I. Introduction (rappels)

- Le paradigme de Programmation Orientée Objet, que nous avons abordé dans le chapitre 1, consiste à rendre possible dans le langage de programmation la définition d'objets qui ressemblent à ceux du monde réel.
- On en déduit que toute **entité identifiable**, concrète ou abstraite, peut être considérée comme un **objet**
- Un **objet** réagit à certains **messages** qu'on lui envoie de l'extérieur. La façon avec laquelle il réagit détermine le **comportement** de l'objet. Il ne réagit pas toujours de la même façon à un même événement, sa réaction dépend de l'**état** dans lequel il se trouve.

- Ainsi, pour faire de la **Programmation Orienté Objet**, il faut définir des modèles d'objets, et créer des objets à partir de ces modèles. Il faut donc savoir **concevoir des classes.** Les classes permettront alors de définir de nouveaux types. Chaque classe définit la façon de créer et de manipuler des objets de ce **type**.

II. Classes en Java

- Une **classe** est un **prototype** ou un patron d'objets qui définit les variables et les méthodes communes à tous les **objets d'un même genre.**
- Un **objet** est une **instance de la classe**
- **Un objet**
  - maintient son état dans des champs (appelées variables d'instance)
  - implémente son comportement à l'aide de méthodes

  Objet informatique = regroupement logiciel de variables et de méthodes

II.1 Définition d'une classe : syntaxe

class **Nom*Classe*** {

```
// Instructions permettant de définir la classe
```

}

- **NomDeLaClasse** représente le type d'objets désignés par la classe.
- Avec Java il ne faut pas ajouter de point-virgule à la fin du bloc de définition de la classe, contrairement au langage C++.
- Par convention le nom d'une classe commence par une majuscule
- On peut placer certaines **classes** dans un **paquetage** en mettant, au début du fichier contenant cette classe :

package nomDuPaquetage;

- Une classe publique est déclarée par les mots-clés **public class**. Elle doit être enregistrée dans un fichier qui porte le même nom qu'elle (voir chapitre 2).
- On ne peut pas avoir plus d'une classe publique dans un fichier donné. En revanche, on peut ajouter d'autres classes, non publiques: **classes internes**

Exemple de classe interne

```
public  class Enseignant {   // doit être écrite dans le fichier Enseignant.java

   // code de la classe Enseignant
}

 class ResponsableCours {   // écrite dans le même fichier

     // code de la classe ResponsableCours
}

// NB : La classe  ResponsableCours est une classe interne. On peut avoir autant
   de classes internes qu'on le souhaite dans un fichier.
```

II.2 Membres d'une classe

- Les **éléments** déclarés à l'intérieur d'une classe Java sont appelés des **membres**. Ils peuvent être :
  - des champs ou attributs (variables d'instances et/ou variables statiques)
  - des méthodes (méthodes d'instances et/ou méthodes statiques)
  - des classes membre (statiques et/ou non statiques)

- Les membres d'une classe peuvent se voir doter d'un niveau de protection (visibilité) par utilisation des mots clés (**public, private et protected**).
  - Avec **public** l'accès est possible depuis toute classe du même paquetage ou autre
  - Avec **protected** l'accès est possible depuis toute classe du même paquetage et toute classe fille
  - Avec **private** l'accès n'est possible que dans la classe elle-même
- Lorsqu'aucune protection n'est mise, l'accès est possible depuis toute classe du même paquetage

Remarques :

- On appelle **membre statique** d'une classe tout élément (champ, méthode, classe) attaché à cette classe plutôt qu'à l'une de ses instances. Un membre statique peut exister, être référencé, ou s'exécuter même si aucune instance de cette classe n'existe.
- Lorsque que l'on déclare un membre statique avec le mot clé **static**, il est placé dans un espace mémoire commun à tous les objets de la classe.

Exemples de membre statiques

1. La valeur d'un **champ statique** est la même pour tous les objets instance de la classe. Si un des objets modifie la valeur d'un champ statique par exemple, tous les objets verront la valeur de ce champ modifiée.
2. De même que pour un champ, une méthode peut être déclarée static. On parle alors de méthode de classe. A titre d'exemple, la méthode main, appelée lors du lancement d'une application Java est une méthode statique.

- Une méthode statique est une méthode qui n'a accès qu'aux membres static de la classe.
- Les membres **non statiques** (ou d'**instance**) ne peuvent exister sans **un objet.**

II.3 Déclaration des membres d'une classe

II.3.1 Les champs

Les champs sont les membres qui permettent de stocker des valeurs de types de base ou des objets (variables d'instance). Un champ est déclaré suivant la syntaxe suivante :

[visibilité] [static] typeDeChamp nomDuChamp [= valeur-initiale] ;

Remarques :

- La visibilité du champ peut être omise (ce qui n'est pas conseillé),
- Le mot-clé **static** indique que le champ est statique. S'il est omis, le champ est non statique.
- Le type du champ peut être un type de base, ou tout nom de classe.
- La valeur-initiale est la valeur affectée dans la variable d'instance lors de la création de l'objet. Si la valeur-initiale n'est pas précisée, le système initialise la variable par défaut (voir chapitre 2 pour les valeurs par défaut de tous les types).

