

# Analyse d'un problème en utilisant l'approche OO

## Etude de cas:

Une société nommée “Direct Clothing” vend des chemises par catalogue.

On veut développer un programme pour cette société.

On sait qu'une chemise possède les caractéristiques suivantes:

Chaque chemise:

- possède un identifiant- code à barre
- est disponible en plusieurs coloris– bleu, gris, etc
- est disponible en plusieurs tailles
- a un prix
- a une description – type du tissu, style, etc
- quantité dans le stock

On peut ajouter ou diminuer des chemises du stock.



Button Front, Blouse  
Blue Denim

# Identification de l'objet



Chemise
Prix
Couleur
Description
Quantité dans le stock
Ajouter chemise dans le stock
Diminuer une chemise du stock

# Identification de l'objet



Une classe n'est pas un objet.

Une classe est un patron d'objet.



Classe « Chemise »

## Objet 1

Id: 101

Prix: 20\$

Couleur: Bleu

Quantite: 200



## Objet 2

Id: 201

Prix: 30\$

Couleur: Rouge

Quantite: 180



# Classe et objet

On appelle classe la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet.

Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule.  
En réalité on dit qu'un objet est une **instanciation** d'une classe



**objet = instance**

Une classe est composée de deux parties :

- Les attributs (parfois appelés *données membres*) : il s'agit des données représentant l'état de l'objet
- Les méthodes (parfois appelées *fonctions membres*) : il s'agit des opérations applicables aux objets

# Déclaration de la classe

```
public class Chemise
{
    /*Déclaration des attributs*/

    /*Déclaration des méthodes*/

    //commentaire sur une seule ligne

    /*commentaires sur
    plusieurs lignes*/
}
```

- Le fichier *.java* doit avoir le même nom que la classe (Chemise.java)
- Le nom de la classe doit commencer par une majuscule

# Déclaration des méthodes

Syntaxe:

```
Type_retour nom_method([arguments])  
{  
}  
}
```

Exemple:

```
void afficherInfoChemise (){
```

```
}
```

- Le nom de la méthode doit commencer par un verbe

# Classe chemise

```
public class Chemise{  
    int id;  
    char couleur;  
    float prix;  
    String description;  
    int quantite;  
    void ajouterChemise (int nombre) {  
        quantite += nombre;  
    }  
  
    void diminuerChemise (int nombre) {  
        quantite -= nombre;  
    }  
  
    void afficherInfoChemise() {  
        System.out.println(id+ couleur);  
    }  
}
```

# Création des objets



17, Rue Carthage



9, Rue Hannibal



23, Rue des anges



# Référence d'un objet

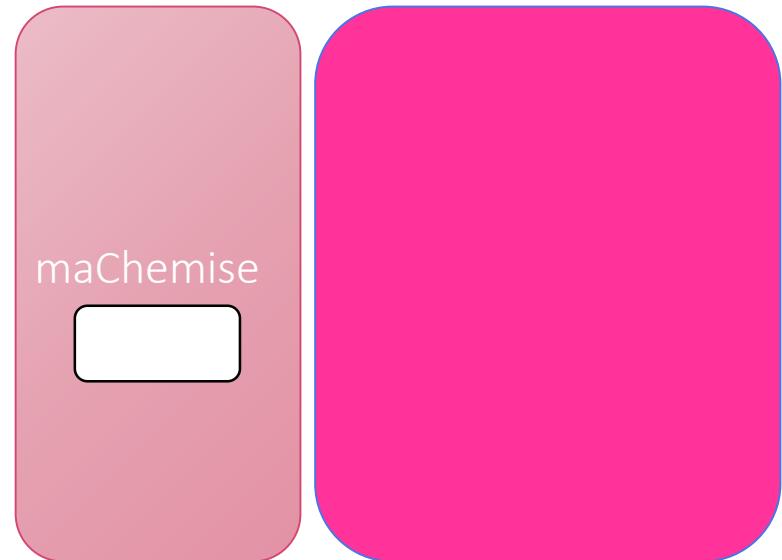
- Une référence nous permet de trouver l'objet.
- En utilisant la référence, on peut accéder aux attributs et méthodes de l'objet.
- Une adresse nous permet de trouver une maison.
- En utilisant l'adresse, on peut envoyer une lettre à cette maison.

# Notion de référence

1/5

```
public class Test{  
  
    public static void main(String[] args){  
  
        Chemise maChemise;  
  
        maChemise=new Chemise();  
  
        maChemise.couleur='R';  
    }  
}
```

Mémo



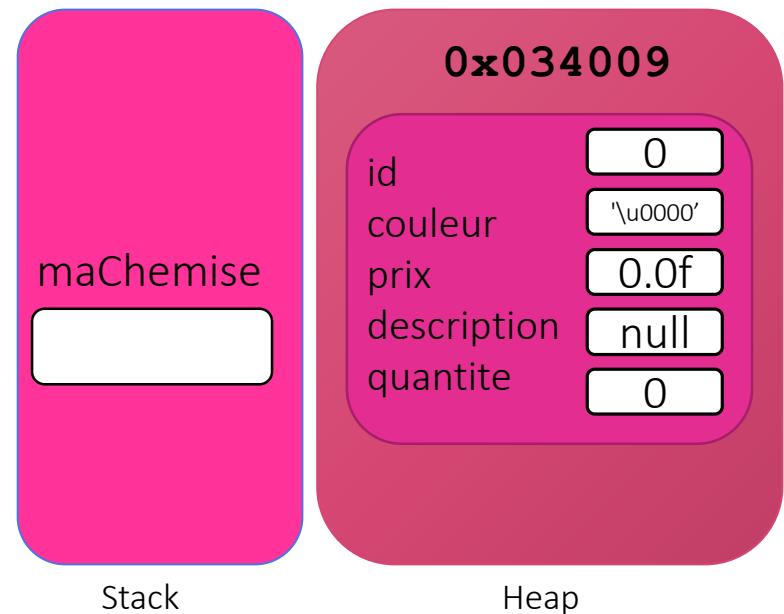
Création d'une variable *maChemise* de type Chemise

# Notion de référence

2/5

```
public class Test{  
  
    public static void main(String[] args){  
  
        Chemise maChemise;  
  
        maChemise = new Chemise();  
  
        maChemise.couleur='R';  
    }  
}
```

Mémoire



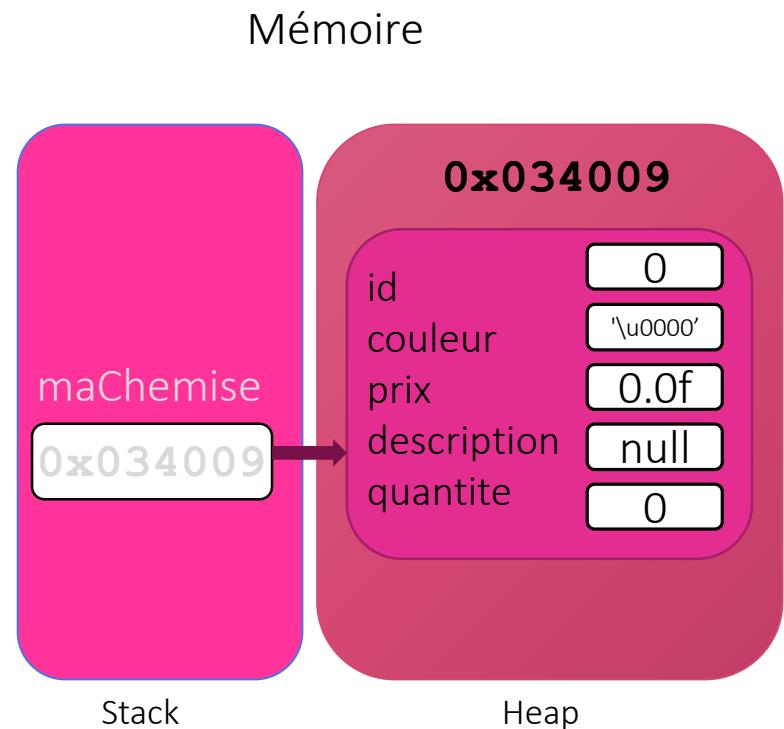
Instanciation de la classe Chemise → Création d'un objet.

Cet objet possède une adresse de son emplacement dans la mémoire (**0x034009**)

# Notion de référence

3/5

```
public class Test{  
  
    public static void main(String[] args){  
  
        Chemise maChemise;  
  
        maChemise=new Chemise();  
  
        maChemise.couleur='R';  
    }  
}
```



Lier l'objet créé et la variable *maChemise*

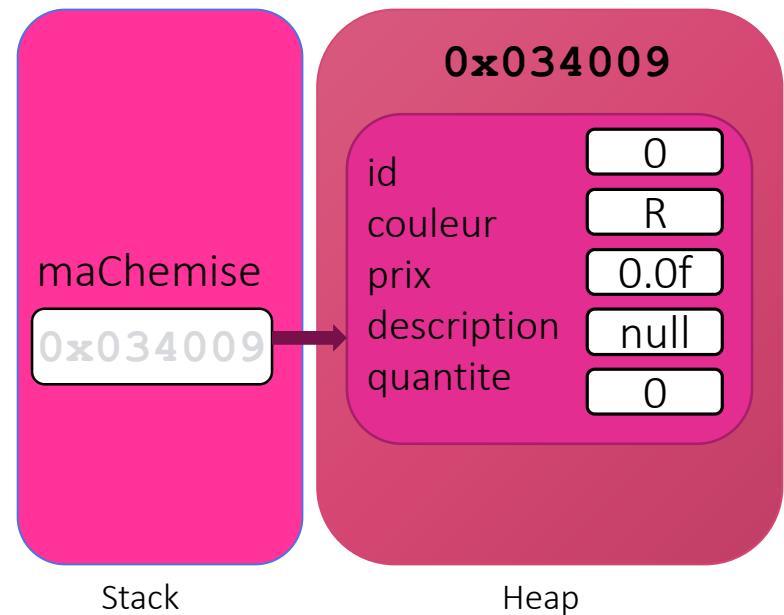
→ *maChemise* est la référence de l'objet créé

# Notion de référence

4/5

```
public class Test{  
  
    public static void main(String[] args){  
  
        Chemise maChemise;  
  
        maChemise=new Chemise();  
  
        maChemise.couleur='R';  
    }  
}
```

Mémoire

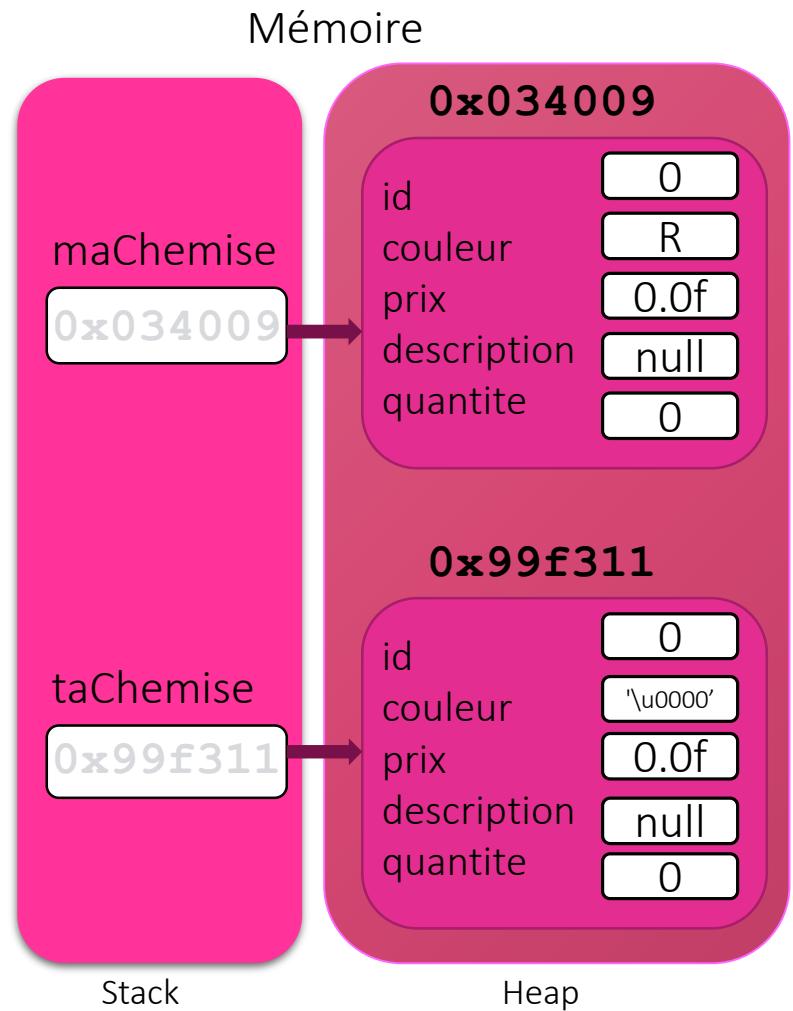


En utilisant la référence `maChemise`, on peut accéder aux attributs de l'objet

# Notion de référence

5/5

```
public class Test{  
    public static void main(String[] args){  
        Chemise maChemise=new Chemise();  
        Chemise taChemise=new Chemise();  
    }  
}
```



# Le mot-clé this

Le mot-clé *this* permet de désigner l'objet courant,

 *this* permet d'accéder aux attributs et méthodes de l'objet courant

- Pour manipuler un attribut de l'objet courant: **this.couleur**
- Pour manipuler une méthode de la super-classe : **this.ajouterChemise (100)**
- Pour faire appel au constructeur de l'objet courant: **this()**

# Les constructeurs 1/4

Pour créer un objet à partir d'une classe, on utilise l'opérateur `new`.

```
public class Test{  
  
    public static void main(String[] args){  
  
        Chemise maChemise;  
  
        maChemise=new Chemise();  
  
    }  
}
```

L'opérateur `new` fait appel au constructeur de la classe

```
class Chemise{  
  
    Chemise () {}  
  
}
```

- un constructeur porte le même nom que la classe dans laquelle il est défini
- un constructeur n'a pas de type de retour (même pas `void`)

# Les constructeurs 2/4

## ■ Constructeur par défaut

```
Chemise(){}  
}
```

Le constructeur par défaut initialise les variables de la classe aux valeurs par défaut.

```
Chemise(){  
    id=0;  
    couleur='B'  
    prix=10.2f;  
}  
}
```

## ■ Constructeur surchargé

```
Chemise(int id, char couleur, float prix){  
    this.id=id;  
    this.couleur=couleur;  
    this.prix=prix;  
}  
}
```

# Les constructeurs 3/4

Si vous ne créez pas un constructeur dans votre classe, le compilateur va automatiquement vous créer un constructeur par défaut implicite

- Si le constructeur surchargé est créé, le constructeur par défaut implicite ne sera plus créé par le compilateur
- La plateforme java différencie entre les différents constructeurs déclarés au sein d'une même classe en se basant sur le nombre des paramètres et leurs types.

On ne peut pas créer deux constructeurs ayant le même nombre et types des paramètres.

```
Chemise(int id) {  
    this.id=id  
}  
Chemise(int id) {  
    this.id=id*2  
}
```

# Les constructeurs 4/4

Quel constructeur va choisir Java lorsque vous allez créer votre objet ?

```
class Chemise{  
    int id;  
    char couleur;  
    float prix;  
    String description;  
    int quantite;
```

Chemise () {}



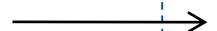
Chemise ch1=new Chemise();

```
Chemise(int id) {  
    this.id=id;  
}
```



Chemise ch1=new Chemise(122);

```
Chemise(int id, char couleur) {  
    this.couleur=couleur;  
}
```



Chemise ch1=new Chemise(122, 'B');

# Plan

Introduction  
Classe et objet  
Encapsulation  
Héritage  
Polymorphisme  
Interfaces

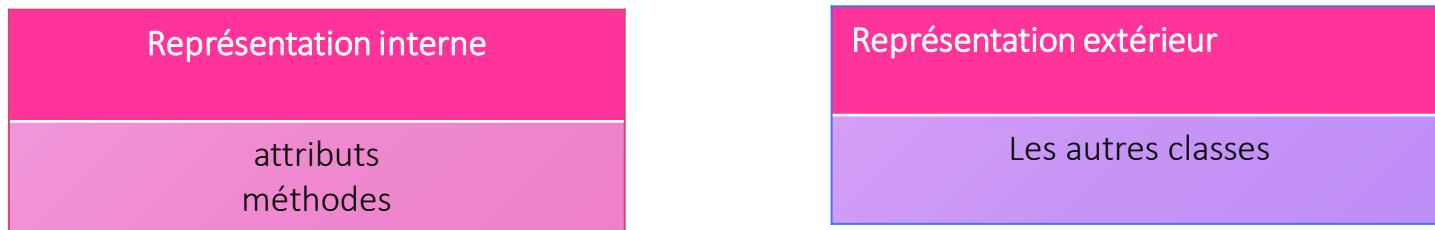
# Objectifs

- ✓ Notion de package
- ✓ Encapsulation des classes
- ✓ Encapsulation des attributs/méthodes
- ✓ Les attributs et méthodes static

# Encapsulation : définition

L' encapsulation : est un des concepts fondamentaux de la POO (Programmation Orientée Objets)

Le principe d'encapsulation : un objet ne doit pas exposer sa représentation interne au monde extérieur.

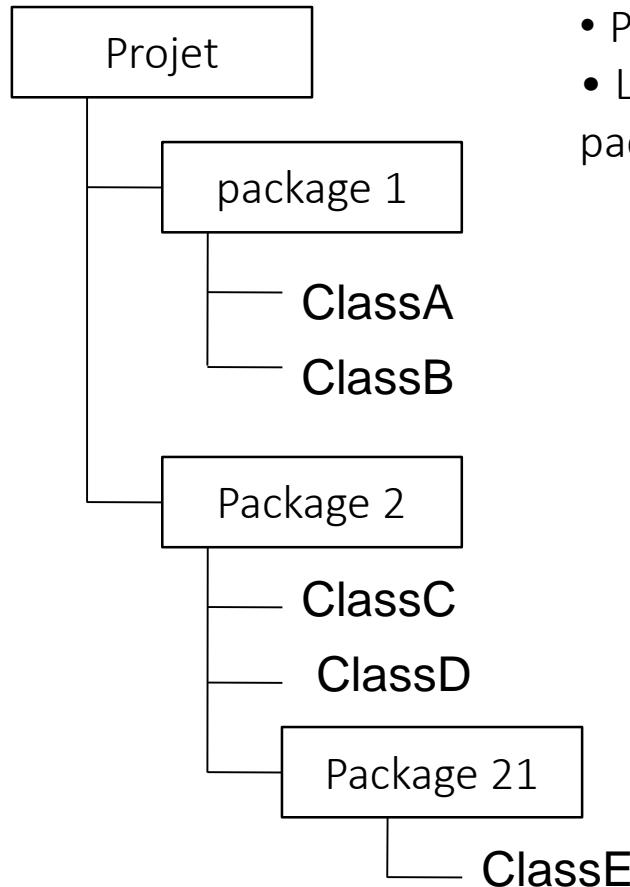


Dans le but de renforcer le contrôle de l'accès aux variables d'une classe, il est recommandé de les déclarer ***private***.

