# Conception Orientée Objet

Par Y.HMOUR (y.himour@univ-dbkm.dz)

Pour M1 AII

# I.Introduction à l’approche objet

# 1.Méthodes orientées objet

- Dans une approche Orientée Objet (OO), le logiciel est considéré comme une collection d'objets dissociés définis par des propriétés.
- Une propriété est soit un attribut de l'objet ou une opération sur l'objet.
- Un objet comprend donc à la fois une structure de données et une collection d'opérations (son comportement).

# 2.Ensemble des concepts dans l'COO

- Abstraction,
- Modularité,
- Encapsulation,
- Polymorphisme.

# Avantages

- Reflète plus finement les objets du monde réel
- Du code : facile à maintenir
- Plus stable: Isolation des changements
- Réutilisation des composantes
- Faciliter le prototypage

# 3.Exemple: système de gestion d'un lycée

**Objets**

- Personnes
  - Etudiant, enseignant, principal, secrétaire
- Diplôme
  - Année, matière, parcours
- Notes
  - Coefficients

**Fonctions**

- Calculer la moyenne
- Calculer les taux d'encadrement
- Calculer le nombre de redoublants
- Calculer le taux de réussite au baccalauréat

# 4.Objectifs des technologies à objets

- **Utiliser le langage du domaine** : Modèle et vocabulaire métier
- **Construire des modèles faciles à**: Etendre, modifier, valider, vérifier
- **Faciliter l'implantation** : Génération facilitée vers les langages à objets
- **Nécessite une méthode et des outils** :example UML

# II.Notions de base

# 1.Les structures de contrôle

Les structures de contrôle permettent d'exécuter un **bloc d'instructions** soit plusieurs fois (instructions itératives) soit selon la valeur d'une expression (instructions conditionnelles ou de choix multiple). Dans tous ces cas, un bloc d'instruction est :

- soit une instruction unique ;
- soit une suite d'instructions commençant par une accolade ouvrante "{" et se terminant par une accolade fermante "}".

# 1.1 Instructions conditionnelles

Syntaxe 1:
if (<condition>)

<bloc1>

[else if (<condition>) <bloc2>]

[else <bloc2>]

Syntaxe 2:
<condition>?<instruction1>:<instruction2>
<condition> doit renvoyer une valeur booléenne. Si celle-ci est vraie c'est <bloc1> (resp.<instruction1>) qui est exécuté sinon <bloc2> (resp. <instruction2>) est exécuté.

```
if (a > b){
 x = a ;
}
else if (a < b) {
x=b;
}
else {
x=30;
}
//
x=(a>b)? a:b;
```

# 1.1 Instructions conditionnelles

L'instruction switch

Syntaxe :
switch( variable) {
case valeur1: instr1;break;
case valeur2: instr2;break;
case valeurN: instrN;break;
default: instr;break;

```
import java.util.Scanner;
public class Test {
public static void main(String[] args) {
System.out.print("Donner un nombre:");
Scanner clavier=newScanner(System.in);
int nb=clavier.nextInt();
switch(nb){
case 1 :
System.out.println("Lundi");break;
case 2 :
System.out.println("Mardi");break;
default:
System.out.println("Autrement");break;
}}}
```

# 1.2 Instructions itératives

Les instructions itératives permettent d'exécuter plusieurs fois un bloc d'instructions, et ce, jusqu'à ce qu'une condition donnée soit fausse.
Les trois types d'instruction itératives sont les suivantes :

- Tant que … Faire
- Faire … Tant que
- Pour … Faire

# 1.2 Instructions itératives

**TantQue...Faire...** L'exécution de cette instruction suit les étapes suivantes :
1-la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on passe à l'étape 2, sinon on passe à l'étape 4;
2-le bloc est exécuté ;
3-retour à l'étape 1;
4-la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

```
while (a != b){
 a++;
}
```

# 1.2 Instructions itératives

**Faire...TantQue...** L'exécution de cette instruction suit les étapes suivantes :
1. le bloc est exécuté ;
2. la condition (qui doit renvoyer une valeur booléenne) est évaluée. Si celle-ci est vraie on retourne à l'étape 1, sinon on passe à l'étape 3 ;
3. la boucle est terminée et le programme continue son exécution en interprétant les instruction suivant le bloc.

Syntaxe :
do <bloc> while (<condition>);

```
do a++
while (a != b);
```

# 1.2 Instructions itératives

**Pour...Faire** Cette boucle est constituée de trois parties :

- (i) une initialisation (la déclaration de variables locales à la boucle est autorisée dans cette partie) ;
- (ii) une condition d'arrêt ;
- (iii) un ensemble d'instructions à exécuter après chaque itération (chacune de ces instructions est séparée par une virgule).

