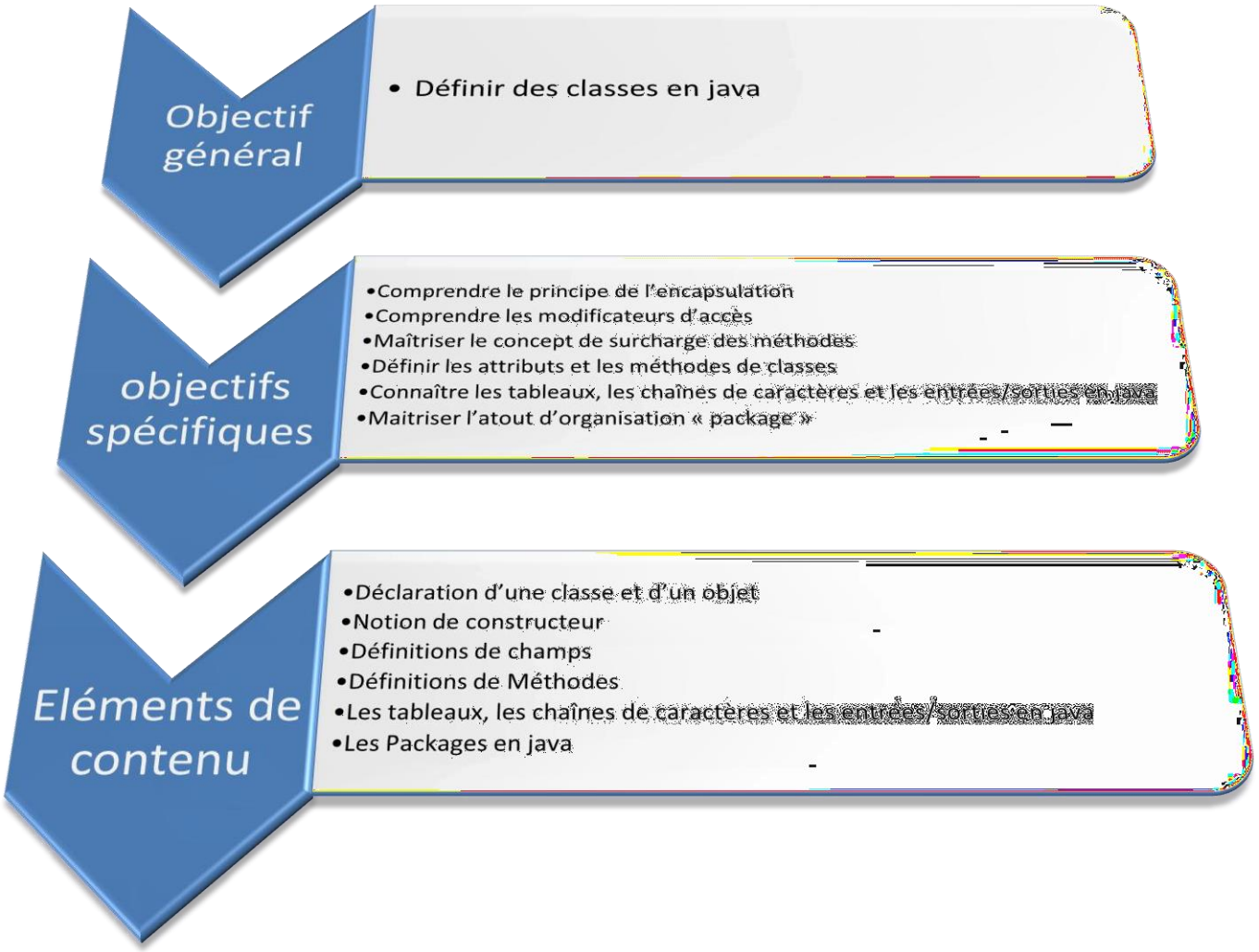
Chapitre 3 : Encapsulation



I. Introduction

L'objet encapsule ses données et les opérations qu'on peut faire sur ses données. La notion fondamentale d'encapsulation est introduite par le concept objet.

L'encapsulation consiste aussi à désigner certain membres (données ou opérations) comme étant privés. Ce qui signifie qu'il n'est plus possible pour le programmeur (utilisateur de l'objet) d'accéder directement aux données mais seulement avec des opérations publiques de cet objet. Le choix des membres privés se fait lors de la conception de l'application, dans le but de préserver la signification des données donc diminuer les risques d'erreur sémantique.

Exemple : Prenant l'exemple d'un objet produit qui a un attribut réel : le prix unitaire si on place une valeur négative comme -50 cette valeur n'a pas de sens pour cet attribut malgré que -50 est un réel.

II. Déclaration d'une classe

2.1 Définition et déclaration d'une classe

Une classe décrit un ensemble de données (attributs, caractéristiques ou **variables d'instance**) et des opérations (**méthodes**) sur les données.

