LES CONCEPTS DE BASE DE LA POO

Plan du chapitre :

1. Introduction
2. Définitions
3. Les trois fondamentaux de la POO

Objectifs du chapitre :

- Connaitre les facteurs de naissance de la POO, ses vanatags et ses définitions de base.
- Comprendre les trois fondamentaux de la POO.

1. INTRODUCTION :

La programmation classique telle que étudiée à travers des langages C, Pascal… définit un programme comme étant un ensemble de données sur lesquelles agissent des procédures et des fonctions.

Les données constituent la partie passive du programme. Les procédures et les fonctions constituent la partie active.

Programmer dans ce cas revenait à :

- ✓ Définir un certain nombre de variables (structures, tableaux…)
- ✓ Ecrire des procédures pour les manipuler sans associer explicitement les unes aux autres.

Exécuter un programme se réduit alors à appeler ces procédures dans un ordre décrit par le séquençage des instructions et en leur fournissant les données nécessaires à l'accomplissement de leurs tâches. Dans cette approche données et procédure sont traitées indépendamment les unes des autres sans tenir compte des relations étroites qui les unissent.

Les questions qu'on peut poser dans ce cas :

- ➢ Cette séparation (données, procédures) est elle utile ?
- ➢ Pourquoi privilégier les procédures sur les données (Que veut-on faire ?) ?
- ➢ Pourquoi ne pas considérer que les programmes sont avant tout des ensembles objets informatiques caractérisé par les opérations qu'ils connaissent ?

Les langages objets sont nés pour répondre à ces questions. Ils sont fondés sur la connaissance d'une seule catégorie d'entités informatiques : les objets.

Un objet incorpore des aspects statiques et dynamiques au sein d'une même notion. Avec les objets ce sont les données qui deviennent prépondérantes.

Un programme est constitué d'un ensemble d'objets chacun disposant d'une partie procédures et d'une partie données. Les objets interagissent par envoie de messages.

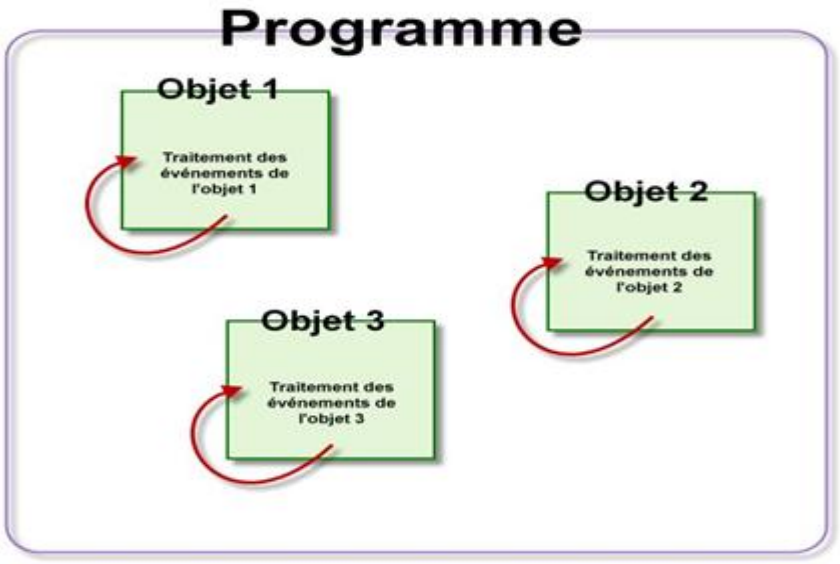


Figure 1: Structure d'un programme Orienté Objet

2. DEFINITIONS :

2.1 POO :

La POO est une méthode d'implémentation dans laquelle les programmes sont organisés sous formes de collections coopératives d'objets, dont chacun représente une instance d'une classe quelconque et dont toutes les classes sont membres d'une hiérarchie de classes unis à travers des relations d'héritage.

2.2 Classes, objets, instances:

Un objet représente une entité du monde réel, ou de monde virtuel dans le cas d'objets immatériels, qui se caractérisent pas une identité, des états significatifs et par un comportement :

- **L'identité d'un objet** : permet de distinguer les objets les uns par rapport aux autres.
- **L'état d'un objet** : correspond aux valeurs de tous les attributs à un instant donné. Ces propriétés sont définies dans la classe d'appartenance de l'objet.
- **Le comportement d'un objet** : se défini par l'ensemble des opérations qu'il peut exécuter en réaction aux messages envoyés (un message = demande d'exécution d'une opération) par les autres objets. Ces opérations sont définies dans la classe d'appartenance de l'objet.

Une classe est l'abstraction d'un ensemble d'objets qui possèdent une structure identique (liste des attributs) et un même comportement (liste des opérations). C'est un modèle décrivant le contenu et le comportement des futurs objets de la classe. Un objet est une *instance* d'une et une seule classe.

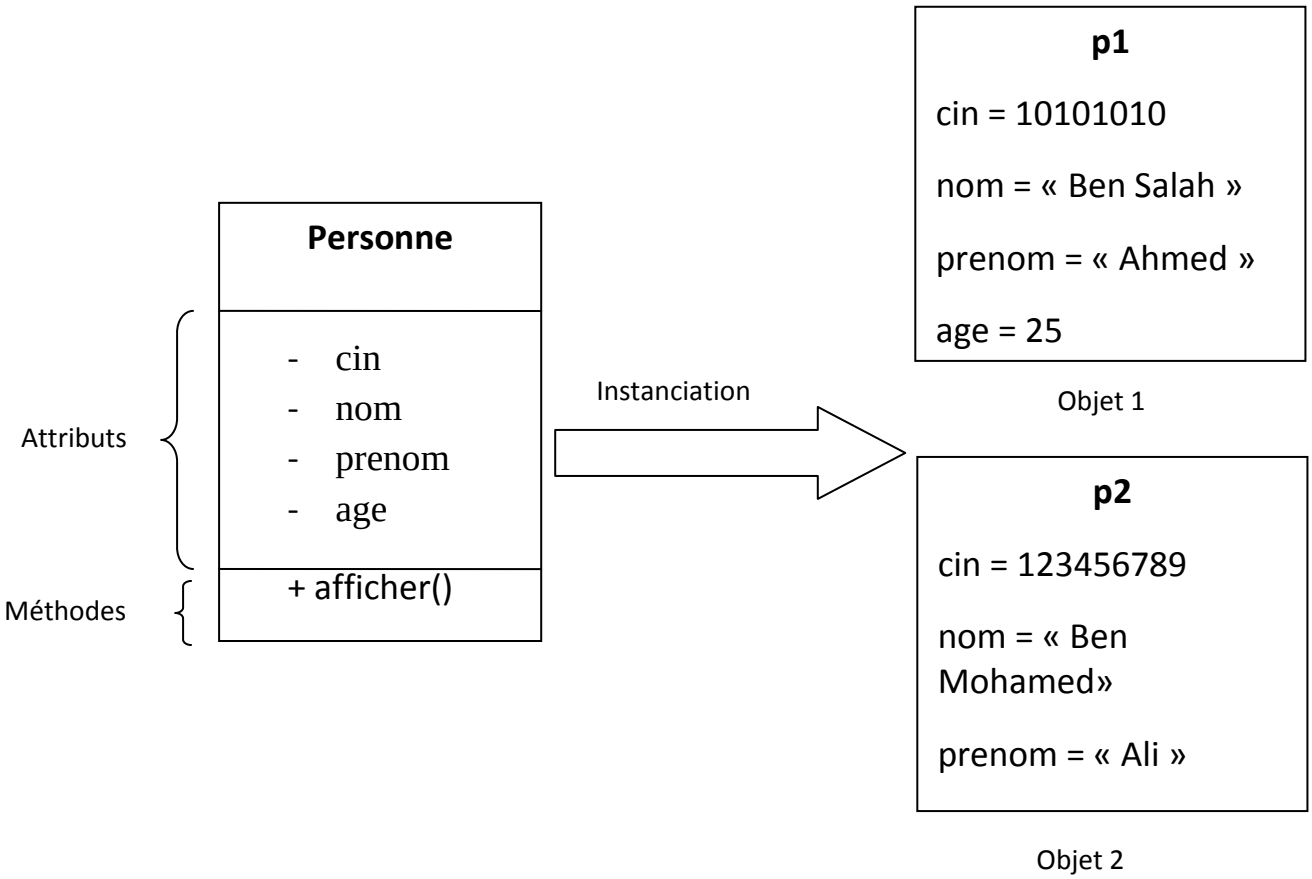


Figure 2: Instanciation d'une classe

2.3 Attribut:

Les attributs d'un objet sont l'ensemble des informations se présentant sous forme de variable et permettant de représenter l'état de l'objet.

2.4 Méthode:

Une méthode est une fonction ou procédure liée à un objet qui est déclenchée à la réception d'un message particulier : la méthode déclenchée correspond strictement au message reçu. La liste des méthodes définies au sein d'un objet constitue l'interface de l'objet pour l'utilisateur : ce sont les messages que l'objet peut comprendre si on les lui envoie et dont la réception déclenche les méthodes correspondantes.

3. LES TROIS FONDAMENTAUX DE LA POO :

La Programmation Orientée Objet est dirigée par trois fondamentaux qu'il convient de toujours garder à l'esprit : **encapsulation**, **héritage** et **polymorphisme :**

3.1 Encapsulation:

L'encapsulation est le faite qu'un objet renferme ses propres attributs et ses méthodes. Les détails de l'implémentation d'un objet sont masqués aux autres objets du système à objets. On dit qu'il ya encapsulation de données et du comportement des objets.

On précise trois modes d'accès aux attributs d'un objet :

- Le mode ***public*** avec lequel les attributs seront accessibles directement par l'objet lui même ou par d'autres objets. Il s'agit du niveau le plus bas de protection.
- Le mode ***private*** avec lequel les attributs de l'objet seront inaccessibles à partir d'autres objets : seules les méthodes de l'objet pourront y accéder. Il s'agit du niveau le plus fort de protection.
- Le mode ***protected :*** cette technique de protection est étroitement associée à la notion d'héritage.

3.2 Héritage :

Permet de spécialiser ou d'étendre une classe, exemple :

- un étudiant est une personne,
- un cercle est un objet graphique,
- un bouton est un objet graphique.

L'héritage est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres. En termes de concepts objets cela se traduit de la manière suivante :

- On associe une classe au concept le plus général, nous l'appellerons *classe de base* ou *classe mère* ou *super - classe*.
- Pour chaque concept spécialisé, on dérive une classe du concept de base. La nouvelle classe est dite *classe dérivée* ou *classe fille* ou *sous-classe.*

On parle d'héritage simple lorsqu'une classe fille ne possède qu'une classe mère :

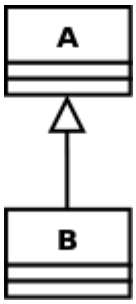


Figure 3: Héritage simple

On parle d'héritage multiple lorsqu'une classe fille possède plusieurs classes mères.

