L'ENCAPSULATION

1. Notion d'encapsulation

a. 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.

b. L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet. **L'exemple 1** suivant présente le concept de l'encapsulation qui se traduit par les les modificateur de visibilite **private, public, protected.**

```
public class Rectangle {
	private    int largeur;
	protected  int longueur;
	public     Rectangle(int  x,inty){this.largeur=x;this.longueur=y;}
	public     void affiche(){System.out.println("lrgeur "+largeur+"
"+"longueur"+longueur);}}
public class Test_Rect{
	public static void main(String[] args) {
		// TODO Auto-generated method stub
				Rectangle r=new Rectangle (45,15);
		r.afficher();            }}
```

Exemple 1 : **Dans cet exemple l'encapsulation se traduit par les mots :**
private, public, protected

2.1. Les avantages de l'encapsulation :

2.1.1. Préservation de l'intégrité des objets:

L'encapsulation permet d'affecter un niveau de visibilité (autorisation d'accès aux classes, interfaces(méthodes) ou attributs d'une classe quelconque) a l'aide des modificateur private,public,protected ou sans modificateur, ce qui rend :

a- les attributs ne peuvent **admettre** toute n'importe quelle valeur
b- **restreindre** les accès aux classes et interface (méthodes)

2.1.2. La gestion d'exception

il est également prévu que toute tentative, lors de l'exécution du programme, visant à violer l'intégrité d'un objet, en lui passant des valeurs d'attributs inadmissibles, puisse faire l'objet d'un mécanisme de gestion d'exception. Ce mécanisme permet, soit à la classe elle-même, soit à son interlocutrice, de prévoir et de prendre en compte la réponse à donner à cette tentative avortée : on interrompt le programme, la classe interlocutrice essaie une autre valeur, on continue comme si de rien n'était, mais, cette fois-ci, sans avoir à effectuer le changement.

La gestion d'exception est un mécanisme de programmation assez sophistiqué, destiné à la réalisation de code plus robuste et qui permet d'anticiper et de gérer les problèmes pouvant survenir lors de l'exécution d'un code dans un contexte sur lequel le programmeur n'a pas tout contrôle. Il pourrait faire l'objet **d'un chapitre à lui tout seul.**

En résumé les avantages de l'encapsulation sont :

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

2. Niveaux de visibilité

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. La table suivante montre les niveaux de visibilité (les droits d'accès) qui peuvent être affectées aux attributs, méthodes et classes :

TABLE – Autorisations d'accès

	public	protected	défaut	private
Dans la même classe	Oui	Oui	Oui	Oui
Dans une classe du même package	Oui	Oui	Oui	Non
Dans une sous-classe d'un autre package	Oui	Oui	Non	Non
Dans une classe quelconque d'un autre package	Oui	Non	Non	Non

2.1. Explication :

- ✓ Pour **la première colonne de la table (public) :** signifie qu'une classe ou un membre de classe (attribut ou méthodes) du modificateur **public** est accessible partout, on peut accéder a ce dernier (dans la même classe, dans une classe du même paquetage , dans sous classe (voir chapitre suivant) d'un autre paquetage et dans une classe quelconque d'un autre paquetage)
- ✓ Pour **la dernière colonne de la table (private) :** signifie qu'un membre de classe du modificateur **private** n'est pas accessible que dans sa classe où il est délardé.

L'exemple 3 dans ce chapitre montre la présente explication

2.3. Tableaux récapitulatifs des droits d'accès

a-Modificateurs d'accès des classes et interfaces

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

b-Modificateurs d'accès pour les membres et les classes internes

Modificateur	Signification pour une classe ou une interface
public	Accès **possible partout où la classe est accessible**
Néant (sans modificateur)	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)**

2.3.4 Notion de paquetage:

Un paquetage (en anglais package) est une bibliothèque qui permet de réunir un grand nombre de classes, fournies par Java SE, implémentent des données et traitements génériques utilisables par un grand nombre d'applications. Ces classes forment l'API (Application Programmer Interface) du langage Java. Une documentation enligne pour l'API java est disponible à l'URL :

http://docs.oracle.com/javase/7/docs/api/

Toutes ces classes sont organisées en packages (ou bibliothèques) dédiés à un thème précis. Parmi les packages les plus utilisés, on peut citer les suivants :