Syntaxe:
for (<init>;<condition>;<instr_post_itération>)
<bloc>

```
for (int i = 0, j = 49 ;
(i < 25) && (j >= 25);
i++, j--)
{
if (tab[i] > tab[j]) {
int tampon = tab[j];
}
}
```

# 1.3 Instructions break et continue

L'instruction break est utilisée pour sortir immédiatement d'un bloc d'instructions (sans traiter les instructions restantes dans ce bloc). Dans le cas d'une boucle on peut également utiliser l'instruction continue avec la différence suivante :

**break :** l'exécution se poursuit après la boucle (comme si la condition d'arrêt devenait vraie) ;
**continue :** l'exécution du bloc est arrêtée mais pas celle de la boucle. Une nouvelle itération du bloc commence si la condition d'arrêt est toujours vraie .

```
for (int i = 0, j = 0 ; i < 100 ; i++)
{
if (i > tab.length) {
        break ;}
if (tab[i] == null) {
        continue ;}
tab2[j] = tab[i];
j++;
}
```

# 2. Les fonctions

Fonctions ne fournissant aucun résultat sont des fonction avec le mot clé **void** dans leurs en-têtes.

Fonctions fournissant un résultat sont des fonction avec le type du résultat dans leurs en-têtes

```
void Hello() {
System.out.println("Hello World");
}
//
double somme(double r1,double r2)
{
double som = 0.0;
som = r1 + r2;
return som;
}
```

# 3 Les Tableaux

Plusieurs éléments de même type peuvent être regroupés (sous un même nom) en tableau. On peut accéder à chaque élément à l'aide d'un d'indice précisant le rang de l'élément dans le tableau.
Le premier élément porte l'indice **0**.
Les tableaux sont alloués dynamiquement en Java. Ils sont initialisés par défaut à 0 pour les nombres et à faux pour les tableaux de booléens.

```
int[] tabEnt1;
int tabEnt2[]; // référence (non initialisée) vers un tableau de int

// Allocation et initialistion
int tabEnt[] = {1, 2, 3};
int[] tabEnt = {1, 2, 3};
```

# 3. Les tableaux

Les deux premières instructions précédentes déclarent et définissent la variable tabEnt comme une référence sur un tableau d'entiers, mais ne réservent pas de place mémoire pour ce tableau.
L'allocation doit se faire avec new en indiquant le nombre d'éléments. Ci-après, on réserve 10 éléments de type
int, numérotés de 0 à 9.

```
tabEnt = new int [10]; //
allocation dynamique de 10
int pour tabEnt


int[] tabEnt = new int[10];
// tableau d'int de 10
éléments
```

# 3. Les tableaux

De la même façon on déclare des tableaux d'objets.

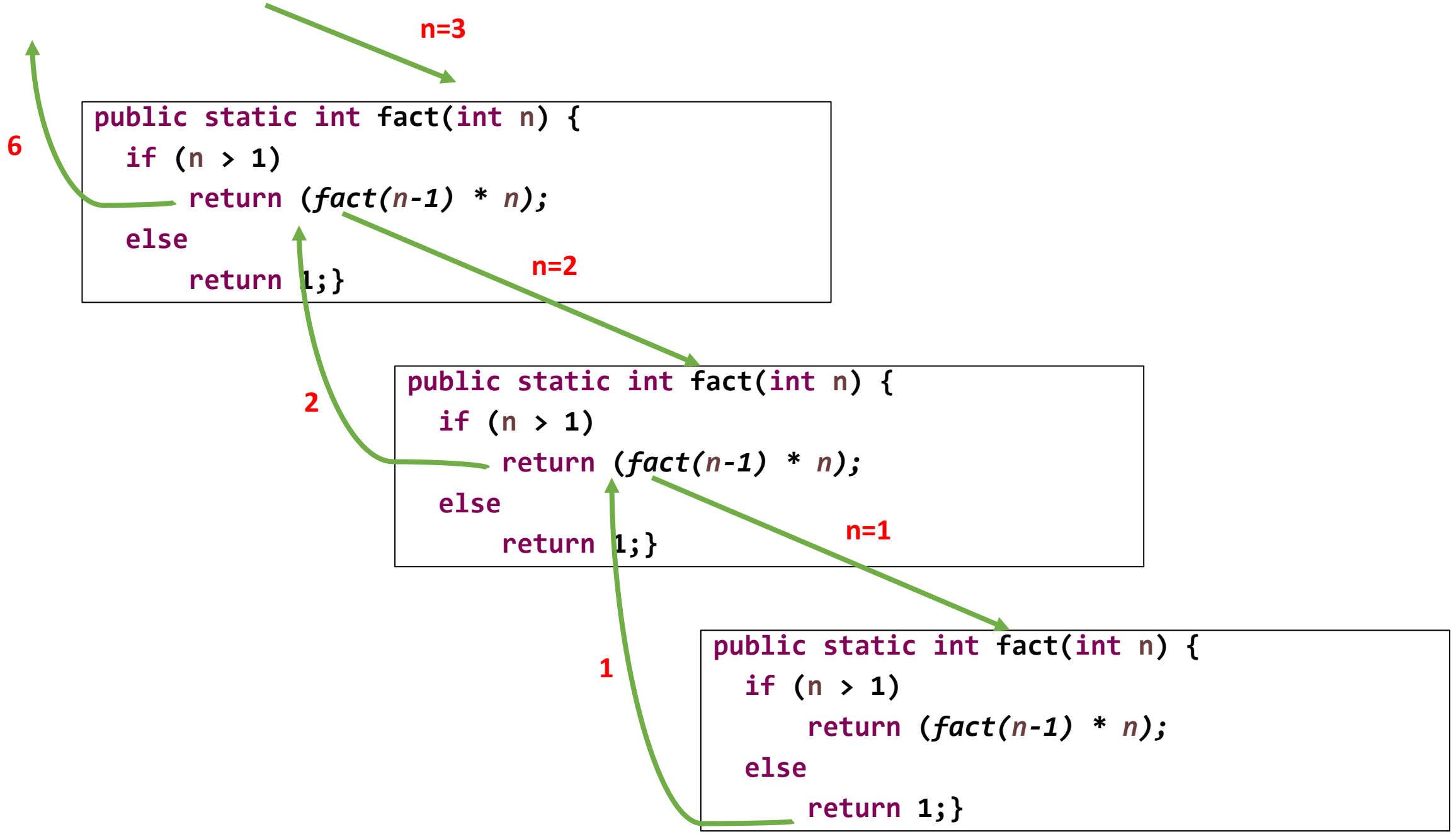
Dans l'exemple en face tabBalle est un tableau d'éléments de type Balle pouvant référencer six objets de type Balle numérotés de 0 à 5. La déclaration du tableau ne crée pas les objets référencés. Les références sur les objets sont nulles.

```
Balle[] tabBalle=new Balle[6];
```

