

# Programmation Avancée et Application

## Bases de la POO en Java

---

Jean-Guy Mailly

`jean-guy.mailly@u-paris.fr`

LIPADE - Université de Paris

<http://www.math-info.univ-paris5.fr/~jmailly/>

## 1. Objets et classes

- Créations de classes et d'objets en Java

- Encapsulation

- Documentation

## 2. Héritage

- Bases de l'héritage

- Classes abstraites

- Interfaces

- Polymorphisme

## 3. Classes de l'API standard

# Objets et classes

---

# Qu'est-ce qu'un objet ?

- Entité créée et manipulée par un programme correspondant à une entité (concrète ou abstraite) du monde réel

5 répertoires téléphoniques avec 100 personnes chacun	5 objets Repertoire
	500 objets Personne
Un cinéma qui diffuse 20 films avec 73 acteurs différents	1 objet Cinema
	20 objets Film
	73 objets Acteur
Une promo de licence avec 200 étudiants qui ont 5 notes pour chacune des 8 unités d'enseignement qu'ils étudient	1 objet Promotion
	200 objets Etudiant
	8 objets UniteEnseignement
	8000 objets Note

# Qu'est-ce qu'une classe ?

- Certains objets représentent des entités du monde réel qui appartiennent à une même catégorie, sont des instances d'un même concept
- La *classe* est l'entité informatique qui correspond à ce concept

# Qu'est-ce qu'une classe ?

- Certains objets représentent des entités du monde réel qui appartiennent à une même catégorie, sont des instances d'un même concept
- La *classe* est l'entité informatique qui correspond à ce concept

Monde réel	Objets	Classe
Marlon Brando	un objet brando	Acteur
Robert De Niro	un objet deNiro	
Al Pacino	un objet pacino	
Talia Shire	un objet shire	
...	...	
Programmation avancée	un objet progAv	UniteEnseignement
Génie logiciel	un objet genieLog	
Algorithmique avancée	un objet algoAv	
...	...	

# Création de classes en Java

- Chaque fichier de code source porte l'extension `.java`
- Le préfixe du nom du fichier doit correspondre au nom de l'unique classe publique définie dans le fichier
- Exemple : fichier `MaClasse.java`

```
public class MaClasse {  
    // ...  
}
```

- On verra dans la suite qu'on peut avoir plusieurs classes dans le même fichier (celles qui ne correspondent pas au nom du fichier ne peuvent pas être déclarées `public`)
- Pour le début du cours, on fait l'hypothèse simplificatrice « 1 fichier = 1 classe »

# Membres d'instance

- On a parlé des membres *de classe* précédemment (`static`), qui ne nécessitent pas de créer d'instances de la classe pour les utiliser
- Les membres (attributs ou méthodes) non `static` sont les membres *d'instance* : on doit créer un objet de cette classe pour les utiliser

Exemple :

```
public class Acteur {  
    private String nom ;  
    private String prenom ;  
    private String nationalite ;  
  
    // ...  
}
```

- Les attributs `nom`, `prenom` et `nationalite` n'ont pas de sens s'ils ne sont pas associés à un acteur précis



- Une méthode est définie par une signature suivie d'un bloc d'instructions (appelé le *corps* de la méthode)
- Signature d'une méthode :
  - Liste de modificateurs
  - Type de retour
  - Nom de méthode
  - Liste de paramètres : type et nom de chaque paramètre (on parle aussi d'arguments)

# Rappel : modificateurs

- Modificateurs d'accès :
  - `public` : on peut utiliser le membre depuis n'importe quel endroit du programme
  - `protected` : on peut utiliser le membre depuis les classes filles et les classes du même package
  - par défaut : sans modificateur d'accès, on peut utiliser le membre depuis les classes du même package
  - `private` : on peut utiliser le membre uniquement à l'intérieur de la classe où elle est définie