L'objet est ainsi vu de l'extérieur comme une **boîte noire**

Protéger l'information contenue dans un objet et ne proposer que des méthodes de manipulation de cet objet

# C'est quoi un package ?



- Package = répertoire.
- Les classes Java peuvent être regroupées dans des packages.

- Déclaration d'un package

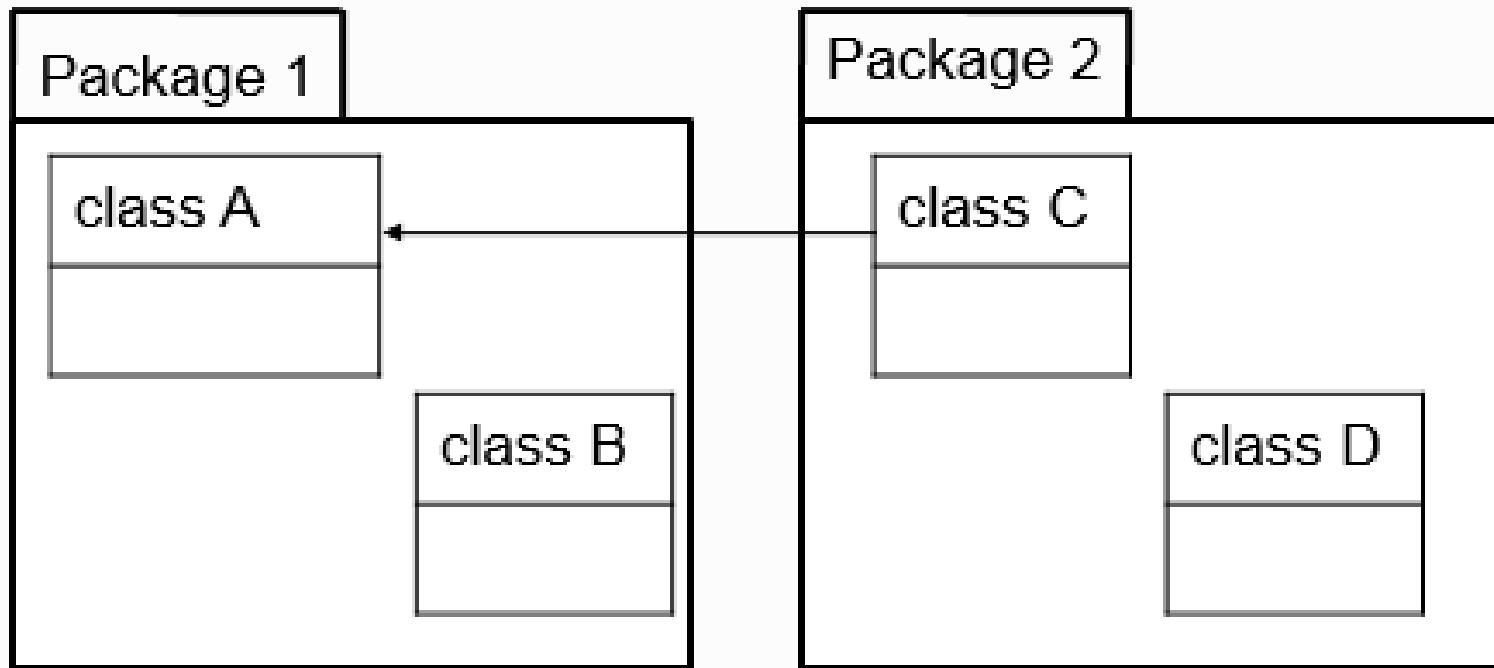
```
package pack1;
```

- Import d'un package

```
import pack2.ClassC;
```

```
import pack2.*;
```

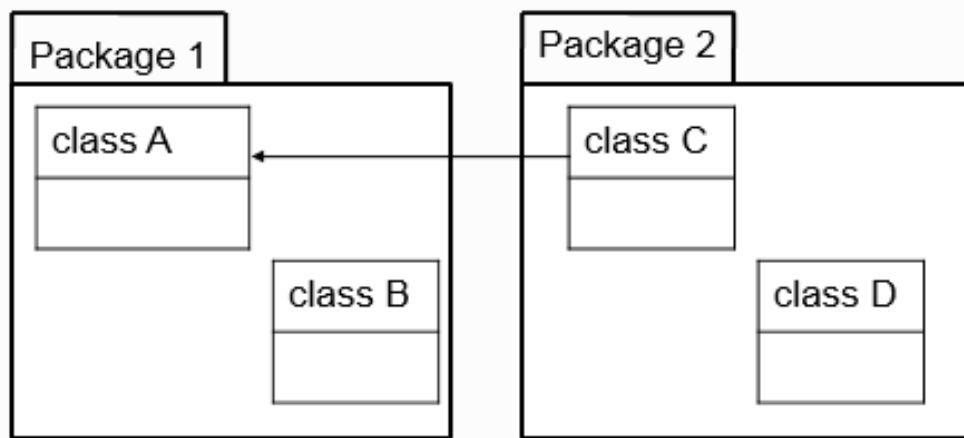
# Encapsulation des classes 1/3



# Encapsulation des classes 2/3

**Classe public :**

```
public class A {  
    ...  
}
```



```
class B {  
    A a;  
    ...  
}
```

```
class C extends A {  
    A a;  
    ...  
}
```

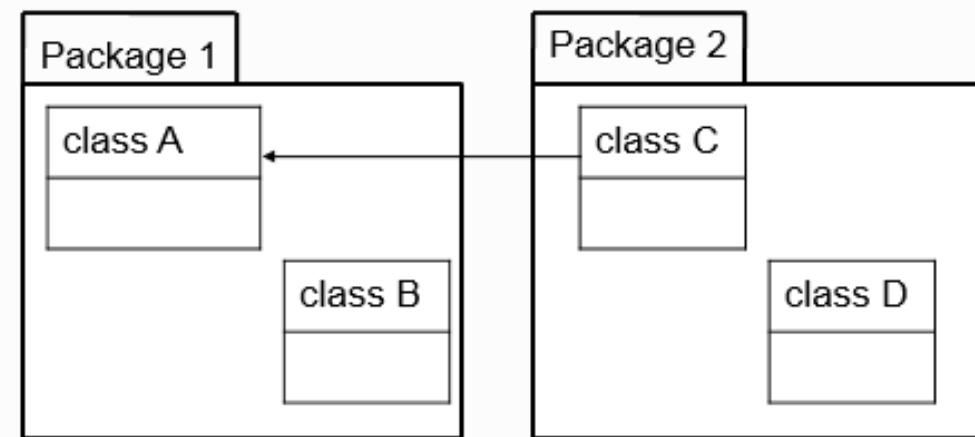
```
class D {  
    A a;  
    ...  
}
```

La classe public est visible depuis n'importe quelle classe du projet.

# Encapsulation des classes 3/3

## Default classe

```
class A {  
...  
}
```



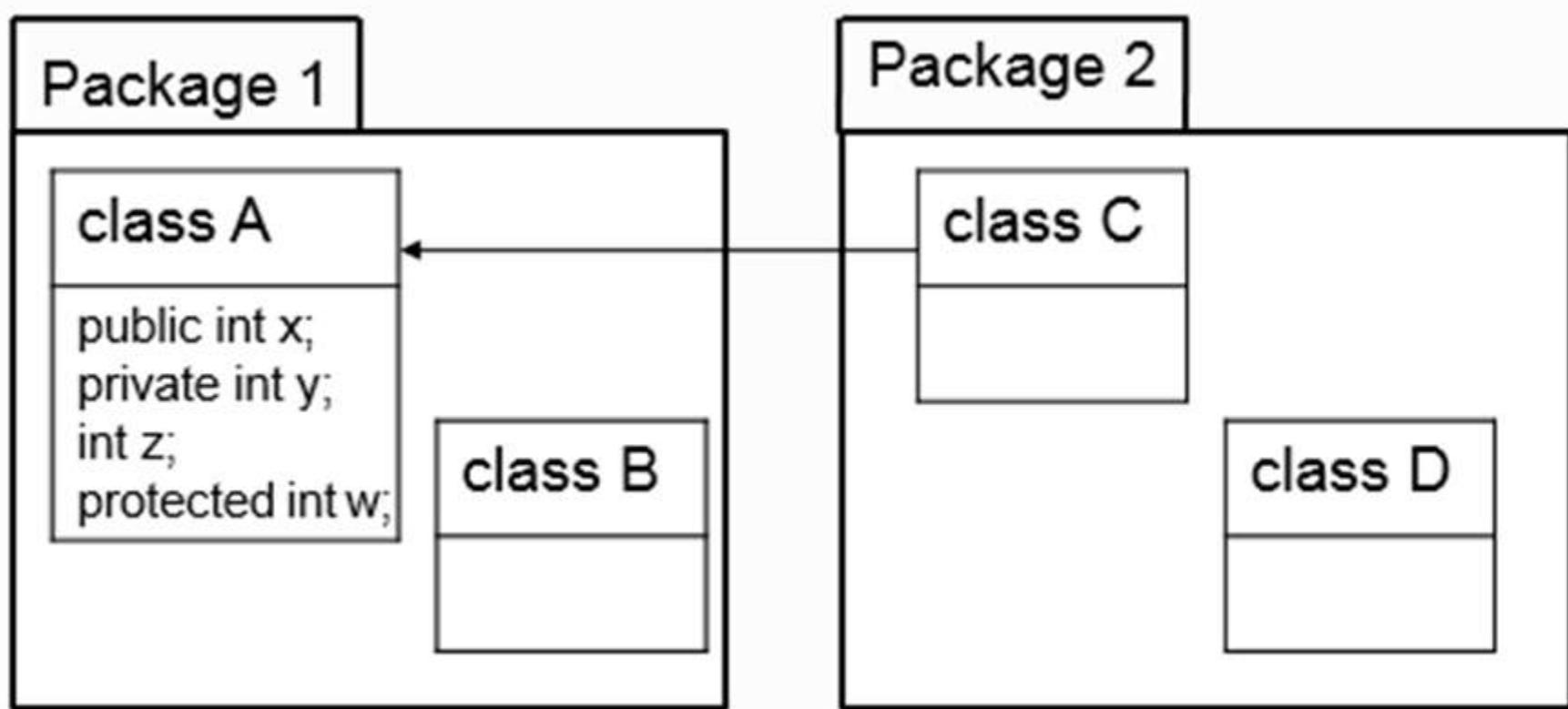
```
class B {  
    A a;  
...  
}
```

```
class C extends A {  
    A a;  
...  
}
```

```
class D {  
    A a;  
...  
}
```

La classe *default* est visible seulement par les classes de son package.

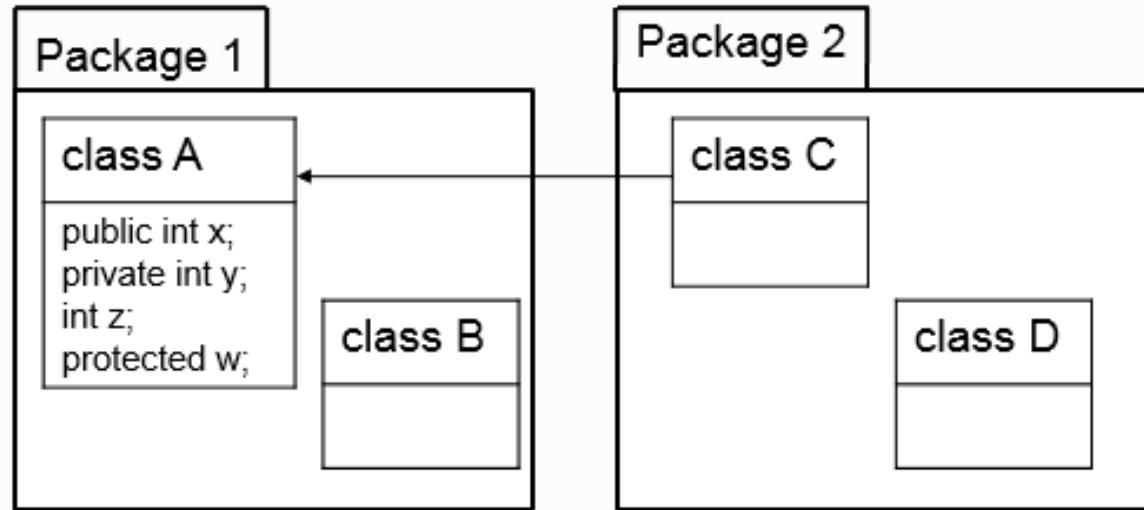
# Encapsulation des attributs 1/5



# Encapsulation des attributs 2/5

## L'attribut public

```
public class A {  
    public int x;  
  
    ...  
}
```



```
class B {  
    A a=new A();  
    a.x = t ;  
}
```

```
class C extends A {  
    A a=new A();  
    a.x = t ;  
    x = t ;  
}
```

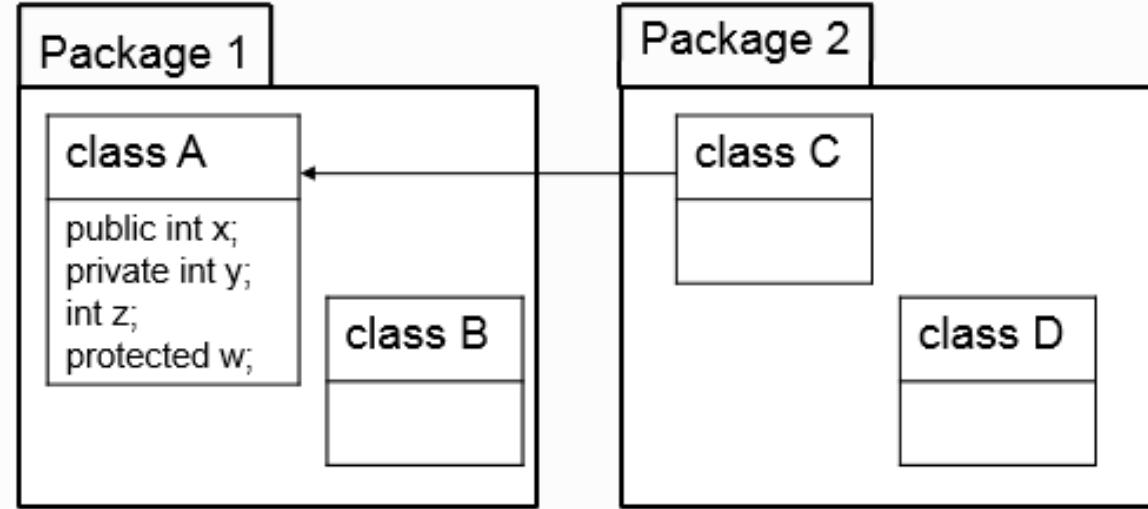
```
class D {  
    A a=new A();  
    a.x = t ;  
}
```

La variable *public* est visible par toutes les classes

# Encapsulation des attributs 3/5

## L'attribut private

**private int y;**



```
class B {
    A a=new A();
    a.y=t;
}
```

```
class C extends A {
    A a=new A();
    a.y=t;
    y=t;
}
```

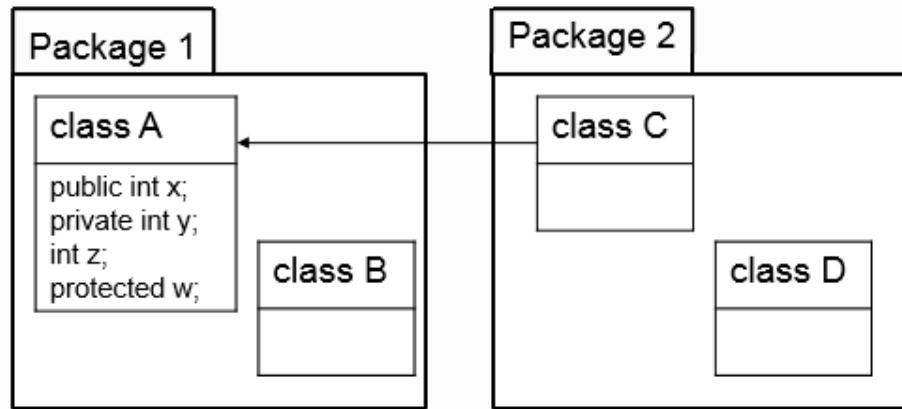
```
class D {
    A a=new A();
    a.y=t;
}
```

La variable **private** n'est accessible que depuis l'intérieur même de la classe.