# 4. Récursivité des méthodes

Une boucle peut aussi s'écrire sous forme récursive. On recommence (appel récursif) le corps de la boucle tant qu'on n'a pas atteint la valeur limite de sortie de la boucle.

```
public static int fact(int n) {
  if (n > 1)
      return (fact(n-1) * n);
  else
      return 1;
}
```


n=3
6
public static int fact(int n) {
if (n > 1)
return (fact(n-1) * n);
else
return 1;}
n=2
2
public static int fact(int n) {
if (n > 1)
return (fact(n-1) * n);
else
return 1;}
n=1
1
public static int fact(int n) {
if (n > 1)
return (fact(n-1) * n);
else
return 1;}

# 5. Les fichiers

Un fichier est représenté par un objet de la classe java.io.File. Le constructeur de cette classe prend en paramètre d'entrée le chemin d'accès du fichier. Le flux d'entrée est alors créé à l'aide de la classe FileInputStream sur lequel on peut lire caractère par caractère grâce à la méthode read().

```
import java.io.* ;
public class fiStream  {
public static void main(String[] args) {
try {
File fichier = new File("D:\\monFichier.txt");
FileInputStream flux = new
FileInputStream(fichier);
int c ;
while ((c = flux.read()) > -1) {
System.out.write(c); }
flux.close();}
catch (FileNotFoundException e) {
e.printStackTrace(); }
 catch (IOException e) {
e.printStackTrace(); } } }
```

# III.Classes et Objets

Référence principale : https://www.jmdoudoux.fr/java/dej/chap-poo.htm#poo-1

# 1.Concept de classe

Une classe est le support de l'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.
- Instancier une classe consiste à créer un objet sur son modèle.

*Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.*

# 1.Concept de classe (suite)

- Java est un langage orienté objet : tout appartient à une classe sauf les variables de types primitives.
- Pour accéder à une classe il faut en déclarer une instance de classe ou objet.
- Une classe comporte sa déclaration, des variables et les définitions de ses méthodes.
- Une classe se compose de deux parties : un en-tête et un corps.
- Le corps peut être divisé en 2 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 gèrent leur accessibilité par les composants hors de la classe.

# 1.1 La syntaxe de déclaration d'une classe

modificateurs **class** nomDeClasse [**extends** classe_mere]  [**implements** Interfaces]

```
public class nomDeClasse {
// contenu de la classe
}
```

| Modificateur | Rôle |
|---|---|
| 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. |
| 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. |
| private | la classe n'est accessible qu'à partir du fichier où elle est définie. |
| public | La classe est accessible partout |

# 2. Les Objets

- Les objets contiennent des attributs et des méthodes.
- Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet.
- En Java, une application est un objet.
- La classe est la description d'un objet.
- Un objet est une instance d'une classe.
- Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

# 2.1 . La création d'un objet : instancier une classe

- Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré.
- L'opérateur **new** se charge de créer une instance de la classe et de l'associer à la variable.

```
NomDeClasse NomDeVariable;
NomDeVariable=new NomDeClasse();
// une autre méthode
NomDeClasse NomDeVariable=new NomDeClasse();
```

# 3. Les modificateurs d’accès

- Ils s'appliquent aux classes, aux méthodes et aux attributs.
- Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.
- Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.