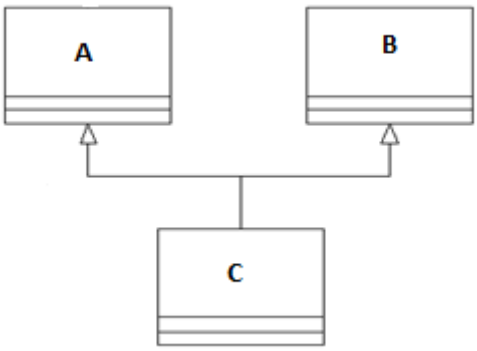


Figure 4: Héritage multiple

3.3 Polymorphisme :

Le polymorphisme traite de la capacité de l'objet à posséder *plusieurs formes*. Cette capacité dérive directement du principe d'héritage. En effet, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir redéfinir une méthode afin de la réécrire, ou de la compléter.

On voit donc apparaître ici ce concept de polymorphisme : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.

Le polymorphisme, en d'autres termes, est donc la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours.

Ainsi, si l'on considère l'exemple suivant :

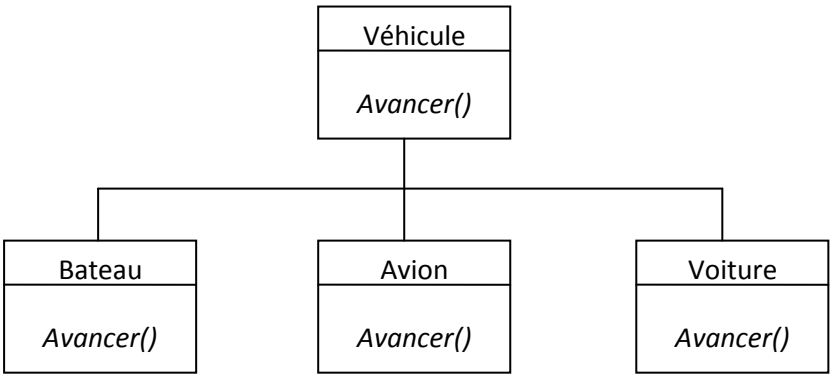


Figure 5: Notion de polymorphisme

Un objet *Véhicule* et ses descendants *Bateau*, *Avion*, *Voiture* possédant tous une méthode *Avancer*, le système appellera la fonction *Avancer* spécifique suivant que le véhicule est un *Bateau*, un *Avion* ou bien une *Voiture*.

Ainsi le polymorphisme se résume par : un même nom, plusieurs implémentations et ses avantages sont les suivants :

- Les fonctions ayant la même sémantique ont le même nom,
- La programmation est plus souple. Si on veut ajouter une classe Camion, il suffit d'ajouter la méthode *Avancer( )* dans cette classe.

LES CONCEPTS DE BASE DU LANGAGE JAVA

Plan du chapitre :

1. Présentation du langage JAVA
2. Interprétation d'un programme JAVA
3. Avantages et caractéristiques du langage JAVA
4. Syntaxe de base du langage JAVA

Objectifs du chapitre :

- Connaitre les caractéristiques principales du langage java
- Utiliser la syntaxe du base du langage JAVA

1. PRESENTATION DU LANGAGE JAVA

Le langage Java est un langage généraliste de programmation synthétisant les principaux langages existants lors de sa création en 1995 par *Sun Microsystems*. Il permet une programmation orientée-objet et reprend une syntaxe très proche de celle du langage C.

Outre son orientation objet, le langage Java a l'avantage d'être modulaire (on peut écrire des portions de code génériques, c'est à dire utilisables par plusieurs applications), rigoureux (la plupart des erreurs se produisent à la compilation et non à l'exécution) et portable (un même programme compilé peut s'exécuter sur différents environnements).

Ce langage est composé de quatre éléments :

- ✓ un langage de programmation
- ✓ une machine virtuelle (JVM)
- ✓ un ensemble de classes standards réparties dans différentes API
- ✓ un ensemble d'outils (jdbc, javadoc, ...)

2. INTERPRETATION D'UN PROGRAMME JAVA

Java est un langage interprété, ce qui signifie qu'un programme compilé n'est pas directement exécutable par le système d'exploitation mais il doit être interprété par un autre programme, qu'on appelle interpréteur :

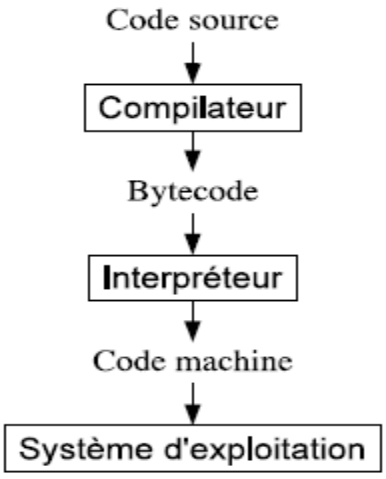


Figure 6: Interprétation d'un programme JAVA

Un programmeur Java écrit son code source, sous la forme de classes, dans des fichiers dont l'extension est .java. Ce code source est alors compilé par le compilateur javac en un langage appelé *bytecode* et enregistre le résultat dans un fichier dont l'extension est .class. Le bytecode

ainsi obtenu n'est pas directement utilisable. Il doit être interprété par la machine virtuelle de Java qui transforme alors le code compilé en code machine compréhensible par le système d'exploitation.

C'est la raison pour laquelle Java est un langage portable : le bytecode reste le même quelque soit l'environnement d'exécution.

3. AVANTAGES ET CARACTERISTIQUES DU LANGAGE JAVA

Les caractéristiques et les avantages de Java se résument selon les termes suivants :

- **Fiable :** Sa conception encourage le programmeur à traquer préventivement les éventuels problèmes, à lancer des vérifications dynamiques en cours d'exécution et à éliminer les situations génératrices d'erreurs...
- **Orienté objet :** Java se concentre sur les objets et sur les interfaces avec ces objets. Java offre de nombreuses classes permettant de définir et de manipuler les objets.
- **Distribué :** Java possède une importante bibliothèque de routines permettant de gérer les protocoles TCP/IP tels que HTTP et FTP. Les applications Java peuvent charger et accéder à des pages sur Internet via des URL avec la même facilité qu'elles accèdent à un fichier local sur le système.
- **Portable :** A la différence du C/C++, on ne trouve pas les aspects de dépendance de la mise en œuvre dans la spécification. Les tailles des types de données primaires sont spécifiées, ainsi que le comportement arithmétique qui leur est applicable.
- **Interprété :** La source est compilé en pseudo code puis exécuté par un interpréteur Java (la Machine Virtuelle Java (JVM)).

4. SYNTAXE DE BASE DU LANGAGE JAVA

4.1 Structure d'un programme JAVA :

Un programme JAVA a l'allure suivante :

```
// Affiche deux phrases à l'écran.
//--------------------------------------------------
public class Affiche
{
    public static void main (String[] args)
    {
        System.out.println ("Voici un exemple de programme
Java.");
        System.out.println ("C'est mon premier programme.");
    }}
```

Après exécution du programme, on peut lire à l'écran :

```
Voici un exemple de programme Java.
C'est mon premier programme.
```

Analyse du programme

- Les lignes du programme, commençant par les caractères //, sont des *commentaires*.
- Le reste du programme, est une *définition de classe* permettant de définir le programme Affiche.
- Tout programme Java doit comporter une méthode *public static void main (String[] args)* qui est la méthode principale, le point d'entrée du programme.
- La méthode *println* permet d'afficher des caractères à l'écran.

- **Commentaire**

Il y a deux manières d'insérer des commentaires dans le code source d'un programme :

- La première manière consiste à utilise les caractères // qui permettent de définir un commentaire sur une ligne.
- L'autre manière permet d'insérer un commentaire qui s'étend sur plusieurs lignes. On marque le début du commentaire avec les caractères /* et la fin de celui-ci avec les caractères */.

- **Identificateurs et mots réservés**

Chaque objet, classe, programme ou variable est associé à un nom : l'identificateur qui peut se composer de tous les caractères alphanumériques et des caractères _ et $. Le premier caractère doit être une lettre, le caractère de soulignement ou le signe dollar.

Les mots réservés font partie intégrante du langage pour lequel ils ont une signification bien définie. Le langage Java compte 50 mots réservés :

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

4.2 Types de donnée élémentaire :

Il existe huit types primitifs en Java :

type élémentaire	*intervalle de variation*	*nombre de bits*
boolean	false , true	1 bit
byte	[-128 , +127]	8 bits
char	caractères unicode (valeurs de 0 à 65536)	16 bits
double	Virgule flottante double précision ~5.10^{308}	64 bits
float	Virgule flottante simple précision ~9.10^{18}	32 bits
int	entier signé : [-2^{31}, $+2^{31}$ - 1]	32 bits
long	entier signé long : [-2^{63}, $+2^{63}$- 1]	64 bits
short	entier signé court : [-2^{15}, $+2^{15}$ -1]	16 bits

Tableau1 : Les types primitifs en JAVA

4.3 Variable et constante :

- **Variable**

Déclaration

Pour déclarer une variable, on utilise la syntaxe suivante :

```
<Type> <nom variable> ;
```

Exemple :

```
int a;
```

Modifier le contenu d'une variable

Pour modifier le contenu d'une variable on utilise *l'opérateur d'affectation* (=) :

a=6 ; =>Pour affecter la valeur 6 à la variable dont le nom a.