Elle sert de modèle pour la création d'objets, en effet elle crée un nouveau type de variable.

La syntaxe générale de la déclaration d'une classe est la suivante :

```
[Liste de modificateurs] class NomDeLaClasse
{
   Code de la classe
}
```

Les modificateurs permettent de déterminer la visibilité de la classe et comment elle peut être utilisée. Parmi les modificateurs disponibles il y a :

- **public** : La classe peut être utilisée par n'importe quelle classe du même package et des autres packages.
- **private :** Accès uniquement possible à l'intérieur du fichier .java (où elle existe)
- Par défaut les classes sont disponibles pour tous les membres qu'elles contiennent, et ceux-ci sont disponibles les uns pour les autres.

Le début de la déclaration de la classe Personne par exemple est le suivant :

```
public class Personne
{
}
```

Remarque

Dans un fichier java, il ne peut y avoir qu'une seule classe publique et ce fichier doit avoir exactement le même nom que la classe.

Si deux classes publiques sont définies dans un même fichier il y a alors une erreur à la compilation.

2.2 Déclarations des attributs

Pour déclarer des attributs d'une classe il suffit de déclarer des variables à l'intérieur du bloc de code de la classe en indiquant la visibilité de ces variables, leur type et leur nom.

La syntaxe générale de la déclaration d'un attribut est la suivante :

```
[private | protected | public] typeDeLaVariable nomDeLaVariable ;
```

La visibilité d'un attribut répond aux règles suivantes :

- **private** : Attribut restreint à la classe où est faite la déclaration
- **public :** Attribut accessible partout où sa classe est accessible. Ce n'est pas recommandé du point de vue encapsulation
- **protected** : Attribut accessible dans la classe où il est déclarée et dans les autres classes faisant partie du même package et dans les classes héritant de la classe où il est déclaré.

  Remarque

  Si aucun modificateur d'accès n'est déclaré, l'attribut est disponible pour toutes les classes se trouvant dans le même package.

La classe personne prend donc la forme suivante :

```
public class Personne {
//Variables d'instance
private String nom ;
private String prenom;
private String adresse;
}
```

2.3 Déclarations des méthodes

Les méthodes sont simplement des fonctions définies à l'intérieur d'une classe. Elles sont en général utilisées pour manipuler les champs de la classe.

La syntaxe générale de déclaration d'une méthode est décrite ci-dessous.

```
[modificateurs] typeDeRetour nomDeLaMethode ([listeDesParamètres])
{ Corps de la méthode }
```

Parmi les modificateurs on peut citer :

- **public** : La méthode est accessible partout où sa classe est accessible.
- **private :** La méthode est accessible à l'intérieur de la classe où elle est définie.
- **protected :** La méthode est accessible à l'intérieur de la classe où elle est définie, dans les classes faisant partie du même package et dans les sous-classes de cette classe.

Remarque

Si aucun modificateur d'accès n'est déclaré, la méthode est disponible pour toutes les classes se trouvant dans le même package.

Exemple

```
class Personne {
        public String Nom;
        public String Prenom;
        private String Adresse;
        public void init_Personne(String n,String p)
        {       Nom=n;
                Prenom=p;
        }
        public void afficher_homme()
        {       System.out.println(Nom);
                System.out.println(Adresse);
                System.out.println(Prenom);
        }
}
```

2.3.1 Les accesseurs

Deux types de méthodes servent à donner accès aux attributs depuis l'extérieur de la classe :

- **Les accesseurs en lecture** pour fournir les valeurs des attributs ou **getter** en anglais, elles sont nommées **get** suivi du nom du champ.
- **Les accesseurs en écriture** pour modifier leur valeur ou **setter** en anglais, elles sont nommées **set** suivi du nom du champ.

Exemples de méthodes

```
class Personne {
class Personne {
	public String Nom;
	public String Prenom;
	private String Adresse;

	public String getNom() {
		return this.Nom;
	}
	 public void setNom(String nom) {
		Nom = nom;
	}
	public String getPrenom() {
		return Prenom;
	}
	public void setPrenom(String prenom) {
		Prenom = prenom;
	}
	public String getAdresse() {
		return Adresse;
	}
	public void setAdresse(String adresse) {
		Adresse = adresse;
	}
}
```

2.3.2 La surcharge de méthodes

Une méthode **surchargée** est une méthode qui porte le même nom qu'une autre méthode de la classe mais porte une signature différente. Les informations suivantes sont prises en compte pour déterminer la signature d'une méthode :

- Le nom de la méthode,
- Le nombre de paramètres attendus par la méthode,
- Le type des paramètres,
- L'ordre des paramètres.

Pour pouvoir créer une méthode surchargée il faut qu'au moins un de ces éléments change par rapport à une méthode déjà existante.

En Java, le type de la valeur de retour de la méthode ne fait pas partie de sa signature (au contraire de la définition habituelle d'une signature)

Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des paramètres fournis.

