CHAPITRE 2 :

# *LES CONCEPTS ORIENTES OBJETS EN JAVA*

**Objectifs spécifiques**

1. Introduire le concept de classe et sa notation (1/2 heure)
2. Maîtriser la définition des attributs et méthodes d'une classe (1 heure 30)
3. Maîtriser l'instanciation des objets et l'accès aux membres d'un objet (2 heures 30)
4. Comprendre le concept des classes internes (1 heure 30)
5. Maîtriser l'atout d'organisation « package » (1 heure 30)

**Eléments de contenu**

**I.** Notion de classe

**II.** Attributs et méthodes

**III.** Objets et Instanciation

**IV.** Membres statiques et mots clés final, this et null

**V.** Les classes internes

**VI.** Les packages

**Volume Horaire :**

**Cours :** 7 heures 30

**Travaux Pratiques :** 15 heures (5 TPs)

## 2.1 Notion de Classe

### 2.1.1 Concept

- Une classe est un support d'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes.
- Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.
- En Java : tout appartient à une classe sauf les variables de types primitifs (int, float…).
- Pour accéder à une classe il faut en déclarer une instance de cette classe (ou un objet).
- Une classe comporte sa déclaration, des variables et la définition de ses méthodes.
- Une classe se compose de deux parties : un en-tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes.

- Les méthodes et les données sont pourvues d'attributs de visibilité qui gère leur accessibilité par les composants hors de la classe.

### 2.1.2 Syntaxe

```
modificateur nom_classe [extends classe_mere]
[implements interface]
{
// Insérer ici les champs et les méthodes
}
```

Les modificateurs de classe (ClassModifiers) sont :

| Modificateur | Rôle |
|---|---|
| public | La classe est accessible partout |
| private | la classe n'est accessible qu'à partir du fichier où elle est définie |
| final | La classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles. |
| abstract | La classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires pour ne plus être abstraite. Ce modificateur sera manipulé dans un prochain chapitre. |

Les modificateurs **abstract** et **final** ainsi que **public** et **private** sont mutuellement exclusifs (c-à-d on ne peut ni avoir public suivi de private ni abstract suivi de final).

Le mot clé **extends** permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage (qui sera manipulé dans un prochain chapitre).

Le mot clé **implements** permet de spécifier une ou des interfaces (qui seront manipulées dans un prochain chapitre) que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

**Remarque :**

Par convention, en Java les noms de classes commencent par une lettre majuscule, si elle est composé de plusieurs mots, ces mots doivent être liés par « _ » et commencent par une lettre majuscule.

## 2.2 Attributs et Méthodes

### 2.2.1 Les Attributs

Les données d'une classe sont contenues dans les propriétés ou attributs. Les attributs sont les données d'une classe, ils sont tous visibles à l'intérieur de la classe.

**Syntaxe générale:**

**[**< modificateur de visibilité>**]** <type> <nom_attribut> [=<expression>] ;

**Exemple**

```
class Rectangle
{private int longueur ;
 private int largeur ;
 public Point Centre ;
}
```

**Remarques**

- Les modificateurs de visibilité sont : « public », « private » et « protected ».
- Dans la syntaxe, « expression » permet d'affecter des valeurs par défaut pour les attributs.

Ils peuvent être des variables d'instances, des variables de classes ou des constantes.

#### *2.2.1.1 Les variables d'instances*

Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

**Exemple :**

```
public class MaClasse {
public int valeur1 ;
int valeur2 ;
protected int valeur3 ;
private int valeur4 ;
}
```

Chaque instance de la classe a accès à sa propre occurrence de la variable.

#### *2.2.1.2 Les variables de classes (static)*

Les variables de classes sont définies avec le mot clé ***static***

**Exemple :**

```
public class MaClasse {
static int compteur ;
}
```

Chaque instance de la classe partage la même variable.

#### *2.2.1.3 Les constantes*

Les constantes sont définies avec le mot clé ***final*** : leur valeur ne peut pas être modifiée.