# Encapsulation des attributs 4/5

L'attribut par défaut : package friendly

```
public class A {  
    int z ;  
    ...  
}
```



```
class B {  
    A a=new A();  
    a.z=t;  
}
```

```
class C extends A {  
    A a=new A();  
    a.z=t;  
    z=t;  
}
```

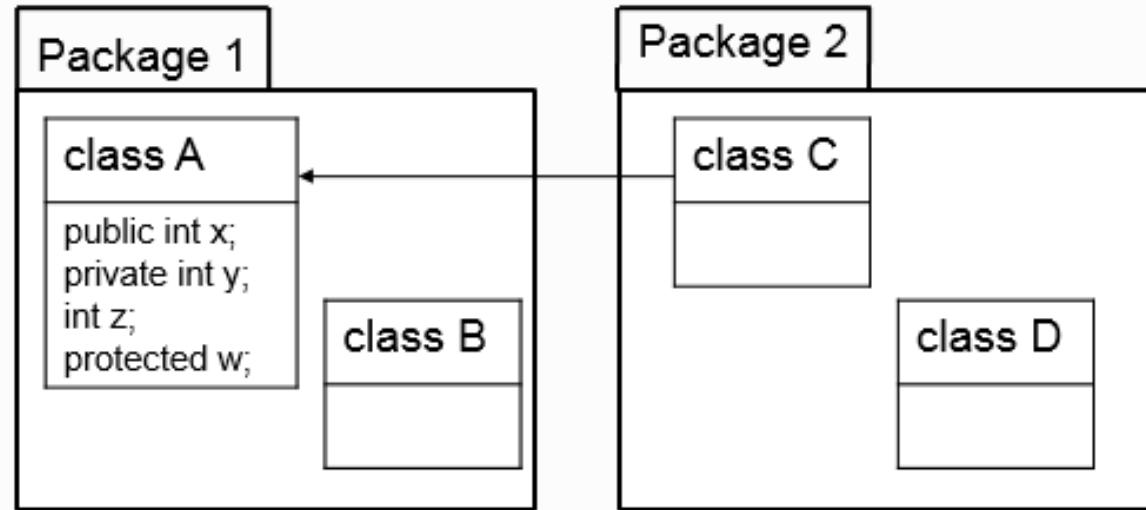
```
class D {  
    A a=new A();  
    a.z=t;  
}
```

La variable *par défaut* n'est accessible que depuis les classes faisant partie du même *package*.

# Encapsulation des attributs 5/5

## L'attribut protected

```
public class A {  
    ...  
    protected int w ;  
}
```



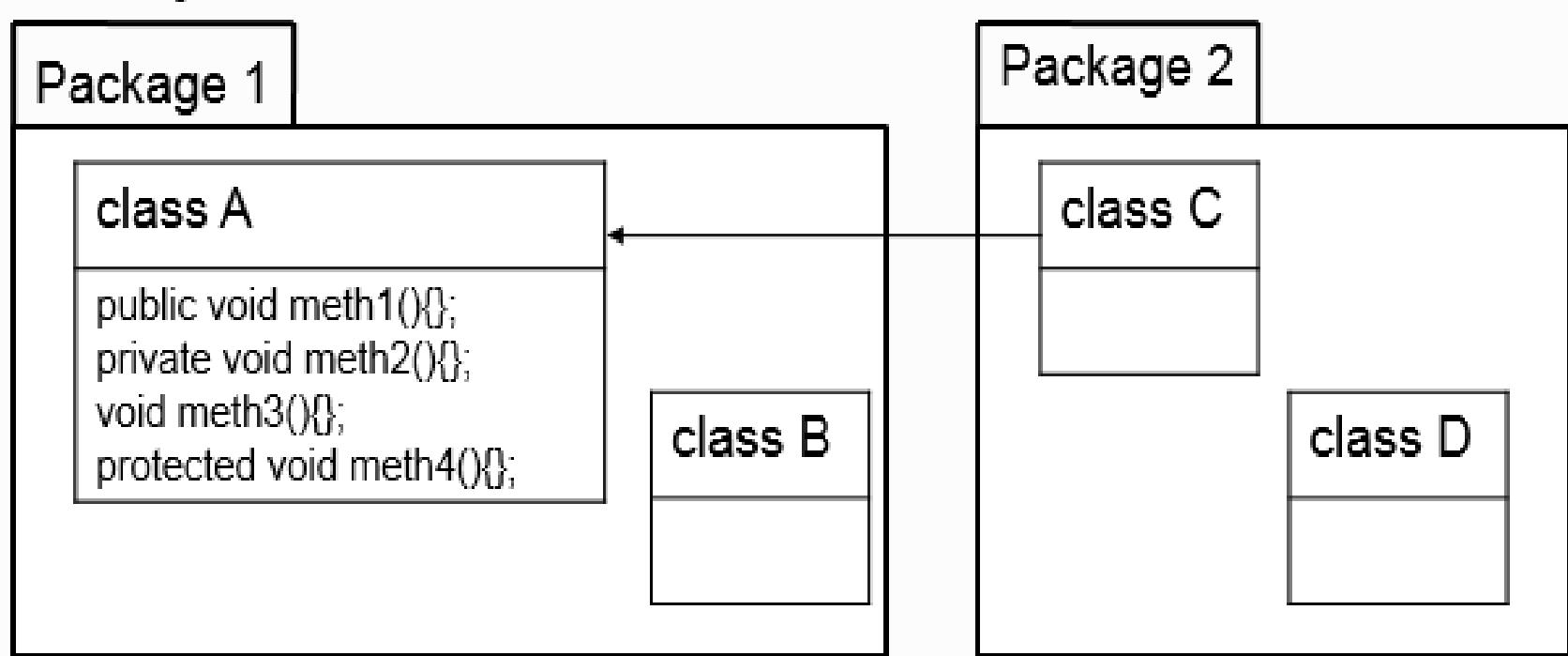
```
class B {  
    A a=new A();  
    a.w= t ;  
}
```

```
class C extends A {  
    A a=new A();  
    a. w = t ;  
    w = t;  
}
```

```
class D {  
    A a=new A();  
    a. z = t ;  
}
```

La variable *protected* est accessible uniquement aux classes d'un package et à ses sous-classes (même si elles sont définies dans un package différent.)

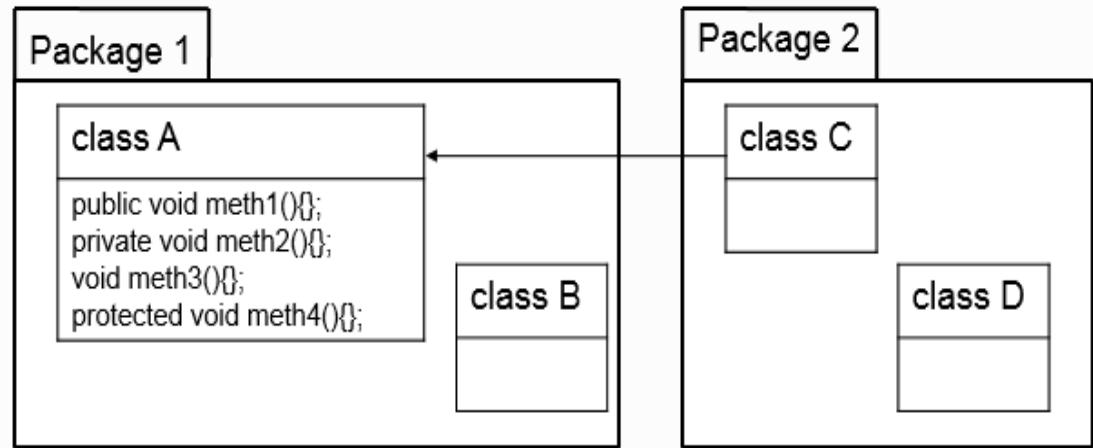
# Encapsulation des méthodes 1/5



# Encapsulation des méthodes 2/5

## Méthode public

```
public class A {  
    public void meth1()  
    { }  
    ...  
}
```



```
class B {  
    A a=new A();  
    a.meth1();  
}
```

```
class C extends A {  
    A a=new A();  
    a.meth1() ;  
    meth1();  
}
```

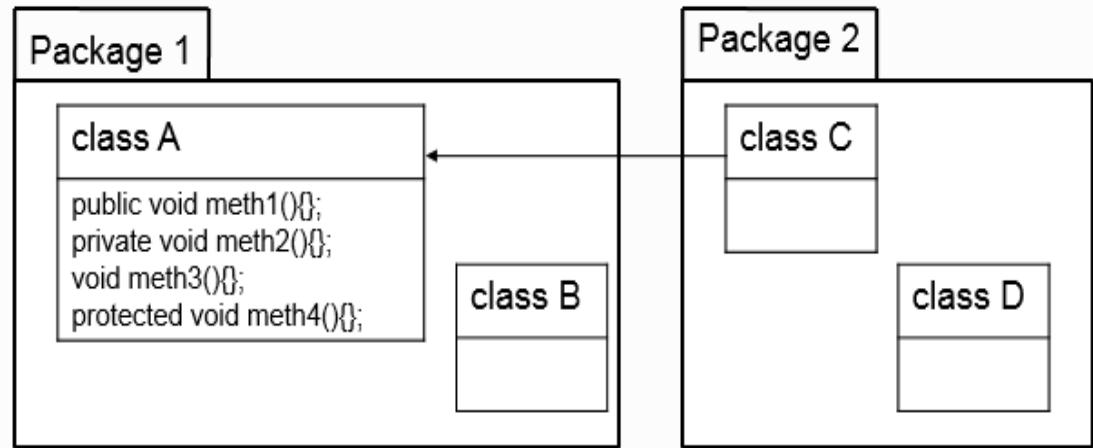
```
class D {  
    A a=new A();  
    a.meth1() ;  
}
```

La méthode *public* est visible par toutes les classes

# Encapsulation des méthodes 2/5

## Méthode private

```
public class A {  
    private void meth2 ()  
    { }  
    ...  
}
```



```
class B {  
    A a=new A();  
    a.meth2();  
}
```

```
class C extends A {  
    A a=new A();  
    a.meth2();  
    meth2();  
}
```

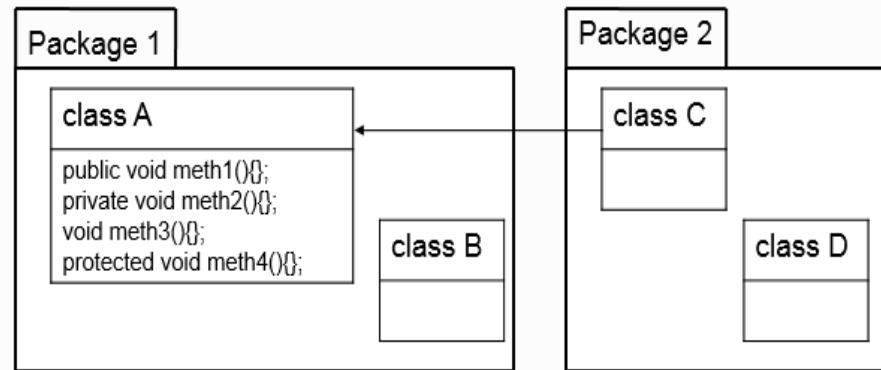
```
class D {  
    A a=new A();  
    a.meth2();  
}
```

La méthode *private* n'est accessible que depuis l'intérieur même de la classe.

# Encapsulation des méthodes 3/5

## Méthode par défaut : package friendly

```
public class A {  
    void meth3 () {}  
    ...  
}
```



```
class B {  
    A a=new A();  
    a.meth3() ;  
}
```

```
class C extends A {  
    A a=new A();  
    a.meth3() ;  
    meth3() ;  
}
```

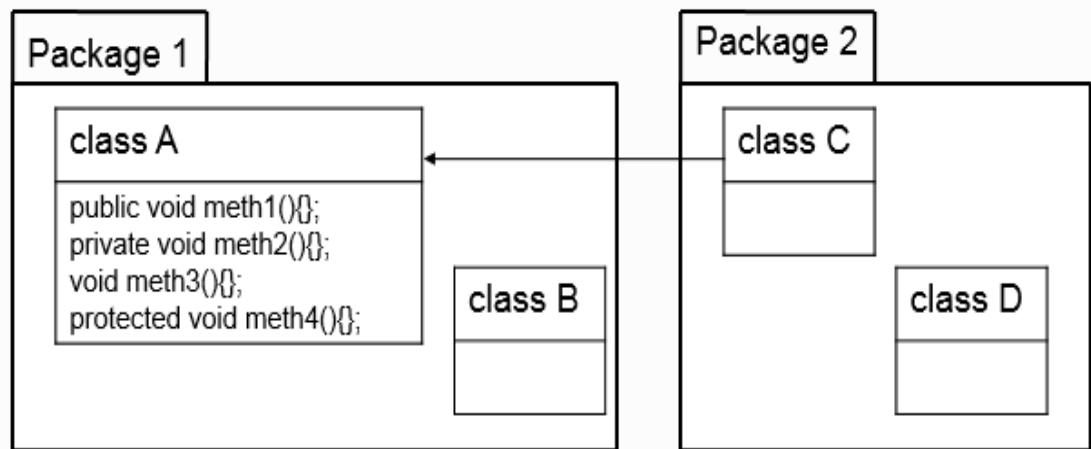
```
class D {  
    A a=new A();  
    a.meth3() ;  
}
```

La méthode *par défaut* n'est accessible que depuis les classes faisant partie du même *package*.

# Encapsulation des méthodes 4/5

## Méthode protected

```
public class A {  
    Protected void meth4()  
    {}  
    ...  
}
```



```
class B {  
    A a=new A();  
    a.meth4() ;  
}
```

```
class C extends A {  
    A a=new A();  
    a.meth4();  
    meth4() ;  
}
```

```
class D {  
    A a=new A();  
    a.meth4();  
}
```

La méthode *protected* est accessible uniquement aux classes d'un package et à ses sous-classes (même si elles sont définies dans un package différent.)

# Encapsulation des attributs/méthodes

Pour la manipulation des attributs *private*, il faut :

- Un **modificateur** (setter): est une méthode qui permet de définir la valeur d'une variable particulière
- Un **accesseur** (getter): est une méthode qui permet d'obtenir la valeur d'une variable particulière.

Les setter et les getter doivent être déclarés *public*

# Encapsulation des attributs/méthodes : Exemples

```
private float prix;  
  
public void setPrix(float prix){  
    this.prix=prix;  
}  
public float getPrix(){  
    return prix;  
}
```

```
private boolean absent;  
  
public void setAbsent(boolean absent){  
    this.absent=absent;  
}  
public boolean isAbsent(){  
    return absent;  
}
```

## Variable d'instance:

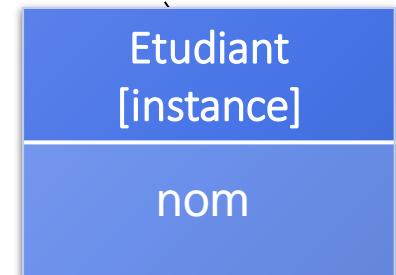
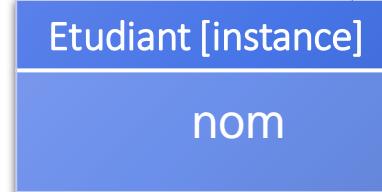
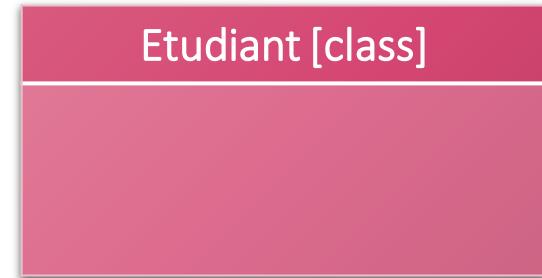
Chaque instance de la classe possède ses propres valeurs des variables.

```
Class Etudiant {
```

```
    String nom;
```

```
    Etudiant(String nom){  
        this.nom=nom;  
    }
```

```
}
```



```
Etudiant etud2 = new Etudiant ("Marwa");
```

# Les attributs static

2/4

```
Class Etudiant {  
    String nom;  
  
    Etudiant(String nom){  
        this.nom=nom;  
    }  
}
```

```
class Test {  
    public static void main(String[] args){  
  
        Etudiant etudiant =new Etudiant();  
  
        System.out.println(etudiant.nom);  
    }  
}
```

On invoque les variables d'instance avec le nom de l'instance

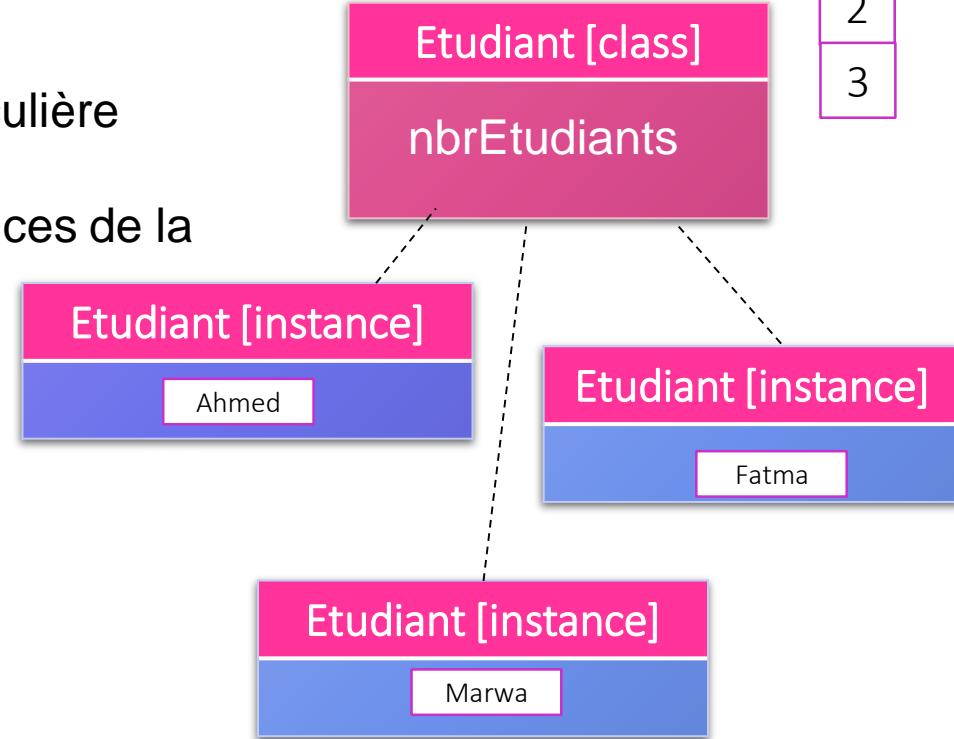
0
1
2
3

## Variab le de classe:

- n'appartient pas à une instance particulière
- elle appartient à la classe.
- elle est partagée par toutes les instances de la classe

Class Etudiant {

```
String nom;  
static int nbrEtudiants;  
  
Etudiant(String nom){  
    this.nom=nom;  
    nbrEtudiants++;  
}
```



```
Etudiant etud1 = new Etudiant ("Ahmed");  
Etudiant etud2 = new Etudiant ("Marwa");  
Etudiant etud3 = new Etudiant ("Fatma");
```

## Variable de classe:

```
class Etudiant{  
    String nom;  
static int nbrEtudiants;  
  
    Etudiant(String nom){  
        this.nom=nom;  
        nbrEtudiants++;  
    }  
}
```

```
class Test{  
    public static void main(String[] args){  
  
        System.out.println(Etudiant.nbrEtudiants);  
    }  
}
```

On invoque les variables **static** avec le nom de la classe

# Les méthodes static 1/2

Le comportement d'une méthode statique ne dépend pas de la valeur des variables d'instance

```
class MaClassMath {  
  
    static int min(int a , int b){  
        if(a<b){  
            return a;  
        }else{  
            return b;  
        }  
    }  
}
```

## Utilisation:

L'appel à une méthode statique se fait en utilisant le nom de la classe.

```
class Test {  
    public static void main ( String [] args ){  
        int x = MaClassMath .min (21 ,4);  
    }  
}
```

# Les méthodes static 2/2

Puisque les méthodes static appartiennent à la classe, elles ne peuvent en aucun cas accéder aux variables d'instances qui appartiennent aux instances de la classe.