Les paramètres des méthodes surchargées doivent être suffisamment différents par le nombre et (ou) par leurs types, pour que la méthode à appeler puisse être déterminée sans ambiguïté.

Exemple

```
double calculerSalaire(int)
double calculerSalaire(int, double)
        indice dans                prime accordées
  la grille des salaires           aux commerciaux
```

- En Java, il est interdit de surcharger une méthode en changeant seulement le type de retour. Autrement dit, on ne peut différencier deux méthodes par leur type retour.

Exemple

```
int calculerSalaire(int)
double calculerSalaire(int)
```

2.4 Les constructeurs et le destructeur

a. Définition d'un constructeur

Un constructeur est une méthode membre de la classe appelée automatiquement au moment de la définition d'une instance de cette classe et qui permet d'effectuer certaines initialisations pour le nouvel objet déjà crée.

Les constructeurs portent le même nom que le nom de la classe et ne retourne aucune valeur (même pas un void).

Toute classe comporte au moins un constructeur et si on n'écrit pas un constructeur, un constructeur est fourni automatiquement. Ce dernier ne prend aucun argument et son corps est vide.

b. Constructeur par défaut

Un Constructeur par défaut appelé aussi un constructeur sans argument est un constructeur qui peut être appelé sans paramètres.

Donc ce constructeur est appelé chaque fois qu'un objet est crée sans qu'il y ait appel explicite d'un constructeur.

Remarques

- Il est important de réaliser que si on ajoute une déclaration de constructeur comportant des arguments à une classe qui n'avait pas de constructeur explicite auparavant on perdra le constructeur par défaut. A partir de là, les appels de new X( ) causeront des erreurs de compilation.

1. Dans une classe on peut avoir plusieurs constructeurs (plusieurs façons d'initialisation), c'est donc la surcharge de constructeur. Le choix se fait alors en fonction de la position des arguments, du nombre des paramètres et du type.

Exemple

```
class Personne
{ ……..
public Personne() //Constructeur par défaut
        {       Nom="";
                Prenom="";
                Adresse="tunis";
        }
public Personne (String n,String p,String a) // un autre constructeur
```

```
        {       Nom=n;
                Prenom=p;
                Adresse=a;
        }
}
```

c. Désigner un constructeur par this( )

Le mot clé **this** désigne l'objet courant sur lequel la méthode est invoquée. On peut par exemple réécrire la méthode init_Personne comme suit :

```
public void init_Personne(String n,String p)
        {       this.Nom=n;
                this.Prenom=p;
        }
```

Pour appeler un constructeur il faut le designer par le **this(….)** en mettant entre parenthèses les paramètres effectifs s'il ne s'agit pas d'un constructeur par défaut.

Exemple

```
public class Personne {
……….
public Personne (String n,String p)
        {       Nom=n;
                Prenom=p;
        }

        public Personne (String n,String p,String a)
        {       this(n,p);
                Adresse=a;}
}
```

d. Le destructeur

Les destructeurs sont d'autres méthodes particulières d'une classe. Elles sont appelées implicitement lors de destruction d'une instance de classe. Il n ya pas de surcharge possible pour les destructeurs. La déclaration d'un destructeur est la suivante :

```
protected void finalize () throws Throwable
        { .....
        }
```

Le code du destructeur doit permettre la libération des ressources utilisées par la classe. On peut par exemple y trouver du code fermant un fichier ouvert par la classe ou la fermeture d'une connexion vers un serveur de base de données.

III. Utilisation d'une classe

L'utilisation d'une classe passe par deux étapes :

- La déclaration d'une variable permettant l'accès à l'objet ;
- La création de l'objet appelée aussi **instanciation**. L'objet créé est donc une **instance** de la classe.

3.1 Déclaration

```
Nom_classe Nom_objet ;
```

Point p ; //Un objet seulement déclaré vaut «**null** »(mot réservé du langage)

Un objet est constitué d'une partie **statique** et d'une partie **dynamique**

- Partie **statique :** Constituée des méthodes de la classe et qui permet d'activer l'objet
- Partie **dynamique :** Constituée d'un exemplaire de chaque attribut de la classe et varie d'une instance de classe à une autre.

3.2 Création et allocation de la mémoire

La création réserve la mémoire et initialise les attributs et renvoie une référence sur l'objet créé : un_Objet != **null**

Exemple

```
public class test_Objet
{ public static void main(String [ ] args)
 { Personne p=new Personne( );}
}
```

Remarque

Un attribut d'une classe peut être déclaré comme étant un objet d'une autre classe

3.3 Accès à un membre d'un objet

L'accès à un membre d'un objet s'effectue à l'aide de l'opérateur "**.** "

```
Nom_objet .membre_i ;
```

Exemple

```
class Test {
        public static void main(String[] args) {
        Personne asma;
        asma=new Personne();
        asma.setNom("farhat");
        asma.set_Adresse("5,rue des olivier Tunis");
        System.out.println("l'adresse est"+asma.getAdresse());}
}
```

3.4 La gestion de la mémoire

Le Garbage Collector (gc) ou ramasse-miettes est chargé de détecter les objets devenus inaccessibles. C'est un système de récupération de mémoire automatique. Par défaut, ce système tourne en arrière-plan pendant l'exécution des programmes. Il repère les objets qui ne sont plus référencés, et libère l'espace mémoire alloué à ceux-ci.

