition)

Un objet est une structure de données valuées qui répond à un ensemble de *messages*. Cette structure de données définit son ***état*** tandis que l'ensemble des messages qu'il comprend décrit son ***comportement***.

La portée de vie de l'objet est celui du programme informatique. Il reste un élément du langage de programmation et non une sorte d'entité plus ou moins autonome qui persisterait au sein de l'ordinateur.

Certains attributs ou tous sont cachés : l'encapsulation. On cache la structuration interne de l'objet pour le monde extérieur ce qui permet son évolution (**interface**) (souvent pour des raisons de performance) sans impact sur les programmes qui utilisent cet objet.

Un objet est typé. Le type définit la structure de l'objet **et** la syntaxe et la sémantique des messages auxquels peut répondre un objet. Ce type est identifié et décrit par la définition d'une **classe**.

Un objet peut appartenir à plusieurs types : le **polymorphisme**.
Cela permet d'utiliser des objets de types différents là où est attendu un objet d'un certain type.

Un objet peut appartenir à un type et à certains de ses sous-types : l' **héritage**.

Un objet appartient à une classe.

10. La classe

La classe est le cadre (ou Frame) de définition de l'objet. On y trouve :
- les **attributs** (les données) ➔ partie statique de l'objet
- les **méthodes** (le code) (traitement, message, fonction, ...)➔ partie dynamique de l'objet.

Les attributs

Les attributs décrivent la structuration de donnée de l'objet. Ils contiennent les données de l'objet et donc du programme informatique.
Le principe de l'encapsulation de ces données est de maitriser, contrôler, rendre transparent ou non l'accès de ces données; que ce soit en "lecture" ou en "écriture", par rapport à celui qui utilise l'objet.

Pour cela il existe des règles de visibilité de ces attributs. On parle d'attribut privé ou d'attribut public.
Un attribut privé est un attribut qui ne peut pas être utilisé à l'extérieur de l'objet.
Un attribut public est un attribut qui peut être utilisé à l'extérieur de l'objet. On dit aussi que l'attribut est visible.

Comme un attribut est une donnée informatique, il est typé (entier, chaine de caractère, double, tableau de valeur, …. et **classe** aussi)

Les méthodes

Les méthodes sont les éléments du langage qui contiennent le code du programme informatique. La démarche d'une bonne programmation objet réside sur la compréhension suivante : dans un L.O.O. la méthode est un traitement qui s'applique sur l'objet c'est-à-dire que le calcul réalisé **change** les valeurs des attributs de l'objet.
En programmation objet, il est donc courant qu'une méthode ne retourne pas de valeur.
De même, il est courant qu'une méthode n'a pas de paramètres d'entrée car les données du calcul sont des attributs de l'objet.

Cette nouvelle façon de programmer induit une phase de conception qui s'oppose à la conception fonctionnelle : la conception objet.

Ainsi dans la programmation objet, appeler un traitement nécessite de connaître deux éléments :
- le nom de la méthode
- l'objet auquel la méthode appartient.

```
objet.methode(…);
```

La conception objet

La première étape dans la conception objet est d'identifier en premier les données manipules par le programme et donc les objets. Ensuite vient l'identification des traitements.

Cette étape de conception est très importante car la plupart du temps l'expression du besoin ou le cahier des charges (ou spécifications) sont exprimés par le client (ou donneur d'ordre) sous une forme fonctionnelle. Un système informatique doit réaliser certains traitements qui sont autant de fonctions (dites de haut niveau) et il faut donc passer d'une description fonctionnelle a une description objet.

Un des premiers aspects de la conception objet va donc être d'identifier les objets et leurs compositions.
Car de même que la décomposition d'un problème (traitement) en sous-problèmes est la bonne démarche dans une programmation fonctionnelle, la décomposition des objets en sous-objets est la bonne démarche dans une programmation objet.

11. L'objet (structuration)

La structuration des données a toujours été dans le monde de l'informatique un concept fort et important. Très rapidement, la plupart des langages informatique savaient manipuler des structures de données (record, struct, …).

Il fallait nommer et structurer les données du programme et donc définir les règles d'accès à ces données.