# 3. Les modificateurs d'accès (suite...)

Il existe 3 modificateurs qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : **public, private** et **protected**. Leur utilisation permet de définir des niveaux de protection différents :

| Modificateur | Rôle |
|---|---|
| public | Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Une seule classe public est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier |
| par défaut : package friendly | Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package. |
| protected | Si une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous-classes pourront y accéder. On ne peut pas qualifier une classe avec protected. |
| private | C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarées abstract car elles ne peuvent pas être redéfinies dans les classes filles. |

# 4.Constructeurs

Appelés par l'opérateur **new** pour créer un objet

- Peuvent avoir des paramètres (avec surcharge)
- Initialisent les objets
- Constructeur par défaut (si aucun constructeur n'est défini)
- Constructeur de copie

# Exemple

```
public class Astre {
private long idNum;
private String nom = "<pasdenom>";
private Astre orbite = null;
private static long nextId = 0;
/** Creation d'une nouvelle
instance of Astre */
private Astre() {
idNum = nextId ++;
}
```

```
/** Création d'une nouvelle
instance avec 2 paramètres*/
public Astre(String nom,
Astre enOrbite){
this();
this.nom=nom;
orbite=enOrbite;
}
public Astre(String nom){
this(nom,null);
}//…
```

# Exemple (suite)

faire des copies

```
public Astre(Astre a){
idNum = a.idNum;
nom=a.nom;
orbite=a.orbite;
}
```

# 5 Static

- Une variable (une méthode) déclarée static est une variable (méthode) de classe: elle est associée à la classe (pas à une instance particulière).
- Statique parce qu'elle peut être créée au moment de la compilation (pas de new()).
- Statique -> les initialisations doivent avoir lieu à la compilation.

# Exemple : Initialisation d'un tableau static

```
public class Puissancedeux {
static int[] tab = new
int[12];
static{
            tab[0]=1;
            for(int i=0; i< tab.length-1;i++)
            tab[i+1]= suivant(tab[i]);
}
static int suivant(int i){
            return i*2;
}
}
```

Example d'applicaton

System.***out.println(Puissancedeux.tab[2]);***

***Le résultat est alors: 4***

# 6.Les “Getters” et “Setters”

- Si un membre de données est déclaré "private", il ne peut être accédé que dans la même classe. Aucune classe extérieure ne peut accéder au membres privés d’une autre  classe. Si vous devez accéder à ces variables, vous devez utiliser les méthodes publiques "getter" et "setter".
- Les méthodes « Getter » et « Setter » sont des méthodes conventionnelles utilisées pour créer, modifier, supprimer et afficher les valeurs des variables privés.

Dans l'éxemple en face, getBalance et setNumber sont les seules méthodes pour lire et écrire resp « account_balance » et « account_number » d'hors de la classe Account.

Par convention on écrit les noms des setters et getters comme suit:

setNomDeLaVariable,

Dans notre cas:

setAccount_number

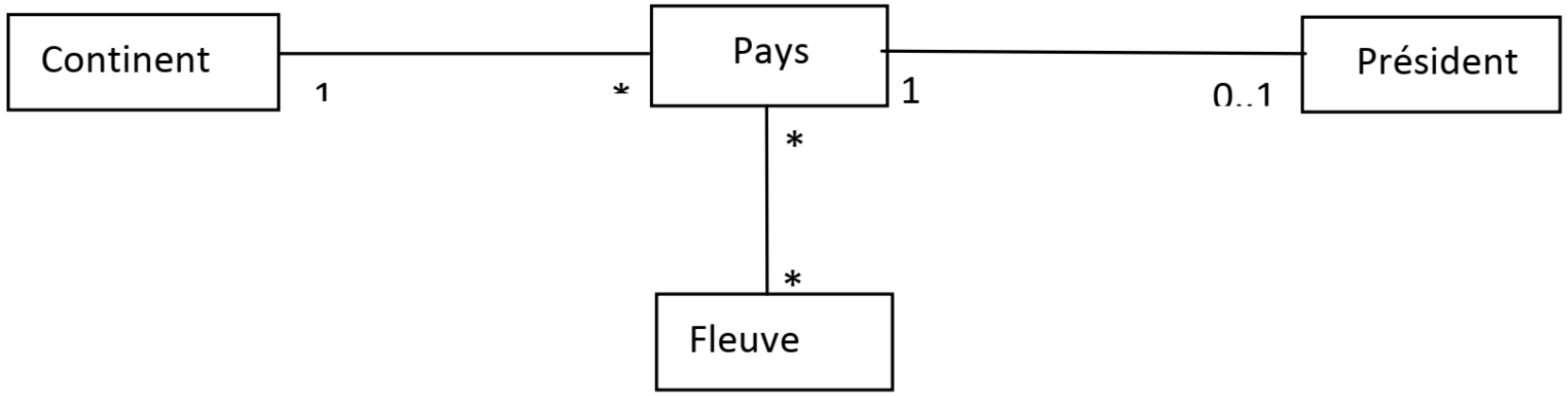
getAccount_balance

```
class Account{
private int account_number;
private int account_balance;
    // getter method
public int getBalance() {
return this.account_balance;}
    // setter method
public void setNumber(int num)
{
 this.account_number = num;}
```