Exemple

```
public class VoitureP {
public String marque = "Peugeot";
private double vitesse;
protected float prix;
String immatriculation;
}
```

Exemple :

```
classe Fraction(){
private  int num ;
private  int  denom ;
public void Afficher(){
system.out.println(num +  "/" +denom) ;
}
}
```

II.3.2 Les méthodes

- Les méthodes sont des fonctions faisant partie d'une classe. Elles permettent d'effectuer des traitements sur (ou avec) les données membres des objets., de faire dialoguer les objets entre eux, d'accéder à des ressources telles que des fichiers, des bases de données, des terminaux graphiques, etc…

a) Signature d'une méthode

Une méthode est caractérisée par sa ***signature*.** La signature d'une méthode est composée :

- de son nom ;
- de la liste ordonnée des paramètres (arguments) qu'elle accepte en entrée.

Remarques :

- Un argument peut être :
  - une constante
  - une variable
  - une expression
  - une autre méthode retournant une valeur
- Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes
- le nom de la méthode suit les mêmes règles que les noms de variables :
  - le nom doit être écrit en minuscule, sauf pour les initiales de chaque nouveau mot
  - un nom de méthode peut comporter des chiffres, mais pas pour le premier caractère
  - les caractères spéciaux _ et $ peuvent être utilisés mais ne devraient pas l'être
  - le nom de la méthode est sensible à la casse (différence entre les minuscules et majuscules)
- Une classe ne peut pas avoir deux méthodes qui possèdent une même signature.
- Plusieurs méthodes peuvent porter le même nom à condition qu'elles se distinguent par leurs paramètres (surcharge).

b) La déclaration d'une méthode

- Avant d'être utilisée, une méthode doit être définie (pour l'appeler dans une classe il faut que le compilateur la connaisse, c'est-à-dire qu'il connaisse son nom, ses arguments et les instructions qu'elle contienne).
- Une méthode est déclarée de la façon suivante.

[visibilité] [static] [TypeDeRetour] [signature] {

// Déclaration des variables locales

// liste d'instructions

}

NB : [signature] = [nomDeLaMethode(Type1 argument1, Type2 argument2,….)]

- Les règles de visibilité d'une méthode sont les mêmes que pour les champs. Une méthode peut donc être private, protected, ou public.
- Si l'on omet le modificateur de visibilité pour une méthode (ce qui n'est pas conseillé), alors elle est package protected .
- Une méthode statique ne peut accéder elle-même aux champs et méthodes non statiques de cette classe. Elle peut bien sûr recevoir en argument des champs non statiques et les traiter.
- TypeDeRetour représente le type de valeur que la méthode va retourner, cela peut-être un type primitif, une classe, ou alors le mot-clé *void* si la méthode ne retourne aucune valeur
- Une méthode pour laquelle on déclare un type de retour doit obligatoirement retourner une valeur. Le compilateur génèrera une erreur s'il se rend compte qu'il existe un chemin de sortie pour cette méthode, qui ne comporte pas de return.
- La syntaxe de l'instruction return est simple :

return valeurDeRetour;

- Le type de valeur retourné doit correspondre à celui qui a été précisé dans la définition
- Lorsque l'instruction return est rencontrée, la méthode évalue la valeur qui la suit, puis la renvoie au programme appelant (classe à partir de laquelle la méthode a été appelée).
- Une méthode peut contenir plusieurs instructions return, ce sera toutefois la première instruction return rencontrée qui provoquera la fin de l'exécution de la méthode et le renvoi de la valeur qui la suit.

Exemple

Erreur de compilation due à une valeur de retour non toujours définie

```
public boolean testParite(int i) {  // erreur de compilation : la méthode ne retourne pas toujours une valeur
        if (i % 2 == 0)
            return true ;
    }
```

Exemple

À l'inverse, la méthode suivante compile correctement.