La programmation objet n'échappe pas à ce principe sachant que c'est la classe qui est une structure de données via ses attributs.

La règle d'accès à ses données est donc :

```
objet.attribut                  de l'extérieur
this.attribut ou attribut       de l'intérieur
```

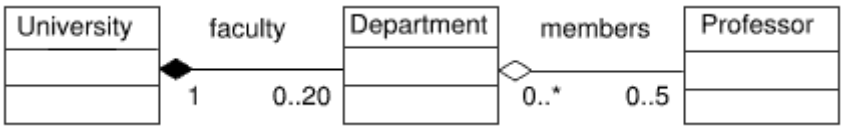
Lorsqu'un attribut n'est pas un type élémentaire (type primitif) (int, char, ..) mais une classe alors on parle de relation avec un autre objet.

Il existe deux formes de relation entre deux objets :
- la composition
- l'agrégation

La composition est utilisé pour décrire qu'un objet se décompose en sous-objet. La vie du sous-objet dépende de la vie de l'objet auquel il appartient. Quand on crée un objet alors ses sous-objets sont également créés. De même pour la destruction.

L'agrégation est utilisée pour préciser une relation forte entre deux objets mais l'un appartient pas à l'autre. Quand on crée (ou détruit) le premier, le deuxième n'est pas créé (ou détruit).

Exemple :
Une université est composée de différents départements.
Un département a des professeurs.
Les départements est une décomposition de l'université. Si l'université est détruite les départements aussi. Mais ce n'est pas parce qu'on ferme un département que les professeurs sont détruits. De plus un professeur peut appartenir à plusieurs départements.



Dans cet exemple, l'université est composée au plus de 20 départements.
Chaque département est lié à une seule université.
Chaque département peut contenir un nombre illimité de professeur. Sachant qu'un professeur appartient au plus à 5 départements.

Le constructeur est une **méthode** de l'objet particulière et propre au langage que vous utilisez. Son rôle est de créer un objet à partir d'une classe donnée. On parle de processus d'instanciation. Un objet est une **instance** de classe.

Le rôle du constructeur est :
- d'initialiser les attributs avec des valeurs par défaut
- d'initialiser les attributs avec de nouvelle création d'objet
- d'initialiser les attributs en fonction des paramètres de la méthode de construction.

Il n'est pas rare de voir des constructeurs complexes, dans lequel il existe de nombreuses lignes de code.

Pour créer un objet :

```
Class obj = new Class(…);
```

Exemple :
Employe e1 = new Employe("LAFONT","Paul",40);

Class est le type d'appartenance de l'objet.
new est une commande du langage objet qui alloue l'objet en mémoire et construit l'objet

12. Exemple

Nous voulons gérer une médiathèque dont les éléments de prêt peuvent être des livres, des dvd, des CD de musique…

```
class Livre                                          //1
{
   // Les attributs

   private static final int MAX_AUTEURS = 5;              //2
   private static int ident_courant = 0;                  //3


   String titre;                                     //4
   String auteurs[];
   int tome;
   Calendar date_creation;
   int ident;

   // Les constructeurs                              //5

   public Livre()                                    //6
   {
      titre = new String("");
      auteurs = new String[MAX_AUTEURS];
      tome = 0;
      date_creation = Calendar.today();              //11
      ident_courant++;
      ident = ident_courant;                         //12
   }

   public Livre(String titre, String auteur)         //7
   {
      this.titre = new String(titre);
      auteurs = new String[MAX_AUTEURS];
      auteurs[0] = new String(auteur);
      tome = 0;
      date_creation = Calendar.today();
      ident_courant++;
      ident = ident_courant;
   }




   public Livre(Livre l)                             //8
   {
      titre = l.titre;
      auteurs = l.auteurs;

      this.tome = l.tome;
      date_creation = Calendar.today();
      ident_courant++;
      ident = ident_courant;
   }

   // Méthode qui convertit le livre en chaîne (pour affichage)
   public String toString()                          //9
   {
```

```
        String str="";

        str = str + this.titre+"\n";
        for(String a : auteurs) str = str + a;
        str = str + "tome "+tome;

        return str;
    }

    // Pour saisir les valeurs d'un livre
    public void saisir()                              //10
    {
       titre = … valeur saisie par l'utilisateur
       …….
    }
}
```

La syntaxe utilisé dans cet exemple est celle de Java qui très proche de celle du C++.