- Les méthodes d'instances accèdent aux variables d'instance et méthodes d'instances
- Les méthodes d'instances accèdent aux variables de classe (static) et méthodes de classe (static)
- Les méthodes de classe (static) accèdent aux variables de classe (static) et méthodes de classe (static)
- Les méthodes de classe (static) n'accèdent pas aux variables de d'instance et méthodes d'instance
- Dans une méthode static on ne peut pas utiliser *this*

# La méthode `toString`

- La méthode `toString` est définie dans la classe `Object`
- La méthode `toString` définie dans la classe `Object` renvoie le nom de la classe de l'objet concerné suivi de l'adresse de cet objet.
- Quand on redéfinit la méthode `toString`, on fait en sorte qu'elle renvoie une chaîne de caractères servant à décrire l'objet concerné.

```
class Chemise{  
    int id;  
    char couleur;  
    float prix;  
    String description;  
    int quantite;  
  
    public String toString() {  
        return id+ " "+  
            couleur + " "+  
            prix + " "+  
            description + " "+  
            quantite;  
    }  
}
```

# Introduction à la Programmation Objet : Interface

## Jeu vidéo

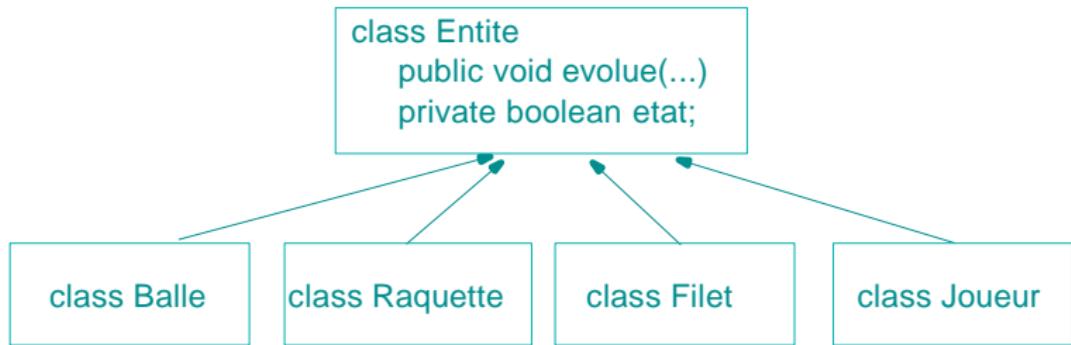
Supposons que l'on souhaite programmer un jeu mettant en scène les entités suivantes :

- 1.Balle
- 2.Raquette
- 3.Filet
- 4.Joueur

Chaque entité sera principalement dotée :

1. d'une méthode `evolve`, gérant l'évolution de l'entité dans le jeu;
2. d'un état indiquant si l'entité participe au jeu;

# Première ébauche de conception (1)



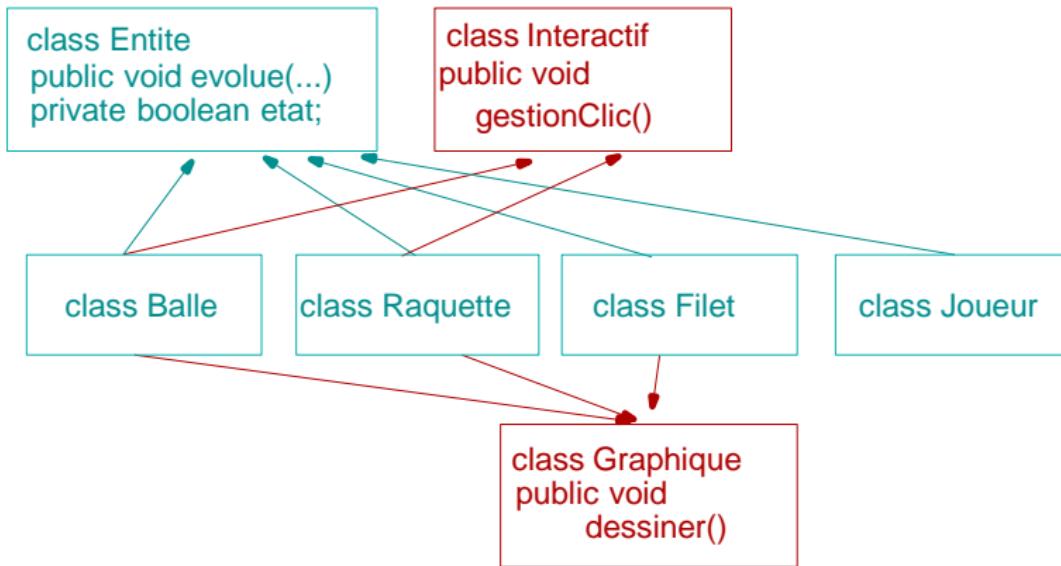
## Première ébauche de conception (2)

Si l'on analyse de plus près les besoins du jeu, on réalise que :

- ▶ Certaines entités doivent avoir une représentation graphique :
    - ▶ *Balle, Raquette, Filet*
  - ▶ ... et d'autres non
    - ▶ *Joueur*
  - ▶ Certaines entités doivent être interactifs (on veut pouvoir les contrôler avec la souris par exemple) :
    - ▶ *Balle, Raquette*
  - ▶ ... et d'autres non
    - ▶ *Joueur, Filet*
- + Comment organiser tout cela ?

# Jeu vidéo impossible

Idéalement, il nous faudrait mettre en place une hiérarchie de classes telle que celle-ci :



Mais . . . **Java ne permet que l'héritage simple** : chaque sous-classe ne peut avoir qu'une seule classe parente directe !

# Héritage simple/multiple

- ▶ Pourquoi pas d'héritage multiple en Java ?
  - ▶ Parfois difficile à compiler et à comprendre
- ▶ Si une variable/méthode est déclarée dans plusieurs super-classes :
  - ▶ Ambiguïté : laquelle hériter ?

## Analyse

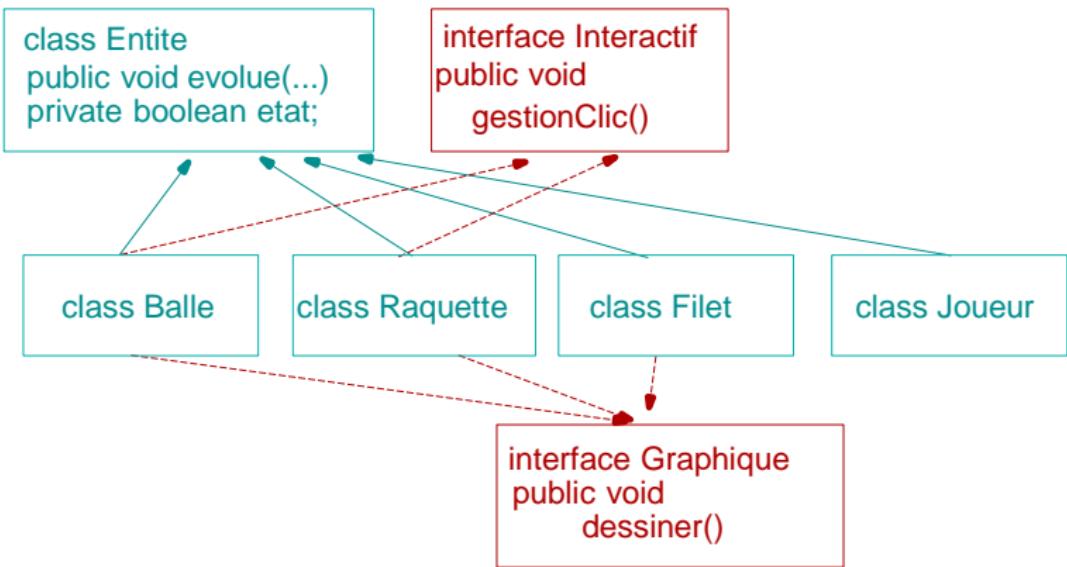
Mais en fait, que souhaitait-on utiliser de l'héritage multiple dans le cas de notre exemple de jeu vidéo ?

- + **Le fait d'imposer à certaines classes de mettre en oeuvre des méthodes communes**

Par exemple :

- ▶ `Balle` et `Raquette` doivent avoir une méthode `gestionClic`;
- ▶ mais `gestionClic` ne peut être une méthode de leur super-classe (car n'a pas de sens pour un `Joueur` par exemple).
- + Imposer un contenu commun à des sous-classes en dehors d'une relation d'héritage est le rôle joué par la notion d'`interface` en Java.

# Alternative possible de jeu vidéo



- ▶ Interface  $f = \text{Classe}$
- ▶ Une interface permet d'imposer à certaines classes d'avoir un contenu particulier sans que ce contenu ne fasse partie d'une classe.

# Interfaces (1)

Une **interface** permet d'abord d'attribuer un type supplémentaire à une classe :

- ▶ Interface = type supplémentaire  $\neq$  classe
- ▶ Solution Java pour imposer un contenu commun à des classes en dehors d'une relation d'héritage

Déclaration :

- ▶ Pas de méthode constructeur  
(car interface  $\neq$  classe)
- ▶ Eventuellement des constantes
- ▶ Eventuellement des méthodes abstraites
- ▶ Rien d'autre

Exemple :

```
interface Graphique {  
}  
interface Interactif {
```

## Interfaces(2)

Attribution d'une interface à une classe :

```
class Filet extends Entite
    implements Graphique {
    ... code habituel
}
```

# Test de type avec une interface

Test du type d'un objet avec `instanceof` :

```
Filet filet = new Filet(...);
boolean b;
b = (filet instanceof Entite); // true
b = (filet instanceof Graphique); // true !
b = (filet instanceof Interactif); // false
```

# Variable de type interface

- ▶ Une interface n'est pas une classe :
  - ▶ Pas de méthode constructeur
  - ▶ Impossible de faire `new`
- ▶ Une interface attribue un type supplémentaire à une classe d'objets :
- ▶ Par contre, on peut :
  - ▶ Déclarer une variable de type interface
  - ▶ Y affecter un objet d'une classe qui implémente l'interface
  - ▶ Faire un transtypage explicite vers l'interface

```
Graphique graphique;  
Balle balle = new Balle(..);  
graphique = balle;  
Entite entite = new Balle(..);  
graphique = (Graphique)entite; // Cast
```

# Plusieurs interfaces

- ▶ Une classe peut implémenter plusieurs interfaces (mais étendre une seule classe)
  - ▶ Séparer les interfaces par des virgules

```
class Balle extends Entite
implements Graphique, Interactif {
... code habituel
}
```

- ▶ On peut déclarer une hiérarchie d'interfaces :
  - ▶ Mot-clé `extends`
  - ▶ La classe qui implémente une interface reçoit aussi le type des super-interfaces

```
interface Interactif { ...}
interface GerableParSouris
    extends Interactif { ...}
interface GerableParClavier
    extends Interactif { ...}
```

# Interface – Résumé (1)

Une interface est un moyen d'attribuer des composants communs à des classes non-liées par une relation d'héritage :

- + Ses composants seront disponibles dans chaque classe qui l'implémente

Composants possibles (Java 7) :

## 1. Variables statiques finales (*assez rare*)

- + Ambiguïté possible, nom unique exigé

## 2. Méthodes abstraites (*courant*)

- + Chaque classe qui implémente l'interface sera obligée d'implémenter chaque méthode abstraite déclarée dans l'interface
- + Une façon de garantir que certaines classes ont certaines méthodes, sans passer par des classes abstraites
- + Aucune ambiguïté car sans instructions

# Evolution des interfaces

Les interfaces jusqu'à Java 7 ne peuvent contenir que :

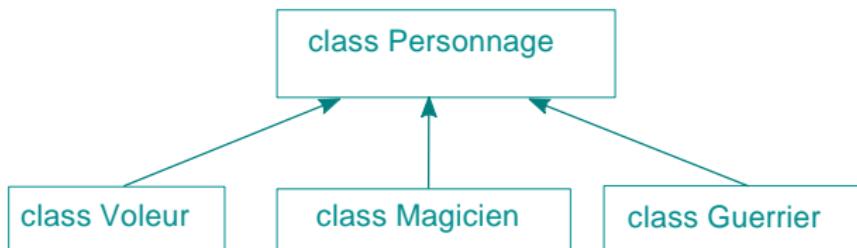
- ▶ des constantes
- ▶ des méthodes abstraites

Nouveautés depuis Java 8, elles peuvent aussi contenir :

1. des définitions par défaut pour les méthodes
2. des méthodes statiques

# Illustration

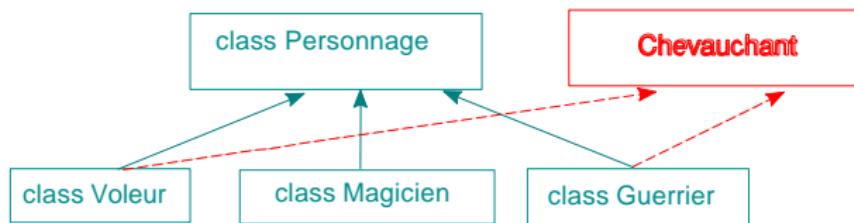
Reprendons notre exemple avec les personnages :



Supposons maintenant que certains personnages puissent chevaucher des montures.

# Illustration

On pourrait donc imaginer la conception suivante :



# Illustration

Pour le contenu de l'interface `Chevauchant`, on pourrait avoir :

```
interface Chevauchant
{
    // effectue le déplacement en chevauchant:
    void seDeplace();
    // peut-on descendre de la monture ou pas ?
    boolean peutDescendre();
}
```

Supposons que dans la plupart des cas un personnage chevauchant une monture ne puisse pas en descendre ...

## Méthode avec définition par défaut

Depuis Java 8, il est possible de donner une définition par défaut à certaines méthodes :

```
interface Chevauchant
{
    void seDeplace();
    Default boolean peutDescendre() { return false; }
}
```

# Méthode avec définition par défaut

## Syntaxe :

```
interface UneInterface  
{  
    default définition par défaut de la méthode  
}
```

Cette nouveauté soulève de nouvelles problématiques :

- ▶ quelles sont les règles d'utilisation des méthodes avec définition par défaut ?
- ▶ comment gérer les ambiguïtés pouvant se produire lors de définitions multiples (par des classes ou des interfaces) ?

# Règle 1 : héritage et définition

Les définitions par défaut des méthodes s'héritent et peuvent être redéfinies

plus bas dans les hiérarchies d'interfaces :

```
interface Cavalier extends Chevauchant
{
    defaultvoid seDeplace() { System.out.println("au trot"); }
}
```

- + Inutile de redonner une définition par défaut à `peutDescendre` si on en est satisfait.

Il est aussi possible de le faire si on le souhaite :

```
ninterface Cavalier extends Chevauchant
{
    defaultvoid seDeplace() { System.out.println("au trot"); }
    defaultboolean peutDescendre() { returntrue ; }
}
```

## Règle 2 : redéfinitions inutiles dans les classes

Une classe n'est plus obligée de redéfinir les méthodes des interfaces qu'elle implémente si celles-ci ont une définition par défaut

- + Si la classe `Guerrier` implémente l'interface `Cavalier` elle est instantiable en l'état

Elle peut néanmoins le faire si nécessaire :

```
class Guerrier extends Personnage implements Cavalier
{
    /**
     * ...
     */
    @Override
    boolean peutDescendre { return false; }
}
```

## Règle 3 : les classes ont la précédence (1)

En cas d'ambiguïté entre une définition faite dans une classe et une définition par défaut dans une interface, c'est la méthode de la classe qui a la priorité :

```
class Personnage
{
    publicvoid seDeplace() {
        System.out.println("en courant");
    }
}
interface Cavalier extends Chevauchant
{
    defaultvoid seDeplace() {
        System.out.println("au trot");
    }
}
class Guerrier extends Personnage implements Cavalier {}
```

- + Appeler `seDeplace` sur une instance de `Guerrier` invoque la méthode de `Personnage`

## Règle 3 : les classes ont la précédence (2)

La classe `Guerrier` peut bien sûr redéfinir la méthode `seDeplace` pour plutôt choisir celle de l'interface :

```
class Guerrier extends Personnage implements Cavalier
{
    public void seDeplace() {
        Cavalier.super.seDeplace();
    }
}
```

## Règle 4 : les classes doivent lever les ambiguïtés (1)

Les conflits entre définitions par défaut sont désormais possibles :

```
interface Dragonier extends Chevauchant
{
    Default void seDeplace() { System.out.println("vole"); }
}

interface SeTeleporte
{
    Default void seDeplace { System.out.println("plop !"); }
}

class MageUltime extends Magicien implements Dragonier,
    // conflit sur la definition de seDeplace !
}
```

- + Les classes qui implémentent des interfaces conflictuelles doivent lever l'ambiguïté

## Règle 4 : les classes doivent lever les ambiguïtés (2)

La classe `MageUltime` doit lever l'ambiguïté :

```
class MageUltime extends Magicien implements Dragonier,  
                      SeTeleporte  
{  
    // un MageUltime se déplace comme un Dragonier  
    Public void seDeplace { Dragonier.super.seDeplace(); }  
}  
  
class MageUltime extends Magicien implements Dragonier,  
                      SeTeleporte  
{  
    // ou se teleporte  
    Public void seDeplace { SeTeleporte.super.seDeplace(); }  
}
```

## Règle 4 : les classes doivent lever les ambiguïtés (2)

La classe `MageUltime` doit lever l'ambiguïté :

```
class MageUltime extends Magicien implements Dragonier,  
                                SeTeleporte  
{  
    // ou se deplace d'une maniere qui lui est propre  
    public void seDeplace {  
        if (peutDescendre()) {  
            SeTeleporte.super.seDeplace();  
        } else {  
            Dragonier.super.seDeplace();  
        }  
    }  
}
```

## Interface ou classe abstraite ?

La différence majeure est que les interfaces ne permettent pas de modéliser des **états** (attributs).