Le ramasse-miettes ou Garbage Collector se met en route automatiquement si :

- Aucune variable ne référence l'objet
- Si le bloc dans lequel il était défini l'objet se termine
- Si l'objet a été affecté à «null »

La récupération de mémoire peut alors être invoquée explicitement par le programmeur à des moments bien précis avec la méthode System.gc()

IV. Les champs et méthodes statiques

4.1 Les champs de classe

Si l'on définit trois objets de type Date, chacun aura évidemment son propre jeu de valeurs pour les champs jour, mois, année. De tels champs sont appelés variables d'instances (ou attributs). Il y a des cas où il est souhaitable d'avoir une donnée commune à tous les objets d'une même classe.

Un champ d'une classe est dit **static** lorsqu'il n'y a qu'un seul exemplaire de ce champ pour tous les objets de cette classe (on dit aussi qu'il est **partagé** entre toutes les instances de la classe) Ce champ existe même s'il n'y a aucune instance de la classe.

Exemple1

```
class Facture
{private final int numéroFacture;
private int jour;
private int annee;
public static int ne=0;
public Facture(int j, int m, int a)
{mois=m; jour=j; annee=a;
 numéroFacture ++; }
 public int getJour( ){return jour;}
 public int getMois( ){return mois;}
 public int getAnnee( ){return annee;}
 public void afficher( )
 {System.out.println(jour+"/"+mois+"/"+annee); } }
class Test
{ public static void main(String[] arg){
 Date aujourdhui=new Date(17,11,2008);
 Date aid=new Date(8,12,2008);
 aujourdhui.afficher( );
 aid.afficher( );
 System.out.println(aid.nbDate);
 System.out.println(Date.nbDate);}
```

```
}
```

Voici le résultat obtenu :

```
18/11/2008
8/12/2000
2
2
```

Exemple2

```
class Date
{private int mois;
private int jour;
private int annee;
private Date suivant;
public static Date tete=null;
public Date(int j, int m, int a)
{jour=j; mois=m; annee=a;
 suivant=tete;
 tete=this; }
 public void afficher( )
 {System.out.println(jour+"/"+mois+"/"+annee); }}
class Test
{public static void main(String[] arg)
 {Date aid=new Date(8,12,2008);
 Date aujourdhui=new Date(17,11,2008);
 for (Date d=Date.tete; d!=null; d=d.suivant) d.afficher(); }
}
```

Remarques

Les champs **static** sont initialisés lors du chargement de la classe qui les contient. Une erreur de compilation se produit lorsque :

– un champ de classe est initialisé relativement à un champ de classe défini plus loin

```
class X
 { static int x = y+1; // erreur, y est déclaré aprés x
   static int y =0;
 }
```

– un champ de classe est initialisé relativement à un champ d'instance

```
class X
  {public int x=120;
    static int y=x+10; // erreur, x variable d'instance}
```

4.2 Les méthodes statiques

4.2.1 Le passage de paramètres

Tous les paramètres sont passés par valeur. Les seuls types possibles de paramètres sont les types primitifs et les références. Autrement dit :

– les types primitifs sont passés par valeur. Une méthode ne peut donc jamais modifier la valeur d'une variable de type primitif,

– les références également sont passées par valeur (valeur de la référence vers l'objet). Si la méthode modifie un champ de l'objet référencé, c'est l'objet qui est modifié, et le code appelant voit donc l'objet référencé modifié.

4.2.2 Les méthodes statiques

Les méthodes vues jusqu'à présent s'appliquent toujours a une référence sur un objet.

Les méthodes qualifiées de **static** sont celles qui n'ont pas besoin d'une instance pour être invoquées.

Comme toute méthode, une méthode de classe est membre d'une classe. Elle est invoquée en lui associant, non pas un objet mais la classe a laquelle elle appartient.

Par exemple, la méthode sqrt qui calcule la racine carrée appartient à la classe Math.

Pour l'invoquer on écrit :

```
Math.sqrt(x) ;
```

Une méthode statique, puisqu'elle ne s'applique pas sur un objet, ne peut pas accéder aux variables d'instances. De même, le mot clé **this** n'a pas de sens dans une méthode statique.