- **Constante**

Pour signaler qu'une variable est une constante, on utilise le mot réservé *final* lors de sa déclaration :

```
final int var1= 6;
```

4.4 Les opérateurs

Opérateurs arithmétiques :

Opérateur	priorité	action	exemples
+	1	signe positif	+a; +(a-b); +7 (unaire)
-	1	signe négatif	-a; -(a-b); -7 (unaire)
*	2	multiplication	5*4; 12.7*(-8.31); 5*2.6
/	2	division	5 / 2; 5.0 / 2; 5.0 / 2.0
%	2	reste	5 % 2; 5.0 %2; 5.0 % 2.0
+	3	addition	a+b; -8.53 + 10; 2+3
-	3	soustraction	a-b; -8.53 - 10; 2-3

Tableau2 : Les opérateurs arithmétiques en JAVA

Exemple :

int nbre1, nbre2, nbre3; //déclaration des variables

nbre1 = 1 + 3; //nbre1 vaut 4

nbre2 = 2 * 6; //nbre2 vaut 12

nbre3 = nbre2 / nbre1; //nbre3 vaut 3

nbre1 = 5 % 2; //nbre1 vaut 1, car 5 = 2 * 2 + 1

nbre2 = 99 % 8; //nbre2 vaut 3, car 99 = 8 * 12 + 3

nbre3 = 6 % 3; //là, nbre3 vaut 0, car il n'y a pas de reste

Incrémentation et décrémentation :

Opérateur	priorité	action	exemples
++	1	post ou pré incrémentation : incrémente de 1 son opérande numérique : short, int, long, char, float, double.	++a; a++; (unaire)
--	1	post ou pré décrémentation : décrémente de 1 son opérande numérique : short, int, long, char, float, double.	--a; a--; (unaire)

Tableau3 : Incrémentation et décrémentation

Exemple :

Les trois syntaxes suivantes correspondent exactement à la même opération :

nbre1 = nbre1 + 1;

nbre1 += 1;

nbre1++;

Opérateurs de comparaison :

Opérateur	priorité	action	exemples
<	5	strictement inférieur	5 < 2 ; x+1 < 3 ; y-2 < x*4
<=	5	inférieur ou égal	-5 <= 2 ; x+1 <= 3 ; etc...
>	5	strictement supérieur	5 > 2 ; x+1 > 3 ; etc...
>=	5	supérieur ou égal	5 >= 2 ; etc...
= =	6	égal	5 = = 2 ; x+1 = = 3 ; etc...
!=	6	différent	5 != 2 ; x+1 != 3 ; etc...

Tableau4 : Les opérateurs de comparaison en JAVA

Opérateurs de logiques :

Les opérateurs logiques utilisables sur des booléens sont :

- **&&** ET logique
- || OU logique
- **!** NEGATION logique

4.5 Les structures conditionnelles

La structure if... else

Syntaxe :

```
if (condition)
     {Instructions}
else
    {Instructions}
```

Exemple :

```
if (x==y)
{System.out.println (˝ x et y sont égaux ˝);}
else
{System.out.println (˝ x et y sont différents ˝) ;}
```

La structure switch...case

Syntaxe :

```
Switch (expression) {
Case cas1 : traitement1 ;
Case cas2 : traitement2 ;
.
.
.
Default : traitement n ;}
```

Exemple :

```
int note = 10;
switch (note)
{
case 10:
System.out.println("Vous avez juste la moyenne.");
break;
case 20:
System.out.println("Parfait !");
break;
default:
System.out.println("Il faut davantage travailler.");
```

4.6 Les structures itératives

La boucle while :

Syntaxe :

```
while (/* Condition */)
{
//Instructions à répéter}
```

Exemple :

```
int i = 100, j = 0, somme = 0;
while (j <= i){
    somme += j;
    j++;}
```

La boucle do ... while

Syntaxe :

```
Do
while (/* Condition */) ;
```

Exemple :

```
int i = 100, j = 0, somme = 0 ;
do{
    somme += j;
    j++;}
while (j <= i);
```

La boucle for(...)

Syntaxe :

```
for(condition initiale; condition de test; incrémentation)
{//Instructions à répéter}
```

Exemple :

```
int somme = 0;
for (int i=0; i<=100; i++)
    somme+= i;
```

4.7 Les Tableaux :

Les tableaux à une dimension :

Les tableaux sont déclarés en post-fixant soit le type des variables du tableau soit le nom de celui-ci avec des crochets.

```
int i[];
int[] i;
```

Ce genre de déclarations est insuffisant pour pouvoir utiliser le tableau. Il est nécessaire de préciser sa dimension (pour que le compilateur réserve la mémoire nécessaire) Ceci est réalisé au moyen de l'instruction ***new***.

```
int i[] = new int[10];
```

Cette instruction crée un tableau de 10 entiers dont les indices vont de 0 à 9. Il est aussi possible d'initialiser un tableau lors de sa création :

```
int impairs[] = {1,3,5,7,9,11}; //tableau à 6 éléments .
int x = impairs[2] ; // x = 5
```

Les tableaux à deux dimension :

La syntaxe est la même que pour les tableaux à une dimension :

```
int tab[][] = new int[4][3]; //indices 0, 1, 2, 3 et 0, 1, 2
float mat[][] = new float[3][3];
```

Comme pour les tableaux à une dimension, Il est possible d'initialiser un tableau lors de sa création :

```
int carre = {{11,12,13},{21,22,23},{31,32,33}};
int x =carre[1][2] ; //x = 23
```

Dans un bloc de valeurs, le séparateur est la virgule. Les blocs sont séparés par des virgules

4.8 Les chaînes de caractères :

Dans beaucoup de langages un string n'est autre qu'un tableau de caractères, alors qu'en Java un string est un objet.

En fait, Java propose 3 classes apparentées aux chaînes de caractères :

- La classe String (chaîne de caractères non modifiables)
- La classe StringBuffer (chaîne de caractères modifiables à volonté)
- La classe StringTokenizer (séparation d'une chaîne en plusieurs entités)

Dans la plupart des cas il convient d'utiliser String pour créer, stocker et gérer des chaînes de caractères.

➢ La classe String :

Cette classe dispose de 11 constructeurs et plus de 40 méthodes pour examiner les caractères d'une chaîne, comparer, chercher, extraire, etc. Parmi ces méthodes, on peut citer

Construction d'un String

Une chaîne de caractères se déclare normalement de la manière suivante :

```
String prenom = new String("Ali");
```

Comme les strings sont un type de donnée très utilisé, Java permet une déclaration plus concise :

```
String prenom = "Ali";
```

Comparaison de strings

Une comparaison de chaînes s'effectue de la manière suivante :

```
if (str1.equals(str2)) ...
```

Concaténation

On peut utiliser la méthode *concat* pour concaténer deux chaînes :

```
String s3 = s1.concat(s2);
```

Il est également possible de faire appel a signe d'addition + pour effectuer une concaténation :

```
String chaine = s1 +s2 ;
```

Extraction de caractères

On peut extraire une sous-chaîne à l'aide de la méthode *substring* de la classe String :

```
public String substring(int debut, int fin)
```

Qui retourne un nouveau string qui débute au caractère à la position *debut* et va jusqu'au caractère à la position *fin* -1.

Exemple :

```
msg = "nouveau texte";
txt = msg.substring(0, 6). Retourne « nouveau »
```

Longueur d'une chaîne de caractères

La longueur d'une chaîne est obtenue à l'aide de la méthode *length()*. Par exemple, *txt.length()* retourne 11.

Chapitre 3

LES ELEMENTS DE LA POO EN JAVA

Plan du chapitre :

1. Notion de classe
2. Attributs et méthodes
3. Objets et Instanciation
4. Les classes internes
5. Les packages

Objectifs du chapitre :

- Maîtriser la syntaxe de base pour écrire des classes enJAVA.
- Comprendre le concept des classes internes.
- Connaitre l'atout d'organisation en « package ».

1. NOTION DE CLASSE :

1.1 Concepts :

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.

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 deux 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.

1.2 Syntaxe :

Pour créer une classe, on utilise la syntaxe suivante :

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

Les modificateurs de classe 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.

Tableau5 : Les modificateurs en JAVA

- ✓ Les modificateurs **abstract** et **final** ainsi que **public** et **private** sont mutuellement exclusifs.
- ✓ 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.

Exemple :

```
class Personne  {
//attributs de la classe
//méthodes de la classe
                    }
```

2. ATTRIBUTS ET METHODES :

2.1 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 Personne  {
 private String nom;
 private String prenom;
 public int age; }
```

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.

➢ 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.

➢ Les variables de classes:

Une variable de classe est une variable qui prend la même valeur pour chaque instance de la classe. C'est-à-dire que ces instances partagent la même variable. Les variables de classes sont définies avec le mot clé ***static.***

Exemple :

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

➢ 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 Les méthodes :

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

- ✓ **Syntaxe de la déclaration :**

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

Remarque :

- 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.
- 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 :

```
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 ;}
```

➢ La transmission de paramètre :

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.

➢ 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.

➢ Les 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.

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 d'une classe avec plusieurs constructeurs:**

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

- 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.

- Les accesseurs:

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; }
```

3. OBJETS ET INSTANCIATION :

3.1 Objet en JAVA:

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.

Exemple : un bouton dans une interface graphique

3.2 Instanciation:

Une instanciation se décompose en trois phases :

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

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 :

```
Nom_classe nom_obj ;
```

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();
```

On peut faire les deux étapes en même temps : **Rectangle r1= new Rectangle (10, 4);**

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

Remarque:

- ✓ Si m1 et m2 désignent deux objets de type MaClasse, l'instruction **m2 = m1** ne définit pas un nouvel objet mais m1 et m2 désignent tous les deux le même objet.
- ✓ 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 distingue 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*/
……
}}
```

- **Cas 2 : 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
 public class Test_Rect{
 public static void main (String [] argv)
 {Rectangle r1, r3;
 Rectangle r2 = new Rectangle ();
 r1 = new Rectangle ();
 r3 = new rectangle (5,2);
 /*correcte car la classe a un constructeur avec deux paramètres*/
 ……}}
```

4. LES CLASSES INTERNES

Une classe en Java peut être insérée dans une autre classe. Ci-dessous, B est une classe interne à la classe A. Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit où une seule autre en a besoin
- définir des classes de type adapté (essentiellement pour traiter des événements émis par les interfaces graphiques)

Il est possible d'imbriquer plusieurs classes internes. Java ne possède pas de restrictions sur le nombre de classes qu'il est ainsi possible d'imbriquer.

Un objet d'une classe interne est associé à l'objet de la classe englobante l'ayant créé. Un objet interne peut accéder aux attributs et méthodes de l'objet externe associé. Un objet externe peut accéder aux champs et méthodes de l'objet qu'il a créé.

Exemple de programme Java :

```
// A.java
public class A {
      int va = 10;
      String mot = "bonjour";
      B nb = new B(); //objet B(classe interne) attribut de la classe A
      A (int va) {
      this.va = va;}
      void a1 () {
      System.out.println ("a1 : création de objet b1");
      B objetb1 = new B(100); // objet de la classe interne
      // acces a l'attribut (vb) de objetb1 de la classe interne B
      System.out.println ("a1 : objetb1.vb : " + objetb1.vb);
      objetb1.b1(); // appel d'une methode de la classe B
      System.out.println ("\n\na1 : creation de objetb2");
      B objetb2 = new B(); // objet de la classe interne
      // acces a l'attribut (vb) de objetb2 de la classe interne B
      System.out.println ("a1 : objetb2.vb : " + objetb2.vb);
      objetb2.b1(); // appel d'une methode de la classe B
      }
      void a2 () {
      System.out.println ("*** methode a2 ");
      }
      // classe interne B
      public class B {
      private int vb = 5; // attribut de la classe interne
      B (int vb) {
      this.vb = vb;
      }
      B () {}
      void b1 () {
      // un objet de la classe B peut accéder a un attribut de A
```