- + Elles sont à privilégier s'il n'y a pas d'état.

## Interface – Résumé (2)

Nous avons vu que l'héritage permet de mettre en place une relation de type "**EST-UN**" ("IS-A") entre deux classes.

Lorsqu'une classe a pour attribut un objet d'une autre classe, il s'établit entre les deux classes une relation de type "**A-UN**" ("HAS-A"), moins forte que l'héritage (on parle de délégation).

Une interface permet d'assurer qu'une classe se conforme à un certain **protocole**.

Elle met en place une relation de type "**SE-COMPORTE-COMME**" ("BEHAVES-AS-A") : une **Balle** est une entité du jeu, elle se-comporte-comme un objet graphique et comme un objet interactif.

# **Programmation Orientée Objet**

**Chapitre 2 : Classe et Objet(Partie 1)**

# PLAN

- Introduction
- **Classe et objet**
- Encapsulation
- Héritage
- Polymorphisme
- Exceptions
- Interfaces
- Collection
- Interface Fonctionnelle
- Expression Lambda
- Stream

## Objectifs du chapitre

- ✓ Notion de classe et d'objet
- ✓ Déclaration de classe
- ✓ Déclarations des attributs et des méthodes
- ✓ Les types des variables (primitives et objets)
- ✓ Notion de référence
- ✓ Les constructeurs

- Etude de cas: Gestion des Elections en Tunisie

Approche procédurale

vs

OO

## Raisonnement par Approche Procédurale

- ✓ Que fait le système ?
- ✓ Comment ajouter un candidat aux élections ?
- ✓ Comment gérer les votes de citoyens ?

## Raisonnement par Approche OO

- ✓ De quoi le système est composé?
- ✓ On doit décortiquer le Système en classes :
  - Candidat
  - Vote
  - Citoyen

## POO: Identification des classes

- les classes sont les composants fondamentaux de la POO

Candidat

- C'est une unité de base de la programmation orientée objet et représente les entités de la vie réelle.

Citoyen

Vote

Systeme de Gestion des Elections en Tunisie

# Notion classe et d'objet

- Le concept d'utilisation de classes et d'objets consiste à encapsuler l'état et le comportement dans une seule unité de programmation.
- Les objets Java sont similaires aux objets du monde réel.
- Par exemple, nous pouvons créer une classe voiture en Java, qui aura des propriétés telles que la vitesse actuelle et l'immatriculation ; et un comportement comme: rouler et changerPneu....



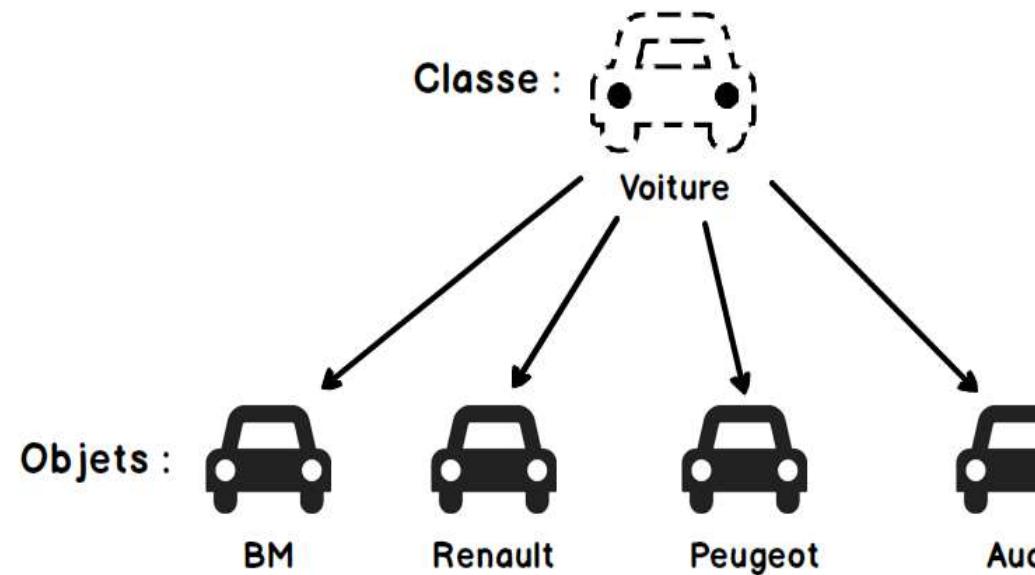
Voiture

# Notion classe et d'objet

- **Qu'est ce qu'une classe?**

Une classe est un plan ou un prototype défini par l'utilisateur à partir duquel des objets sont instanciés.

Il représente l'ensemble des propriétés ou méthodes communes à tous les objets d'un même type.



## Notion classe :

### Nom de la classe

Lorsque vous créez une classe java, vous devez suivre cette règle: le nom du fichier et le nom de la classe doivent être les mêmes.

Voiture écrit avec une majuscule V n'est pas la même chose que voiture, écrit avec une minuscule v.

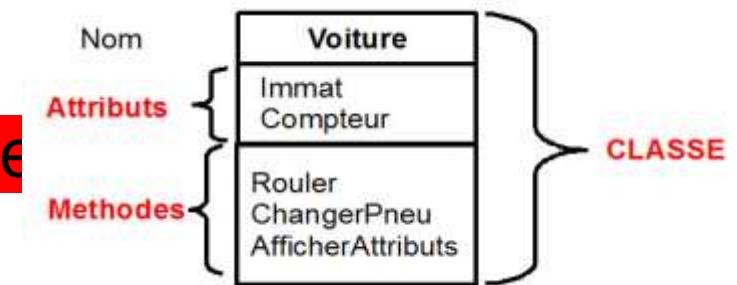
```
public class Voiture {  
  
    int vitesse;  
    String model;  
  
    public Voiture(String model) {  
        this.model = model;  
    }  
  
    public void accelerer() {  
        // ajoute 10 miles par heure à la vitesse actuelle  
        vitesse = vitesse + 10;  
    }  
  
    public void freiner() {  
        // déduire 10 miles par heure à la vitesse actuelle  
        vitesse = vitesse - 10;  
    }  
}
```

## Notion classe : Noms de la classe

Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule.

En réalité on dit qu'un objet est une instantiation d'une classe

objet = instance



Une classe est composée de deux parties :

- Les attributs (parfois appelés données membres) :
  - il s'agit des données représentant l'état de l'objet
- Les méthodes (parfois appelées fonctions membres): il s'agit des opérations applicables aux

# Manipulation de variables

## Quel nom choisir pour notre variable?

**Syntaxe:**

```
type nom_variable =valeur;
```

```
int id = 0;
```

Les noms des variables sont case-sensitive

Les espaces ne sont pas permis

Le nom de la variable doit commencer par une lettre minuscule.

# Mots clés JAVA

12

Évitez les mots réservés de java tel que :

abstract	double	int	strictfp	**
boolean	else	interface	super	
break	extends	long	switch	
byte	final	native	synchronized	
case	finally	new	this	
catch	float	package	throw	
char	for	private	throws	
class	goto	*	protected	transient
const	*	if	public	try
continue	implements	return		void
default	import	short		volatile
do	instanceof	static	while	

\* Indique un mot clé qui est peu utilisé

\*\* A partir de la plate-forme Java2

# Déclaration des variables

## ▪ Constante

- On déclare une constante avec le mot final

Exemple:

```
final int CAPACITE=2000;
```

- Le nom de la constante doit être en majuscule.

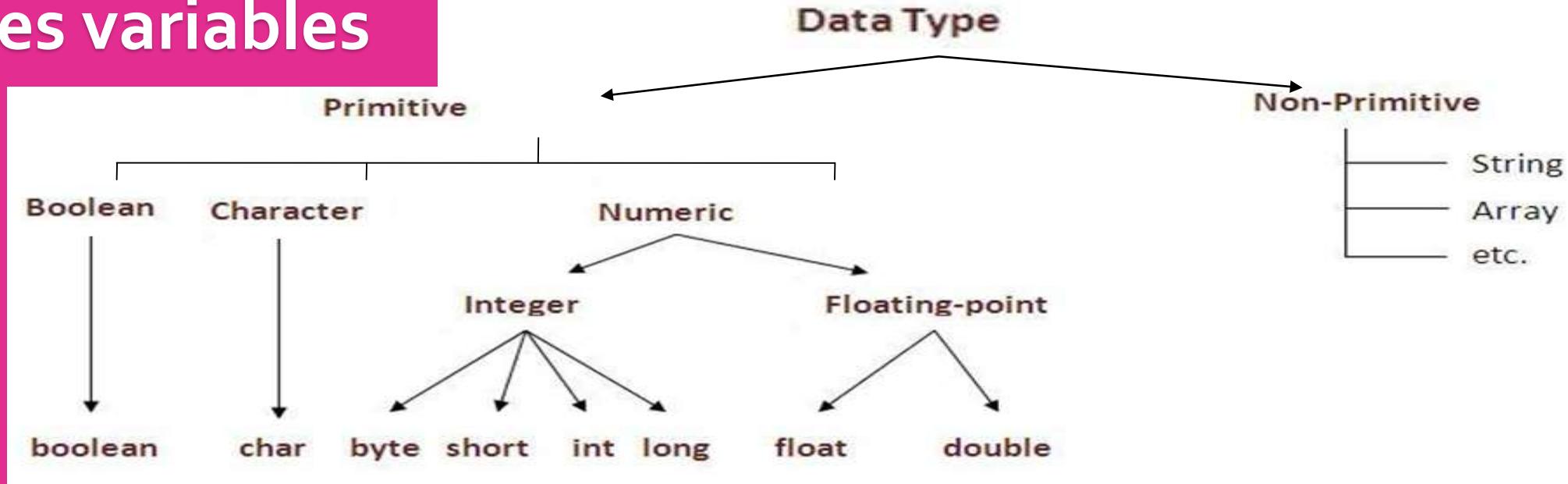
- Si le nom est composé de plusieurs mots, on utilise \_ pour la séparation des mots

Exemple:

```
final int TAILLE;
```

```
final int MAX_STOCK;
```

# Déclaration des variables



Il existe deux sortes de variables :

- Une variable primitive contient des bits qui représentent sa valeur.  
Exemple : int, float , boolean.
- Une variable référence contient des bits qui indiquent comment accéder à l'objet.

# Déclaration des variables

## Les variables primitives

- Déclaration d'une variable primitive

```
int age;  
boolean inscrit;  
char genre;
```

Réservation de l'espace mémoire.

age

inscrit

sexe

- Affectation de valeur

```
age=18;  
inscrit=true;  
genre='M';
```

• Variable sera stockée dans l'espace mémoire réservé.

age

Inscrit

genre

18

true

M

# Déclaration des variables

## Les types entiers

	Valeur minimale	Valeur maximale	Codé sur
<b>byte</b>	- 128	127	8 bits
<b>short</b>	- 32 768	32767	16 bits
<b>int</b>	- 2 147 483 648	2 147 483 647	32 bits
<b>long</b>	-9223372036854775808	9223372036854775807	64 bits

## Les types réels

	Valeur minimale	Valeur maximale	Codé sur
<b>float</b>	1.4E45	3.4028235E38	4 octets
<b>double</b>	4.9E324	1.7976931348623157E308	8 octets

# Déclaration des variables

## Le type caractère char

```
char sexe='M';
```

## Le type boolean

```
boolean Inscrit=true;
```

## Le type chaîne de caractère String

```
String message = "Hello Word";
```

# Déclaration des variables

## Les valeurs par défaut des variables

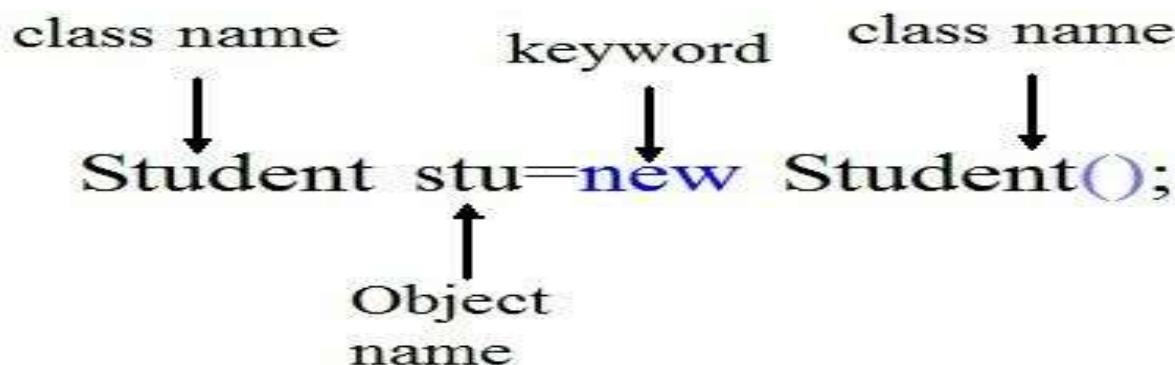
Type	Valeur par défaut
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

# Déclaration d'une classe et Manipulation des constructeurs

# Instanciation de Classe

Pour instancier une classe, c'est-à-dire créer un objet à partir d'une classe, il s'agit d'utiliser l'opérateur **new**.

En réalité l'opérateur *new*, lorsqu'il est utilisé, fait appel à une méthode spéciale de la classe: **le constructeur**.



# Le constructeur

- Le rôle d'un constructeur est de déclarer et de permettre d'initialiser les données membres de la classe, ainsi que de permettre différentes actions (définies par le concepteur de la classe) lors de l'instanciation.
- Un constructeur se définit comme une méthode standard, mais ne renvoie aucune valeur.
- Ainsi, le constructeur d'un objet porte le même nom que la classe et ne possède aucune valeur de retour (même pas *void*).

# Le constructeur

- un constructeur porte le même nom que la classe dans laquelle il est défini
- un constructeur n'a pas de type de retour (même pas *void*)
- un constructeur peut avoir des arguments
- la définition d'un constructeur n'est pas obligatoire lorsqu'il n'est pas nécessaire

# Le constructeur

## ▪ Constructeur par défaut

```
Candidat(){}
```

```
Candidat(){  
    id=0;  
    nom='BA'  
    NBVote=10.2f;  
}
```

## ▪ Constructeur surchargé

```
Candidat(int id, String couleur, float vote){  
    this.id=id;  
    this.nom=nom;  
    this.vote=vote;  
}
```

- Le constructeur par défaut initialise les variables de la classe aux valeurs par défaut.
- Si vous ne créez pas un constructeur dans votre classe, le compilateur va automatiquement vous créer un constructeur par défaut implicite
- Si le constructeur surchargé est créé, le constructeur par défaut implicite ne sera plus créé par le compilateur
- La plateforme java différencie entre les différents constructeurs déclarés au sein d'une même classe en se basant sur le nombre des paramètres et leurs types.

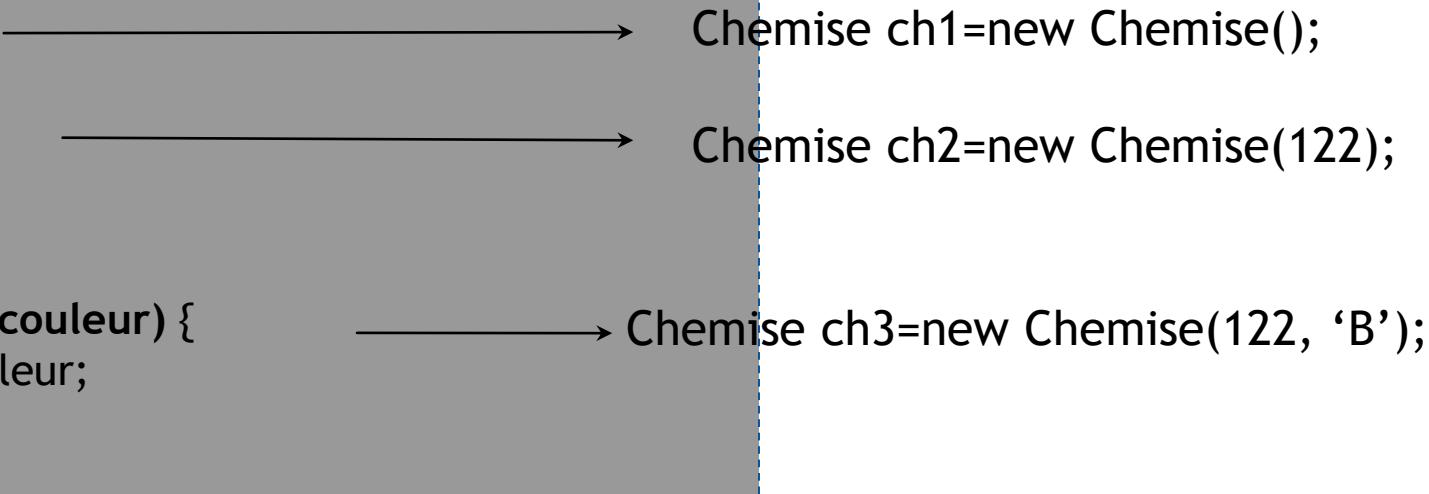
On ne peut pas créer deux constructeurs ayant le même nombre et types des paramètres.

# Le constructeur

Quel constructeur va être déterminé lorsque vous allez créer votre objet ?

```
class Chemise{  
    int id;  
    char couleur;  
    float prix;  
    String description;  
    int quantite;  
  
    Chemise () {}  
  
    Chemise(int id) {  
        this.id=id;  
    }  
  
    Chemise(int id, char couleur) {  
        this.couleur=couleur;  
    }  
}
```

## Utilisation:



# Utilisation de This

- Le mot-clé *this* permet de désigner l'objet courant,
- Pour manipuler un attribut de l'objet courant:

`this.couleur`

- Pour manipuler une méthode de la classe courante :

`this.ajouterChemise (100)`

- Pour faire appel au constructeur de l'objet courant: `this()`

```
class Chemise{  
    int id;  
    char couleur;  
  
    Chemise(int id) {  
        this.id=id;  
    }  
  
    Chemise(int id, char couleur) {  
        this.couleur=couleur;  
    }  
}
```

# Manipulation de Méthodes

# Déclaration des méthodes

- Le nom de la méthode doit commencer par un verbe
- Une méthode est une fonction faisant partie d'une classe.
- Elle permet d'effectuer des traitements sur (ou avec) les données membres des objets.

