

Programmation Orientée Objet Java

M.WANE

Ingénieur Informaticien

Analyste, Concepteur

Développeur d'Applications

Cours 2 : Programmation orientée Objet Java

- ❖ Objet et Classe
- ❖ Héritage et accessibilité
- ❖ Polymorphisme
- ❖ Collections

Méthode orientée objet

La méthode orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux par des relations.

Lorsque que l'on programme avec cette méthode, la première question que l'on se pose plus souvent est :

«**qu'est-ce que je manipule ?** », Au lieu de
«**qu'est-ce que je fait ?** ».

L'une des caractéristiques de cette méthode permet de concevoir de nouveaux objets à partir d'objets existants.

On peut donc réutiliser les objets dans plusieurs applications.

La réutilisation du code fut un argument déterminant pour vanter les avantages des langages à objets.

Pour faire la programmation orientée objet il faut maîtriser les fondamentaux de l'orienté objet à savoir:

- Objet et classe**
- Héritage**
- Encapsulation (Accessibilité)**
- Polymorphisme**

Objet

Un objet est une structure informatique définie par un état et un comportement.

Objet=état + comportement

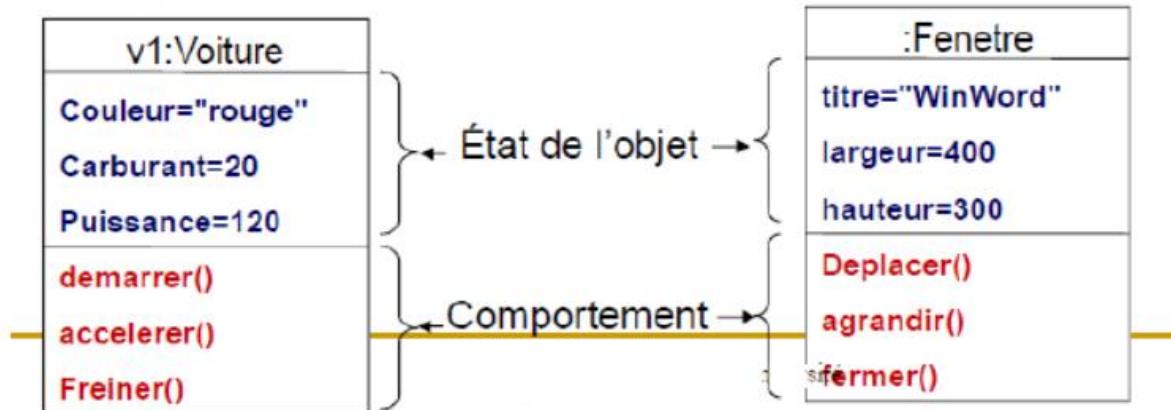
L'état regroupe les valeurs instantanées de tous les attributs de l'objet.

Le comportement regroupe toutes les compétences et décrit les actions et les réactions de l'objet. Autrement dit le comportement est défini par les opérations que l'objet peut effectuer.

L'état d'un objet peut changer dans le temps.

Généralement, c'est le comportement qui modifie l'état de l'objet

Exemple



Identité d'un objet

En plus de son état, un objet possède une identité qui caractérise son existence propre.

Cette identité s'appelle également référence ou handle de l'objet.

En terme informatique de bas niveau, l'identité d'un objet représente son adresse mémoire.

Deux objets ne peuvent pas avoir la même identité: c'est-à-dire que deux objets ne peuvent pas avoir le même emplacement mémoire.

Classes

Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelée classe.

La classe décrit le domaine de définition d'un ensemble d'objets.

Chaque objet appartient à une classe

Les généralités sont contenues dans les classes et les particularités dans les objets.

Les objets informatiques sont construits à partir de leur classe par un processus qui s'appelle l'instanciation.

Tout objet est une instance d'une classe.

Caractéristique d'une classe

Une classe est définie par:

Les attributs

Les méthodes

Les attributs permettent de décrire l'état de des objets de cette classe.

Chaque attribut est défini par:

Son nom

Son type

Éventuellement sa valeur initiale

Les méthodes permettent de décrire le comportement des objets de cette classe.

Une méthode représente une procédure ou une fonction qui permet d'exécuter un certain nombre d'instructions.

Parmi les méthodes d'une classe, existe deux méthodes particulières:

Une méthode qui est appelée au moment de la création d'un objet de cette classe. Cette méthode est appelée **CONSTRUCTEUR**

Une méthode qui est appelée au moment de la destruction d'un objet.