# 7.Association entre Classes

- Une application réalisée selon l'approche objet est constituée d'objets qui collaborent entre eux pour exécuter des traitements.
- Ces objets ne sont donc pas indépendants, ils ont des relations entre eux.
- Comme les objets ne sont que des instances de classes, il en résulte que les classes elles-mêmes sont reliées.
- Deux classes sont en association lorsque certains de leurs objets ont besoin de s'envoyer des messages pour réaliser un traitement.
- Une association possède des **valeurs de multiplicité**, qui représentent le nombre d'instances impliquées.
- UML (Unified Modeling Language) permet de représenter grâce à son diagramme de classes, les liens entre les classes.


- On lit ici : PERSONE est propriétaire d'un COMPTE.
- Les valeurs de multiplicité « 0..1 » indiquent qu'une personne a au minimum zéro compte et au maximum 1 compte.
- Les valeurs de multiplicité « 1 » signifient q 'un compte appartient a une personne et une seule. On peut aussi écrire « 1..1 ».



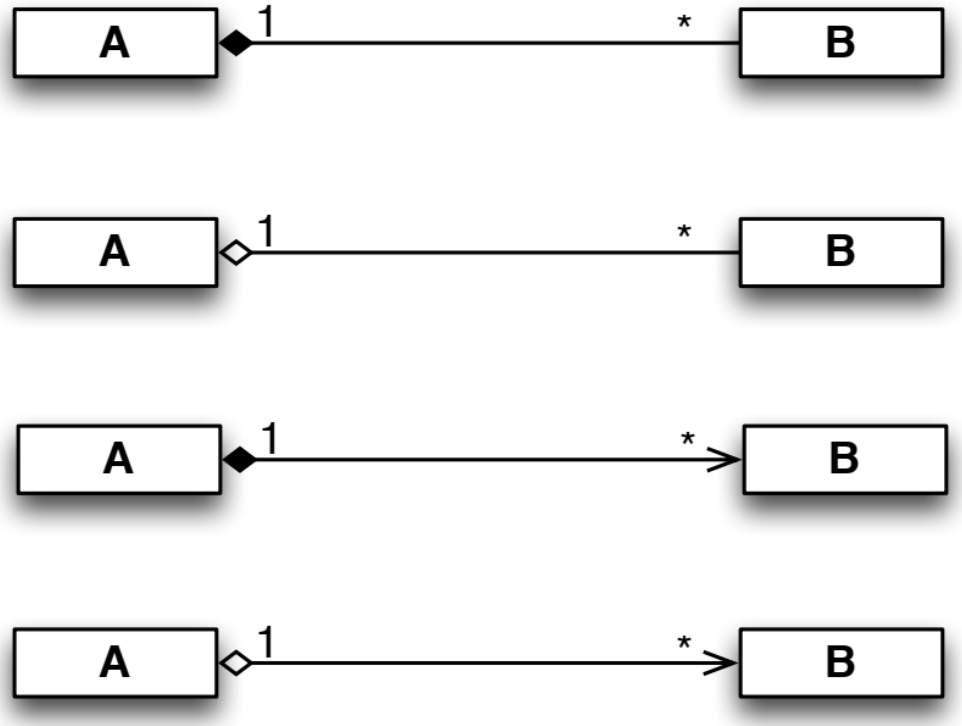

*(notation simplifiée des classes sans indiquer d'attributs et de méthodes)*

| | |
|---|---|
| 1..1 | Un et un seul |
| 1 | Un et un seul |
| 0..1 | Zéro ou un |
| m..n | De m à n |
| * | De zéro à plusieurs |
| 0..* | De zéro à plusieurs |
| 1..* | De un à plusieurs |

Ensemble des valeurs de multiplicité possibles en UML :

# 7.1 Agrégation et association

- Deux sous-types d'associations permettent de préciser un sens particulier à ces relations : l'agrégation et la composition. Elles peuvent également être dirigées.



# 7.1 Agrégation et association(suite…)

L'**agrégation** est une association avec relation de subordination, souvent nommée possède représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe subordonnée) par un *losange creux*. Une des classes "regroupe" d'autres classes. On peut dire que l'objet A utilise ou possède une instance de la classe B.