Commentaires :

1/ la classe est structurée en deux parties : les attributs et les méthodes (constructeurs + traitements). Il est une convention de mettre les attributs avant les méthodes.

2/ il existe des attributs qui sont des constantes (static final).

3/ cet attribut (le 2/ aussi) est static : attribut commun à tous les objets de la classe

4/ il y a des attributs de type élémentaire (int) et des attributs d'objets (String, Calendar)

5/ plusieurs constructeurs permettent de construire un objet différemment

6/ ce constructeur crée un livre "vide" (il doit être ensuite correctement initialisé, via une IHM par exemple)

7/ 2ème constructeur qui crée un livre à partir d'un titre et d'un auteur (constructeur le plus courant)

8/ 3ème constructeur qui crée un nouvel exemplaire d'un livre

9/ cette méthode retourne sous la forme d'une chaine de caractère la description d'un livre. Ceci est souvent utilisé pour débuguer ou journaliser une application

10/ cette méthode a pour rôle d'initialiser certains champs d'un livre (créé au préalable) en demandant à l'utilisateur de saisir ces champs à l'écran (Ihm, console, ..)

11/ cet attribut est la date de création du livre qui est initialisée par le system via l'utilisation d'une autre classe, Calendar

12/ cet attribut permet de donner un numéro unique à chaque livre car un livre peut être en plusieurs exemplaires. L'identification d'un livre est la date de création + ident; On peut penser que ce numéro est remis à 0 en début de chaque année.

Par cet exemple on comprend l'importance de connaître le rôle de chaque attribut et d'en déduire certains traitements

13. L'héritage

Le but ici n'est pas de rentrée dans le détail mais de présenter le principe générale de l'héritage qui est le concept fort des langages objets. Un chapitre entier lui sera consacré par la suite.

On parle ici d'héritage de classe.

Quand on dit que la classe B hérite de la classe A, cela signifie que dans toutes les méthodes de B il est possible d'utiliser les attributs de A et les méthodes de A comme si B était un A.

Prenons un exemple :

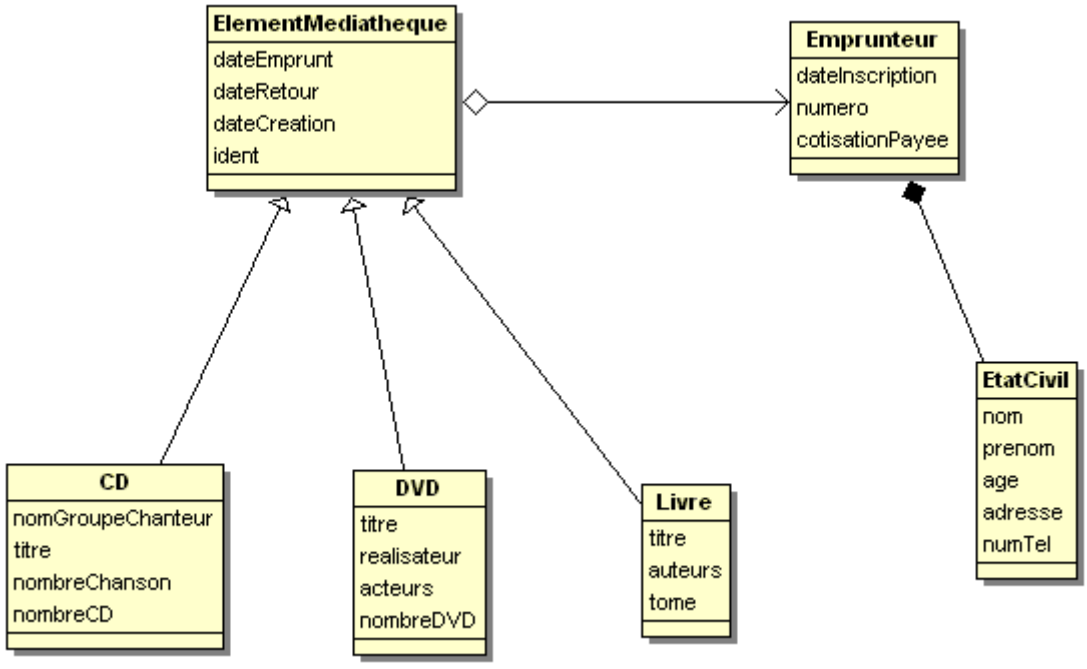
La classe (**ElementMediatheque**) des éléments pouvant être empruntés dans une médiathèque indépendamment des caractéristiques propres est caractérisé par :