## Syntaxe:

```
Niveau d'accès Type_retour nom_method([arguments])  
{  
}  
}
```

# Appel des méthodes

Pour exécuter une méthode, il suffit de faire appel à elle en écrivant l'objet auquel elle s'applique (celui qui contient les données), le nom de la méthode (en respectant la casse), suivie de ses arguments entre parenthèse :

```
objet.nomDeLaMethode(argument1,argument2);
```

Le passage d'arguments à une méthode se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la méthode.

Le nombre et le type d'arguments dans la déclaration, le prototype et dans l'appel doit correspondre, sinon, on risque de générer une erreur lors de la compilation...



Merci pour votre attention



# LA CLASSE OBJECT

# LA CLASSE OBJECT

” Toutes les classes en java héritent de la classe **Object**.



La classe Object est la classe parente de toutes les classes en java



La classe Object permet donc de présenter le comportement minimal de tout objet ainsi que le code par défaut pour ce comportement minimal.



Quand on déclare une classe en java sans hériter explicitement de la classe Object, le compilateur se charge de le rajouter.



Toutes les méthodes de la classe Object sont donc disponibles dans toutes les classes Java existantes.

# Les méthodes de la classe Object

```
public class Object {  
    public Object() {...} // constructeur  
    public String toString() {...}  
    protected native Object clone() throws CloneNotSupportedException {...}  
    public equals(java.lang.Object) {...}  
    public native int hashCode() {...}  
    protected void finalize() throws Throwable {...}  
    public final native Class getClass() {...}  
  
    // méthodes utilisées dans la gestion des threads  
    public final native void notify() {...}  
    public final native void notifyAll() {...}  
  
    public final void wait(long) throws InterruptedException {...}  
    public final void wait(long, int) throws InterruptedException {...}  
}
```

# Les méthodes les plus usuelles de la classe Object

- Dans ce cours, on va s'intéresser à quelques méthodes de la classe Object qui sont très usuelles :
  - `getClass()`
  - `toString()`
  - `equals()`

# La méthode getClass()

- La méthode getClass() retourne la classe d'un objet de type Class . Il existe donc une classe **Class**, qui modélise les classes Java.
- Parmi les méthodes les plus usuelles de la classe **Class**
  - **getName()** : retourne le nom de la classe incluant le package
  - **isInterface()** : Indiquer si la classe est une interface
  - **isArray()** : Indiquer si la classe est un tableau
  - **getSuperClass()** : Renvoyer la classe mère de la classe

# Exemple d'utilisation de la méthode getClass()

```
if (objet1.getClass()==objet2.getClass()) {  
    //traitement si la Classe de l'objet1 et l'objet2 est la même  
}
```

Il est possible d'utiliser getClass() pour vérifier si **la classe de deux objets est la même**. Si objet1 et objet2 ne sont pas dans la même hiérarchie de classe (lien d'héritage), cette instruction provoque une erreur à la compilation.

# La méthode `toString()`

La méthode `toString()` retourne le nom de la classe, suivie du caractère @, et une adresse en hexadécimal, qui est l'adresse mémoire où l'objet considéré est enregistré.

System.out prend en paramètre un objet de type String. → Java devrait convertir `rec1` en String → appel implicite de la méthode `toString` pour retourner la description de l'object `rec1`

The screenshot shows a Java application window. The code in the editor is:

```
2 public class Geometrie {  
3  
4    public static void main(String[] args) {  
5        Rectangle rec1 =new Rectangle(4, 10);  
6        System.out.println("La description du rectangle : " + rec1.toString());  
7    }  
8 }
```

The console output is:

```
<terminated> Geometrie (3) [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe 2020 13 |  
La description du rectangle : Rectangle@6d06d69c
```

A large blue double-headed arrow points from the text "C'est équivalent" to the code line `System.out.println("La description du rectangle : " + rec1);`.

The screenshot shows a Java application window. The code in the editor is identical to the one above:

```
2 public class Geometrie {  
3  
4    public static void main(String[] args) {  
5        Rectangle rec1 =new Rectangle(4, 10);  
6        System.out.println("La description du rectangle : " + rec1);  
7    }  
8 }
```

The console output is:

```
<terminated> Geometrie (3) [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe (19)  
La description du rectangle : Rectangle@6d06d69c
```

# Redéfinie la méthode `toString()`

Généralement, on redéfinie la méthode `toString()` pour afficher une description (les attributs) de l'objet

```
public class Rectangle {  
    //....  
    @Override  
    public String toString() {  
        return "Rectangle [largeur=" + largeur + ", hauteur=" + hauteur + "]";  
    }  
    //....  
}
```

```
public class Geometrie {  
    public static void main(String[] args) {  
        Rectangle rec1 =new Rectangle(4, 10);  
        System.out.println(rec1.toString());  
    }  
}
```

Console :

```
Rectangle [largeur=10.0, hauteur=4.0]
```

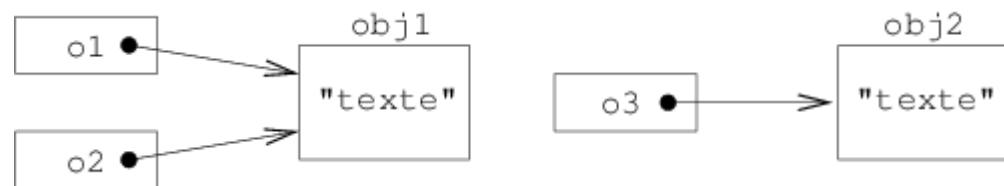
# La méthode equals()

Elle permet de **comparer** deux objets, et notamment de savoir s'ils sont égaux.

Telle qu'elle est implémentée dans la classe Object, cette méthode retourne true si les deux Objects pointent vers la même zone mémoire et false dans le cas contraire.

Les objets sont manipulés par des références ⇒

- `o1 == o2`
  - Teste si les références sont égales
- `o1.equals(o3)`
  - Teste si le contenu de l'objet référencé par o3 est comparable au contenu de l'objet référencé par o1



# Exemple de redéfinition de la méthode equals

```
class Rectangle
{
    private double hauteur;
    private double largeur;
    //...
@Override
public boolean equals(Object autre)
{
    if (this == obj) {
        return true;
    }
    if (!(autre instanceof Rectangle))
        return false;

    Rectangle autreRectangle = (Rectangle) autre;
    return (hauteur == autreRectangle.hauteur && largeur == autreRectangle.largeur);
}

//.. par exemple dans la méthode main()

Rectangle r1 = new Rectangle(4.0, 5.0); Rectangle r2 = new Rectangle(4.0, 5.0);

if (r1.equals(r2)) {
    System.out.println("Rectangles identiques");
}
```

# La méthode compareTo()

C'est une méthode abstraite de **l'interface Comparable**. La classe qui implémente cette interface devraient définir la méthode compareTo()

**Paramètre :** un autre objet du même type que la classe.

**Retour :**

- un entier négatif si objet1 plus petit que objet2
- un entier positif si objet1 plus grand que objet2
- 0 si objet1 est égal à objet2

Elle est appelée ainsi : objet1.compareTo(objet2)

# Exemple d'implémentation de la méthode compareTo()

```
public class Rectangle implements Comparable{
    //..

    @Override
    public int compareTo(Object autre) {
        int compare;
        if (this.surface() == ((Rectangle)autre).surface()) {
            compare = 0;
        } else if (this.surface() < ((Rectangle)autre).surface()) {
            compare = -1;
        } else {
            compare = 1;
        }
        return compare;
    }
    //..
}
```

# Généricité

- Toute classe dérive de la classe Object. Une variable de la classe Object peut contenir une référence de n'importe quel type d'objet grâce au polymorphisme.
- Ce qui nous permet de programmer d'une manière générique.
- En java une collection est un regroupement d'objets. On peut dire qu'un tableau est une forme de collection : on peut construire un tableau d'objets de type quelconque (universel)  
→On laisse le choix au programmeur de spécifier une classe différente comme classe d'objets stockés.

# Généricité

Donc on peut créer un tableau universel qui peut être utilisé pour stocker et gérer des objets de différentes classes : Employé, Article, ...

Le tableau sera un **tableau d'objet** de la classe **Object**.

**MAIS, SI ON VEUT UN TABLEAU D'ENTIER, DOUBLE, OU AUTRE TYPE SIMPLE !!**

**Ce tableau d'objet ne peut pas les contenir car ils ne sont pas des objets !**

**Solution : les classes enveloppes**

# Classes enveloppes ou emballages ou conteneurs ou wrapper class

Les classe enveloppes (*Wrappers*) sont des classes qui **encapsulent les données de type primitif**, afin de pouvoir les utiliser comme des objets ordinaires.

Type primitif	Classe enveloppe
boolean	Boolean
char	Character
byte	Byte    Sous-classes de la classe Number
short	Short
int	Integer
long	Long
float	Float
double	Double

# Classes enveloppes : Récupérer la valeur

Toutes les classe enveloppes possèdent des méthodes pour récupérer à leur valeur :

- *byte* **`byteValue()`**,
- *short* **`shortValue()`**,
- *int* **`intValue()`**,
- *long* **`longValue()`**,
- *double* **`doubleValue()`**

# Classes enveloppes : exemple

```
public class Wrapper {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int entier1 = 10, entier2;  
        double db1 = 15, db2;  
        char c1='B', c2;  
        Integer objetEntier;  
        Double objetDouble;  
        Character objetChar;  
  
        //Créer des objets de classes enveloppes  
        objetEntier = new Integer(entier1);  
        objetDouble = new Double(db1);  
        objetChar = new Character(c1);  
  
        //Utiliser la valeur des objets de classes enveloppes  
        entier2 = objetEntier.intValue();  
        db1 = objetDouble.doubleValue();  
        c2 = objetChar.charValue();  
    }  
}
```

# Exemple de tableau générique et son utilisation

Voir le cours du Christian Mongeon p13→p24 et le code de l'exemple Doc8Ex.

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Introduction à la Programmation : Héritage

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Objectifs du cours

- ▶ Introduire la notion d'**héritage** en POO
- ▶ Comment cette notion se met en pratique en Java
- ▶ Héritage et droits d'accès
- ▶ Constructeurs et héritage
- ▶ Masquage dans une hiérarchie de classes
- ▶ Introduction au polymorphisme (résolution dynamique des liens)

# Passons aux choses sérieuses ...

On considère le jeu suivant:



[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Ebauche de conception....

- ▶ Une classe pour le programme principal : *Jeu*
- ▶ Une classe pour représenter le *Joueur*
- ▶ Une classe pour représenter une *Partie* (avec le joueur et tous les personnages/avatars virtuels qu'il va rencontrer)
- ▶ Quatre classes pour des types d'avatars particuliers :
  - 1.des *Orcs*,
  - 2.des *Elfes*,
  - 3.des *Magiciens*
  - 4.des *Sorciers*.
- ▶ et plein de petites classes utilitaires pour représenter des accessoires : *Arme*, *Baguette* etc.

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Classes pour les Avatars

class Orc

String nom  
int energie  
int dureeVie

Arme arme

rencontrer(Joueur)

class Elfe

String nom  
int energie  
int dureeVie

rencontrer(Joueur)

class Magicien

String nom  
int energie  
int dureeVie

Baguette baguette

rencontrer(Joueur)

class Sorcier

String nom  
int energie  
int dureeVie

Baguette baguette

Baton baton

rencontrer(Joueur)

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

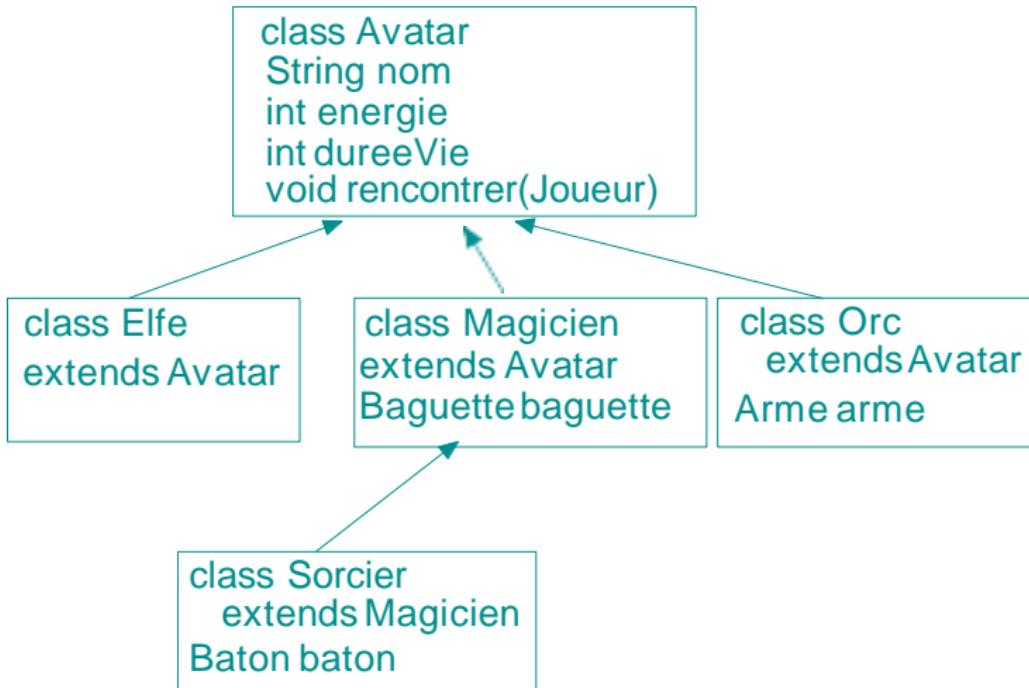
# Classes pour les Avatars



On duplique beaucoup de code d'une classe à l'autre (espace et temps perdus . . . mais surtout, problèmes de maintenance)!

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

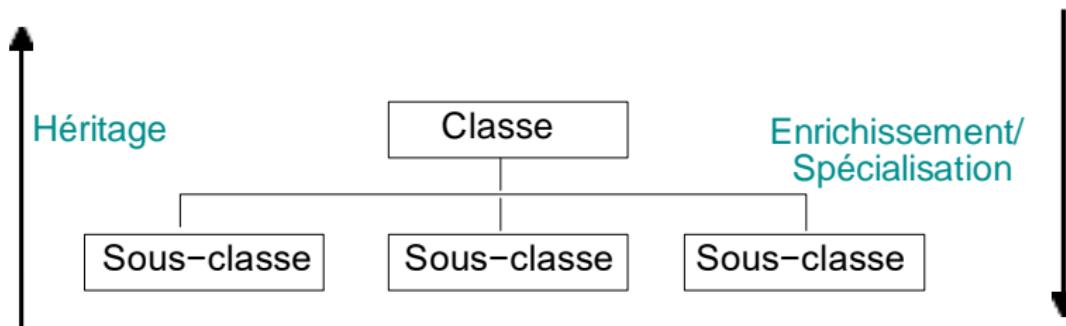
# Héritage



[Encours](#)[Prélude](#)[Héritage](#)[En Java](#)[Héritage et droits d'accès](#)[Constructeurs et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Héritage (1)

- ▶ Après les notions d'**encapsulation** et d'**abstraction**, le troisième aspect essentiel des objets est la notion d'**héritage**
- ▶ L'héritage est une **technique extrêmement efficace** pour créer des classes plus spécialisées, appelées **sous-classes**, à partir de classes plus générales déjà existantes, appelées **sur-classes**.



Plus précisément, lorsqu'une sous-classe **SousClasse** est créée à partir d'une classe **SuperClasse**, **SousClasse** va **hériter** de l'ensemble :

- ▶ des attributs de **SuperClasse**
- ▶ des méthodes de **SuperClasse**
- + Les attributs et méthodes de **SuperClasse**, aussi appelés *membres* de **SuperClasse**, vont être disponibles pour **SousClasse** sans que l'on ait besoin de les redéfinir explicitement dans **SousClasse**.

De plus, des attributs et/ou méthodes supplémentaires peuvent être définis par la sous-classe **SousClasse**

- + Ces nouveaux membres constituent l'**enrichissement** apporté par cette sous-classe.

[Encours](#)[Prélude](#)[Héritage](#)[En Java](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Héritage (3)

L'héritage permet donc :

- ▶ D'expliciter des relations structurelles et sémantiques entre classes.
- ▶ De réduire les redondances de description et de stockage des propriétés.

Attention, l'héritage doit être utilisé :

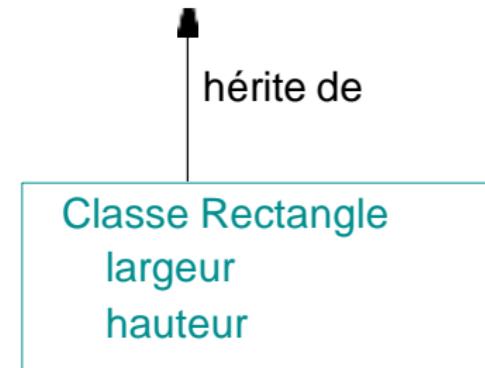
- ▶ pour décrire une relation "**est-un**" entre les classes ;
- ▶ il ne doit **jamais** décrire une relation "**a-un**" ("possède-un")

[Encours](#)[Prélude](#)[Héritage](#)[En Java](#)[Héritage et droits d'accès](#)[Constructeurs et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Héritage (4)

Par exemple, grâce à l'héritage, on peut étendre une classe `FormeGeometrique`, caractérisée par un attribut `position`, avec une sous-classe `Rectangle` ayant pour attributs `largeur` et `hauteur`.

Classe `FigureGeometrique`  
`position`



- + Un rectangle "**est-une**" forme géométrique
- + Une forme géométrique "**possède-une**" position

[Encours](#)[Prélude](#)[Héritage](#)[En Java](#)[Héritage et droits d'accès](#)[Constructeur et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Héritage (5)

Par **transitivité**, les instances d'une sous-classe possèdent :

- ▶ Les attributs et méthodes de l'ensemble des classes parentes (classe parente, classe parente de la parente etc ...)