**Exemple :**

```
public class MaClasse {
final double pi=3.14 ;
}
```

### 2.2.2 Les méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe.

#### *2.2.2.1 La syntaxe de la déclaration*

La syntaxe de la déclaration d'une méthode est :

```
modificateurs type_retourné nom_méthode ( arg1, ... ) {... }
// Définition des variables locales et du bloc d'instructions
// Les instructions
}
```

Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise **void.**

Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis. Il n'est pas possible d'indiquer des valeurs par défaut dans les paramètres. Les arguments sont passés par valeur : la méthode fait une copie de la variable qui lui est locale. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine et qui est une copie de la variable. Il est possible de modifier l'objet grâce à ces méthodes mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre : ce changement n'aura lieu que localement à la méthode.

Une méthode représente l'équivalent d'une procédure ou fonction dans les langages classiques de programmation.

**Exemple**

```
class Rectangle {
                    private int longueur ;
                    int largeur ;
                    public int calcul_surface ( ) {
                        return (longueur*largeur) ;
                    }
                    public void initialise (int x, int y) {
                            longueur =x ; largeur = y ;
                    }
}
```

#### 2.2.2.2 Remarques

- Toutes les méthodes sauf les «constructeurs » doivent avoir un type de retour.
- Les méthodes qui ne renvoient rien, utilisent « void » comme type de retour.
- Le code de la méthode est implémenté à la suite de l'accolade ouvrante « { ».
- Une méthode peut ne pas avoir besoin d'arguments, ni de variables à déclarer.
- Les méthodes agissent sur les attributs définis à l'intérieur de la classe.
- Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.
- La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle ci : toutes les instructions qui suivent return sont donc ignorées.

**Exemple :**

```
int add(int a, int b) {
return a + b;
}
```

Il est possible d'inclure une instruction return dans une méthode de type void : cela permet de quitter la méthode.

La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante:

**Déclaration d'une méthode main () :**

```
public static void main (String args[]) { ... }
```

Si la méthode retourne un tableau alors les [] peuvent être précisés après le type de retour ou après la liste des paramètres :

**Exemple :**

```
int[] valeurs() { … }
int valeurs()[] { … }
```

### *2.2.3 La transmission de paramètres*

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié via un message (appel d'une méthode).

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

Si un objet o transmet sa variable d'instance v en paramètre à une méthode m, deux situations sont possibles :

- si v est une variable primitive alors elle est passée par valeur : il est impossible de la modifier dans m pour que v en retour contienne cette nouvelle valeur.

- si v est un objet alors m pourra modifier l'objet en utilisant une méthode de l'objet passé en paramètre.

### *2.2.4 L'émission de messages*

Un message est émis lorsqu'on demande à un objet d'exécuter l'une de ses méthodes.
La syntaxe d'appel d'une méthode est : nom_objet.nom_méthode(parametre, … ) ;
Si la méthode appelée ne contient aucun paramètre, il faut laisser les parenthèses vides.

### 2.2.5 Constructeurs

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.
Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur.
La définition d'un constructeur est facultative. Si elle n'est pas définie, la machine virtuelle appelle un constructeur par défaut vide créé automatiquement. Dès qu'un constructeur est explicitement défini, Java considère que le programmeur prenne en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, est supprimé. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres.
Il existe plusieurs manières de définir un constructeur :

- ***Le constructeur simple*** : ce type de constructeur ne nécessite pas de définition explicite: son existence découle automatiquement de la définition de la classe.

**Exemple :**

```
public MaClasse() {}
```

- ***Le constructeur avec initialisation fixe*** : il permet de créer un constructeur par défaut

**Exemple :**

```
public MaClasse() {
nombre = 5;
}
```

- ***Le constructeur avec initialisation des variables*** : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur

**Exemple :**

```
public MaClasse(int valeur) {
nombre = valeur;
```

```
}
```

**Exemple**

```
class  Rectangle
{private int longueur ;
 private int largeur ;
Rectangle () {longueur =10 ; largeur = 5 ;}     // C'est un constructeur sans paramètres
Rectangle (int x, int y) {longueur =x ; largeur = y ;}  // constructeur avec 2 paramètres
Rectangle (int x) {longueur =2*x ; largeur = x ;}  // constructeur avec 1 paramètre
public int calcul_surface ( ) {return (longueur*largeur) ;}
}
```

### *2.2.6 Les destructeurs*

Un destructeur permet d'exécuter du code lors de la libération de la place mémoire occupée par l'objet. En java, les destructeurs appelés finaliseurs (finalizers), sont automatiquement appelés par le garbage collector. Pour créer un finaliseur, il faut redéfinir la méthode finalize() héritée de la classe Object.