Cette méthode s'appelle le **DESTRUCTEUR**

Représentation UML d'une classe

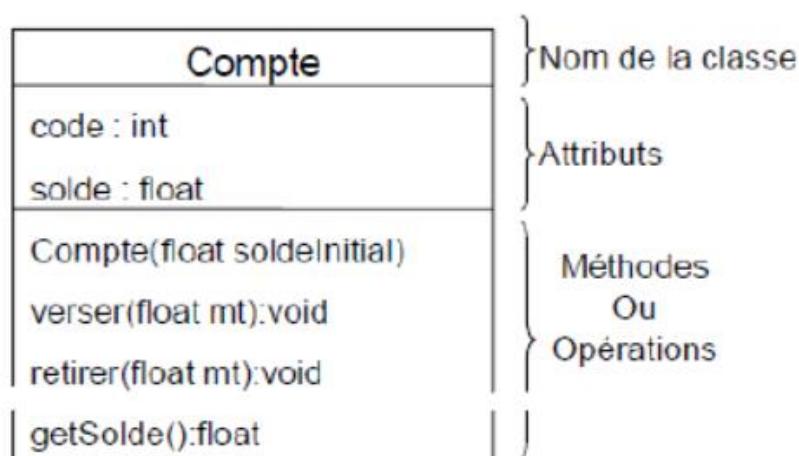
Une classe est représentée par un rectangle à 3 compartiments:

Un compartiment qui contient le nom de la classe

Un compartiment qui contient la déclaration des attributs

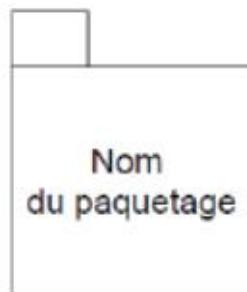
Un compartiment qui contient les méthodes

Exemples:



Les classes sont stockées dans des packages

- Les packages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de la modélisation
- Chaque package est représenté graphiquement par un dossier
- Les packages divisent et organisent les modèles de la même manière que les dossier organisent le système de fichier



Accessibilité aux membres d'une classe

Dans java, il existe 4 **niveaux de protection** :

private (-) : Un membre privé d'une classe n'est accessible qu'à l'intérieur de cette classe.

protected (#) : un membre protégé d'une classe est accessible à :

- L'intérieur de cette classe

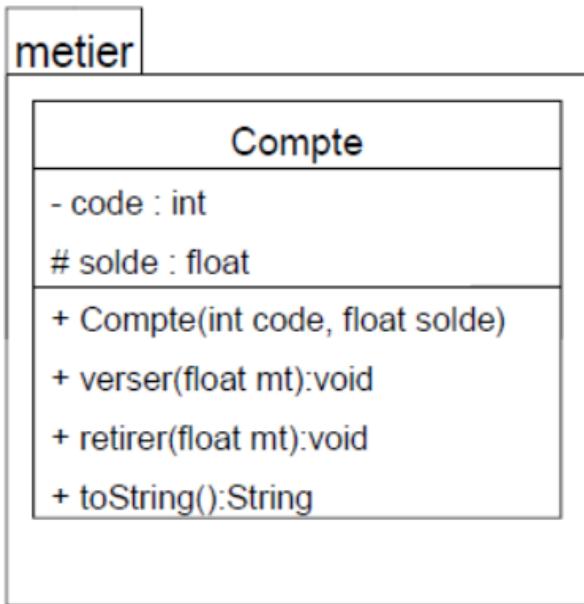
- Aux classes dérivées de cette classe.

- Aux classes du même package.

public (+) : accès à partir de toute entité interne ou externe à la classe

Autorisation par défaut : dans java, en l'absence des trois autorisations précédentes, l'autorisation par défaut est **package**. Cette autorisation indique que uniquement les classes du même package ont l'autorisation d'accès.

Exemple d'implémentation d'une classe avec Java



```
package metier;
public class Compte {
    // Attributs
    private int code;
    protected float solde;
    // Constructeur
    public Compte(int c, float s) {
        code=c;
        solde=s;
    }
    // Méthode pour verser un montant
    public void verser(float mt) {
        solde+=mt;
    }
    // Méthode pour retirer un montant
    public void retirer(float mt) {
        solde-=mt;
    }
    // Une méthode qui retourne l'état du compte
    public String toString() {
        return(" Code="+code+" Solde="+solde);
    }
}
```

Création des objets dans java

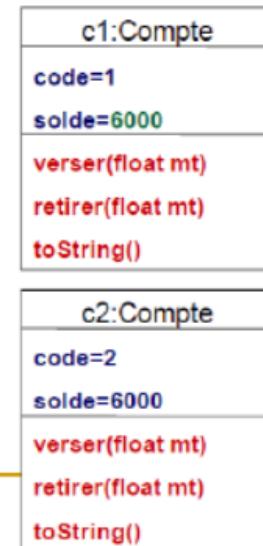
Dans java, pour créer un objet d'une classe , On utilise la commande **new suivie du constructeur de la classe.**

La commande new Crée un objet dans l'espace mémoire et retourne l'adresse mémoire de celui-ci.