Package Description	Package Description
java.awt	Classes graphiques et de gestion d'interfaces
java.io	Gestion des entrées/sorties
java.lang	Classes de base (importé par défaut)
java.util	Classes utilitaires
javax.swing	Autres classes graphiques

Pour accéder à une classe d'un package donné, il faut préalablement importer cette classe ou son package. Par exemple, la classe **Date** appartenant au package **java.util** qui implémente un ensemble de méthodes de traitement sur une date peut être importée de deux manières :

- une seule classe du package est importée : **import java.util.Date;**

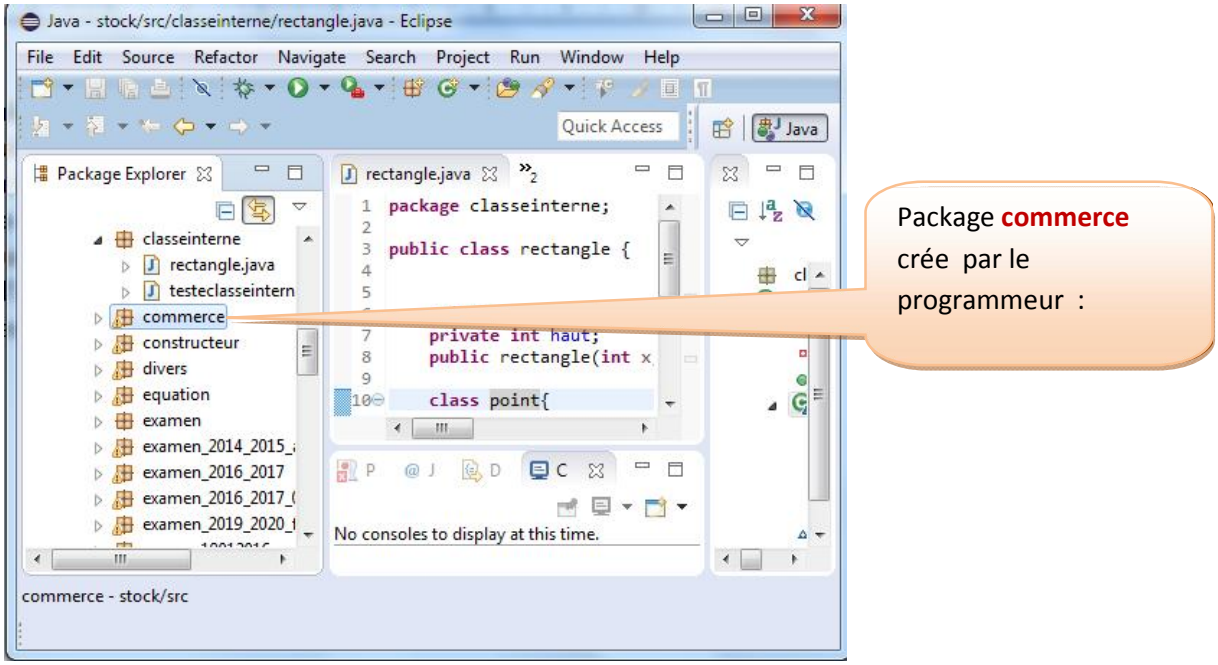
- toutes les classes du package sont importées (même les classes non utilisées) : **import java.util.*;**

Le programme suivant utilise cette classe pour afficher la date actuelle :

```
import java.util.Date;
public class DateMain{
public static void main(String[]args){
Datetoday= newDate();
System.out.println("Noussommes le"+today.toString());}}
```

2.3.4.1 package avec IDE : Eclipse

L'image suivante présente un exemple d'un package crée par un programmeur



a. Encapsulation de données (attributs) et méthode (message)

L'encapsulation permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière (voir chapitre suivant), ou bien d'une classe quelconque. Il existe trois niveaux de visibilité:

- **publique**: les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité *public*. Il s'agit du plus bas niveau de protection des données
- **protégée**: l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées

- **privée**: l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

Exemple 2 : on reprend **l'exemple 1** de la classe Rectangle

- 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 de 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
```

3. Encapsulation en Java

a. Contrôle d'accès (public, private)

Dans les exemples précédents, le mot-clé **public** apparaît parfois au début d'une déclaration de classe ou de méthode, qui permet d'autoriser n'importe quel objet à utiliser la classe ou la méthode déclarée comme publique. La portée de cette autorisation dépend de l'élément à laquelle elle s'applique. **Le tableau 2** présente la portée des autorisations

Élément	Autorisations
Variable	Lecture et écriture
Méthode	Appel de la méthode
Classe	Instanciation d'objets de cette classe et accès aux variables et méthodes de classe

Le tableau 2 : la portée des autorisations

Le mode **public** n'est, bien sûr, pas le seul type d'accès disponible en Java. Deux autres mot clés peuvent être utilisés en plus du type d'accès **par défaut** : **protected** et **private**. Le **tableau 3** récapitule ces déférents types d'accès (la notion de sous-classe est expliquée dans le chapitre suivant).

	public	protected	défaut	private
Dans la même classe	Oui	Oui	Oui	Oui
Dans une classe du même package	Oui	Oui	Oui	Non
Dans une sous-classe d'un autre package	Oui	Oui	Non	Non
Dans une classe quelconque d'un autre package	Oui	Non	Non	Non

Tableau 3 : Autorisations d'accès

Exemple 3 :
Soit le programme suivant, qui comporte deux package p1 et p2
Le p1 comporte trois classe C1, C2 et C3, dont C2 est une sous classe de C1 (voir l'héritage dans le chapitre suivant)
Le p2 comporte deux classe C4 et C5 , dont C4 est sous classe de C1

```
package p1 ;
class C1 {
public int a;
protected int b;
int c;
private int d;}
class C2 extends C1 {}
class C3 {}
```

```
package p2 ;
import C1.p1;
class C4 extends C1 {}
class C5 {}
```

Le tableau 4 présente l'accessibilité pour les variables a,b,c,d par les classes C1,C2,C3,C4,C5

	Variable (a)	Variable (b)	Variable (c)	Variable (d)
l'accessibilité de C2	oui	oui	oui	oui
l'accessibilité de C3	oui	oui	oui	non
l'accessibilité de C4	oui	oui	non	non
l'accessibilité de C5	oui	non	non	non

Le tableau 4 :l'accessibilité des variables par classe selon le programme de l'exemple 3

b. Accesseurs (get et set)

pour sécuriser le maximum les attributs d'une classe on utilise toujours le modificateur de visibilité **private**, ce qui rend l'attribut inaccessible (que se soit lecture ou écriture) en dehors de sa classe où il est déclarés, ce qui nécessite d'implémenter les accesseurs de lectures et écriture ce qu'on appel **les setters et getters.**

Exemple 4 : on reprend **l'exemple 1** de la classe Rectangle avec une petite modification

```
public class Rectangle
{ private int longueur;
 private int largeur;
Rectangle (int l, int a) //Le premier constructeur
{longueur= l;          largeur= a;}
Rectangle () // Le deuxième constructeur
{longueur= 20;         largeur= 10;}
 Rectangle (int x) //Le troisième constructeur
{longueur= 2*x;        largeur= x;}
public int getlong () //pour retourner la valeur de longueur get
{return (longueur);}
public int getlarg () //Pour retourner la largeur get
{return (largeur);}
public void setlong (int l) //pour modifier la longueur set
{longueur=l;}
public void setlarg (int l) //pour modifier la largeur set
{largeur=l;}
public int surface() //Pour calculer la surface
{return(longueur*largeur);}
public int périmètre() //pour calculer le périmètre
{return((largeur+longueur)*2);}
public void allonger(int l) //Pour allonger la longueur d'un rectangle
{longueur+=l;}
public void affiche() //pour afficher les caractéristiques d'un rectangle
{System.out.println("Longueur=" + longueur + " Largeur =" + largeur );}}
```

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();}}
```

c. Accès à l'instance (this)

On peut utiliser **this** comme un objet ou bien **this** (paramètres) comme une méthode.