### *2.2.7 Les accesseurs*

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées *private* à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de message ».

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée.

Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

**Exemple :**

```
private int valeur = 13;
public int getValeur(){
return(valeur);
}
public void setValeur(int val) {
valeur = val;
}
```

### 2.2.8 L'enchaînement de références à des variables et à des méthodes

**Exemple :**

```
System.out.Println("bonjour");
```

Deux classes sont impliqués dans l'instruction : System et PrintStream. La classe System possède une variable nommée out qui est un objet de type PrintStream.Println() est une méthode de la classe PrintStream. L'instruction signifie : « *utilise la méthode Println() de la variable out de la classe System* ».

## 2.3 Objets et Instanciation

### 2.3.1 Objet en Java

L'objet est l'unité de base de la conception orientée objet d'un système informatique. Il représente des entités réelles (personne, animal…) ou conceptuelles (cercle, point, graphique…).

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En java, une application est un objet. La classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seul les données sont différentes à chaque objet.

Un objet est caractérisé par :

- Un ensemble de données (**attributs**) et d'actions (**méthodes**) formant un tout indépendant.
- Les **attributs** portent des valeurs attachées à l'objet. L'ensemble de ces valeurs à un instant donné constitue l'état de l'objet à cet instant,
- Les **méthodes** permettent de faire "vivre" l'objet (en lui faisant effectuer des actions et peut-être changer son état). Elles définissant son comportement et ses réactions aux stimulations externes et implémentent les algorithmes invocables sur cet objet,
- Une identité qui permet de le distinguer des autres objets, de manière unique.
- L'objet est "l'instanciation d'une classe"

**Exemple** : un bouton dans une interface graphique

- Attributs : une image, un label, une couleur de fond, une police de caractères.
- Méthodes : se dessiner, réagir quand on clique dessus.

### 2.3.2 Instanciation

La création d'un objet à partir d'une classe est appelée instanciation. Cet objet est une instance de sa classe.

Une instanciation se décompose en trois phases

1. Création et initialisation des attributs en mémoire, à l'image d'une structure.
2. Appel des méthodes particulières : les constructeurs qui sont définies dans la classe
3. Renvoi d'une référence sur l'objet (son identité) maintenant créé et initialisé.

**Exemple :** on définit une classe Cercle, contenant entre autres un attribut rayon et un attribut epaisseur_trait. Chaque cercle instancié de cette classe possédera son propre rayon et sa propre épaisseur du trait.

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme **classe nom_variable.**

**Exemple :**

```
MaClasse m;
```

L'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable.

**Exemple :**

```
m = new MaClasse();
```

Il est possible de tout réunir en une seule déclaration.

**Exemple :**

```
MaClasse  m = new MaClasse();
```

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d'effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type MaClasse, l'instruction **m2 = m** ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

L'opérateur **new** est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : **le constructeur**. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire, il lève l'exception **OutOfMemoryError**.

**Remarque**:

Un objet String est automatiquement créé lors de l'utilisation d'une constante chaîne de caractères sauf si celle-ci est déjà utilisée dans la classe. Ceci permet une simplification dans l'écriture des programmes.

**Exemple :**

String chaine = "bonjour" et String chaine = new String("bonjour") sont équivalents.

Pour instancier un objet on passe par les 2 étapes suivantes :

- Déclarer une variable de la classe que l'on désire instancier. Cette variable est une référence sur l'objet que l'on va créer. Java y stockera l'adresse de l'objet.

**Exemple** : Rectangle r 1; Rectangle r2 ;….

- Allouer la mémoire nécessaire à l'objet et l'initialiser. cette opération se fait par l'intermédiaire de l'opérateur new auquel on passe l'appel au constructeur de la classe désirée.

**Exemple**: r1 = new Rectangle (10, 4); r2 = new Rectangle ();

**Remarque** : on peut faire les deux étapes en même temps :

**Rectangle r1= new Rectangle (10, 4);**

L'objet est détruit lorsque il n' y a plus des références pour cet objet (sortie d'un bloc de visibilité, etc....).