La **composition** est une association liant le cycle de vie des deux classes concernées. Une association de composition s'interprète comme une classe est composée de un ou plusieurs élément de l'autre classe. Elle est représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe composant) par un *losange plein*. On peut dire que l'objet A est composé instance de la classe B, et donc si l'objet de type A est détruit, les objets de type B qui le composent sont également détruit. Ce sera également souvent les objets de type A qui créeront les objets de type B

# 7.2 Traduction des associations en JAVA

- Les classes associées possèdent en attribut une ou plusieurs références vers l'autre classe.
- Le nombre de référence dépend de la cardinalité.
- Lorsque la cardinalité est « 1 » ou « 0..1 », il n'y a qu'une seule référence de l'autre classe

# 7.2 Traduction des associations en JAVA (suit..)

```
public class Compte{
private int numero;
private float solde;
private Personne proprietaire; //l'attribut proprietaire est de type de
la classe Personne

public Compte(int num, Personne propr){ //constructeur
numero = num;
proprietaire = propr;
solde = 0;
}
public void crediter(){
…
}
… //autres méthodes
```

# 7.2 Traduction des associations en JAVA (suit..)

```
public class Compte{
private int numero;
private float solde;
private Personne proprietaire;
public Compte(int num){ //constructeur
numero = num;
proprietaire = new Personne();
solde = 0;
}
public void crediter(){
…
}
… //autres méthodes
```

# 7.2 Traduction des associations en JAVA (suit..)

```
public class Personne
{
private String nom;
private String prenom;
private float revenu;
private Compte cpt; //l'attribut cpte est une référence à un objet Compte

public Personne(String n, String p, float r){
nom = n;
prenom = p;
revenu = r;
}
//autres méthodes
}
//Remarque : cpt n’est pas initialisé dans le constructeur car une
personne peut ne pas avoir de compte
```

# 7.2 Traduction des associations en JAVA (suit..)

**Et si une personne pouvait avoir plusieurs comptes?**

Dans la classe personne, on n'aurait plus une seule référence d'un compte, mais un ensemble de références vers des objets comptes

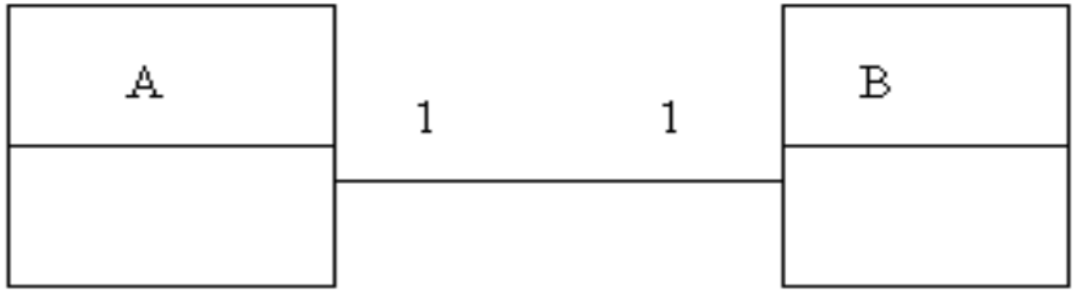
- Soit sous la forme d'un tableau (si le nombre de compte est fixé dès le départ)
- Soit sous la forme d'une collection (ArrayList), qui est un type de tableau particulier dont la taille peut varier.

# 7.2 Traduction des associations en JAVA (suit..)

```
class Personne
{
private String nom;
private String prenom;
private float revenu;
private ArrayList comptes = new ArrayList();
…
}

//Remarque : La collection de comptes est construite, mais
pas les comptes à l’intérieur !
```

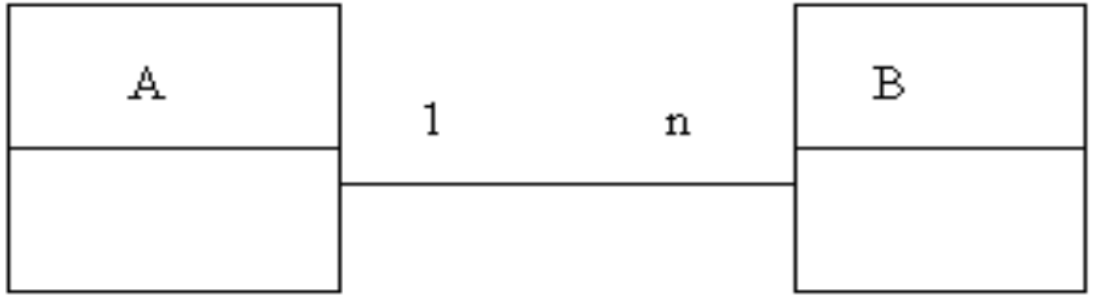
Ci-après, les grandes lignes de passage entre un modèle UML et Java dans ses cas les plus courants :