Exemple

```
class Date_Exemple{
```

```
private int mois;
private int jour;
private int annee;
private Date_Exemple suivant;
public static Date_Exemple tete=null;
public Date_Exemple (int j, int m, int a)
{ jour=j; mois=m; annee=a;
 suivant=tete;
 tete=this; }
public Date_Exemple getSuivant()
 {return suivant;}
public void afficher( )
 {System.out.println(jour+"/"+mois+"/"+annee); }
public static void afficher_Liste( )
 { for (Date_Exemple d= Date_Exemple.tete; d!=null; d=d.getSuivant( ))
 d.afficher();}}
class Test {
public static void main(String[] arg){
 Date_Exemple aid=new Date_Exemple (8,10,2012);
 Date_Exemple aujourdhui=new Date_Exemple (23,10,2012);
 Date_Exemple.afficher_Liste();
}}
```

V. Les Tableaux, les chaines de caractères et les entrées/sorties

5.1 Les tableaux

5.1.1 Definition

Un Tableau est un objet référencé, une variable de type tableau n'est pas un type élémentaire (c'est une référence sur un tableau). Un tableau est un objet possédant l'attribut **length** : c'est la taille du tableau.

Un tableau est déclaré comme suit :

```
type NomTableau [] ; Ou
```

```
type [] NomTableau ;
type NomTableau[][] ; // Pour le cas d'un tableau à deux dimensions
type NomTableau[][][]; // Pour le cas d'un tableau à trois dimensions
```

5.1.2 Allocation et initialisation d'un tableau de types primitifs

Une déclaration de tableau ne doit pas préciser de dimensions, en effet le nombre d'éléments du tableau sera déterminé quand l'objet tableau sera effectivement créé en utilisant le mot clé **new**.

La taille déterminée à la création du tableau est fixe, elle ne pourra plus être modifiée par la suite.

Exemple1:

```
int[] monTableau; // Déclaration
monTableau = new int[3]; // Dimensionnement
```

La création d'un tableau par new

- ✓ Alloue la mémoire en fonction du type de tableau et de la taille
- ✓ Initialise le contenu du tableau par les valeurs par défaut des types

```
int T1[]=new int[8]; // Allocation de 8 entiers pour T1
int T2[]={5,7,-12,8};
double [][][] T3= new double [3][20][12] ;
// Tableau référençant 3 tableaux de 20 autres de 12 éléments primitifs
```

Exemple2

```
class test1{
public static void main(String arg[]){
float[][] taux = new float[2][2];
taux[1][0]=0.24F;
taux[1][1]=0.33F;
System.out.println(taux[1].length); // 2
System.out.println(taux[1][1]); //0.33F}}
}}
```

Exemple3

```
int[][]T= {{1, 3}, {7, 5}};
```

5.1.3 Allocation d'un tableau de types objets

En java, un tableau d'objets ne contient que des références qui sont initialisée à **null** jusqu'à ce qu'on leur affecte des objets réels.

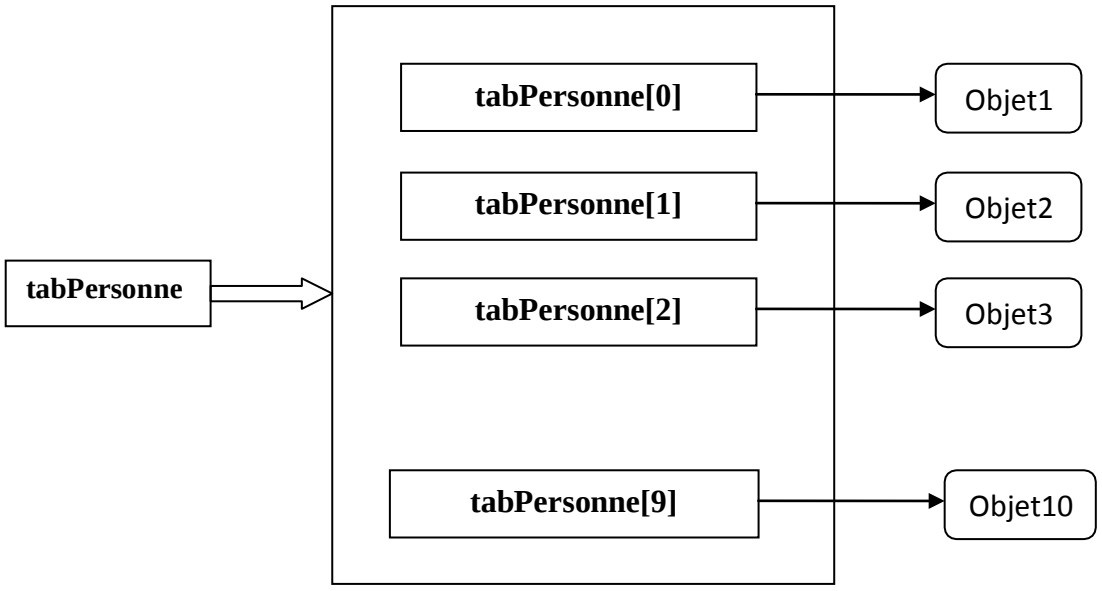
Pour un tableau d'objets il y a trois étapes à faire :