```
      // (ex : acces a l'attribut "va" de la classe A)
      System.out.println (" b1 : mot " + mot + ", va " + va + ", vb " +
vb);
      a2(); // appel d'une methode de la classe A
}
} // fin de la classe interne B
public static void main (String[] args) {
      System.out.println ("\n\ncreation de objeta1");
      A objeta1 = new A(10);
      objeta1.a1();
      System.out.println();
      System.out.println ("\n\ncreation de objeta2");
      A objeta2 = new A(50);
      objeta2.a1();
} // fin de class A
```

4. LES PACKAGES

La notion de paquetage (*package* en anglais) s'apparente à la notion de **bibliothèque de programmes**. Plusieurs classes sont regroupées logiquement dans un même paquetage (un même répertoire). L'accès aux classes et donc aux méthodes du paquetage (pour les classes extérieures à ce paquetage) se fait alors en indiquant explicitement ou implicitement le ou les paquetages à consulter. Si plusieurs méthodes portent le même nom, il ne peut y avoir conflit que pour un même paquetage.

Si des méthodes portant le même nom sont définies dans deux paquetages différents, elles sont identifiées sans ambiguïté grâce au nom de paquetage. L'ensemble des paquetages peut être **hiérarchisé** sur plusieurs niveaux en insérant les paquetages dans des sous-paquetages (sous-répertoires du répertoire du paquetage initial).

Toute la hiérarchie des paquetages peut être réunie dans un seul fichier (d'extension .zip ou .jar). Ainsi, la bibliothèque standard des classes Java est hiérarchisée en paquetage comme suit (seule une partie de la hiérarchie est mentionnée) :

Hiérarchie	les paquetages	les classes
java		
awt	java.awt	*interfaces graphiques (boutons, fenêtres, etc.)*
event	java.awt.event	*événements (clic de souris, clavier, etc.)*
image	java.awt.image	*traitement d'images*
io	java.io	*input-output (entrées-sorties)*
lang	java.lang	*classes de base (String, Math, System, etc.)*
util	java.util	*utilitaires (Random, Vector, Calendar, etc.)*
applet	java.applet	*traitement des applets*
net	java.net	*accès au réseau (URL, Socket, etc.)*

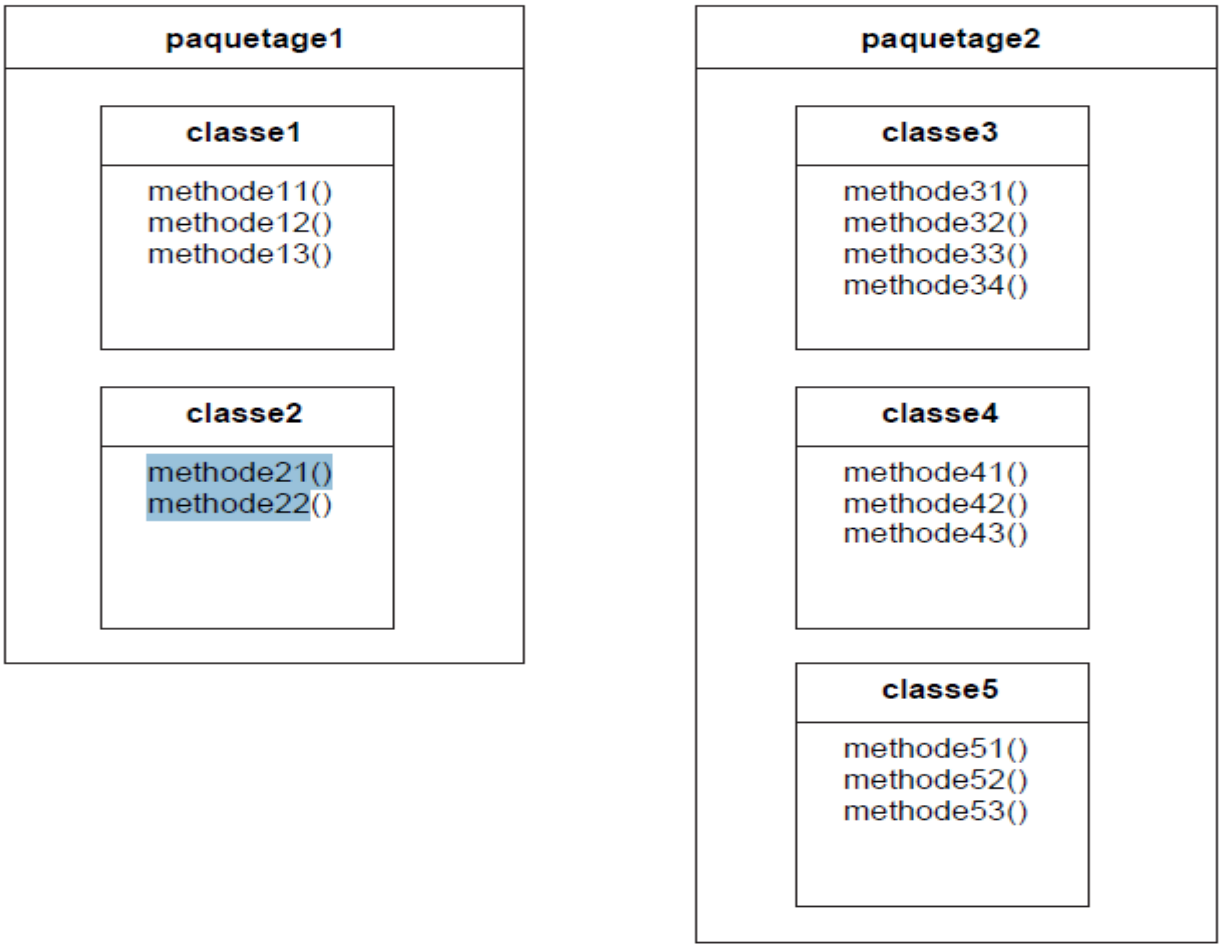


Figure7: Notion de package

5.1. La définition d'un package

Pour réaliser un package, on écrit un nombre quelconque de classes dans plusieurs fichiers d'un même répertoire et au début de chaque fichier on met la directive ci-dessous ou nom-du-package doit être composé des répertoires séparés par un caractère point :

```
package nom-du-package;
```

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).

5.2. 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 deux méthodes si le chemin de recherche est correctement renseigné : préciser un nom de classe ou interface qui sera l'unique entité importée ou mettre un * indiquant toutes les classes et interfaces définies dans le package.

Exemple	Rôle
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

Remarque : 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 ... )
```

Chapitre 4

L'ENCAPSULATION ET LA SURCHARGE

Plan du chapitre :

1. L'encapsulation
2. Modificateurs de visibilité et accès
3. Surcharge des méthodes

Objectifs du chapitre :

- Introduire la notion d'encapsulation et ses intérêts pratiques
- Comprendre les droits d'accès aux membres et aux classes
- Maîtriser le concept de surcharge des méthodes

1. L'ENCAPSULATION :

L'encapsulation est la possibilité de ne montrer de l'objet que ce qui est nécessaire à son utilisation. L'encapsulation permet d'offrir aux utilisateurs d'une classe la liste des méthodes et éventuellement des attributs utilisables depuis l'extérieur. Cette liste de services exportables est appelée l'interface de la classe et elle est composée d'un ensemble des méthodes et d'attributs dits publics **(Public).**

Les méthodes et attributs réservés à l'implémentation des comportements internes à l'objet sont dits privés **(Private).** Leur utilisation est exclusivement réservée aux méthodes définies dans la classe courante.

2. MODIFICATEURS DE VISIBILITE ET ACCES :

2.1 Modificateurs de visibilité :

Les attributs et les méthodes sont précédés lors de la déclaration par l'un des modificateurs de visibilité suivants : **« public », « private », « protected »** et **Néant.**

- Une méthode, classe ou attribut sont déclarés comme publiques « **public** » s'ils doivent être visibles à l'intérieur et à l'extérieur quelque soit leur package.
- Une méthode, classe ou attribut ne sont pas précédés par un modificateur de visibilité explicite (Néant) ne vont être visibles qu'à l'intérieur de même package. C'est-à-dire seules les classes de même package peuvent accéder aux attributs et méthodes de classes « amies ». Ce modificateur de visibilité est aussi appelé « modificateur de package » ou modificateur « **freindly** ».
- Une méthode ou attributs définis comme étant privés **« private »** s'ils sont accessibles uniquement par les méthodes de la classe en cours. Ils ne sont pas accessibles ailleurs.
- Une méthode ou attribut sont définis comme protégés **« protected »** s'ils ne peuvent être accessibles qu'à travers les classes dérivées ou les classes de même package.

Tableaux récapitulatifs des droits d'accès

Modificateur	***Signification pour une classe ou une interface***
Public	Accès toujours possible
Néant	Accès possible depuis les classes du même paquetage

Tableau6 : Modificateurs d'accès des classes et interfaces

Modificateur	*Signification pour les membres et les classes internes*
public	Accès possible partout où la classe est accessible
néant	Accès possible depuis toutes les classes du même paquetage
protected	Accès possible depuis toutes les classes de même paquetage ou depuis les classes dérivées
private	Accès restreint à la classe où est faite la déclaration (du membre ou de la classe interne)

Tableau7 : Modificateurs d'accès pour les membres et les classes internes

2.2 Accès aux membres privés :

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_objet>.<nom_attribut>

Exemple 1 :

- Si longueur et largeur étaient des attributs publics de Rectangle, on peut écrire le code suivant dans la méthode « main » de la classe Test_Rect :

```
Rectangle r = new Rectangle ();
r.longueur = 20;
r.largeur = 15;
int la = r.longueur ;
```

- Si longueur et largeur étaient des attributs privés « private », les instructions précédentes seront refusées par le compilateur. Il faut dans ce cas définir des méthodes d'accès « setlong(int) » et « setlarg (int) » qui permettent de modifier les valeurs des attributs et les méthodes d'accès « getlong()» et « getlarg ()» pour retourner les valeurs de longueur et de largeur d'un objet Rectangle. Dans ce cas, les instructions seront les suivantes :

```
r.setlong(20); //juste
r.setlarg(15); //juste
int la = r.getlong() ; //juste
```

Exemple 2 :

```
public class Rectangle {
private int longueur;
private int larg;

Rectangle (int l, int a) //Le premier constructeur
 { longueur= l;
   larg= a;}

Rectangle() // Le deuxième constructeur
 {longueur= 20;
larg= 10;}

Rectangle (int x) //Le troisième constructeur
 {longueur= 2*x;
larg= x;}

int getlong () //pour retourner la valeur de longueur
{return (longueur);}

int getlarg () //Pour retourner la largeur
{return (larg);}

void setlong (int l) //pour modifier la longueur
{longueur=l;}

void setlarg (int l) //pour modifier la largeur
 {larg=l;}

int surface() //Pour calculer la surface {return(longueur*larg);}

int périmètre()     //pour calculer le périmètre

{return((larg+longueur)*2);}

void allonger(int l)   //Pour allonger la longueur d'un rectangle

{longueur+=l;}

void affiche()  //pour afficher les caractéristiques d'un rectangle

{ System.out.println("Longueur=" + longueur + " Largeur =" + larg );
}
}
```

Code de la classe Test_Rect

```
class Test_Rec {
public static void main(String []args) {

Rectangle r = new Rectangle(10,5);

Rectangle r3;

r3= new Rectangle (14);

Rectangle r2 = new Rectangle();
```

```
r.affiche();
r2.affiche();
r3.affiche() ;
r2.setlong(50);
r2.setlarg(30);
r2.affiche();
System.out.println("Rectangle1" ); System.out.println("Surface= " +
r.surface()); System.out.println("Périmetre= " + r.perimetre());
r.allonger(40);
System.out.println("Aprés allongement");
r.affiche();
}}
```

3. Surcharge des méthodes :

La surcharge est la capacité des objets d'une même hiérarchie de classes à répondre différemment à une méthode de même nom, **mais avec des paramètres différents**.