- l'emprunteur
- la date d'emprunt
- la date de retour
- l'identification de l'élément emprunté
- le type d'élément emprunté

Ainsi les classes **Livre**, **CD**, **DVD** vont hériter de la classe **ElementMediatheque**.

La classe **ElementMediatheque** contient les attributs communs aux livres, CD et DVD.

Ces classes héritent aussi des méthodes et aussi des constructeurs; Sauf que pour les constructeurs l'appel se fait automatiquement par le langage : le constructeur de Livre va appeler automatiquement le constructeur de ElementMediatheque.




```
public class ElementMediatheque
{
    Emprunteur emprunteur;
    Calendar   dateEmprunt;
    Calendar   dateRetour;
    Calendar   dateCreation;
    int        ident;
}
```

```
public class Livre extends ElementMediatheque
{
    String titre;
    String auteurs[];
    int tome;
}

public class CD extends ElementMediatheque
{
    String nomGroupeChanteur;
    String titre;
    int nombreChanson;
    int nombreCD;  // Dans le cas d'un coffret ou album
}

public class DVD extends ElementMediatheque
{
    String titre;
    String realisateur;
    String acteurs[];
    int nombreDVD;
}

public class Emprunteur
{
    EtatCivil emprunteur;
    Calendar  dateInscription;
    int       numero;
    boolean   cotisationPayee;
}

public class EtatCivil
{
    String   nom;
    String   prenom;
    String   age;
    String   adresse;
    String   numTel;
}
```

La relation entre ElementMediatheque et Emprunteur est une relation d'agrégation car justement l'objet emprunteur vit indépendamment des éléments qu'il a pu ou non emprunter. Quand un livre est détruit on ne détruit pas l'emprunteur.

La relation entre Emprunteur et Individu est une relation de composition car Individu est une décomposition en un sous-objet de Emprunteur. Quand on détruit l'emprunteur on détruit son état civil

L'héritage est le moyen de factoriser des attributs qui sont communs à différentes classes. Les méthodes associées à ces attributs sont également elles aussi factorisées.

Nous verrons plus loin que ces méthodes peuvent être surchargées par les classes en-dessous (on parle de spécialisation).

14. Un objet bien formé

Un objet bien formé est une notion essentielle car elle correspond à des règles de qualité qui, si elles sont respectées, permet de faire une "bonne" programmation objet.

Une première règle, largement partagée, est d'imposer que tous les attributs soient privés. Il est ainsi donner à la discrétion du programmeur de rendre ces attributs accessibles via des méthodes permettant de lire ou d'écrire ses attributs.

On appelle, respectivement ces méthodes : des getteurs et des setteurs.

Exemple :

```
public class Livre extends ElementMediatheque
{
    private String titre;
    private String auteurs[];
    private int tome;

    public String getTitre()
    {
       return titre;
    }

    public String[] getAuteurs()
    {
       return auteurs;
    }

    public int getTome()
    {
        return(tome);
    }

    public void setTome(int tome)
    {
       this.tome = tome;
    }
}
```

Il existe de nombreuses autres règles.
On peut citer les suivantes :
- créer une donnée dans un programme objet, consiste à créer un objet
- éviter autant que possible de modifier un objet passé en paramètre d'une méthode
- ne pas faire une méthode de plus de 20 lignes de code
- ne pas faire d'héritage multiple
- désallouer les objets dans l'ordre inverse de leurs allocations (pas JAVA)
- créer des classes privées pour les données propres à une classe
- donner des noms explicites aux méthodes et aux attributs publics