Cette adresse mémoire devrait être affectée à une variable qui représente l'identité de l'objet. Cette référence est appelée handle.

```
package test;  
import metier.Compte;  
public class Application {  
    public static void main(String[] args) {  
        Compte c1=new Compte(1,5000);  
        Compte c2=new Compte(2,6000);  
        c1.verser(3000);  
        c1.retirer(2000);  
        System.out.println(c1.toString());  
    }  
}
```

Code=1 Solde= 6000



Constructeur par défaut

Quand on ne définit aucun constructeur pour une classe, le compilateur crée le constructeur par défaut.

Le constructeur par défaut n'a aucun paramètre et ne fait aucune initialisation.

Exemple de classe :

```
public class Personne {  
    // Les Attributs  
    private int code;  
    private String nom;  
    // Les Méthodes  
    public void setNom(String n){  
        this.nom=n;  
    }  
    public String getNom(){  
        return nom;  
    }  
}
```

Instanciation en utilisant le constructeur par défaut :

```
Personne p=new Personne();  
p.setNom("AZER");  
System.out.println(p.getNom());
```

Surcharge

Dans une classe, on peut définir plusieurs constructeurs. Chacun ayant une signature différente (paramètres différents)

On dit que le constructeur est surchargé

On peut également surcharger une méthode. Cela peut dire qu'on peut définir, dans la même classe plusieurs méthodes qui ont le même nom et des signatures différentes;

La signature d'une méthode désigne la liste des arguments avec leurs types.

Dans la classe CompteSimple, par exemple, on peut ajouter un autre constructeur sans paramètre

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot **this()** avec des paramètres éventuels

Surcharge de constructeurs

```
public class CompteSimple extends Compte {  
    private float découvert;  
    //Premier constructeur  
    public CompteSimple(float découvert) {  
        super();  
        this.découvert=découvert;  
    }  
    //Deuxième constructeur  
    public CompteSimple() {  
        this(0);  
    }  
}
```

On peut créer une instance de la classe CompteSimple en faisant appel à l'un des deux constructeur :

```
CompteSimple cs1=new CompteSimple(5000);
```

```
CompteSimple cs2=new CompteSimple();
```

Getters et Setters

Les attributs privés d'une classe ne sont accessibles qu'à l'intérieur de la classe. Pour donner la possibilité à d'autres classes d'accéder aux membres privés, il faut définir dans la classes des méthodes publiques qui permettent de :

- Lire les variables privées. Ce genre de méthodes s'appelle les **accesseurs** ou **Getters**
- Modifier les variables privés. Ce genre de méthodes s'appelle les **mutateurs** ou **Setters**

Les getters sont des méthodes qui commencent toujours par le mot **get** et finissent par le nom de l'attribut en écrivant en majuscule la lettre qui vient juste après le get.

Les getters retournent toujours le même type que l'attribut correspondant.

Par exemple, dans la classe **CompteSimple**, nous avons défini un attribut privé :

private String nom;

Le getter de cette variable est :

```
public String getNom(){  
    return nom;  
}
```

Les setters sont des méthodes qui commencent toujours par le mot **set** et finissent par le nom de l'attribut en écrivant en majuscule la lettre qui vient juste après le set.

Les setters sont toujours de type **void** et reçoivent un **paramètre** qui est de **meme type que la variable**:

Exemple:

```
public void setNom( String n ){  
    this.nom=n;  
}
```

Encapsulation

```
public class Application {  
    public static void main(String[] args) {  
        Personne p=new Personne();  
        p.setNom("AZER");  
        System.out.println(p.getNom());  
    }  
}
```