**Remarques**

La création des objets se fait généralement dans les classes qui utilisent la classe à instancier et plus particulièrement dans la méthode qui constitue le point d'entrée « main ». On va distinguer 2 cas :

- **cas1 : la classe à instancier ne possède pas des constructeurs**

***Exemple***

```
class  Rectangle
{private int longueur ;
private int largeur ;
public int calcul_surface ( )
{return (longueur*largeur) ;}
public void initialise (int x, int y) {
longueur =x ; largeur = y ;}
public void initialise (int x) {
longueur =2*x ; largeur = x ;}
}
public class Test_Rect
{public static void main (String [] argv)
{Rectangle r1, r3;
 Rectangle r2 = new Rectangle ();
 r1 = new Rectangle ();
 r3 = new rectangle (5);
/*erreur car la classe ne définit pas des
constructeurs*/
 ……
}
}
```

- **cas2 : la classe à instancier possède au moins un constructeur**

**Exemple**

```
class  Rectangle
{private int longueur ;
 private int largeur ;
Rectangle (int x, int y) {longueur =x ; largeur = y ;}
// constructeur avec 2 paramètres
```

```
Rectangle (int x) {longueur =2*x ; largeur = x ;}
 // constructeur avec 1 paramètre
public int calcul_surface ( ) {return (longueur*largeur) ;}
}
Rectangle  r1 = new Rectangle (10,4)
// juste car on a un constructeur avec 2 paramètres.
Rectangle  r1 = new Rectangle (10)
// juste car on a un constructeur avec 1 paramètre.
Rectangle  r1 = new Rectangle ()
// faux car on a pas un constructeur sans paramètres. Le constructeur
par défaut n'est plus valable car la classe possède au moins un
constructeur.
```

### 2.3.3 La durée de vie d'un objet

Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

La durée de vie d'un objet passe par trois étapes :

- la déclaration de l'objet et l'instanciation grâce à l'opérateur new

  **Exemple :**

  ```
  nom_classe nom_objet = new nom_classe( ... );
  ```

- l'utilisation de l'objet en appelant ces méthodes
- la suppression de l'objet : elle est automatique en java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (**garbage collector**). Il n'existe pas d'instruction delete comme en C++.

### 2.3.4 La création d'objets identiques

**Exemple :**

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1;
```

m1 et m2 contiennent la même référence et pointent donc tous les deux sur le même objet : les modifications faites à partir d'une des variables modifient l'objet.

Pour créer une copie d'un objet, il faut utiliser la méthode **clone()** : cette méthode permet de créer un deuxième objet indépendant mais identique à l'original. Cette méthode est héritée de la classe **Object** qui est la classe mère de toutes les classes en Java.

**Exemple :**

```
MaClasse m1 = new MaClasse();
MaClasse m2 = m1.clone();
```

m1 et m2 ne contiennent plus la même référence et pointent donc sur des objets différents.

### 2.3.5 Les références et la comparaison d'objets

Les variables déclarés de type objet ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrive c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1 : c1 et c2 réfèrent au même objet (ils pointent sur le même objet). L'opérateur == compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

**Exemple :**

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'un méthode à cette effet : la méthode **equals** héritée de **Object**.

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode **getClass()** de la classe **Object** dont toutes les classes héritent.

**Exemple :**