- Déclaration du tableau
- Création du tableau avec l'opérateur **new**
- Création des objets du tableau avec l'opérateur **new**

Exemple : On veut créer un tableau de 10 objets Personnes

```
class Test {
        public static void main(String[] args) {
                // Déclaration du tableau
                Personne [] tabPersonne;
                // Création du tableau
                tabPersonne= new Personne[10];
                // Création des objets du tableau
                for(int i=0;i<10;i++)
                        tabPersonne[i]=new Personne();
        }
}
```

La figure suivante illustre le tableau tabPersonne :



5.2 Les chaînes de caractères

5.2.1 La définition d'une chaîne de caractères

Les variables de type String sont des objets. Partout ou des constantes chaînes figurent entre guillemets, le compilateur Java génère un objet de type String avec le contenu spécifié.

Exemple1

```
String texte = "bonjour";
```

Les chaînes ne sont pas des tableaux : il faut utiliser les méthodes de la classe String d'un objet instancié pour effectuer des manipulations.

Il est impossible de modifier le contenu d'un objet String construit à partir d'une constante. Cependant, il est possible d'utiliser les méthodes qui renvoient une chaîne pour modifier le contenu de la chaîne

Exemple2

```
String texte = "Java Java Java";
texte = texte.replace('a','o');
System.out.println(texte);
```

→ Jovo Jovo Jovo

5.2.2 L'addition des chaînes

L'addition de chaînes Java admet l'opérateur + comme opérateur de concaténation de chaînes de caractères.

L'opérateur + permet de concaténer plusieurs chaînes. Il est possible d'utiliser l'opérateur +=

Exemple 1

```
String texte = "";
texte += "Hello";
texte += " World3";
System.out.println(texte);
```

→Hello World3

Cet opérateur sert aussi à concaténer des chaînes avec tous les types de bases. La variable ou constante est alors convertie en chaîne et ajoutée à la précédente. La condition préalable est d'avoir au moins une chaîne dans l'expression sinon le signe "+" est évalué comme opérateur mathématique.

Exemple2

```
System.out.println("La valeur de Pi est : " + Math.PI);
int duree = 121;
System.out.println("durée = " + duree);
```

→La valeur de Pi est : 3.141592653589793

→durée = 121

5.2.3 La comparaison de deux chaînes

Il faut utiliser la méthode **equals**()

Exemple

```
String texte1 = "texte 1";
String texte2 = "texte 2";
System.out.println( texte1.equals(texte2) ) ;
```

→false

5.2.4 La détermination de la longueur d'une chaîne

La méthode length() permet de déterminer la longueur d'une chaîne.

Exemple

```
String texte = "texte";
int longueur = texte.length();
System.out.println( longueur ) ;
```

→ 5

5.2.5 La modification de la casse d'une chaîne

Les méthodes Java toUpperCase() et toLowerCase() permettent respectivement d'obtenir une chaîne tout en majuscule ou tout en minuscule.

Exemple

```
String texte = "texte";
String textemaj = texte.toUpperCase();
System.out.println(textemaj) ;
```

→ TEXTE

5.3 Entrées/Sorties

5.3.1 Ecriture sur écran

La syntaxe de l'instruction d'écriture sur l'écran est la suivante :

```
System.out.println(expression);
```

Où expression est tout type de données qui puisse être converti en chaîne de caractères pour être affiché à l'écran. Dans l'exemple précédent, nous avons vu deux instructions d'écriture :

```
System.out.println(taux[1].length);
System.out.println(taux[1][1]);
```

→**2**

→0.33

5.3.2 Lecture de données tapées au clavier

Pour que Java puisse lire ce qu'on tape au clavier, il ya plusieurs méthodes entre autres l'utilisation de la classe **Scanner**.

Cet objet peut prendre différents paramètres, mais ici, nous n'en utiliserons qu'un. Celui qui correspond justement à l'entrée standard en Java **: System.in.**

Donc, avant de dire à Java de lire ce que nous allons taper au clavier, nous devrons instancier un objet Scanner :

```
Scanner s=new Scanner(System.in);
```

Pour pouvoir utiliser un objet Scanner, nous devons dire à Java où trouver cet objet :

import java.util.Scanner;

Exemple

```
import java.util.Scanner;
class exemple
{ public static void main(String [] args)
 {Scanner s=new Scanner(System.in);
  int choix=s.nextInt();
  String ch=s.next() ;}
}
```