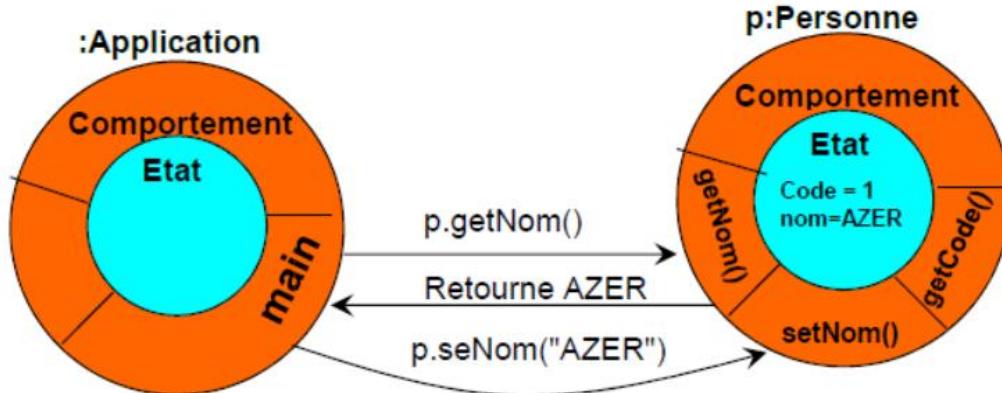
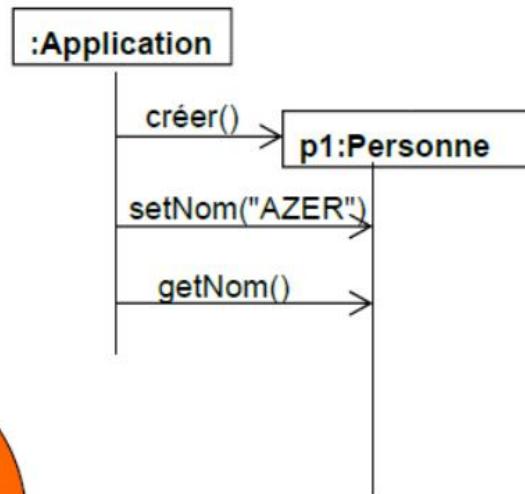


Diagramme de séquence :



- Généralement, l'état d'un objet est privé ou protégé et son comportement est public
- Quand l'état de l'objet est privé Seules les méthodes de ses qui ont le droit d'y accéder
- Quand l'état de l'objet est protégé, les méthodes des classes dérivées et les classes appartenant au même package peuvent également y accéder

Membres statiques d'une classe.

Dans l'exemple de la classe Compte, chaque objet Compte possède ses propres variables code et solde. Les variables code et solde sont appelées variables d'instances.

Les objets d'une même classe peuvent partager des mêmes variables qui sont stockées au niveau de la classe. Ce genre de variables, s'appellent les variables statiques ou variables de classes.

Un attribut statique d'une classe est un attribut qui appartient à la classe et partagé par tous les objets de cette classe.

Comme une méthode peut être déclarée statique, ce qui signifie qu'elle appartient à la classe et partagée par toutes les instances de cette classe.

Dans la notation UML, les membres statiques d'une classe sont soulignés.

Exemple:

- Supposant nous voulions ajouter à la classe Compte une variable qui permet de stocker le nombre de comptes créés.
- Comme la valeur de variable nbComptes est la même pour tous les objets, celle-ci sera déclarée statique. Si non, elle sera dupliquée dans chaque nouveau objet créé.
- La valeur de nbComptes est au départ initialisée à 0, et pendant la création d'une nouvelle instance (au niveau du constructeur), nbComptes est incrémentée et on profite de la valeur de nbComptes pour initialiser le code du compte.

Compte	
- code : int	
# solde : float	
- nbComptes:int	
+ Compte(float solde)	
+ verser(float mt):void	
+ retirer(float mt):void	
+ toString():String	
+ getNbComptes():int	

```
package metier;
public class Compte {
    // Variables d'instances
    private int code;
    private float solde;
    // Variable de classe ou statique
    private static int nbComptes;
    public Compte(float solde) {
        this.code=++nbComptes;
        this.solde=solde;
    }
    // Méthode pour verser un montant
    public void verser(float mt) {
        solde+=mt;
    }
    // Méthode pour retirer un montant
    public void retirer(float mt) {
        solde-=mt;
    }
    // retourne l'état du compte
    public String toString() {
        return(" Code="+code+" Solde="+solde);
    }
    // retourne la valeur de nbComptes
    public static int getNbComptes() {
        return(nbComptes);
    }
}
```

Application de test

```
package test;  
import metier.Compte;  
public class Application {  
    public static void main(String[] args) {  
        Compte c1=new Compte(5000);  
        Compte c2=new Compte(6000);  
        c1.verser(3000);  
        c1.retirer(2000);  
        System.out.println(c1.toString());  
        System.out.println(Compte.nbComptes);  
        System.out.println(c1.nbComptes);  
    }  
}
```

Classe Compte
<u>nbCompte=2</u>
<u>getNbComptes()</u>

c1:Compte	c2:Compte
code=1	code=2
solde=6000	solde=6000
verser(float mt)	verser(float mt)
retirer(float mt)	retirer(float mt)
toString()	toString()

Code=1 Solde= 6000
2
2

Destruction des objets : Garbage Collector

Dans certains langages de programmation, le programmeur doit s'occuper lui-même de détruire les objets inutilisables.

Java détruit automatiquement tous les objets inutilisables en utilisant ce qu'on appelle le **garbage collector (ramasseur d'ordures)**. Qui s'exécute automatiquement dès que la mémoire disponible est inférieure à un certain seuil.