Types de retour correctement définis

```
public boolean testParite1(int i) {
    if (i % 2 == 0)
        return true ;
    else
        return false ;
}
```

- Les méthodes ne peuvent pas être imbriquées. Elles sont déclarées les unes après les autres.

c) Appel de méthode

Pour exécuter une méthode, il suffit de faire appel à elle en écrivant l'objet auquel elle s'applique (celui qui contient les données), le nom de la méthode (en respectant la casse), suivie de ses arguments entre parenthèse :

objet.nomDeLaMethode(argument1,argument2,…);

Remarques :

- Une méthode est exécutée par un objet lorsqu'il reçoit le message correspondant.
- Dans une méthode, l'objet qui l'exécute est accessible avec la référence **this**.
- Si vous exécutez une méthode sur l'objet courant (**this**), c'est à dire que vous utilisez dans une classe, une méthode de la même classe, alors il est inutile de préciser que **this** est l'objet auquel s'applique la méthode.
- Si jamais vous avez défini des arguments dans la déclaration de la méthode, il faudra veiller à les inclure lors de l'appel de la méthode (le même nombre d'arguments séparés par des virgules !), mais sans avoir à préciser leur type. Le nombre et le type d'arguments dans la déclaration et dans l'appel doit correspondre, sinon, une erreur est générée lors de la compilation...
- Les paramètres tels que nommés lors de la déclaration de la méthode sont appelés **paramètres formels**. Les paramètres utilisés lors de l'appel de la méthode sont appelés **paramètres effectifs.**
- Le mécanisme qui lie les paramètres effectifs aux paramètres formels en Java est **l'appel par valeur**. Cela signifie que les paramètres effectifs sont évalués avant l'exécution de la méthode et les paramètres formels prennent la valeur des paramètres effectifs au début de l'exécution de la méthode.

Exemple :

```
int v = 3 ;
g.setXY(12+1,v) ;
```

Remarque : Le paramètre formel x de setXY(int x, int y) prend la valeur 13 et y prendra la valeur 3. Notons bien que y est lié à la valeur de v et non à la variable v. Une modification du paramètre formel y pendant l'exécution de la méthode n'aura aucun impact sur v.

d) La surcharge de méthode

- Un des apports les plus intéressants de la conception objet, est la possibilité d'appeler plusieurs méthodes avec le même nom, à condition que leurs arguments diffèrent (en type et/ou en nombre).

- Ce principe est appelé **surcharge de méthode**. Il permet de donner le même nom à des méthodes comportant des paramètres différents et simplifie donc l'écriture de méthodes sémantiquement similaires sur des paramètres de types différents.

Exemples

```
int somme(int p1, int p2) {
 return p1+p2;
}
 int somme(int p1, int p2, int p3) {
 return p1+p2+p3;
}
 double somme(double p1,double p2){
 return p1+p2;
 }
```

Remarque : Avec la surcharge, Il est ainsi possible de définir une méthode réalisant la même opération sur des variables différentes en nombre ou en type.

II.3.3 Classe membre :

- Une **classe membre** est une classe java définie dans la partie déclaration des membres d'une autre classe qui la contient (nommée classe englobante : ex classe NomClasse).
- Une classe membre est instanciable.
- Une classe membre est associée à un objet instancié de la classe englobante.

II.4 Constructeurs d'une classe

II.4.1 Définition

- Un **constructeur** d'une classe est une **méthode particulière**. Elle n'a pas de type de retour, et porte le même nom que la classe dans laquelle elle se trouve.

Remarques

- Une classe peut avoir autant de constructeurs (surcharge) que l'on veut créer, dès l'instant qu'ils ont des signatures différentes, c'est-à-dire des paramètres différents.
- Il n'est toutefois pas conseillé de multiplier les constructeurs, ni de créer des constructeurs prenant des listes interminables de paramètres.
- Une classe qui ne déclare aucun constructeur explicitement en possède en fait toujours un : le constructeur par défaut, qui ne prend aucun paramètre.
- Si l'on définit un constructeur explicite, alors ce constructeur par défaut n'existe plus ; on doit le déclarer explicitement si l'on veut encore l'utiliser.

II.4.1 Rôle du constructeur

Le constructeur permet de déclarer et d'initialiser les données membres de la classe. Il permet également différentes actions (définies par le concepteur de la classe) lors de l'instanciation.

Remarque :

- la définition d'un constructeur n'est pas obligatoire lorsqu'il n'est pas nécessaire

Exemples : pour la classe Fraction on a les deux constructeurs 1 et 2

Constructeur1

```
public Fraction (int N, int D)
numerateur=N ;
If (denominateur==0)
{system.out.println (« le dénominateur doit être différent de zéro »)
System.exit (1) ;
}
else denominateur=D ;
}
```

Constructeur2

```
public Fraction (int N)
{
Numerateur=N ;
Denominateur=1 ;
}
```

- Il est possible pour un constructeur d'appeler un autre constructeur. Cela se fait avec le mot clé this. Attention : cet appel doit être la **première instruction**

II.5 getters et setters

Deux types de méthodes ont un statut particulier en Java : les ***getters*** et les ***setters***. Afin de respecter le principe d'**encapsulation**, les **champs non statiques** d'une méthode sont déclarés **private**. Afin de pouvoir lire ces champs, et éventuellement d'en modifier la valeur, on ajoute à la classe un getter et un setter pour chacun de ses champs private.