La notion d'**enrichissement par héritage** :

- ▶ Crée un **réseau de dépendances** entre classes.
  - ▶ Ce réseau est organisé en une **structure arborescente** où chacun des noeuds hérite des propriétés de l'ensemble des noeuds du chemin remontant jusqu'à la racine.
- 
- + Ce réseau de dépendance définit une **hiérarchie de classes**

# Hiérarchie de classes

[Encours](#)

[Prélude](#)

[Héritage](#)

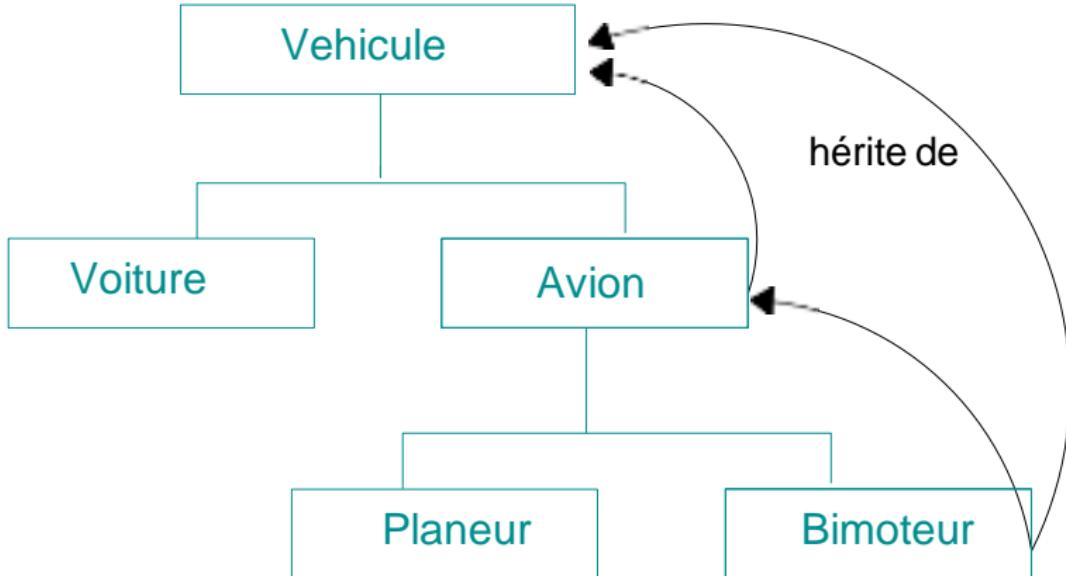
[En Java](#)

[Héritage et droits d'accès](#)

[Constructeurs et héritage](#)

[Masquage dans une hiérarchie](#)

[Polymorphisme](#)



[Encours](#)[Prélude](#)[Héritage](#)[En Java](#)[Héritage et droits d'accès](#)[Constructeur et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Super- et sous-classes

Une **super-classe** :

- ▶ est une classe "parente"
- ▶ déclare les variables/méthodes communes
- ▶ peut avoir plusieurs sous-classes

Une **sous-classe** est :

- ▶ une classe "enfant"
- ▶ étend **une seule** super-classe
- ▶ hérite des **variables**, des **méthodes** et du **type** de la super-classe

Une variable/méthode héritée peut s'utiliser :

- ▶ comme si elle était déclarée dans la sous-classe au lieu de la super-classe (en fonction des droits d'accès, voir plus loin)
- + On évite ainsi la **duplication de code**

[Encours](#)

[Prélude](#)

[Héritage](#)

[En Java](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Passons à la pratique...

Définition d'une sous-classe en Java :

## Syntaxe :

```
class NomClasseEnfant extends NomClasseParente
{
    /* Déclaration des attributs et méthodes
       spécifiques à la sous-classe */
    //...
};
```

## Exemple :

```
class Rectangle extends FormeGeometrique
{
    int largeur; // attributs spécifiques
    int hauteur;
    //...
```

# Accès aux membres d'une sous-classe

Jusqu'à maintenant, l'accès aux membres (attributs et méthodes) d'une classe pouvait être :

- ▶ soit **public** : visibilité totale à l'intérieur et à l'extérieur de la classe (mot-clé **public**)
- ▶ soit **privé** : visibilité uniquement à l'intérieur de la classe (mot-clé **private**)
- ▶ soit par défaut (aucun modificateur) : visibilité depuis toutes les classes du même paquetage (est aussi valable pour le paquetage par défaut que vous utilisez en TP)

Un troisième type d'accès régit l'accès aux attributs/méthodes au sein d'une hiérarchie de classes :

- ▶ l'accès **protégé** : assure la visibilité des membres d'une classe dans les classes de sa descendance (et dans autres classes du même paquetage). Le mot clé est «**protected**».

(Voir <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html> pour

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Droits d'accès protégé

- ▶ Une sous-classe n'a **pas de droit d'accès** aux membres (attributs ou méthodes) **privés** hérités de ses super-classes
  - + elle doit alors utiliser les getter/setters prévus dans la super-classe
- ▶ Si une super-classe veut permettre à ses sous-classes d'accéder à un membre donné, elle doit le déclarer non pas comme privé (**private**), mais comme protégé (**protected**).

**Attention :** La définition d'attributs protégés nuit à une bonne encapsulation d'autant plus qu'en Java un membre protégé est aussi accessible par toutes les classes d'un même paquetage

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Utilisation des droits d'accès

- ▶ Membres *publics* : accessibles pour les **programmeurs utilisateurs** de la classe
- ▶ Membres *protégés* : accessibles aux **programmeurs d'extensions** par héritage de la classe (ou travaillant dans le même paquetage)
- ▶ Membres *privés* : pour le **programmeur de la classe** : structure interne, **modifiable** si nécessaire **sans répercussions** ni sur les utilisateurs ni sur les autres programmeurs.

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et droits d'accès](#)[Constructeurs et héritage](#)[Construction de copie](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Constructeurs et héritage

Lors de l'instanciation d'une sous-classe, il faut initialiser :

- ▶ les attributs *propres à la sous-classe*
- ▶ les attributs *hérités des super-classes*

**MAIS...**

...il ne doit pas être à la charge du concepteur des sous-classes de réaliser lui-même l'*initialisation des attributs hérités*

L'**accès** à ces attributs peut notamment être **interdit** ! (*private*)

L'initialisation des attributs hérités doit donc se faire au niveau des classes où ils sont explicitement définis.

**Solution :** l'initialisation des attributs hérités doit se faire en invoquant les *constructeurs des super-classes*.

[Encours](#)  
[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeur et  
héritage](#)

[Construction de  
copie](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Constructeurs et héritage (2)

L'invocation du constructeur de la super-classe se fait au **début du corps du constructeur de la sous-classe** au moyen du mot clé **super**.

## Syntaxe :

```
SousClasse(liste d'arguments)
{
    super(...);
    // corps du constructeur
}
```

## Règles :

1. Chaque constructeur d'une sous-classe doit appeler **super(...)**
2. Les arguments fournis à **super** doivent être ceux d'au moins un des constructeur de la super-classe.
3. L'appel doit être la toute **1ère instruction**
4. Erreur si l'appel vient plus tard ou 2 fois

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et droits d'accès](#)[Constructeur et héritage](#)[Construction de copie](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Appel obligatoire à super(...) (2)



Et si l'on oublie l'appel à `super(...)` ?

- ▶ Appel automatique à `super()`
- ▶ Pratique parfois, mais erreur si le **constructeur sans paramètres** n'existe pas

Rappel : le constructeur sans paramètres est particulier

- ▶ Il existe par défaut pour chaque classe qui n'a aucun autre constructeur
- ▶ Il disparaît dès qu'il y a un autre constructeur

Pour éviter des problèmes avec les hiérarchies de classes :

- ▶ Toujours déclarer au moins un constructeur
- ▶ Toujours faire l'appel à `super(...)`

# Constructeurs et héritage : exemple (2)

```
class Rectangle {  
    private double largeur;  
    private double hauteur;  
    // il y a un constructeur par defaut !  
    public Rectangle()  
        { largeur =0; hauteur =0; }  
    // le reste de la classe...  
};  
  
class Rectangle3D extends Rectangle {  
    private double profondeur;  
    public Rectangle3D(double p)  
        {profondeur=p;}  
    // le reste de la classe...  
}
```

Ici il n'est pas nécessaire d'invoquer explicitement le constructeur de la classe parente puisque celle-ci admet un constructeur par défaut.

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeur et  
héritage  
Construction de  
copie](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Encore un exemple

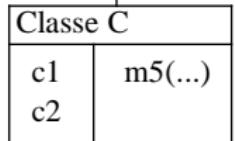
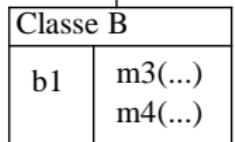
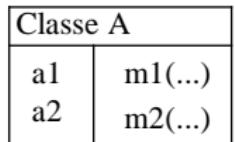
Il n'est pas nécessaire d'avoir des attributs supplémentaires...

```
class Carre extends Rectangle {  
  
    public Carre(double taille)  
    {  
        super(taille, taille);  
    }  
  
    // et c'est tout ! (sauf s'il y avait des  
    // "methodes set" dans Rectangle)  
}
```

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et droits d'accès](#)[Constructeur et héritage](#)[Construction de copie](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Ordre d'appel des constructeurs

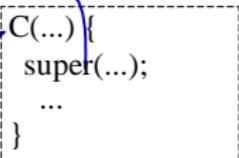
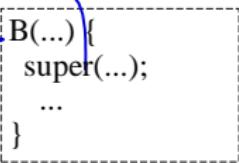
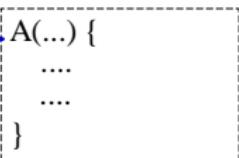
Hiérarchie de classes



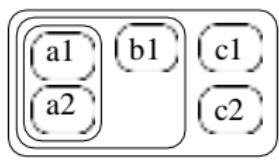
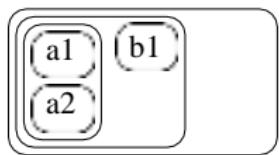
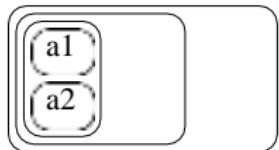
instanciation :

`C monC = new C(...);`

Constructeurs



Instance



monC

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeur et  
héritage](#)[Construction de  
copie](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# super(...) ≠ new

Nous connaissons maintenant deux façons d'appeler le constructeur d'une classe.

1. `new Rectangle(uneLargeur, uneHauteur)`

- ▶ Façon générale de construire un objet
- ▶ Réservation de mémoire + exécution des instructions

2. `super(largeur, hauteur)`

- ▶ invoqué uniquement par le constructeur d'une sous-classe de `Rectangle`
- ▶ Exécution des instructions seulement
- ▶ Réservation de mémoire déjà faite dans la sous-classe

[Encours](#)  
[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeur et  
héritage](#)

[Construction de  
copie](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# super(...) ≠ this(2)

```
class Rectangle
{
    private double largeur;
    Private double hauteur;

    public Rectangle()
    {
        largeur =0.0;
        hauteur =0.0;
    }

    public Rectangle(double uneLargeur, double uneHauteur)
    {
        largeur = uneLargeur;
        hauteur = uneHauteur;
    }

    public Rectangle(double uneValeur)
    {
        // invoque la constructeur precedent
        this(uneValeur, uneValeur);
    }
}
```

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeur et  
héritage](#)[Construction de  
copie](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Héritage et constructeur de copie

Le constructeur de copie d'une sous-classe doit invoquer explicitement le constructeur **de copie** de la super-classe

- + Sinon c'est le constructeur **par défaut** de la super-classe qui est appelé!!

## Exemple:

```
RectangleColore(RectangleColore autreRectangleColore)
{
    // Appel au constructeur de copie de la super-classe
    super(autreRectangleColore);
    // ....
}
```

[Encours](#)  
[Prélude](#)

[Héritage](#)

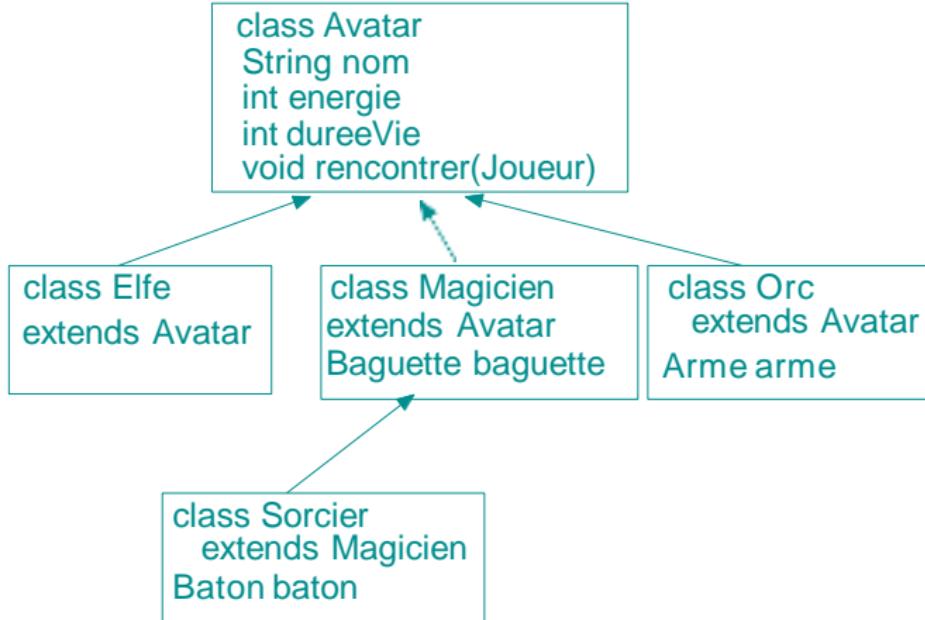
[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Appel à une méthode dans une hiérarchie



Comment se passe l'appel à la méthode `rencontrer(Joueur)` sur un objet de la sous-classe `Sorcier`?

# Appel à une méthode dans une hiérarchie

Exemple : appel à `rencontrer(Joueur)` sur une objet de type `Sorcier`

```
Joueur toto = new Joueur(...);  
Sorcier oz = new Sorcier(...);  
  
oz.rencontrer(toto);
```

1. Recherche de la méthode dans la classe de l'objet
2. Pas trouvée dans `Sorcier`
3. Recherche de la méthode dans la hiérarchie, en commençant par la super-classe directe
4. Pas trouvée dans `Magicien`
5. Trouvée dans `Avatar`, exécution de la méthode

Conclusions :

- ▶ C'est la variable/méthode de la classe **la plus proche** de l'objet qui sera utilisée
- ▶ et si un `Orc` avait une autre façon de rencontrer le joueur ?

[Encours](#)  
[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Les Orc font bande à part

## ► Pour un Orc

```
class Orc
{ //...
    private Arme monArme;
    Public void rencontrer(Joueur lePauvre)
    {
        frapper(lePauvre); // avec monArme bien sur
    }
}
```

## ► Pour tous les autres :

```
class Avatar
{ //...
    Public void rencontrer(Joueur unJoueur)
    {
        saluer(unJoueur);
    }
}
```

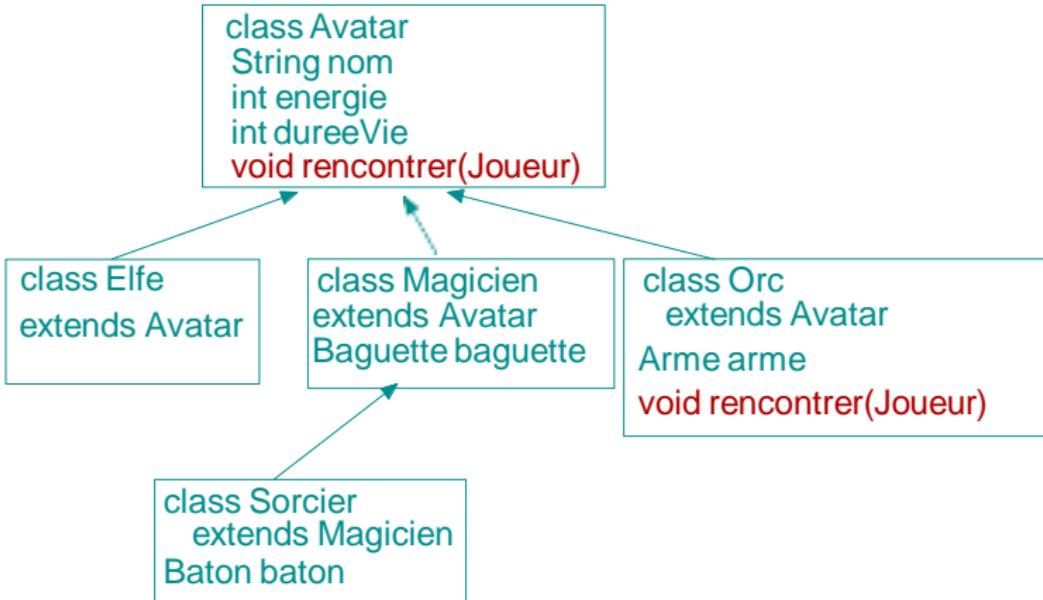


Faut-il re-concevoir toute la hiérarchie ?