Tous les objets qui ne sont pas retenus par des handles seront détruits.

Ce phénomène ralenti parfois le fonctionnement de java.

Pour signaler au garbage collector que vous voulez détruire un objet d'une classe, vous pouvez faire appel à la méthode `finalize()` redéfinie dans la classe.

Héritage et accessibilité

Héritage

Dans la programmation orientée objet, l'héritage offre un moyen très efficace qui permet la réutilisation du code.

En effet une classe peut hériter d'une autre classe des attributs et des méthodes.

L'héritage, quand il peut être exploité, fait gagner beaucoup de temps en termes de développement et en terme de maintenance des applications.

La réutilisation du code fut un argument déterminant pour venter les méthodes orientées objets.

Exemple de problème

Supposons que nous souhaitions créer une application qui permet de manipuler différents types de **comptes bancaires**: les **compte simple**, les **comptes épargnes** et les **comptes payants**.

Tous les types de comptes sont caractériser par:

Un code et un solde

Lors de la création d'un compte, son code qui est défini automatiquement en fonction du nombre de comptes créés;

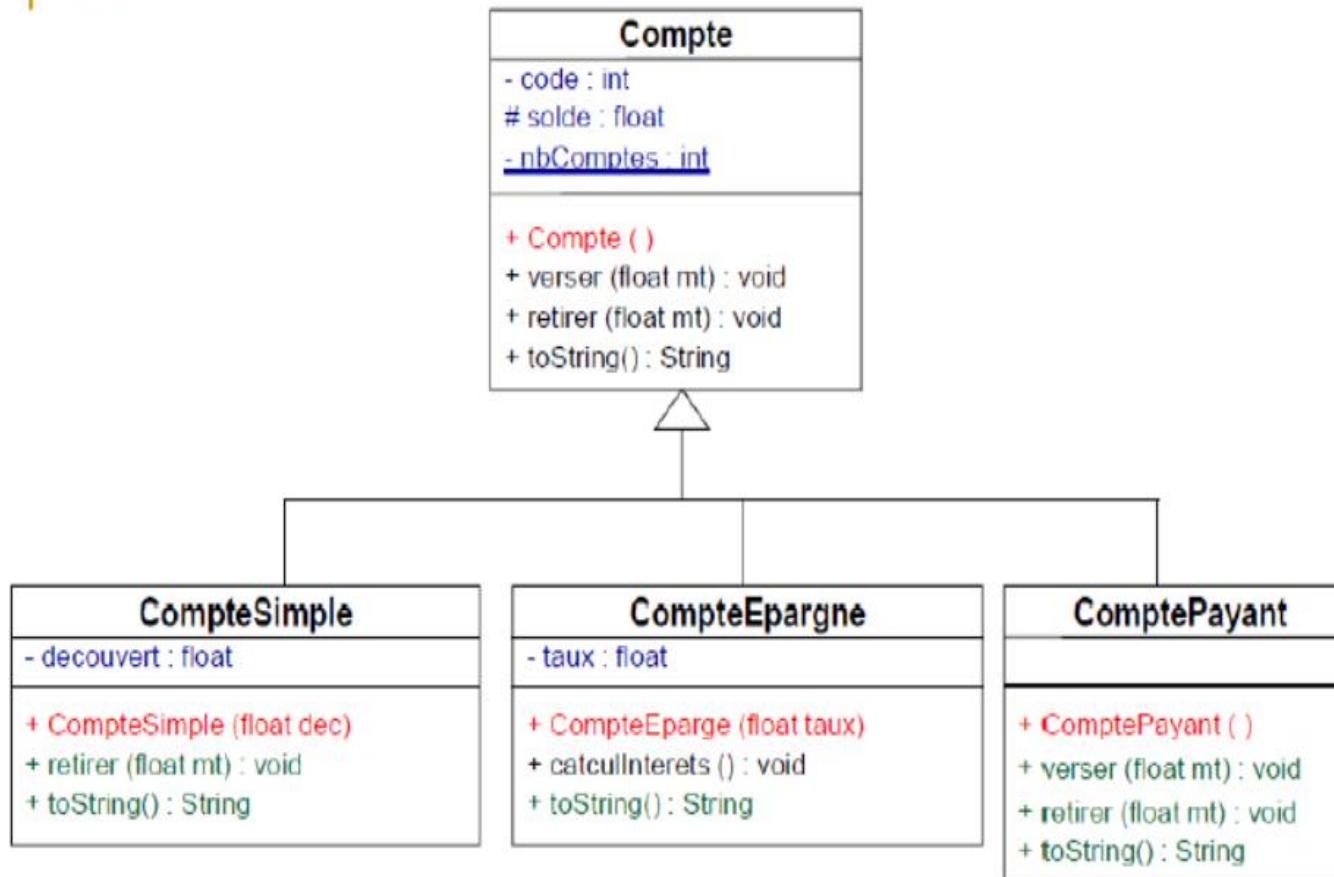
Un compte peut subir les opérations de **versement et de retrait**. Pour ces deux opérations, il faut connaître le montant de l'opération.