```
(obj1.getClass().equals(obj2.getClass())
```

### 2.3.6 Utilisation des objets

L'utilisation des objets se manifeste selon 2 axes : l'accès aux attributs et l'accès aux méthodes.

#### 2.3.6.1 L'accès aux attributs

- Pour accéder aux attributs de l'intérieur de la classe, il suffit d'indiquer le nom de l'attribut que l'on veut y accéder.
- Pour accéder de l'extérieur de la classe, on utilise la syntaxe suivante :
  **<nom_méthode>.<nom_attribut>**

#### 2.3.6.2 L'accès aux méthodes

- A l'intérieur d'une classe, les méthodes de cette classe seront appelées en indiquant le nom et la liste des paramètres effectifs de la méthode.
- A l'extérieur de la classe, l'invocation d'une méthode se fait comme suit :
  **<nom_classe>.<nom_méthode> ( [<liste des arguments effectifs>] ) ;**
- La liste des arguments effectifs doit concorder en nombre et en type avec la liste des arguments formels.

**Exemple**

Dans la méthode **« main »** de la classe **Test_Rect**, on peut appeler les méthodes ainsi :

**Rectangle r = new Rectangle (10,5) ;**

**r.affiche()** ; // Appel de la méthode affiche

**int s = r.calcul_surface()** ; //Appel de la méthode calcul_surface

**r.setlong(20)** ; //Appel de la méthode setlong pour modifier la longueur

## 2.4 Membres statiques et Autres mots clés (final, this, null)

### 2.4.1 Les attributs statiques

Les attributs statiques sont des attributs marqués par le mot clé **« static »,** ils désignent les attributs communs entre tous les objets.

On accède à un attribut statique via :

- Une instance quelconque de la classe.
- Le nom de la classe directement.

Si on souhaite sauvegarder le nombre des objets crées et attribuer à chaque objet son numéro d'ordre. Pour satisfaire à cet objectif, on a certainement besoin d'un attribut de type statique qui est commun à tous les objets et qui sauvegarde le nombre d'objets déjà crées.

On va reformuler l'exemple précèdent, en ajoutant un attribut statique appelé **count** qui va indiquer le nombre d'objets crées et un attribut qui s'appelle **num** qui indique le numéro d'ordre d'un objet, cet attribut n'est pas statique car sa valeur est propre à chaque objet.

**Exemple :**

```
class Rectangle
{static int count =0;
private int longueur, larg;
private int num;
Rectangle (int x, int y) {
Count++ ; num = count ; longueur = x ; largeur = y ;}
…….Les autres méthodes
}

class Test_Rec
{
public static void main (String [] args)
{System.out.println ("Nombre des objets = " + Rectangle.count) ; //  affiche 0
Rectangle r = new Rectangle (10,5);
System.out.println ("Nombre des objets = " + Rectangle.count) ; // affiche 1
System.out.println ("Numéro de  l'objet crée = " + r.num) ; //affiche 1
Rectangle r3;
r3= new Rectangle (14);
```

```
System.out.println (“Nombre des objets = ” + Rectangle.count) ; //affiche 2
System.out.println (“Numero de  l’objet crée = ” + r3.num) ; //affiche 2
Rectangle r2 = new Rectangle();
System.out.println (“Nombre des objets = ” + Rectangle.count) ; //affiche 3
System.out.println (“Numero de  l’objet crée = ” + r2.num) ; //affiche 3
}
}
```

### 2.4.2 Les méthodes statiques

S’appellent aussi méthodes de classe, elles désignent les méthodes dont les actions concernent la classe entière et non pas un objet particulier.

De même, l’accès à ces méthodes ne nécessite pas la création d’un objet de la classe, car on peut appeler directement la méthode à travers le nom de la classe. On peut utiliser aussi ces méthodes statiques pour manipuler des données statiques

**Exemple**

On va définir une classe Math qui peut contenir des fonctions mathématiques. De préférence, on déclare ces méthodes comme étant statiques pour pouvoir les utiliser sans instancier des objets de cette classe.

```
Public class Math {

static double PI = 3.14 ; //c’est un attribut statique
static double  getPI () {Return (PI) ;} // La méthode statique manipule des attributs statiques
static  double diametre (double r) {return (PI*2*r)} // c’est une méthode statique
static double puissance (double x) {return (x*x);}  // c’est une méthode statique
}
class  Test_Math {
public static void main (String [] args) {
double i= Math.power (6);
System.out.println (“I=” + i);
Math m = new Math ();
double d =  m.rayon (5.1);
System.out.println (“Le diametre = “ + d);
}
}
```

On remarque que l’accès à une méthode se fait de deux manières soit à travers le nom de la classe (le premier appel de  la méthode power), soit à travers un objet  de la classe (Le deuxième appel de la méthode statique  rayon).

### 2.4.3 Le mot clé « final »

Le mot clé final s'applique aux variables, aux méthodes et aux classes.

Une variable qualifiée de **final** signifie que la variable est **constante**. Une variable déclarée final ne peut plus voir sa valeur modifiée. On ne peut déclarer de variables **final** locales à une méthode. Les constantes sont qualifiées de **final et static**.

**Exemple :**