- ✓ Dans un constructeur ou une méthode d'instance, l'objet représenté par **this** est l'objet avec lequel on est entrain de travailler.
- ✓ **this** est généralement utilisée pour accéder à un attribut d'instance masqué par un argument de la méthode ou une donnée locale.
- ✓ On peut aussi utiliser **this** pour passer l'objet sur lequel on est entrain de travailler en paramètre à une méthode.
- ✓ Invoquer une méthode **this**(paramètres) ne peut se faire qu'en première ligne d'un constructeur. On invoque alors un autre constructeur dont la liste de paramètres corresponde aux classes des paramètres indiqués dans la parenthèse.

Exemple 5:

```
class Cercle
int x,y, rayon;
Cercle(int x, int y, int rayon){   this.x=x;this.y=y;   this.rayon=rayon;//
this est utilisé comme un objet}
Cercle(int x, int y) { this(x,y,10);// this est utilisé comme une méthode  }}
```

d. Variables et méthodes de classe (static)

d.1.les variables d'instance :

- ✓ sont déclarées en dehors de toute méthode
- ✓ conservent l'état d'un objet, instance de la classe
- ✓ sont accessibles et partagées par toutes les méthodes de la classe

d.2. variables et méthodes statiques

- ✓ Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur **static**)
- ✓ pas besoin de créer un objet (instances de classe)

d.3. méthode statique

- ✓ fournissent une fonctionnalité à une classe entière
- ✓ cas des méthodes non destinées à accomplir une action sur un objet individuel de la classe
- ✓ exemples : Math.random(), Integer.parseInt(String s) , main (String args[])
- ✓ les méthodes statiques ne peuvent pas accéder aux variables d'instances

d.4.Le mot réservée static (appelé modificateur) :

Ce modificateur s'applique aux attributs et aux méthodes ; lorsque ce modificateur est utilisé, on dit qu'il s'agit d'un attribut ou d'une méthode de classe.

- ✓ On rappelle qu'une méthode ou un attribut auxquels n'est pas appliqué le modificateur **static** sont dits d'instance.
- ✓ Un attribut déclaré **static** existe dès que sa classe est évoquée, en dehors et indépendamment de toute instanciation. Quelque soit le nombre d'instanciation de la classe (0, 1, ou plus) un attribut de classe, i.e. statique, existe en un et un seul exemplaire. Un tel attribut sera utilisé un peu comme une variable globale d'un programme non objet.
- ✓ Une méthode, pour être de classe (i.e. static) ne doit pas manipuler, directement ou indirectement, des attributs non statiques de sa classe. En conséquence, une méthode de classe ne peut utiliser directement dans son code aucun attribut ou aucune méthode non statique de sa classe une erreur serait détectée à la compilation. Autrement dit, une méthode qui utilise (en lecture ou en en écriture) des attributs d'instance ne peut être statique, et est donc nécessairement une méthode d'instance.

  De l'extérieur d'une classe ou d'une classe héritée(voir le chapitre suivant), un attribut ou une méthode de classe pourront être utilisés précédés du nom de leur classe :

  nom_d'une_classe.nom_de_la_donnée_ou_méthode_de_classe

  ou bien (mais cela est moins correct) du nom d'une instance de la classe.
  Signalons enfin qu'une méthode **static** ne peut pas être redéfinie (voir le chapitre suivant), ce qui signifie qu'elle est automatiquement **final**.

Exemple 6 :

```
public class MaClasse() {
static int compteur = 0; }
```

L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.

Suite de l'exemple 6 :

```
MaClasse m = new MaClasse();
int c1 = m.compteur;
int c2 = MaClasse.compteur;
```

c1 et c2 possèdent la même valeur.

Ce type de variable est utile pour par exemple compter le nombre d'instanciation de la classe qui est faite.