Pour consulter un compte on peut faire appel à sa méthode **toString()**

- ❖ Un **compte simple** est un compte qui possède un découvert. Ce qui signifie que ce compte peut être débiteur jusqu'à la valeur du découvert.
- ❖ Un **compte Epargne** est un compte bancaire qui possède en plus un champ «tauxInterêt» et une méthode `calculIntérêt()` qui permet de mettre à jour le solde en tenant compte des intérêts.

Un **ComptePayant** est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 5 % du montant de l'opération.

Diagramme de classes



Implémentation java de la classe Compte

```
public class Compte {  
    private int code; protected  
    float solde; private static  
    int nbComptes;  
  
    public Compte( ){  
        ++nbComptes;  
        code=nbComptes;  
        this.solde=0;  
    }  
    public void verser(float mt){  
        solde+=mt;  
    }  
    public void retirer(float mt){  
        if(mt<solde) solde-=mt;  
    }  
    public String toString(){  
        return("Code="+code+" Solde="+solde); }  
}
```

Héritage : extends

La classe CompteSimple est une classe qui hérite de la classe Compte.

Pour désigner l'héritage dans java, on utilise le mot **extends**

```
public class CompteSimple extends Compte {
```

```
}
```

La classe CompteSimple hérite de la classe CompteBancaire tous ses membres sauf le constructeur.

Dans java une classe hérite toujours d'une seule classe. Si une classe n'hérite pas explicitement d'une autre classe, elle hérite implicitement de la classe Object.

La classe Compte hérite de la classe Object.

La classe CompteSimple hérite directement de la classe Compte et indirectement de la classe Object.

Définir les constructeur de la classe dérivée

Le constructeur de la classe dérivée peut faire appel au constructeur de la classe parente en utilisant le mot **super()** suivi de ses paramètres.

```
public class CompteSimple extends Compte {  
    private float découvert;  
    //constructeur  
    public CompteSimple(float découvert) {  
        super();  
        this.découvert=découvert;  
    }  
}
```

Redéfinition des méthodes

Quand une classe hérite d'une autre classe, elle peut redéfinir les méthodes héritées.

Dans notre cas la classe CompteSimple hérite de la classe Compte la méthode retirer().

Nous avons besoin de redéfinir cette méthode pour prendre en considération la valeur du découvert.

```
public class CompteSimple extends Compte {  
    private float découvert;  
  
    // constructeur public  
    CompteSimple(float découvert) {  
        super(); this.découvert=découvert;  
    }  
    // Redéfinition de la méthode retirer  
    public void retirer(float mt) { if(mt-  
        découvert<=solde)  
            solde-=mt;  
    }  
}
```

Redéfinition des méthodes

Dans la méthode redéfinie de la nouvelle classe dérivée, on peut faire appel à la méthode de la classe parente en utilisant le mot **super** suivi d'un point et du nom de la méthode

Dans cette nouvelle classe dérivée, nous allons redéfinir également la méthode `toString()`.

```
public class CompteSimple extends Compte {
    private float découvert;

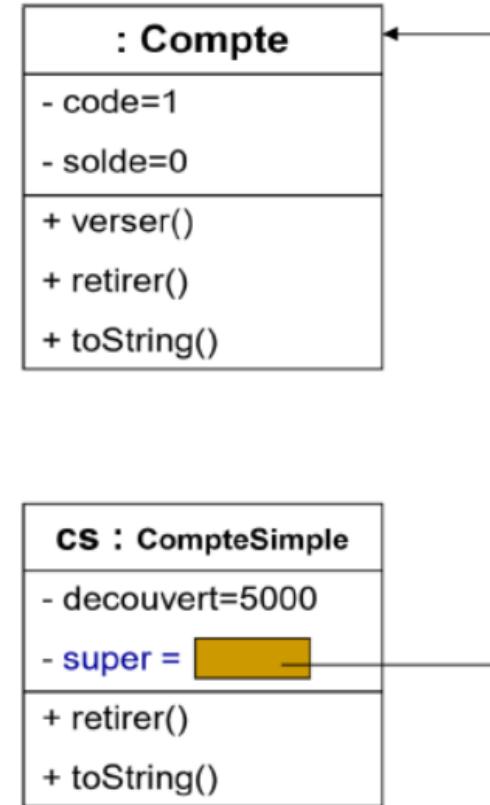
    // constructeur
    // Redéfinition de la méthode retirer
    public void retirer(float mt) {
        if(mt+découvert>solde)
            solde-=mt;
    }
    // Redéfinition de la méthode toString
    public String toString() { return("Compte
Simple
"+super.toString()+"Découvert="+découvert)
;
}
}
```

Héritage à la loupe : Instanciation

- Quand on crée une instance d'une classe, la classe parente est automatiquement instanciée et l'objet de la classe parente est associé à l'objet créé à travers la référence « **super** » injectée par le compilateur.

```
CompteSimple cs=new CompteSimple(5000);
```

- Lors de l'instanciation, l'héritage entre les classes est traduit par une composition entre un objet de la classe instanciée et d'un objet de la classe parente qui est créé implicitement.