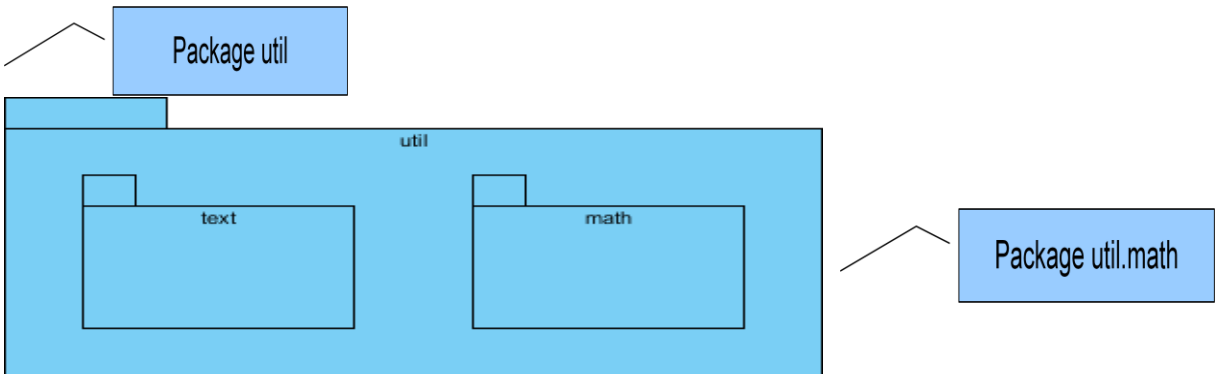
VI. Les Packages en java

6.1 Définition

Java permet de regrouper les classes en ensembles appelés packages afin de :

- Augmenter la modularité
- Réduire la complexité
- Simplifier la maintenance

Il faut noter qu'un package peut contenir d'autres packages



6.2 Exemples de packages java

Parmi les paquetages de Java vous avez :

java.applet Classes de base pour les applets

java.awt Classes d'interface graphique AWT

java.io Classes d entrées/sorties (flux, fichiers)

java.lang Classes de support du langage

java.math Classes permettant la gestion de grands nombres.

java.net Classes de support réseau (URL, sockets)

java.rmi Classes pour les méthodes invoquées à partir de machines virtuelles non locales.

java.security Classes et interfaces pour la gestion de la sécurité.

java.sql Classes pour l'utilisation de JDBC.

java.text Classes pour la manipulation de textes, de dates et de nombres dans plusieurs langages.

java.util Classes d'utilitaires (vecteurs, hashtable)

javax.swing Classes d'interface graphique

6.3 Utilisation des packages

6.3.1 Nomination

Chaque paquetage porte un nom. Ce nom est soit un simple identificateur ou une suite d'identificateurs séparés par des points. L'instruction permettant de nommer un paquetage doit figurer au début du fichier source (.java) comme suit:

---------- Fichier **Exemple.java** ----------

package nomtest; // 1ere instruction

class UneClasse {}

class UneAutreClasse {}

public class Exemple {// blabla suite du code ...}

---------- Fin du Fichier ----------

Ainsi par cette opération nous venons d'assigner toutes les classes du fichier Exemple.java au paquetage nomtest.

Si l'instruction package était absente du fichier, alors c'est le paquetage par défaut qui est pris en considération. Ainsi toutes les classes de ce fichier (Exemple.java) vont appartenir au paquetage par défaut.

6.3.2 Utilisation

Si vous instanciez un objet d'une classe donnée, le compilateur va chercher cette classe dans le fichier où elle a été appelée, nous dirons que cette recherche a eu lieu dans le paquetage par défaut même si ce dernier porte un nom.

Si votre appel fait référence à une classe appartenant à un autre paquetage, vous devez aider le compilateur à retrouver le chemin d'accès vers cette classe en procédant comme suit:

- En mentionnant le chemin complet d'accès à la classe : En citant les noms complets des paquetages nécessaires:

```
nomtest.UneautreClasse mm = new nomtest.UneautreClasse();
```

C'est une opération fastidieuse si vous avez affaire à une arborescence de paquetages ou bien à un nombre important de classes dans un même paquetage.

- La solution c'est soit importer les paquetages utiles : Vous devez utiliser pour cela l'instruction import doit être comme suit:

---------- **Fichier Test.java** ----------

```
import nomtest.UneautreClasse;
public class Test {
// balblabla ...
UneautreClasse mm = new UneautreClasse();
}
// blablabla ....
```

----------**Fin Fichier**----------

- Soit en important toutes les classes se trouvant dans le paquetage et ceci dans le cas où vous avez besoin d'utiliser d'autres classes se trouvant dans le même paquetage nomtest.

Pour cela vous devez écrire ceci :

```
import nomtest.*;
```

8.3.3 Remarques

- Le paquetage **java.lang** est importé automatiquement par le compilateur.
- import nomtest.*;

  Cette instruction ne va pas importer de manière récursive les classes se trouvant dans nomtest et dans ses sous paquetages. Elle va uniquement se contenter de faire un balayage d'un **SEUL** niveau. Elle va importer donc que les classes du paquetage nomtest.