- + Non, on ajoute simplement une méthode `rencontrer(Joueur)` spéciale dans la sous-classe `Orc`

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Les Orc font bande à part : Masquage



[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Masquage dans une hiérarchie

**Masquage** ("shadowing" pour les variables/ "overriding" pour les méthodes) :

- ▶ Même nom de variable utilisé sur plusieurs niveaux
- ▶ Même signature de méthode utilisée sur plusieurs niveaux
- ▶ Peu courant pour les variables (... et à éviter!)
- ▶ Très utiles pour les méthodes

[Encours](#)  
[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Masquage dans une hiérarchie : exemple simple

- ▶ Rectangle3D hérite de Rectangle
- ▶ calcul de la surface pour les Rectangle3D
$$\begin{aligned} & 2 * (\text{largeur} * \text{hauteur}) + 2 * (\text{largeur} * \text{profondeur}) \\ & + 2 * (\text{hauteur} * \text{profondeur}) \end{aligned}$$
- ▶ calcul de la surface pour tous les autres Rectangle :
$$(\text{largeur} * \text{hauteur})$$

[Encours](#)  
[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Masquage dans une hiérarchie : exemple simple (2)

```
class Rectangle
...
double surface() {
    return(largeur*hauteur);
}
```

```
class ... Rectangle3D extends Rectangle
double surface(){
    return(2*largeur*hauteur +
    2*largeur*...);
```

La méthode **surface** de **Rectangle3D** **masque** celle de **Rectangle**

- ▶ Un objet de type **Rectangle3D** n'utilisera donc **jamais** la méthode **surface** de la classe **Rectangle**
- ▶ Vocabulaire OO :
  - ▶ Méthode héritée = méthode générale, *méthode par défaut*
  - ▶ Méthode qui masque la méthode héritée = *méthode spécialisée*

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et droits d'accès](#)[Constructeurs et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Accès à une méthode masquée

- ▶ Il est parfois souhaitable d'accéder à une méthode/un attribut caché(e)
- ▶ Exemple :
  - ▶ surface des `Rectangle3D` ayant une profondeur nulle (`largeur*hauteur`)
    - + identique au calcul de surface pour les `Rectangle`
- ▶ Code désiré :
  1. Objet non- `Rectangle3D`:
    - ▶ Méthode générale (`surface de Rectangle`)
  2. Objet `Rectangle3D`:
    - ▶ Méthode spécialisée (`surface de Rectangle3D`)
  3. Objet `Rectangle3D` de profondeur nulle :
    - ▶ D'abord la méthode spécialisée
    - ▶ Ensuite appel à la méthode générale depuis la méthode spécialisée

## Accès à une méthode masquée (2)

- ▶ Pour accéder aux attributs/méthodes caché(e)s de la super-classe :

`super + . + variable/méthode`  
`super.reconstruire(leJoueur)`

```
class Rectangle3D extends Rectangle {  
    //... constructeurs, attributs comme avant  
  
    public double surface () {  
        if (profondeur == 0.0)  
            // Acces a la methode masquee  
            return super.surface();  
        else  
            return (2.0 * (largeur*hauteur)  
                    + 2.0 * (largeur*profondeur)  
                    + 2.0 * (hauteur*profondeur));  
    }  
}
```

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Les avatars rencontrent le joueur

```
public static void main(String[] args)
{
    Joueur leJoueur = new Joueur(...);
    Avatar[] lesAvatars = new Avatar[3];

    lesAvatars[0] = new Elfe(...); // Correct ?
    lesAvatars[1] = new Orc(...);
    lesAvatars[2] = new Sorcier(...);

    for (int i=0; i < lesAvatars.length; ++i)
    {
        lesAvatars[i].rencontrer(leJoueur);
    }
}
```



Peut-on mettre un **Sorcier** dans un tableau d'**Avatar** ?

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)

# Héritage du type des super-classes

Dans une hiérarchie de classes :

- ▶ Un objet d'une sous-classe hérite le type de sa super-classe
- ▶ L'héritage est transitif
- ▶ Un objet peut donc avoir **plusieurs types**

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et droits d'accès](#)[Constructeurs et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# L'opérateur instanceof

L'opérateur logique `instanceof` permet de **tester le type d'un objet** :

```
Sorcier oz = new Sorcier(...);
boolean b;
b = (oz instanceof Sorcier); // true
b = (oz instanceof Magicien); // true
b = (oz instanceof Avatar); // true
b = (oz instanceof Elfe); // false
```

Il est donc permis d'affecter `Sorcier` à une variable de type `Magicien` ou `Avatar` :

```
Sorcier oz = new Sorcier(...);
Magicien unMagicien = oz; // OK
Avatar unAvatar = oz; // OK
Orc unOrc = oz; // Erreur
```

# Choix de la méthode à exécuter

Supposons que la classe `Orc` redéfinisse `rencontrer(Joueur)` et soit le code suivant :

```
Joueur leJoueur = new Joueur(...);  
Avatar unAvatar = new Orc(...); // un objet de type Orc  
                                // est affect'e a' une  
                                // variable de type Avatar  
unAvatar.rencontrer(leJoueur);;
```

Quelle méthode `rencontrer(Joueur)` va être exécutée ?

En fait, la méthode à exécuter peut être choisie de 2 façons :

1. Résolution *statique* des liens :

- ▶ Le *type apparent*(type de la variable) est déterminant
- ▶ `unAvatar` est déclarée comme une *variable* de type `Avatar`
- ▶ Choix de la méthode de la classe `Avatar` (l'avatar salue le joueur)

2. Résolution *dynamique* des liens :

- ▶ Le *type effectif*(celui de l'objet effectivement stocké dans la variable) est déterminant
- ▶ `unAvatar` contient un *objet* de type `Orc`
- ▶ Choix de la méthode de la classe `Orc` (l'orc use de son arme sur le joueur)

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Résolution dynamique des liens

Java met en oeuvre le principe de la "**résolution dynamique des liens**"

- + C'est le type effectif et non le type apparent qui est pris en compte

Un petit exemple (cruel mais illustratif) sur les transparents suivants ...

[Encours](#)  
[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Résolution dynamique des liens - Exemple (1)

Soit la hiérarchie de classes suivante :

```
class Animal {  
    protected boolean mort;  
    ...  
    void mourir() {  
        mort = true;  
    }
```

```
class Chien {  
}
```

Le chien n'a pas de 2eme chance!

```
class Phenix {  
    void mourir() {  
        mort = false;  
    } }
```

Le phenix renait de ses cendres

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

# Résolution dynamique des liens - Exemple (2)

Soit maintenant la classe suivante :

```
class Assassin {  
    public tuer(Animal a1, Animal a2) {  
        a1.mourir();  
        a2.mourir();  
    }  
}
```

Que ce passe-t-il lors de l'exécution du code suivant ?

```
Assassin leMechant = new Assassin(...);  
Chien pauvreChien = new Chien(...);  
Phenix pauvrePhenix = new Phenix(...);  
leMechant.tuer(pauvreChien, pauvrePhenix);
```

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et droits d'accès](#)[Constructeurs et héritage](#)[Masquage dans une hiérarchie](#)[Polymorphisme](#)

# Résolution dynamique des liens - Exemple (3)

- ▶ Avec la résolution "statique" des liens, dans `Assassin.tuer`, ce serait toujours `Animal.mourir()` qui serait appelé (**c'est le type apparent des variables qui décide**)
  - + Le phénix meurt comme un vulgaire animal
- ▶ Avec la résolution "dynamique" des liens, dans `Assassin.tuer`, `Animal.mourir()` est appelée pour le chien (car il ne sait mourir que de cette façon) mais `Phenix.mourir()` est appelée pour le phénix (**c'est le type effectif qui décide**)
  - + Le chien meurt, le phénix survit!
  - + C'est ce qui va se passer en Java

[Encours](#)[Prélude](#)[Héritage](#)[Héritage et  
droits d'accès](#)[Constructeurs et  
héritage](#)[Masquage dans  
une hiérarchie](#)[Polymorphisme](#)[Résolution  
dynamique des  
liens](#)[Compléments :  
Paquetages](#)

# Polymorphisme

Les deux ingrédients :

- ▶ héritage du type dans une hiérarchie de classes,
- ▶ et résolution dynamique des liens

permettent de mettre en oeuvre ce que l'on appelle le **polymorphisme**.

- ▶ Un même code s'exécute de façon différente selon la donnée à laquelle il s'applique.
- + Nous y reviendrons plus en détail au cours prochain

[Encours](#)

[Prélude](#)

[Héritage](#)

[Héritage et  
droits d'accès](#)

[Constructeurs et  
héritage](#)

[Masquage dans  
une hiérarchie](#)

[Polymorphisme](#)

[Résolution  
dynamique des  
liens](#)

[Compléments :  
Paquetages](#)

# Au sujet de instanceof

**Attention :** L'opérateur `instanceof` est à utiliser avec précaution.

- + Il peut vous amener à ne pas utiliser le polymorphisme là où il faudrait, nous y reviendrons aussi au prochain cours

# Ce que j'ai appris aujourd'hui

- ▶ Que l'on peut réduire la duplication de code et reproduire de bon modèles de la réalité en utilisant des **hiérarchies de classe**
- ▶ Qu'une sous-classe hérite des membres de ses classes parentes ainsi que de leurs types
- ▶ Que la construction d'objets dans le cadre de l'héritage obéit à des règles précises (`super(...)` en Java)
- ▶ Qu'en Java, pour que la notion d'héritage soit établie il faut avoir recours à `extends`
- ▶ Par quel mécanisme on peut gérer le masquage de variables et de méthodes dans une hiérarchie de classes
- ▶ Que Java implémente le principe de **résolution dynamique des liens** : le type de l'objet, plutôt que le type de la variable qui référence l'objet détermine la méthode à exécuter
- + Je peux maintenant **améliorer la modularisation** de mes programmes orientés objet en Java



## Héritage



Spécifier un *lien d'héritage*:

```
class SousClasse extends SuperClasse {...}
```

*Droits d'accès*: `protected` accès autorisé au sein de la hiérarchie (et dans toutes les classes du même paquetage)

*Masquage/Shadowing*: un même attribut (ou méthode statique) peut être présent dans une sous-classe et une super-classe

*Redéfinition/Overriding*: une méthode d'instance peut-être présente dans une sous-classe et une super-classe

Accès à un *membre masqué/redéfini*: `super.membre`

Le constructeur d'une sous classe doit faire appel au *constructeur de la super classe*:

```
class SousClasse extends SuperClasse {  
    SousClasse(liste de paramètres) {  
        super(arguments); // première instruction  
        ...  
        ...  
    }  
}
```

# *Accès aux bases de données en Java*

## Introduction à JDBC

# Introduction

- JDBC : Java Data Base Connectivity
- Framework permettant l'accès aux bases de données relationnelles dans un programme Java
  - Indépendamment du type de la base utilisée (mySQL, Oracle, Postgres ...)
    - Seule la phase de connexion au SGBDR change
  - Permet de faire tout type de requêtes
    - Sélection de données dans des tables
    - Création de tables et insertion d'éléments dans les tables
    - Gestion des transactions
- Packages : `java.sql` et `javax.sql`

# Principes généraux d'accès à une BDD

## ■ Première étape

- Préciser le type de driver que l'on veut utiliser
  - Driver permet de gérer l'accès à un type particulier de SGBD

## ■ Deuxième étape

- Récupérer un objet « Connection » en s'identifiant auprès du SGBD et en précisant la base utilisée

## ■ Etapes suivantes

- A partir de la connexion, créer un « statement » (état) correspondant à une requête particulière
- Exécuter ce statement au niveau du SGBD
- Fermer le statement

## ■ Dernière étape

- Se déconnecter de la base en fermant la connexion

# Connexion au SGBD

## Classe `java.sql.DriverManager`

- Gestion du contrôle et de la connexion au SGBD

## Méthodes principales

- `static void registerDriver(Driver driver)`
  - Enregistre le driver (objet `driver`) pour un type de SGBD particulier
  - Le driver est dépendant du SGBD utilisé
- `static Connection getConnection( String url, String user, String password)`
  - Crée une connexion permettant d'utiliser une base
  - `url` : identification de la base considérée sur le SGBD
    - Format de l'URL est dépendant du SGGB utilisé
  - `user` : nom de l'utilisateur qui se connecte à la base
  - `password` : mot de passe de l'utilisateur

# Gestion des connexions

- Interface `java.sql.Connection`
- Préparation de l'exécution d'instructions sur la base, 2 types
  - Instruction simple : classe `Statement`
    - On exécute directement et une fois l'action sur la base
  - Instruction paramétrée : classe `PreparedStatement`
    - L'instruction est générique, des champs sont non remplis
    - Permet une pré-compilation de l'instruction optimisant les performances
    - Pour chaque exécution, on précise les champs manquants
- Pour ces 2 instructions, 2 types d'ordres possibles
  - Update : mise à jour du contenu de la base
  - Query : consultation (avec un select) des données de la base

# Gestion des connexions

## ■ Méthodes principales de Connection

- Statement createStatement ()
  - Retourne un état permettant de réaliser une instruction simple
- PreparedStatement prepareStatement (
  - ▶ Retourne un état permettant de réaliser une instruction paramétrée et pré-compilée pour un ordre ordre
  - ▶ Dans l'ordre, les champs libres (au nombre quelconque) sont précisés par des « ? »
- Ex : ' 'select nom from clients where ville=?' 'Lors de l'exécution de l'ordre, on précisera la valeur du champ
- void close ()
  - Ferme la connexion avec le SGBD

# Instruction simple

## Classe Statement

- ? ResultSet executeQuery(String ordre)
  - ? Exécute un ordre de type SELECT sur la base
  - ? Retourne un objet de type ResultSet contenant tous les résultats de la requête
- ? int executeUpdate(String ordre)
  - ? Exécute un ordre de type INSERT, UPDATE, ou DELETE
- ? void close()
  - ? Ferme l'état

# Instruction paramétrée

## QUESTION Classe PreparedStatement

QUESTION Avant d'exécuter l'ordre, on remplit les champs avec

QUESTION void set [Type] (int index, [Type] val)

QUESTION Remplit le champ en i<sup>ème</sup> position définie par index avec la valeur val de type [Type]

QUESTION [Type] peut être : String, int, float, long ...

QUESTION Ex : void setString(int index, String val)

QUESTION ResultSet executeQuery ()

QUESTION Exécute un ordre de type SELECT sur la base

QUESTION Retourne un objet de type ResultSet contenant tous les résultats de la requête

QUESTION int executeUpdate ()

QUESTION Exécute un ordre de type INSERT, UPDATE, ou DELETE

# Lecture des résultats

## Classe ResultSet

- Contient les résultats d'une requête SELECT
  - Plusieurs lignes contenant plusieurs colonnes
  - On y accède ligne par ligne puis valeur par valeur dans la ligne
- Changements de ligne
  - `boolean next()`
    - Se place à la ligne suivante s'il y en a une
    - Retourne `true` si le déplacement a été fait, `false` s'il n'y avait pas d'autre ligne
  - `boolean previous()`
    - Se place à la ligne précédente s'il y en a une
    - Retourne `true` si le déplacement a été fait, `false` s'il n'y avait pas de ligne précédente
  - `boolean absolute(int index)`
    - Se place à la ligne numérotée `index`
    - Retourne `true` si le déplacement a été fait, `false` sinon

# Lecture des résultats

## Classe ResultSet

- Accès aux colonnes/données dans une ligne
- [type] get [Type] (int col)
  - Retourne le contenu de la colonne col dont l'élément est de type [type] avec [type] pouvant être String, int, float, boolean ...
    - Ex : String getString(int col)
- Fermeture du ResultSet
- void close()

# Exception SQLException

- Toutes les méthodes présentées précédemment peuvent lever l'exception SQLException
  - Exception générique lors d'un problème d'accès à la base lors de la connexion, d'une requête ...
  - Plusieurs spécialisations sont définies (voir API)
- Opérations possibles sur cette exception
  - `int getErrorCode ()` : le code de l'erreur renvoyé par le SGBD (et dépendant du type du SGBD)
  - `SQLException getNextException ()` : si plusieurs exceptions sont chaînées entre elles, retourne la suivante ou null s'il n'y en a pas
  - `String getSQLState ()` : retourne « l'état SQL » associé à l'exception

# Exemple

- Accès à une base Oracle contenant 2 tables
    - categorie (codecat, libellecat)
    - produit (codprod, nomprod, codecatt\*)
  - Source de l'exemple : A. Lacayrelle
- 
- Paramètres de la base
    - Fonctionne sur la machine ladybird sur le port 1521
    - Base s'appelle « test »
    - Utilisateur qui se connecte : « étudiant », mot de passe : « mdpetud »

# Exemple

## ■ Crédit de la connexion à la base

■ Connection con;

// chargement du driver Oracle

```
DriverManager.registerDriver(  
    new oracle.jdbc.driver.OracleDriver());
```

// création de la connexion

```
con = DriverManager.getConnection(  
    'jdbc :oracle :thin :@ladybird :1521  
    :test', ''etudiant'', ''mdpetud'');
```

// note: la syntaxe du premier argument dépend du type  
// du SGBD

# Exemple

## Exécution d'une instruction simple de type SELECT Lister toutes les caractéristiques de toutes les catégories

```
? Statement req;
ResultSet res;
String libelle;
int code;

► req = con.createStatement();
► res = req.executeQuery(
    ► "select codcat, libellecat from categorie");

► while(res.next()) {
    code = getInt(1);
    ► libelle = getString(2);
    System.out.println(
        ► " produit : "+code +","+ libelle);
}

req.close();
```

# Exemple

- Exécution d'une instruction simple de type UPDATE
  - Ajouter une catégorie « céréales » de code 5 dans la table catégories
  - Statement req;  
int nb;
  - ▶ req = con.createStatement();
  - ▶ nb = req.executeUpdate("
  - ▶     insert into categories values (5, 'cereales'));
  - ▶ System.out.println(
    - ▶     " nombre de lignes modifiées : "+nb);
  - ▶ req.close();

# Exemple

## Instruction paramétrée de type SELECT

- Retourne tous les produits de la catégorie céréales

```
? PreparedStatement req;
```

```
ResultSet res;
```

```
String nom;
```

```
int code;
```

```
req = con.prepareStatement("select codprod, nomprod  
from categorie c, produit p where c.codcat=p.codcat  
and libellecat = ?");
```

```
req.setString(1, "cereales");
```

```
res = req.executeQuery();
```

```
while(res.next()) {
```

```
    code = getInt(1);
```

```
    libelle = getString(2);
```

```
    System.out.println(
```

```
        " produit : "+code +", "+ libelle); }16
```

```
req.close();
```

# Exemple

## Instruction paramétrée de type UPDATE

### Ajout de 2 nouvelles catégories dans la table catégorie

PreparedStatement req;

int nb;

```
req = con.prepareStatement(
```

```
    'insert into categories values (?,?)');
```

```
req.setInt(1, 12);
```

```
req.setString(2, "fruits");
```

```
nb = req.executeUpdate();
```

```
req.setInt(1, 13);
```

```
req.setString(2, "légumes");
```

```
nb = req.executeUpdate();
```

```
req.close();
```

# Transaction

## ▣ Fonctionnalité avancée

- ▣ Gestion des transactions

- ▣ Transaction

- ▣ Un ensemble d'action est effectué en entier ou pas du tout

- ▣ Voir documentation spécialisée