Accessibilité

Les trois critères permettant d'utiliser une classe sont *Qui*, *Quoi*, *Où*. Il faut donc :

Que l'utilisateur soit autorisé (*Qui*).

Que le type d'utilisation souhaité soit autorisé (*Quoi*).

Que l'adresse de la classe soit connue (*Où*).

Pour utiliser donc une classe, il faut :

Connaitre le package où se trouve la classe (*Où*)

Importer la classe en spécifiant son package.

Qu'est ce qu'on peut faire avec cette classe:

Est-ce qu'on a le droit de l'instancier

Est-ce qu'on a le droit d'exploiter les membres de ses instances

Est-ce qu'on a le droit d'hériter de cette classe.

Est-ce qu'elle contient des membres statiques

Connaitre qui a le droit d'accéder aux membres de cette instance.

Les packages (Où)

Nous avons souvent utilisé la classe System pour afficher un message : **System.out.println()** ,

En consultant la documentation de java, nous allons constater que le chemin d'accès complet à la classe **System est java.lang.System.**

La classe System étant stockée dans le sous dossier lang du dossier java.

java.lang.System est le chemin d'accès qui présente la particularité d'utiliser un point « . » comme séparateur.

Java.lang qui contient la classe System est appelé « **package** »

Notion de package:

- ❖ Java dispose d'un mécanisme pour la recherche des classes.
- ❖ Au moment de l'exécution, La JVM recherche les classes en priorité :
 - Dans le répertoire courant, c'est-à-dire celui où se trouve la classe appelante, si la variable d'environnement **CLASSPATH** n'est pas définie ;
 - Dans les chemins spécifiés par la variable d'environnement **CLASSPATH** si celle-ci est définie.

L'instruction package:

Si vous souhaitez qu'une classe que vous avez créée appartienne à un package particulier, vous devez le spécifier explicitement au moyen de l'instruction **package**, suivie du nom du package.

Cette instruction doit être la première du fichier.

Elle concerne toutes les classes définies dans ce fichier.

L'instruction import

Pour utiliser une classe, il faut

Soit écrire le nom de la classe précédée par son package.

Soit importer cette classe en la déclarant dans la clause import. Et dans ce cas là, seul le nom de la classe suffit pour l'utiliser.

Les packages (Où)

Les fichiers .jar

- Les fichiers .jar sont des fichiers compressés comme les fichiers .zip selon un algorithme particulier devenu un standard.
- Ils sont parfois appelés *fichiers d'archives* ou, plus simplement, *archives*. Ces fichiers sont produits par des outils de compression tels que Pkzip (sous DOS) ou Winzip (sous Windows), ou encore par **jar.exe**.
- Les fichiers .jar peuvent contenir une multitude de fichiers compressés avec l'indication de leur chemin d'accès.
- Les packages standard de Java sont organisés de cette manière, dans un fichier nommé **rt.jar** placé dans le sous-répertoire **lib** du répertoire où est installé le JDK.
- Dans le cas d'une installation standard de Java 6 sur le disque C:, le chemin d'accès complet à la classe **System** est donc : c:\jdk1.6\jre\lib\rt.jar\java\lang\System

Création de vos propres fichiers .jar ou .zip

Vous pouvez utiliser le programme jar.exe du jdk pour créer les fichiers .jar

Syntaxe : jar [options] nomarchive.jar fichiers

Exemple qui permet d'archive le contenu du dossier a :

```
C:\AJ2\TP_ACC\bin> jar cf archive.jar a
```

Ce qui peut être fait(Quoi)

Nous avons maintenant fait le tour de la question *Où* ?

Pour qu'une classe puisse être utilisée (directement ou par l'intermédiaire d'un de ses membres), il faut non seulement être capable de la trouver, mais aussi qu'elle soit adaptée à l'usage que l'on veut en faire.

Une classe peut servir à plusieurs choses :

Créer des objets, en étant **instanciée**.

Créer de nouvelles classes, en étant **étendue**.

On peut utiliser directement ses membres statiques (sans qu'elle soit instanciée.)

On peut utiliser les membres de ses instances.