Association de n à n

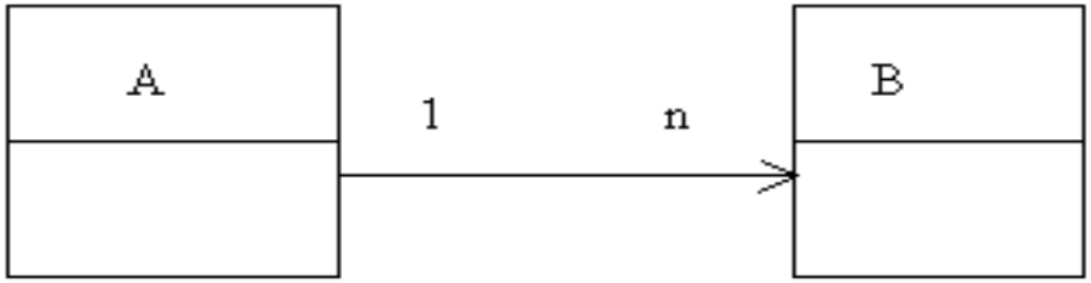
```
public class A{
private B leB;
}

public class B{
private A leA;
}
```




Association de 1 à n

```
public class A{
private ArrayList lesB = new
ArrayList() ;
}
public class B{
private A leA;
}
```




Navigation restreinte

```
public class A{
private ArrayList lesB = new
ArrayList() ;
}
public class B{
// pas de référence à un objet de la
classe A
}
```

# IV.Héritage et polymorphisme

# 1.L’héritage

- L'héritage est un mécanisme qui a été introduit dans la programmation orienté objet dans le but d'éviter la réécriture inutile de code lorsqu'une classe n'est qu'un cas spécial d'une autre classe.
- La classe la plus spécialisée est appelée sous-classe (ou classe fille)et la classe la plus générale, classe mère.
- En java, pour signifier qu'une classe fille hérite d'une classe mère, on utilise dans sa déclaration le mot clé « extends »:

```
class fille extends mere
```

# 1.L’héritage

- La classe **`Object`** est la classe de base de toutes les autres. C'est la seule classe de Java qui ne possède pas de classe mère.
- Tous les objets en Java, quelle que soit leur classe, sont du type **`Object.`** Cela implique que tous les objets possèdent déjà à leur naissance un certain nombre d'attributs et de méthodes dérivées d'**`Object`**.
- Dans la déclaration d'une classe, si la clause **`extends`** n'est pas présente, la surclasse immédiatement supérieure est donc Object

# 1.1 Règles d'héritage (en Java)

- L'héritage multiple comme l'héritage cyclique ne sont pas permis.

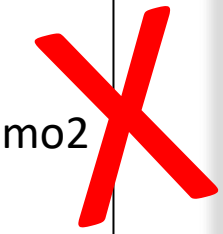
```
class Demo1 extends Demo2
{
//code here
}

class Demo2 extends Demo1
{
//code here
}
```

```
class Demo1
{
//code here
}

class Demo2
{
//code here
}

class Demo3 extends Demo1, Demo2
{
//code here
}
```



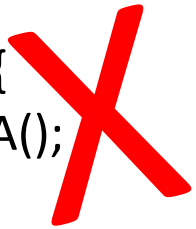
# 1.1 Règles d'héritage (en Java) (suite ..)

- Les membres privés (private) ne sont pas hérités
- Les constructeurs aussi ne sont pas hérités
- On peut assigner une référence mere à un objet fille

```
class A {
private int prive;
  A();}
class B extends A{
  B(){
this.prive=10;
}
}
```

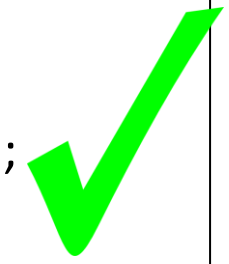
```
class A {
  A();}
class B extends A{
  B();}

class App{
B b=new A();
}
```



```
class A {
  A();}
class B extends A{
  B();}

class App{
A b=new B();
}
```



# 1.2 Utilisation d'un constructeur de la classe mère

- On peut faire appel au constructeur de la classe mère en utilisant le mot-clé **`super( )`** avec les paramètres requis. **`super`** remplace le nom du constructeur de la classe mère.
- L'instruction **`super( )`** doit être obligatoirement la première instruction des constructeurs de la classe fille.
- Tout constructeur d'une classe fille appelle forcément l'un des constructeurs de la classe mère : si cet appel n'est pas explicite, l'appel du constructeur par défaut(sans paramètre) est effectué implicitement. Ainsi, un objet dérivé est toujours construit sur la base d'un objet de sa classe mère.

# 1.3.Le mot clé final

- Un attribut déclaré **final** est un attribut constant. Il faut l'initialiser dès sa déclaration (comme les const en C++). Par convention, un attribut **final** est écrit en majuscule. ex:

```
final int MAX = 1000;
```

- Une classe **final** ne peut pas être étendue. Mais une classe **final** peut évidemment être la fille d'une autre classe (non final!).
- Une méthode **final** est une méthode qui ne peut pas être redéfinie dans les sous-classes.

# 1.4.Les classes abstraites

Une classe abstraite est totalement l'opposé d'une classe final:

- Une classe abstraite est une classe qui ne peut pas être instanciée directement. Elle n'est utilisable qu'à travers sa descendance. Une classe abstraite doit toujours être dérivée pour pouvoir générer des objets.
- Une classe abstraite est précédée du mot clé abstract:

```
abstract class classeabstraite
```

- L'utilité d'une classe abstraite est de permettre le regroupement (la factorisation) des attributs et méthodes communs à un groupe de classes.