```
public static final float PI = 3.1416f;
```

Une méthode final ne peut pas être redéfinie dans une sous classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous classée. Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.

Il existe 2 types d'utilisations des attributs finaux : ou bien on initialise la valeur de la variable dés la déclaration, ou bien on initialise plus tard cette variable et son contenu ne sera jamais modifiée.

***Remarque*** : Le mot clé final peut aussi être utilisé avec les méthodes et les classes. Cette utilisation va être détaillée dans les autres chapitres.

### 2.4.4 Le mot clé « this »

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

La référence « this » peut être utile :

- Lorsqu'une variable locale (ou un paramètre) cache, en portant le même nom, un attribut de la classe.
- Pour appeler un constructeur depuis un autre constructeur.

**Exemple**

```
class Rectangle {
private int longueur, largeur ;
private String couleur ;
Rectangle (int lo, int largeur)
{longueur = lo ;
this.largeur = largeur ;}
//1er cas : même nom entre le paramètre et l'attribut
Rectangle (int lo, int lar, String coul) {
this (lo, lar);
// 2 ème cas : appel d'un constructeur à partir d'un autre
couleur = coul;}
}
```

This est aussi utilisé quand l'objet doit appeler une méthode en se passant lui même en paramètre de l'appel.

### 2.4.5 Le mot clé « null »

L'objet null est utilisable partout. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre. Il permet de représenter la référence qui ne référence rien. C'est aussi la valeur par défaut d'initialisation des attributs représentant des références.

null ne peut pas être utilisé comme un objet normal : il n'y a pas d'appel de méthodes et aucune classe ne puisse en hériter.

Le fait d'initialiser une variable référent un objet à null permet au ramasseur de miettes (garbage collector) de libérer la mémoire allouée à l'objet.

**Exemple**

```
class Test_Rec {
public static void main (String [] args) {
Rectangle r;
if (r==null)
r= new Rectangle (10,12);
r.affiche();
}
}
```

## 2.5 Les classes internes

Une classe interne (ou imbriquée) est une classe définie à l'intérieur d'une autre classe au même niveau qu'une méthode ou attribut.

- Les classes internes sont visibles uniquement par les méthodes de la classe dont laquelle sont définies.
- Elles ont accès à tous les éléments de la classe qui les englobe (même privé).
- On ne peut pas accéder directement aux attributs et aux méthodes de la classe interne. Toutefois, on peut les appeler indirectement à travers la classe externe qui l'englobe.

**Exemple**

Supposons qu'on veut représenter à l'intérieur de la classe Rectangle une classe Point qui permet de représenter le centre de rectangle. Cette classe Point sera interne à la classe Rectangle.

```
Class Rectangle {
private int longueur;
private int larg;
Rectangle(int l, int a)
{longueur= l;
larg= a;}
class Point {
```