Les différents modificateurs qui permettent d'apporter des restrictions à l'utilisation d'une classe sont: **abstract**, **final**, **static**, **synchronized** et **native**

Classe abstraite

Une classe abstraite est une classe qui ne peut pas être instanciée. La classe Compte de notre modèle peut être déclarée abstract pour indiquer au compilateur que cette classe ne peut pas être instancié. Une classe abstraite est généralement créée pour en faire dériver de nouvelle classe par héritage.

```
public abstract class Compte {  
    private int code; protected  
    float solde; private static  
    int nbComptes;  
    // Constructeurs  
    // Méthodes  
}
```

Les méthodes abstraites

Une méthode abstraite peut être déclarée à l'intérieur d'une classe abstraite.

Une méthode abstraite est une méthode qui n'a pas de définition.

Une méthode abstraite est une méthode qui doit être redéfinie dans les classes dérivées.

Exemple :

On peut ajouter à la classe Compte une méthode abstraite nommée afficher () pour indiquer que tous les comptes doivent redéfinir cette méthode.

Les méthodes abstraites

```
public abstract class Compte {  
    // Membres  
    ...  
    // Méthode abstraite  
    public abstract void afficher();  
}
```

```
public class CompteSimple extends Compte {  
    // Membres  
    ...  
    public void afficher(){  
        System.out.println("Solde="+solde+ "  
        Découvert="+decouvert);  
    }  
}
```

Interfaces

Une interface est une sorte de classe abstraite qui ne contient que des méthodes abstraites.
Dans java une classe hérite d'une seule classe et peut hériter en même temps de plusieurs interfaces.

On dit qu'une classe implémente une ou plusieurs interfaces.

Une interface peut hériter de plusieurs interfaces. Exemple d'interface:

```
public interface Solvable {  
    public void solver();  
    public double getSoile();  
}
```

Pour indiquer que la classe CompteSimple implémente cette interface on peut écrire:

```
public class CompteSimple extends Compte implements Solvable {  
    private float decouvert; public void afficher() {  
        System.out.println("Solde="+solde+" Découvert="+decouvert);  
    }  
    public double getSoile() {  
        return solde;  
    }  
    public void solver() {  
        this.solde=0;  
    } }
```

Exercice 1 :

Soit un service caractérisé par son code et son libelle.

Soit un employé caractérisé par leur Matricule, Prénom, Nom, Sexe, Nombre d'enfants, Ancienneté, Salaire de base, Statut (Cadre, Maîtrise, Agent d'exécution), Prime spéciale, Prime, IPRES et son service.

- Une Indemnité représentant 5% du salaire de base est accordée à chaque employé.
- Salaire imposable = Salaire de base + Indemnités + Prime spéciale
- Caisse de sécurité sociale est égale 3% du salaire imposable
- Pour le calcul de l'impôt sur le revenu, les règles sont les suivantes :
 - Si l'ancienneté > 15 et Nombre d'enfants ≥ 3 alors 5% du salaire imposable
 - Sinon 8% du salaire imposable
- Tout agent verse pour le régime général IPRES 8,4% du salaire imposable et 3,6% du salaire imposable pour le régime cadre s'il s'agit d'un cadre
- Retenues = Impôt + IPRES+CSS
- Net à percevoir = Salaire imposable-Retenues

Ecrire un programme qui contient les modules suivants () :

1. Créer le Diagramme de Classe
2. Créer une classe Employé
3. Créer une classe Service
4. Créer une Application Test qui permet d'ajouter un Service et un Employé puis de les Affiche ;

Exercice 2

Une cercle est défini par :

Un point qui représente son centre : centre(x,y) et un rayon.

On peut créer un cercle de deux manières :

Soit en précisant son centre et un point du cercle.

Soit en précisant son centre et son rayon

Les opérations que l'on souhaite exécuter sur un cercle sont :

getPerimetre() : retourne le périmètre du cercle getSurface() :

retourne la surface du cercle.

appartient(Point p) : retourne si le point p appartient ou non à l'intérieur du cercle.

toString() : retourne une chaîne de caractères de type CERCLE(x,y,R)

1. Etablir le diagramme de classes
2. Créer les classe Point définie par:
 - Les attributs x et y de type int
 - Un constructeur qui initialise les valeurs de x et y .
 - Une méthode `toString()`.
3. Créer la classe Cercle
4. Créer une application qui permet de :
 - a. Créer un cercle défini par le centre $c(100,100)$ et un point $p(200,200)$
 - b. Créer un cercle défini par le centre $c(130,100)$ et de rayon $r=40$
 - c. Afficher le périmètre et le rayon des deux cercles.
 - d. Afficher si le point $p(120,100)$ appartient à l'intersection des deux cercles ou non.