# 1.4.Les classes abstraites (suite…)

- Dans une classe abstraite, on peut trouver des méthodes abstraites, c'est à dire des méthodes qui n’ont pas d'implémentation possible dans la classe abstraite, mais qui doivent obligatoirement être implémentées différemment dans toutes les classes filles.
- L'intérêt des méthodes abstraites vient du fait que l'on peut les appeler de la même façon pour tous les objets dérivés : on est en plein dans le polymorphisme. (vous comprendrez cet intérêt par la pratique).
- Une classe qui possède une (ou plusieurs) méthode abstraite est obligatoirement abstraite.

# 1.5.Les interfaces

- L'interface possède un niveau d'abstraction supérieur à la classe abstraite avec
  - des méthodes uniquement abstraites
  - aucun champs, à l'exception des constantes.

```
public interface Interf
{
final int MAX=2;
double operation();
void affichage();
}
```

- Une interface ne peut pas être héritée mais implémentée
- Une classe implémente une interface en utilisant le mot clé **implements** :

```
class A implements MyInterface
{
// Redéfinition de operation();
double operation(double a, double b) {
return a+b;
}

// Redéfinition de affichage();
  void affichage() {
  System.out.println(operation(3,4));
  }

}
```

- Une même classe peut implémenter plusieurs interfaces.

```
public interface I1
{
void operation();
}

public interface I2
{
void affichage();
}

class A implements I1, I2
{
// Redéfinition de operation();
// Redéfinition de affichage();
}
```

# 1.6.Les classes internes (classes imbriquées)

Les classes internes (inner classes) sont des classes qui sont définies dans une autre classe. Les difficultés dans leur utilisation concernent leur visibilité et leur accès aux membres de la classe dans laquelle elles sont définies.

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 adapter (pour traiter des événements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale.

```
public class ClassePrincipale {

    public class ClasseInterne {
    // Contenu de la classe interne
    }
    public static void main(String[] args) {
    ClassePrincipale cp = new ClassePrincipale();
    ClassePrincipale.ClasseInterne ci = cp. new ClasseInterne() ;
    System.out.println(ci.getClass().getName());
     }
}
```

Resultat:

ClassePrincipale$ClasseInterne

# 1.7 Chainage des méthodes

```
class A {
private int a;
private float b;
A() {
System.out.println("Calling The Constructor");}

public A setint(int a) {
this.a = a;
return this; }

public A setfloat(float b) {
this.b = b;
return this; }

void display()  {
System.out.println("Display="+ a + " " + b);
} }
```

```
//Driver code
public class Example {
public static void main(String[] args)
{

//This is the "method chaining".
new A().setint(10).setfloat(20).display();
```

# 2.Polymorphisme

Polymorphe se dit d'une chose qui peut se présenter sous différentes formes.

En programmation orientée objet, le polymorphisme concerne les méthodes et les objets(réellement ça concerne les références vers les objets)

# 2.1Polymorphisme et objets:

Un objet d'une certaine classe peut être manipulé comme un objet d'une autre classe. Par exemple un objet de la classe « Etudiant » qui hérite de la classe « Personne », peut être manipulé comme s'il appartenait à cette dernière:

```
Etudiant etudiant= new Etudiant();
```

Ou

```
Personne etudiant =new Etudiant();
```

Ce type de polymorphisme est très utile dans le passage des paramètres

Supposons par exemple qu'on a un « Medecin » qui « examiner() » (de la même manière) les « Etudiant », « Employe » et « Enseignant » (on parle des classes et objets), il est plus commode ainsi que la méthode « examiner () » de la classe « Medecin » aie un parametre de type « Personne » de qui héritent les classes des objets à manipuler (etudiant, employe et enseignant).

```
Personne etudiant= new Etudiant();
Personne enseignant= new Enseignant();
Personne employe= new Employe();
medecin.examiner(etudiant);
medecin.examiner(enseignant);
medecin.examiner(employe);
```

# 2.2 Polymorphisme et méthodes

- Une méthode de la classe mère peut être implémentée différemment dans une classe fille: la méthode est dite redéfinie. Aucun signe n'indique en java qu'une méthode est redéfinie (contrairement à Pascal ou à C++).
- La redéfinition d'une méthode dans une classe fille cache la méthode d'origine de la classe mère. Pour utiliser la méthode redéfinie de la classe mère et non celle qui a été implémentée dans la classe fille, on utilise le mot-clé super suivi du nom de la méthode. ex: `super.machin();` fait appel à la méthode **`machin()`** implémentée dans la classe mère et non à l'implémentation de la classe fille.
- Lorsqu'on redéfinit une méthode, on peut changer sa signature (les paramètres et leurs types) mais pas sa sortie (le type du résultat retourné)