Par exemple, la classe **Rectangle** peut avoir plusieurs méthodes **Allonger ()** acceptant des paramètres différents en nombre et/ou en types.

En fonction du type et de nombre de paramètres lors de l'appel, la méthode correspondante sera choisie.

Remarque :

- Le mécanisme de surcharge permet de réutiliser le nom d'une méthode déjà définie, pour une autre méthode qui en différera par ses paramètres.
- La méthode surchargée doit conserver la même "intention sémantique".
- La surcharge peut se faire sur une méthode définie localement ou héritée.

Exemple :

```
void allonger(int l)    //Pour allonger la longueur d'un rectangle
{   longueur+=l; }
void allonger(int l, int k)    //Pour allonger un rectangle
{ longueur+=l;
larg+= k}
```

Lors de l'appel des méthodes surchargées, le compilateur sait distinguer entre les méthodes en considérant le nombre et les types des paramètres :

```
Rectangle r = new Rectangle (5,10) ; //Appel de constructeur à 2
//paramètres
r.allonger (10) ; // Appel de la 1ère méthode surchargée
r.allonger (10, 3) ;  // Appel de la 2ème méthode surchargée
```

Chapitre 5

L'HERITAGE

Plan du chapitre :

1. Le principe d'héritage
2. Syntaxe
3. L'accès aux propriétés héritées
4. Construction et initialisation des objets dérivés
5. Redéfinition et surcharge de membres
6. Mots clés «final » et « super »

Objectifs du chapitre :

- Comprendre le concept d'héritage : intérêt et notation
- Maitriser la construction d'un objet dérivé
- Comprendre et utiliser la notion de redéfinition

1. PRINCIPE DE L'HERITAGE :

L'héritage, l'un des mécanismes les plus puissants de la programmation orientée objet, permet de reprendre des membres d'une classe (appelée *superclasse* ou *classe mère*) dans une autre classe (appelée *sous-classe*, *classe fille* ou encore *classe dérivée*), qui en hérite. De cette façon, on peut par exemple construire une classe par héritage successif. L'héritage facilite alors la réutilisation du code et la gestion de son évolution. Ainsi grâce à l'héritage :

- Les objets d'une classe ont accès aux données et aux méthodes de la classe parent et peuvent les étendre. Les sous classes peuvent redéfinir les variables et les méthodes héritées.
- Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.
- L'héritage successif de classes permet de définir une hiérarchie de classe qui ce compose de super classes et de sous classes.
- Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en java.

Remarque :

Object est la classe parente de toutes les classes en java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritage successif toutes les classes héritent d'Object.

2. SYNTAXE :

Pour définir une relation d'héritage entre deux classes, on utilise la syntaxe suivante :

```
class Fille extends Mere { ... }
```

On utilise le mot clé ***extends*** pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe parent.

Exemple :

```
//Classe de base
public class Vehicule {
private String marque;
private String couleur;
private double vitesse; // Vitesse actuelle
private int etat; // 0:arrêt, 1:marche, ...
public Vehicule(String marque, String couleur)
```

```
{ this.marque = marque;
this.couleur = couleur;
vitesse = 0.0;
etat = 0; }
public void demarrer() { etat = 1; }
public void arreter() { if (vitesse == 0) etat = 0; }
public void accelerer() { if (etat == 1) vitesse += 5.0; }
public void freiner()
{ if (vitesse >= 5.0) vitesse -= 5.0;
else vitesse = 0.0; }
 }
//Classe dérivée1
public class Voiture extends Vehicule
{
private String modele; // Nouveau champ
private int nbPortes; // Nouveau champ
public Voiture(String marque, String modele, String couleur, int nbPortes)
{super(marque, couleur); // Appelle le constructeur de la classe parente
this.modele = modele;
this.nbPortes = nbPortes; }
 }
//Classe dérivée2
public class Camion extends Vehicule
{
private double chargemax;
private double poidChargement;
public Camion(String marque, String modele, double chargemax, double poidChargement)
{super(marque, couleur);
this.chargemax = chargemax;
this.poidChargement = poidChargement; }
public void charger(double poids)
{poidChargement = poidChargement+poids; }
public void decharger(double poids)
{poidChargement = poidChargement-poids; }}
```

Interprétation :

- ✓ La classe Voiture est une sous-classe de Vehicule. Elle hérite de tous les attributs et méthodes de Vehicule (marque, couleur, ..., freiner()) en y ajoutant deux nouveaux attributs (modele et nbPortes).
- ✓ La classe Camion est également une sous-classe de Vehicule et hérite de tous ses attributs et méthodes. La classe Camion étend la classe parente (Vehicule) en y ajoutant deux nouveaux attributs (chargeMax et poidsChargement) ainsi que deux nouvelles méthodes (charger() et decharger()) .
- ✓ Dans les sous-classes Voiture et Camion, on peut utiliser les attributs et les méthodes héritées (par exemple couleur ou freiner()) comme si ces membres avaient été déclarés dans chacune des sous-classes (sous réserve, naturellement, que les droits d'accès le permettent).

Remarques :

- ➢ Tout objet peut être vu comme instance de sa classe et de toutes les classes dérivées.
- ➢ Toute classe en Java peut hériter directement d'une seule classe de base. Il n'y a pas la notion d'héritage multiple en Java, mais il peut être implémenté via d'autres moyens tels que les interfaces (Chapitre suivant).
- ➢ Pour invoquer une méthode d'une classe parent, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe parent il suffit d'écrire super(paramètres) avec les paramètres adéquats.
- ➢ En java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

3. L'ACCES AUX PROPRIETES HERITEES :

Les variables et méthodes définies avec le modificateur d'accès **public** restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur **private** est bien héritée mais elle n'est pas accessible directement, elle peut l'être via les méthodes héritées.

Si l'on veut conserver pour une variable d'instance une protection semblable à celle assurée par le modificateur private, il faut utiliser le modificateur **protected**. La variable ainsi définie sera héritée dans toutes les classes descendantes qui pourront y accéder librement mais ne sera pas accessible hors de ces classes directement.

Exemple1 : La classe dérivée et la classe de base sont dans le même package

```
package Formes_geo ;
class graphiques {
```

```
private x, y ;
public String couleur ;
void affiche () {
System.out.println (« Le centre de l'objet = ( » + x + « , » + y + « )
» ) ; }
double surface ( ) {return (0) ;} }
```

```
package Formes_geo ;
class cercle extends graphiques{
double rayon ;
void changer_centre ( int x1, int y1, double r)
{x = x1 ;
y = y1 ;// faux car x et y sont déclarés private dans la classe de
base
rayon =r ;
public double surface ( ) {return (rayon * 2* 3.14) ;}
public void affichec (){
affiche () ; // juste car le modificateur de visibilité de ce membre
est freindly
System.out.println (« Le rayon = » + rayon + « La couleur = « +
couleur ) ;}}
```

Interprétation :

Dans le premier cas, la classe dérivée est la classe de base sont dans le même package, donc la classe cercle a pu accéder aux membres (attributs et méthodes) publiques (l'attribut couleur), de même elle a pu accéder aux membres « freindly » c'est-à-dire qui n'ont pas un modificateur de visibilité (la méthode affiche), mais elle n'a pas pu accéder aux membres privés (x et y).

Exemple 2 :

La classe dérivée et la classe de base ne sont pas dans le même package :

```
package Formes_geo ;
class graphiques {
private x, y ;
public String couleur ;
void affiche () {System.out.println (« Le centre de l'objet = ( » + x
+ « , » + y + « ) » ) ; }
 double surface ( ) {return (0) ;} } //fin de la classe graphiques
```

```
package FormesCirc ;
class cercle extends graphiques{
double rayon ;
void changer_centre ( int x1, int y1, double r)
{x = x1 ; y = y1 ; // faux car x et y sont déclarés private dans la
```

```
classe de base
 rayon =r ;
public double surface ( ) {return (rayon * 2* 3.14) ;}
public void affichec (){affiche () ; // FAUX car le modificateur de
visibilité de ce membre est freindly
System.out.println (« Le rayon =  » + rayon  + « La couleur = «  +
couleur ) ;}}
```

Interprétation :

Dans le deuxième cas, la classe dérivée est la classe de base ne sont pas dans le même package, donc la classe cercle a pu accéder aux membres (attributs et méthodes) publiques (l'attribut couleur), mais, elle n'a pas pu accéder aux membres « freindly » c'est-à-dire qui n'ont pas un modificateur de visibilité (la méthode affiche) et aux membres privés (x et y).

4. CONSTRUCTION ET INITIALISATION DES OBJETS DERIVES :

En java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur. Le constructeur de la classe de base est désigné par le mot clé « super ».

Exemple :

```
Class graphique
 {private x,y ;
  public graphiques (int x1, int y1)
{x = x1 ;
 y = y1 ;}
//autres méthodes
     }
class cercle extends  graphiques{
private double rayon ;
cercle ( int a, int b, double r) {
super (a, b) ;
//appelle le constructeur de la classe de base
rayon = r ; }
//autres méthodes

}
```

Remarque :

Dans le cas général, une classe de base et une classe dérivée possèdent chacune au moins un constructeur. Le constructeur de la classe dérivée complète la construction de l'objet dérivé, et ce, en appelant en premier lieu le constructeur de la classe de base.