  Exemple

```
import java.awt.*;
import java.awt.event.*;
```

  La première instruction importe toutes les classes se trouvant dans le paquetage awt. Elle ne va pas importer par exemple les classes se trouvant dans le sous paquetage event. Si vous avez besoin d'utiliser les classes de event, vous devez les importer aussi.
- Si deux paquetages contiennent le même nom de classe il y a un problème! Par exemple la classe List des paquetages java.awt et java.util ou bien la classe Date des paquetages java.util et java.sql etc.

  Pour corriger ce problème vous devez impérativement spécifier le nom exact du paquetage à importer (ou à utiliser).

  java.awt (ou) java.util ; java.util (ou) java.sql

  import java.awt.*; // ici on importe toutes les classes dans awt

  import java. util.*; // ici aussi

  import java.util.List; // ici on précise la classe List à utiliser

QCM3 : Encapsulation et constructeur

Répondez aux questions en cochant la ou les bonne(s) réponse(s).

1. Lequel parmi les types primitives suivants est considéré comme type réel en Java ? (2 bonnes réponses)
   - ☐ int
   - ☐ short
   - ☐ double
   - ☐ float
   - ☐ char

2. Lesquelles parmi ces prépositions sont vraies ? (2 bonnes réponses)
   - ☐ Java a 8 types primitifs
   - ☐ Java.lang.Object est un type primitif
   - ☐ Intger, Long et Double sont des types primitifs
   - ☐ int, long et double sont des types primitifs

3. lesquels parmi ces types sont primitifs ? (3 bonnes réponses)
   - ☐ int
   - ☐ float
   - ☐ long
   - ☐ Character
   - ☐ Double
   - ☐ String

4. Lesquels parmi les types suivants sont des entiers ? (3 bonnes réponses)
   - ☐ float
   - ☐ long
   - ☐ short
   - ☐ double
   - ☐ int

5. La protection des attributs en les rendant privés, est connu comme le concept de
   - ☐ Héritage des données
   - ☐ Implémentation des données
   - ☐ Encapsulation des données
   - ☐ Privatisation des données

6. Voici un constructeur de la classe Person, y-a-t'il un quelconque problème ?

```
public Person (String n)
{name = n ;
return;}
```

   - ☐ On ne peut pas utiliser return dans un constructeur puisqu'il ne doit rien renvoyer
   - ☐ Un constructeur doit renvoyer une instance, il faut donc mettre return new Person (n);
   - ☐ Aucun problème, return permet simplement de quitter la méthode
   - ☐ Il faut explicitement mettre void si l'on veut pouvoir faire return

7. Lorsque plusieurs méthodes ont le même nom (surcharge), comment la machine virtuelle Java sait-elle laquelle on veut invoquer ?
   - ☐ Elle les essaie toute une à une et prend la première qui fonctionne
   - ☐ Elle ne devine pas, il faut lui spécifier lorsqu'on compile le code
   - ☐ On indique le numéro de la méthode que l'on veut invoquer
   - ☐ Elle se base sur les types des paramètres

8. Soit la classe D définie comme suit:

```
class D {
   public int x;
   public D() {
          x = 3; }
   public D(int a) {
          this();
          x = x + a;}
   public D(int a, int b) {
          this(b);
          x = x - a;}}
```

Qu'affichera le code suivant?

```
D a=new D(5,6);
System.out.println(a.x);
```

- ☐ 1
- ☐ 2
- ☐ 3
- ☐ 4

9. On considère la classe:

```
class X{
 static int i=0; int j=2; /*...*/}
```

parmi ces prépositions, laquelle est vrai ?

- ☐ On peut ajouter à cette classe: void f(){i=i+j;}OK
- ☐ On peut ajouter à cette classe: static void g(){i=j+i;}
- ☐ On peut ajouter à cette classe: static void h(){i=X.j+i;}
- ☐ On peut ajouter à cette classe: static void i(){this.i=this.i+this.j;}

(Le this ne peut être utilisé dans un contexte statique et la méthode statique ne peut utiliser que des attributs statiques)

10. Le code:

```
class A{
static int i=0;int j=10;
static void g(){
System.out.println(j);}}
```

avec le morceau de code : (new A()).g();

- ☐ provoque une erreur à la compilation
- ☐ affiche 10 NON
- ☐ provoque une erreur à l'exécution

11. Le code:

```
class A{ static int i=0;int j=10;
void f(){System.out.println("i="+i);System.out.println(" j="+j);}}
```

avec le morceau de code: (new A()).f();

- ☐ provoque une erreur à la compilation
- ☐ affiche i=0 j=10
- ☐ provoque une erreur à l'exécution

12. Le code:

```
class A{ static int i=0;int j=10;
void f(){System.out.println("i="+i);
System.out.println(" j="+j);}
 static void g(){this.f();}}//cannot use this in a static context
```

avec le morceau de code: (new A()).g();

- ☐ provoque une erreur à la compilation
- ☐ affiche i=0 j=10
- ☐ provoque une erreur à l'exécution

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: cannot use this in a static context
```