1. Un getter commence par get, suivi d'un nom, qui est en général le nom du champ, commençant par une majuscule (Souvent on utilise des noms de la forme getXXX). Il retourne une valeur en général du même type que le champ associé.
2. Le setter associé commence par set, suivi du même nom que le get. Il prend en paramètre un unique argument, du même type que le type de retour du getter associé. La présence d'un getter définit dans une classe une propriété. Si aucun setter associé n'est présent, alors cette propriété est en lecture seule.

Exemple de méthodes setters et getters

```
public  class Enseignant {

    private String nomEnseignant ;
    private String prenomEnseignant ;

    public Enseigant(String nom) {
       nomEnseignant = nom ;
    }

    public String getNom() {
       return nomEnseignant ;
    }

    public String getPrenom() {
       return nomEnseignant ;
    }

    public  void setPrenom(String prenom) {
       this. prenomEnseignant = prenom ;
    }
}
```

NB : Cette classe Enseignant comporte deux propriétés :

- nom, qui est en lecture seule. Le nom d'un Enseignant est fixé lors de sa construction, et ne peut plus être modifié ;
- prenom, que l'on peut lire et modifier.

On remarquera que le nom des champs n'est pas pris en compte lors de la détermination du nom d'une propriété.

Remarque:

- Le nom des deux méthodes (getters et setters) est fixé par une convention universellement suivie. Il convient donc de la respecter !

III. Instanciation d'objets

- Précédemment, nous avons vu qu'un type d'objet est décrit par une classe. Généralement la classe décrit :
  - les attributs : nom et type de valeur ;
  - les méthodes utilisées pour répondre aux messages.
- Le programmeur peut créer des objets à partir de la classe. C'est le processus d'**instanciation**. On dit que les **objets** sont des **instances de la classe** ou que les objets appartiennent à la classe.
- Java gère l'**instanciation** des objets en deux étapes. Tout d'abord, la machine virtuelle Java (JVM) **alloue un espace de stockage** à l'objet lorsqu'il exécute une instruction d'instanciation. Ensuite, elle **appelle la méthode du constructeur** correspondant définie dans la classe de l'objet.
- Java dispose de plusieurs[1] méthodes pour créer une instance d'un objet, la plus courante étant l'opérateur **new**. Il renvoie une nouvelle instance d'un objet qui s'initialise en fonction du constructeur avec les arguments correspondants, comme suivant:

```
NomClasse identificateurObjet;
identificateur = new NomClasse();
```

Est équivalent à:

```
NomClasse  identificateurObjet = new NomClasse();
```

Par exemple pour créer un objet de type Fraction, on écrit :

```
Fraction F ;
F= new Fraction(2,4) ; //On utilise le Constructeur1
F= new Fraction(4) ;   // On utilise le Constructeur 2
```

Remarques :

- Chaque objet crée doit avoir :
  - un **nom** (référent) « qui lui est propre » pour l'identifier. La **référence (handler)** permet d'accéder à l'objet, mais n'est pas l'objet lui-même. Elle contient l'adresse de l'emplacement mémoire dans lequel est stocké l'objet. C'est, en quelque sorte, un pointeur sur l'espace mémoire alloué pour stocker l'état de l'objet.
  - Un **état**, c'est à dire des valeurs particulières pour les variables d'instances de la classe auquel il appartient.
  - Des **méthodes** qui vont agir sur son état.
- Un objet crée possède :
  - une **interface** constituée des opérations qu'on peut lui demander de faire (accessibles) ;
- une **partie interne et protégée**, les données qu'il contient et les méthodes dont il se réserve l'usage(non accessibles).
- Le fonctionnement du programme résulte de l'interaction entre les objets «instanciés ».

[1] Vous pouvez également créer des instances d'objets à l'aide des classes intégrées de Java. Par exemple, vous pouvez utiliser la méthode newInstance() de la classe java.lang.Class

IV Destruction d'objets et Désallocation mémoire

- Un même objet peut être référencé par plusieurs variables. Les variables cessent de référencer un objet quand on leur affecte une autre valeur, ou null. Il faut qu'aucune variable ne référence plus un objet pour qu'il soit détruit. Les objets qui ne sont plus référencés peuvent être « récupérés » pour « recycler » l'espace mémoire qu'ils occupent.

- En Java, il n'y a pas de méthode qui permet de libérer la mémoire occupée par un objet non référencé. Par contre il existe un processus qui est lancé automatiquement (de façon régulière) de l'exécution d'un programme Java et récupère la mémoire non utilisé. Ce processus s'appelle le ramasse miettes (Garbage collector en anglais). L'utilisateur peut appeler le ramasse miette en appelant la méthode System.gc();.