Toutefois, il existe des cas particuliers qui sont :

- **cas1 : La classe de base ne possède aucun constructeur**

Dans ce cas, si la classe dérivée veut définir son propre constructeur, elle peut optionnellement appeler dans la 1ère instruction de constructeur la clause « super () » pour désigner le constructeur par défaut de la classe de base.

Exemple :

```
Class A {
// pas de constructeur
}
class B
extends A {
public B(….)
{super() ; //appel de constructeur par défaut de A
……
}
B b = new B(…);  // Appel de  constructeur1 de A
```

- **Cas2 : La classe dérivée ne possède pas un constructeur**

Dans ce cas, le constructeur par défaut de la classe dérivée doit initialiser les attributs de la classe dérivée et appeler :

- Le constructeur par défaut de la classe de base si elle ne possède aucun constructeur.
- Le constructeur sans argument si elle possède au moins un constructeur.
- Dans le cas où il n'y a pas un constructeur sans argument et il y a d'autres constructeurs avec arguments, le compilateur génère des erreurs.

Exemple :

```
Class A {
      public A() {….} //constructeur1
      public A ( int n) {…..} //constructeur2
}
class B extends A {// …..pas de constructeur }
B b = new B();  // Appel de  constructeur1 de A
```

- ***Dérivations successives :***

D'une même classe peuvent être dérivées plusieurs classes différentes. Les notions de classe de base et de classe dérivée sont relatives puisqu'une classe dérivée peut, à son tour servir de base pour une autre classe. Autrement dit, on peut rencontrer des situations telles que :

- La classe **Voiture** est une classe descendante directement de la classe **Véhicule_A_Moteur** et une classe descendante de **Véhicule.**
- La notion de l'héritage multiple n'existe pas directement en Java, mais elle peut être implémentée via la notion des interfaces.

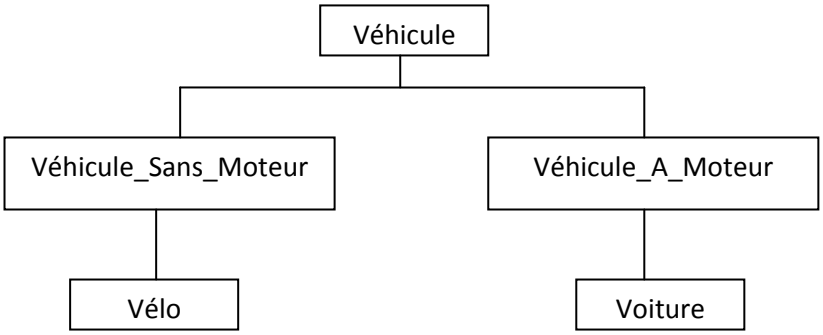


Figure8 : Dérivations successives

5. REDEFINITION ET SURCHARGE DE MEMBRES :

Dans le cadre de l'héritage une classe dérivée pourra à son tour surcharger une méthode d'une classe ascendante. La ou les nouvelles méthodes ne deviendront utilisables que par la classe dérivée ou ses descendantes, mais pas par ses ascendantes.

De même une classe dérivée peut fournir une nouvelle définition d'une méthode de même nom et aussi de même signature et de même type de retour.

5.1 Redéfinition d'une méthode :

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parent (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques) et fournir dans une sous-classe une nouvelle implémentation de la méthode.

Cette nouvelle implémentation masque alors complètement celle de la super-classe. Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge.

Remarque : Grâce au mot-clé **super**, la méthode redéfinie dans la sous-classe peut réutiliser du code écrit dans la méthode de la super-classe, qui n'est plus visible autrement.

Exemple :

```
//Super-classe
class graphiques {
private x, y ;
public String couleur ;
```

```
public void affiche () {
System.out.println (« Le centre de l'objet = ( » + x + « , » + y + « )
» ) ; }
 double surface ( ) {return (0) ;}
 } //fin de la classe graphiques
//classe dérivée
class cercle extends graphiques{
 double rayon ;
public double surface ( ) {return (rayon * 2* 3.14) ;} //redéfinition
de « surface »
public void affiche () //redéfinition de la méthode « affiche »
{super.affiche () ; // appel de affiche de la super-classe
 System.out.println (« Le rayon =  » + rayon  + « La couleur = «  +
couleur ) ;}
    }
```

5.2 Redéfinition de méthode et dérivations successives :

Soit l'arborescence suivante :

La présence d'un astérisque (*) signale la définition ou la redéfinition d'une méthode f :

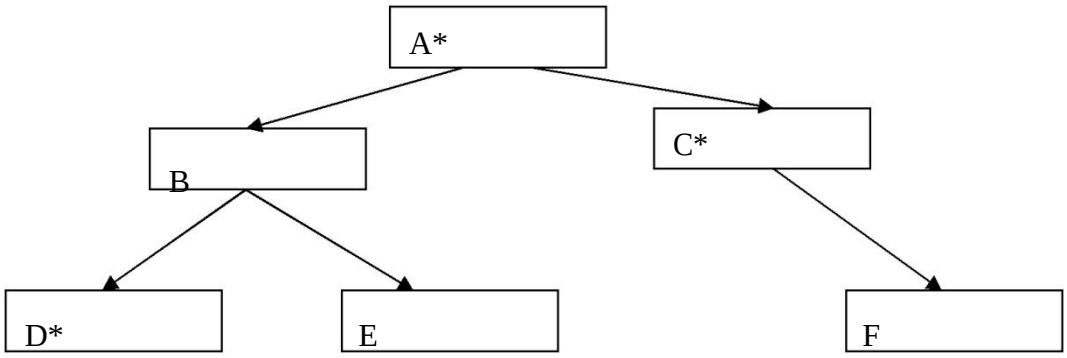


Figure 9: Redéfinition de méthode et dérivations successives

L'appel de la fonction f conduira pour chaque classe à l'appel de la méthode indiquée :

Classe A : Méthode f de A

Classe B : Méthode f de A

Classe C : Méthode f de C

Classe D : Méthode f de D

Classe E : Méthode f de A

Classe F : Méthode f de C

5.3 Surcharge des méthodes :

En Java, une méthode dérivée peut surcharger une méthode d'une classe ascendante :

Exemple :

```
Class A{
public void f (int n) {…..} …}

Class B extends A{
public void f (float n) {…..} …}
A a , B b ;
 int n;
float x;
a.f(n) // appel f(int) de A
a.f(x) // Erreur, on a pas dans A
f(float) b.f(n) // appelle f(int)
de A
v b.f(x) // appel f(float) de B
```

5.4 Utilisation simultanée de surcharge et de redéfinition :

Ci-dessous, un exemple utilisant la surcharge et la redéfinition en même temps :

```
Class A {
public void f (int n) {…..}
public void f (float n) {…..} …}
Class B extends A{
public void f (int n) {…..} // redéfinition de la méthode f( int
//) de A
public void f (double n) {…..} // Surcharge de la méthode f de A
//et de B
…}
A a , B b ;
int n , float x,
double y ;
//Appel de f(int) deA
a.f(n);
a.f(x) ;//Appel de f(float) de A
//Erreur, il n' y a pas
//f(double) dans A b.f(n) ;
a.f(y) ;
//Appel de f(int) de B car
```

```
//f(int) est redéfinie dans B
a.f(x) ; //Appel de f(float) de
A
a.f(y) ;//Appel de f(double) de B
```

6. MOTS CLES « FINAL » ET « SUPER »

6.1 Le mot clé « super » :

Le mot clé « super » peut être utilisé pour :

- Appeler un constructeur de la classe de base dans le constructeur de la classe dérivée. Cet appel doit être la 1ère instruction dans le constructeur de la classe dérivée.

  Syntaxe : `super (<liste des arguments>) ;`

- Accéder dans une méthode d'une classe dérivée à une méthode de la super-classe (optionnel sauf dans le cas de redéfinition).

  Syntaxe : `super.nom_méthode (<liste des arguments>) ;`

- Accéder dans une méthode dérivée à un attribut de la classe de base (optionnel)

  Syntaxe : `super.nom_attribut`

6.2 Le modificateur « final »

Le modificateur « final » est utilisé dans les cas suivants :

Cas1 : Le modificateur « final » placé devant une classe interdit sa dérivation

Exemple :

```
final class A { //Méthodes et attributs }

class B extends A //Erreur car la classe A est déclarée finale
{//....}
```

Cas 2 : Le modificateur « final » placé devant une méthode permet d'interdire sa redéfinition

Exemple :

```
Class A {

final Public void f(int x) {…..}

 //…Autres méthodes et attributs

}
```

```
Class B  {

//Autres méthodes et attributs

public  void f (int x) {…} // Erreur car la méthode f est déclarée
finale. Pas de redéfinition

}
```

Les classes et les méthodes peuvent être déclarées finales pour des raisons de sécurité pour garantir que leur comportement ne soit modifié par une sous-classe.

LE POLYMORPHISME

Plan du chapitre :

1. Concept de polymorphisme
2. Exemple et interprétation
3. Conversion des arguments effectifs
4. Les conversions explicites des références

Objectifs du chapitre :

- Comprendre Le concept de polymorphisme
- Utiliser la conversion des arguments et la conversion explicite des références en rapport avec le polymorphisme

1. CONCEPT DE POLYMORPHISME :

Le polymorphisme est un concept extrêmement puissant en POO, il permet de manipuler des objets sans en connaître tout à fait le type tout en se basant sur la relation d'héritage.

Le polymorphisme en Java, se traduit par :

- La compatibilité entre un type dérivée et un type ascendant.
- La ligature dynamique des méthodes, dans le sens où il permet d'obtenir le comportement adapté à chaque type d'objet, sans avoir besoin de tester sa nature de quelque façon que ce soit.

Le polymorphisme se base sur cette affirmation : un objet a comme type non seulement sa classe mais aussi n'importe quelle classe dérivée.