```
int x, y;
Point ( int x1, int y1)
{x=x1; y=y1;}
void affichep ()
{System.out.println ("Le centre se trouve dans " + x + " " + y);
}
}
Point p = new Point (10,5);
void affiche () {
System.out.println ("Longueur=" + longueur + " Largeur =" + larg);
}
}
class test_Rec{
public static void main(String [] args){
Rectangle r = new Rectangle (4,6);
Point p = new Point(12,14); //faux car la classe Point est invisible de l'extérieur
r.p.affichep(); //on peut accéder à une méthode de la classe interne à travers l'attribut p  de l'objet r.
r.affiche() ;
}
}
```

## 2.6 Les packages

En java, il existe un moyen de regrouper des classe voisines ou qui couvrent un même domaine : ce sont les packages. Un package peut être considéré comme une bibliothèque des classes : il permet de regrouper un certain nombre des classes relativement liées. Les packages peuvent être organisés d'une manière hiérarchique  en sous-packages et ainsi de suite

**Avantages**

- Facilite le développement des bibliothèques et des modules autonomes.
- Les classes d'un même package travaillent conjointement sur un même domaine.
- Facilite la réutilisabilité.

**Organisation**

- Les classes d'un package se trouvent dans un répertoire décrit par le nom du package.
- Ainsi les classes du package Java.lang se trouvent dans un sous-répertoire java/lang accessible par la variable d'environnement CLASSPATH.

### 2.6.1 Création d'un package

Pour réaliser un package :

1. Ajouter dans chaque fichier ".java" composant le package la ligne :

   package <nom_du_package>;
2. Enregistrer le fichier dans un répertoire portant le même nom que le package.
3. Ajouter le chemin de package dans la variable CLASSPATH.

La hiérarchie d'un package se retrouve dans l'arborescence du disque dur puisque chaque package est dans un répertoire nommé du nom du package.

**Remarques :**

- Il est préférable de laisser les fichiers source .java avec les fichiers compilés .class
- D'une façon générale, l'instruction package associe toutes les classes qui sont définies dans un fichier source à un même package.
- Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).
- Le compilateur et la JVM utilisent la variable CLASSPATH pour localiser les emplacements des répertoires servant de racine à une arborescence de packages. Donc, il faut ajouter les localisations des répertoires à cette variable.
- Java importe automatiquement le package « java.lang » qui contient la classe « System ».

### 2.6.2 L'utilisation d'un package

Pour utiliser ensuite le package ainsi créé, on l'importe dans le fichier :

**import nomPackage.*;**

Pour importer un package, il y a trois méthodes si le chemin de recherche est correctement renseigné :

| **Exemple** | Rôle |
|---|---|
| import nomPackage; | les classes ne peuvent pas être simplement désignées par leur nom et il faut aussi préciser le nom du package |
| import nomPackage.*; | toutes les classes du package sont importées |
| import nomPackage.nomClasse; | appel à une seule classe : l'avantage de cette notation est de réduire le temps de compilation |

**Attention :** l'astérisque n'importe pas les sous paquetages. Par exemple, il n'est pas possible d'écrire import java.*.

Il est possible d'appeler une méthode d'un package sans inclure ce dernier dans l'application en précisant son nom complet :

**nomPackage.nomClasse.nomméthode(arg1, arg2 ... )**

Il existe plusieurs types de packages : le package par défaut (identifié par le point qui représente le répertoire courant et permet de localiser les classes qui ne sont pas associées à un package particulier), les packages standards qui sont empaquetés dans le fichier classes.zip et les packages personnels.
Le compilateur implémente automatiquement une commande import lors de la compilation d'un programme Java même si elle ne figure pas explicitement au début du programme :
import java.lang.*; Ce package contient entre autre les classes de base de tous les objets java dont la classe Object.
Un package par défaut est systématiquement attribué par le compilateur aux classes qui sont définies sans déclarer explicitement une appartenance à un package. Ce package par défaut correspond au répertoire courant qui est le répertoire de travail.

### 2.6.3 Quelques packages prédéfinis

- Le package **math** permet d'utiliser les fonctions mathématiques ;
- Dans **io** tout ce qu'il faut pour manipuler les entrées, les sorties, ...
- **sql** contient les classes et méthodes pour interfacer vos applications avec des bases de données (interface JDBC).

### 2.6.4 La collision de classes

Deux classes entrent en collision lorsqu'elles portent le même nom mais qu'elles sont définies dans des packages différents. Dans ce cas, il faut qualifier explicitement le nom de la classe avec le nom complet du package.

### 2.6.5 Les packages et l'environnement système

Les classes Java sont importées par le compilateur (au moment de la compilation) et par la machine virtuelle (au moment de l'exécution). Les techniques de chargement des classes varient en fonction de l'implémentation de la machine virtuelle. Dans la plupart des cas, une variable d'environnement CLASSPATH référence tous les répertoires qui hébergent des packages susceptibles d'être importés.

**Exemple sous Windows :**

```
CLASSPATH = .;C:\Java\JDK\Lib\classes.zip; C:\rea_java\package
```

L'importation des packages ne fonctionne que si le chemin de recherche spécifié dans une variable particulière pointe sur les packages, sinon le nom du package devra refléter la structure du répertoire où il se trouve. Pour déterminer l'endroit où se trouvent les fichiers .class à importer, le compilateur utilise une variable d'environnement dénommée CLASSPATH. Le compilateur peut lire les fichiers .class comme des fichiers indépendants ou comme des fichiers ZIP dans lesquels les classes sont réunies et compressées.