CHAPITRE 3 :

Encapsulation et surcharge

Objectifs spécifiques

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

Eléments de contenu

I. L'encapsulation

II. Modificateurs de visibilité et accès

III. Surcharge des méthodes

Volume Horaire :

Cours : 3 heures

Travaux Pratiques : 9 heures

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.

Les avantages de l'encapsulation sont :

- Simplification de l'utilisation des objets,
- Meilleure robustesse du programme,
- Simplification de la maintenance globale de l'application

Modificateurs de visibilité et Accès

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

Modificateurs d'accès des classes et interfaces

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

Modificateurs d'accès pour les membres et les classes internes

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)

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_méthode>.<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 suivantes seront refusées par le compilateur

```
r.longueur = 20; //faux
r.largeur = 15; //faux
int la = r.longueur ; //faux
```

➔ Il fallait dans le deuxième cas définir des méthodes d'accés « 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();
}
}
```

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.

- 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**

TD1

Exercice 1:

Voici la source de la classe Livre :

```
public class Livre {
// Variables
private String titre, auteur;
private int nbPages

// Constructeur
public Livre(String unAuteur, String unTitre) {
auteur = unAuteur;
titre = unTitre;
}

// Accesseur
public String getAuteur() {
return auteur;
}

// Modificateur
void setNbPages(int n) {
nbPages = nb;
}
```

a- Corrigez quelques petites erreurs

b- ajoutez une méthode main() pour Créer 2 livres, Faire afficher les auteurs de ces 2 livres.

Exercice 2:

Accesseurs et modificateurs

1) Modifiez la classe Livre :

- Ajoutez un accesseur pour la variable titre et la variable nbPages.
- Ajoutez un modificateur pour les variables auteur et titre.
- Changez le modificateur de nbPages : il ne devra changer le nombre de pages que si on lui passe en paramètre un nombre positif, et ne rien faire sinon, en affichant un message d'erreur.

2) Dans la méthode main(),

a- indiquez le nombre de pages de chacun des 2 livres,
b- faites afficher ces nombres de pages,
c- calculez le nombre de pages total de ces 2 livres et affichez-le.

Exercice 3:

1) Dans La classe Livre, ajoutez une méthode afficheToi() qui affiche une description du livre (auteur, titre et nombre de pages).
2) Ajoutez une méthode toString() qui renvoie une chaîne de caractères qui décrit le livre.Modifiez la méthode afficheToi() pour utiliser toString(). Voyez ce qui est affiché maintenant par l'instruction System.out.println(livre).

3) Ajoutez 2 constructeurs pour avoir 3 constructeurs dans la classe :

- Un constructeur qui n'a pas de paramètre
- Un qui prend en paramètre l'auteur et le titre du livre,
- et l'autre qui prend en plus le nombre de pages.

Utilisez les 3 constructeurs (et éventuellement d'autres méthodes) pour créer 3 livres de 300 pages dans la méthode main() de la classe TestLivre.

Exercice 4

1) Contrôle des variables private par les modificateurs

a- Ajoutez un prix aux livres (nombre qui peut avoir 2 décimales ; type Java float ou double) avec 2 méthodes getPrix et setPrix pour obtenir le prix et le modifier.

b- Ajoutez au moins un constructeur qui prend le prix en paramètre.

c- Testez. Si le prix d'un livre n'a pas été fixé, la description du livre (toString()) devra indiquer "Prix pas encore fixé".

d- On bloque complètement les prix : un prix ne peut être saisi qu'une seule fois et ne peut être modifié ensuite (une tentative pour changer le prix ne fait qu'afficher un message d'erreur). Réécrivez la méthode setPrix (et autre chose si besoin est). Vous ajouterez une variable booléenne prixFixe (pour "prix fixé") pour savoir si le prix a déjà été fixé.

e- Réécrire la méthode main () et prévoir le deux cas (prix non fixé ou bien prix fixé plusieurs fois) afficher le résultat de l'exécution.