2. EXEMPLES ET INTERPRETATIONS :

Exemple 1 :

```
//Classe de base
class graphique {
private int x, y;
public graphique ( int x, int y) { this.x = x ; this.y = y ;}
public void identifie () {System.out.println (« Je suis une forme géometrique ») ;}
public void affiche () {this.identifie() ;
System.out.println («  Le centre de l'objet se trouve dans :+x «  et » + y) ;}
double surface () {return (0) ;}
}
// fin de la classe graphique
//Classe dérivée1
class Cercle extends graphique {
 private double rayon =1 ;
public Cercle ( int x, int y, double r) {super(x,y) ; rayon = r ;}
public void identifie () { System.out.println (« Je suis un cercle ») ;}
double surface () {return ( rayon * 2* 3.14) ;} }
//Classe dérivée2
class Rectangle extends graphique {
 private int larg, longueur ;
Rectangle ( int x, int y, int l1, int l2) {
```

```
super (x,y) ;
longueur = l1 ; larg = l2 ;}
double surface () {return (longueur*largeur) ;}
 public void identifie() {
System.out.println(« Je suis un rectangle ») ;} }// fin de la classe
Rectangle
//Classe de test
class test_poly {
public static void main (String [] args) {
graphique g = new graphique (3,7);
g.identifie ();
g= new Cercle (4,8,10) ;
// compatibilité entre le type de la classe et de la classe dérivéé
g.identifie() ;
g= new Rectangle (7,9,10,3) ;
// compatibilité entre le type de la classe et de la classe dérivéé
 g.identifie () ;}}
```

Résultat de l'exécution

Je suis une forme géométrique

Je suis un cercle

Je suis un rectangle

Interprétation

Le même identificateur « g » est initialisé dans la 1ère instruction avec une référence de type « graphique », puis on a changé la référence de cette variable dans l'instruction3 en lui affectant une référence de type « Cercle », puis dans l'instruction 5, on a changé sa référence avec une référence de classe dérivée « Rectangle ».

Ces affectations confirment la compatibilité entre un type d'une classe de base et la référence d'une classe dérivée.

L'autre point qui semble plus important c'est la ligature dynamique des méthodes, dans le sens où le résultat de la méthode « identifie » a changé dans chaque appel selon le type effectif de la variable « g ».

Exemple 2 : Tableau des objets

Dans cet exemple, on va exploiter les possibilités de polymorphisme pour créer un tableau hétérogène d'objets, c'est à dires dans lequel les éléments peuvent être de types différents.

```
class test_poly2 {
public static void main (String [] args) {
graphique [] tab = new graphique [6];
tab[0] = new graphique (3,2);
tab[1] = new Cercle (10,7,3) ;
tab [2] = new Rectangle (4,7,8,6) ;
tab [3] = new graphique (8,10);
tab[4] = new Cercle (8,5,3) ;
tab[5]= new Rectangle (10,17,3,8) ;
for (i=0 ; i <=5 ; i++) {tab[i].affiche();}}}
```

Résultat de l'exécution

Je suis une forme géométrique Le centre de l'objet se trouve dans : 3 et 2
Je suis un cercle Le centre de l'objet se trouve dans : 10 et 7
Je suis un rectangle Le centre de l'objet se trouve dans : 4 et 7
Je suis une forme géométrique Le centre de l'objet se trouve dans : 8 et 10
Je suis un cercle Le centre de l'objet se trouve dans : 8 et 5
Je suis un rectangle Le centre de l'objet se trouve dans : 10 et 17

Remarques :

- Le polymorphisme se base essentiellement sur le concept d'héritage et la redéfinition des méthodes. Toutefois, si dans le même contexte d'héritage, on a à la fois une redéfinition et une surcharge de la même méthode, les règles de polymorphisme deviennent de plus en plus compliquées.
- La notion de polymorphisme peut être généralisée dans le cas de dérivations successives.

3. CONVERSION DES ARGUMENTS EFFECTIFS :

La conversion d'un type dérivé en un type de base est aussi une conversion implicite légale, elle va donc être utilisée pour les arguments effectifs d'une méthode.

Exemple1 :

```
class A{
public void identifie(){System.out.println (« Objet de type A ») ; }
}
class B extends A{
// pas de redéfinition de identitie ici
}
class Test {
static void f(A a ) { a.identitie(); }
}
…..
A a = new A();
B b = new B ();
Test.f(a);
// OK Appel usuel; il affiche "objet de type A"
Test.f(b) ;
/*OK une référence à un objet de type B est compatible avec une référence à un objet de type A. L'appel a.identifie affiche : « Objet de type A » */
```

4. LES CONVERSIONS EXPLICITES DE REFERENCES :

Compatibilité entre référence à un objet d'un type donné et une référence à un objet d'un type ascendant. Dans le sens inverse, la compatibilité n'est pas implicite.

Exemple :

```
Class graphique {…….}
Class Cercle {……..}
…..
graphique g,g1 ;
Cercle c,c1 ;
C = new graphique (….) ; //Erreur de compilation
g= new Cercle (…) //Juste
g1 = new graphique (….) ; //évident
c1= new Cercle(…) ; //évident
c1 = g1 //faux (Conversion implicite illégale)
g1= c1 ; //juste  (conversion implicite légale)
```

Il faut réaliser une conversion explicite c1 = (Cercle) g1 ;

Toutefois cette syntaxe qui est acceptée à la compilation peut être refusée à l'exécution et le traitement peut être arrêtée si il n 'y a pas compatibilité effective entre les types.

LES CLASSES ABSTRAITES ET LES INTERFACES

Plan du chapitre :

1. Les classes abstraites
2. Les interfaces
3. Interface et dérivations

Objectifs du chapitre :

- Comprendre la notion de classe abstraite, les règles de définition et l'intérêt pratique
- Comprendre le concept d'interfaces, leurs intérêts pratiques et leurs relations avec l'héritage

1. LES CLASSES ABSTRAITES :

1.1 Concept des classes abstraites

Une classe abstraite est une classe qui ne permet pas d'instancier des objets, elle ne peut servir que de classe de base pour une dérivation.

Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée et on peut trouver des méthodes dites « abstraites » qui fournissent uniquement la signature et le type de retour.

Syntaxe :

```
abstract class A{
public void f() {……} //f est définie dans A
public abstract void g (int n) ; //g est une méthode abstraite elle
//n'est pas définie dans A }
```

Utilisation :

```
A a ;
//on peut déclarer une référence sur un objet de type A ou dérivé
a = new A (….) ;
//Erreur  pas d'instanciation d'objets d'une classe abstraite
```

Si on dérive une classe B de A qui définit les méthodes abstraites de A, alors on peut :

```
a = new B(…) ; //Juste car B n'est pas une classe abstraite
```

1.2 Règles des classes abstraites

- Dès qu'une classe abstraite comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot clé « abstract » devant sa déclaration.
- Une méthode abstraite doit être obligatoirement déclarée « public », ce qui est logique puisque sa vacation est d'être redéfinie dans une classe dérivée.
- Les noms d'arguments muets doivent figurer dans la définition d'une méthode abstraite :

```
public abstract void g(int) ;
//Erreur : nom argument fictif est obligatoire
```

  - Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites, elle peut ne redéfinir aucune, mais elle reste abstraite tant qu'il y a encore des méthodes abstraites non implémentées.
  - Une classe dérivée d'une classe non abstraite peut être déclarée abstraite.

1.3 Intérêt des classes abstraites

Le recours aux classes abstraites facilite largement la conception orientée objet. On peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour les classes descendantes :

- Soit sous la forme d'une implémentation complète de méthodes (non abstraites) et de champs (privés ou non) lorsqu'ils sont communs à tous les descendants,
- Soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

1.4 Exemple :

```
abstract class graphique {
private int x, y;
graphique ( int x, int y) { this.x = x ; this.y = y ;}
void affiche () {System.out.println («  Le centre de l'objet se trouve
dans :   » + x + «  et » + y) ;}
Abstract public double surface () ;  // méthode abstraite
} // fin de la classe graphique
//Classe dérivée1
class Cercle extends graphique {
private double rayon =1 ;
void affiche () {
System.out.println (« C'est un cercle  de rayon » + rayon) ;
Super.affiche() ; }

double surface () {return ( rayon * 2* 3.14) ;} }
//Classe dérivée2
class Rectangle extends graphique {
private int larg, longueur ;
Rectangle ( int x, int y, int l1, int l2){
super (x,y) ;
longueur = l1 ;
larg = l2 ;}
double surface () {return (longueur*largeur) ;}
}// fin de la classe Rectangle
// Classe de test
class test_poly2 {
public static void main (String [] args) {
```

```
graphique [] tab = new graphique [6];
//tab[0] = new graphique (3,2); Erreur car une classe abstraite ne
peut pas être instanciée
tab[0] = new Cercle (3,2,7);
tab[1] = new Cercle (10,7,3) ;
tab[2] = new Rectangle (4,7,8,6) ;
tab[3] = new Rectangle (8,10,12,10);
tab[4] = new Cercle (8,5,3) ;
tab[5] = new Rectangle (10,17,3,8) ;
for (int i=0 ; i <=5 ; i++) {tab[i].affiche();}
}}
```

Une classe abstraite peut ne comporter que des méthodes abstraites et aucun champ. Dans ce cas, on peut utiliser le concept de **l'interface**.

2. Les interfaces :

2.1 Concept d'interface :

Si on considère une classe abstraite n'implantant aucune méthode et aucun champ (sauf les constantes), on aboutit à la notion d'interface. En effet, Une interface définit les entêtes d'un certain nombre de méthodes, ainsi que des constantes.

L'interface est plus riche qu'un simple cas particulier des classes abstraites :

- Une classe peut implémenter plusieurs interfaces (alors qu'une classe ne pouvait dériver que d'une seule classe abstraite).
- La notion d'interface va se superposer à celle de dérivation, et non s'y substitue.
- Les interfaces peuvent se dériver.
- Une classe dérivée peut implémenter 1 ou plusieurs interfaces, elle peut même réimplémenter une interface déjà implémentée par la superclasse.
- On pourra utiliser des variables de type interface.

2.2 Syntaxe :

- ***Définition d'une interface :***

```
public interface I {

void f (int n) ; //public abstract facultatifs

 void g () ; //public abstract facultatifs

}
```

Toutes les méthodes d'une interface sont abstraites. On peut ne pas mentionner le modificateur de visibilité (par défaut public) et le mot clé « abstract »

- ***Implémentation d'une interface :***

Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot clé « implements » de la manière suivante :

```
public interface I1 {….}
public interface I2 {…}
class A implements I1, I2
{…..A doit définir les méthodes de I1 et I2}
```

Exemple :

```
interface Affichable {void affiche() ; }
class Entier implements Affichable {
private int val ;
public Entier (int n) {val = n ;}
public void affiche() {System.out.println (« Je suis un entier de
valeur  » + val) ;}
}
class Flottant implements Affichable {
private float val ;
public Flottant (float n) {val = n ;}
public void affiche() {System.out.println (« Je suis un flottant de
valeur  » + val) ; }
}
public class TestInterface {
public static void main (String [] args) {
Affichеable [] tab = new Affichеable [2];
tab[0] = new Entier (25);
tab [1] = new Flottant (1.25);
tab [0].affiche(); tab [1].affiche; } }
```

Résultat de l'exécution

Je suis un entier de valeur 25

Je suis un flottant de valeur 1.25

- ***Interface et constantes***

Une interface peut renfermer aussi des constantes symboliques qui seront accessibles à toutes les classes implémentant l'interface :

```
public interface I{
void f (int n) ; //public abstract facultatifs
 void g () ; //public abstract facultatifs
static final int max = 100 ;
}
```

Les constantes déclarées sont toujours considérées comme « static » et « final ».

2.3 Interface et dérivations :

L'interface est totalement indépendante de l'héritage : Une classe dérivée peut implémenter une ou plusieurs interfaces.

Exemple :

```
interface I1 {…..}
interface I1 {…..}
class A implements I1 {……}
class B extends A implements I1, I2 {…..}
```

On peut définir une interface comme une généralisation d'une autre. On utilise alors le mot clé « extends »

Exemple :

```
interface I1 {
void f(int n) ;
static final int max = 100;
}
interface I2 extends I1 {void g (int n);
static final int min = 10;
}
```

En fait la définition de I2 est équivalente à :

```
 interface I2{
void g (int n);
static final int min = 10;
void f(int n) ;
static final int max = 100;
}
```

- ***Conflits des noms :***

Soit l'exemple suivant :

```
interface I1{
void f(int n) ;
void g();
}
interface I2{
void f (float x);
void g();
}
class A implements I1, I2{
/*A doit implémenter la méthode g qui existe dans
 les 2 interfaces une seule fois*/}
```

Supposons que la méthode « g » est présente de 2 façons différentes dans I1 et I2 comme dans l'exemple suivant :

```
interface I1{
void g(); } // g est une méthode abstraite
de I1
interface I2{
int g(); } // g est aussi une méthode
abstraite de I2
class A implements I1, I2 {
/* Erreur car void g() et int g() ne peuvent pas coexister au sein de
la même classe*/
```

LA GESTION DES EXCEPTIONS

Plan du chapitre :

1. Introduction
2. Les exceptions prédéfinies
3. Les exceptions définies par l'utilisateur

Objectifs du chapitre :

- Comprendre la notion d'exception, les règles d'utilisation et de définition des exceptions
- Distinguer entre les exceptions prédéfinies et les exceptions définies par l'utilisateur

1. INTRODUCTION:

Un programme peut être confronté à une condition exceptionnelle (ou exception) durant son exécution. Une exception est une situation qui empêche l'exécution normale du programme. Quelques exemples de situations exceptionnelles :

- Un fichier nécessaire à l'exécution du programme n'existe pas ;
- Division par zéro ;
- Débordement dans un tableau ;
- Etc.

Java propose un mécanisme de gestion des exceptions afin de distinguer l'exécution normale du traitement de celles-ci afin d'en faciliter leur gestion.

On distingue deux types d'exceptions : les exceptions prédéfinies et les exceptions définies par l'utilisateur.

2. LES EXCEPTIONS PREDEFINIES :

En Java, les exceptions sont de véritables objets. Ce sont des instances de classes qui héritent de la classe **Throwable**.

Lorsqu'une exception est levée, une instance de la classe **Throwable** est créée. Voici un aperçu de la hiérarchie des classes pour les exceptions**.**

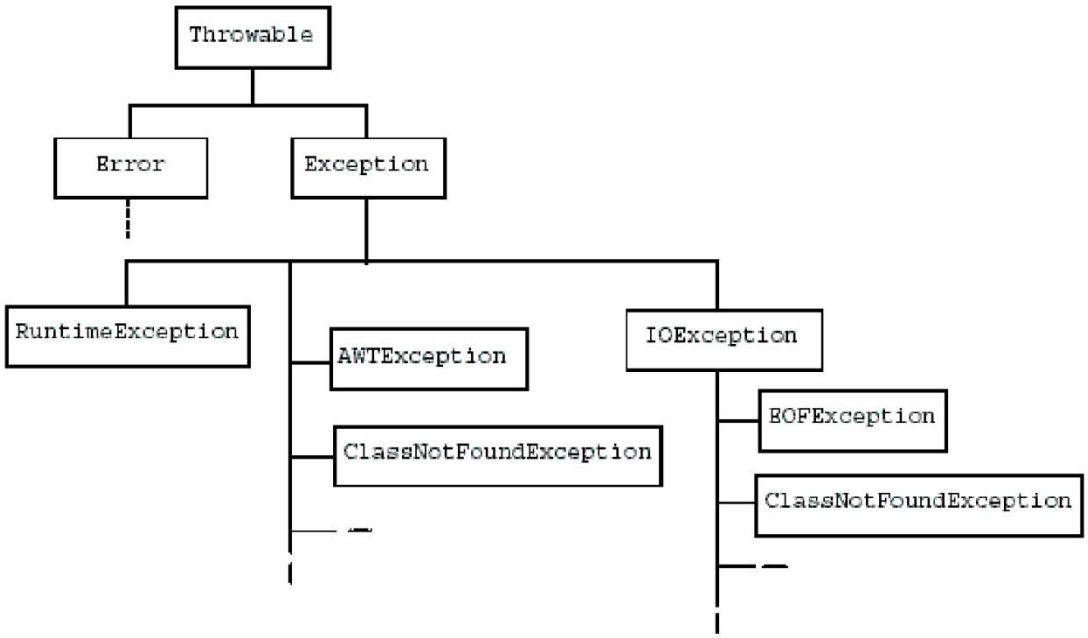


Figure 10: Hiérarchie des classes pour les exceptions

Erreurs et Exceptions :

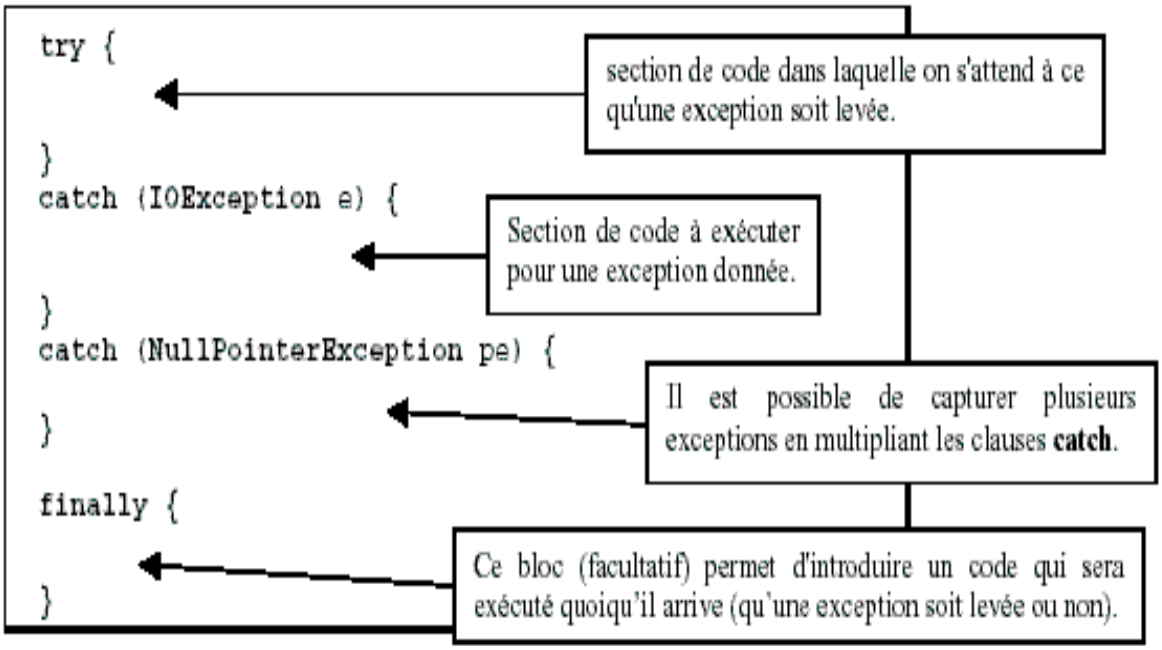
- **Erreur:** indications de problèmes irrécupérables dus au matériel ou au système d'exploitation.
- **Exception:** erreurs résiduelles dans le programme qui peuvent être traitées par une sortie propre ou une récupération (arithmétique, pointeurs, index, i/o, etc.)

➢ Les instances de la classe **Error** sont des erreurs internes à la machine virtuelle Java. Elles sont rares et fatales.

➢ Les sous-classes de la classe Exception sont réparties en deux catégories :

- Les exceptions d'exécution (runtime) sont souvent l'effet du manque de robustesse du code. Par exemple l'exception NullPointerException est levée lorsque l'on manipule un objet non instancié (oubli de l'instruction new) ;
- Les autres exceptions correspondent à des événements anormaux échappant au contrôle du programme. Par exemple, l'exception EOFException est levée si on essaie de lire au-delà d'un fichier.

Pour traiter les exceptions levées, on utilise les mots clés **try, catch et finally** de la manière suivante:



Pour intercepter une exception, on utilise le mot clé **throws** : Si une méthode est susceptible de lever une exception et si elle ne peut pas la traiter, elle se doit de prévenir le système qu'elle relaye cette tâche. Pour ce faire, on utilise le mot clé **throws** dans la définition de la méthode. Ce mot clé permet d'avertir le

système qu'une certaine catégorie d'exception ne sera pas traitée en local (dans l'exemple suivant, l'ensemble des exceptions liées aux entrées/sorties).

```
public void ma_methode (int x) throws IOEXception {     …    }
```

Il est également possible de désigner l'interception de plusieurs exceptions :

```
public void ma_methode (int x) throws IOException, EOFException{ …  }
```

3. LES EXCEPTIONS DEFINIES PAR L'UTILISATEUR :

Jusqu'à présent on parlé des exceptions prédéfinies qui se déclenchent toutes seules. Java offre au programmeur la possibilité de définir ses propres exceptions. Ces exceptions doivent hériter d'une autre exception de la hiérarchie des classes Java. Le programmeur doit lui-même lever ses exceptions.

Pour lever (déclencher) une exception, on utilise le mot-clé **throw** :

```
if (problem) throw new monException("error");
```

Pour capturer une exception, on utilise la syntaxe **try/catch** :

```
try { /* Problème possible */ }
catch (monException e) { /* traiter l'exception. */ }
```

Exemple :

- ✓ Il suffit d'étendre la classe Exception (ou une classe qui l'étend) :

```
public class monException extends Exception {
private int nbr;
public monException(int nbr)
{ this.nbr = nbr; }
public String Affiche()
{ return "Erreur "+nbr; } }
```

- ✓ Pour capturer une exception, on utilise la syntaxe try/catch :

```
public static void test(int valeur) {
System.out.print("A ");
try {
System.out.println("B ");
if (valeur > 12) throw new monException(valeur);
System.out.print("C ");
 }
catch (monException e)
```

```
{ System.out.println(e); } System.out.println("D"); }
```

Résultat d'exécution:

test(11) :

A B

C D

test(13) :

A B

monException: Erreur 13

D