# Méthodes et surcharge

- Plusieurs méthodes qui servent globalement à la même chose peuvent porter le même nom, à condition d'avoir des paramètres différents (types et/ou nombre de paramètres) : c'est la surcharge
- Exemple :

```
public class Film {  
    public void ajoutActeur(Acteur acteur){  
        // ...  
    }  
  
    public void ajoutActeur(String nom, String prenom,  
                            String nationalite){  
        // ...  
    }  
}
```

- Les constructeurs sont des méthodes particulières qui servent à créer des instances de la classe
- La définition d'un constructeur est différente de la définition d'une méthode « classique »
  - Pas de type de retour
  - Même nom que la classe (y compris la majuscule au début !)
- Principale utilité : initialiser les attributs

## Constructeurs : premier exemple

```
public class Acteur {  
    private String nom ;  
    private String prenom ;  
    private String nationalite ;  
  
    public Acteur(String n, String p, String nat){  
        nom = n ;  
        prenom = p ;  
        nationalite = nat ;  
    }  
}
```

## Constructeurs : surcharge

- Les constructeurs peuvent aussi être surchargés
- Exemple : on initialise un acteur sans connaître sa nationalité

```
public class Acteur {  
    // ...  
  
    public Acteur(String n, String p, String nat){  
        nom = n ;  
        prenom = p ;  
        nationalite = nat ;  
    }  
    public Acteur(String n, String p){  
        nom = n ;  
        prenom = p ;  
        nationalite = "" ;  
    }  
}
```

- Le mot-clé **this** a plusieurs rôles, dont :
  - lever les ambiguïtés si un attribut porte le même nom qu'une variable locale
  - faire appel à un constructeur depuis un autre constructeur

## Le mot-clé **this** pour lever l'ambiguïté

- Distingue un attribut d'une variable locale (ou paramètre de méthode) qui porte le même nom

```
public class Acteur {  
    // ...  
  
    public Acteur(String nom, String prenom,  
                  String nationalite){  
        this.nom = nom ;  
        this.prenom = prenom ;  
        this.nationalite = nationalite ;  
    }  
}
```



## Le mot-clé `this` et les constructeurs

```
public class Acteur {  
    // ...  
  
    public Acteur(String nom, String prenom,  
                  String nationalite){  
        this.nom = nom ;  
        this.prenom = prenom ;  
        this.nationalite = nationalite ;  
    }  
  
    public Acteur(String nom, String prenom){  
        this(nom, prenom, "") ;  
    }  
}
```

- Instanciation : création d'un objet appartenant à une classe
- Appel au constructeur avec le mot-clé **new**
- Exemple :<sup>1</sup>

```
Acteur deNiro = new Acteur("De_Niro", "Robert",  
                             "Americain");
```

- Création d'une zone de la mémoire dédiée aux données de l'objet créé, retourne une référence vers cet objet

---

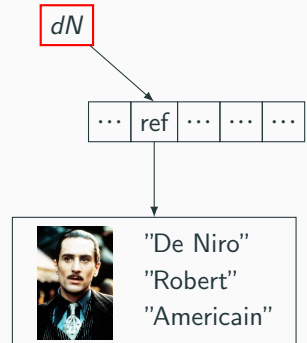
1. Je triche un peu, Robert De Niro a également la nationalité italienne depuis 2006...

# Affectation et passage de paramètres

- Rappel : en Java, l'affectation se fait en recopie et le passage de paramètres se fait par valeur... pour les types simples
- En Java, on ne manipule jamais directement les objets, mais seulement des références vers ces objets ( $\simeq$  des pointeurs)
- Lors d'une affectation ou d'un passage de paramètre, c'est donc la référence qui est copiée, pas l'objet lui même

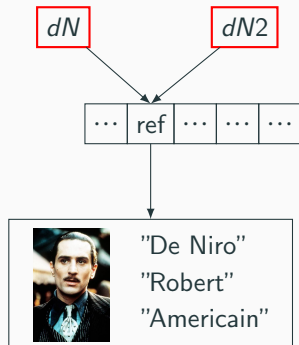
## Exemple d'affectation d'objet

```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");
```



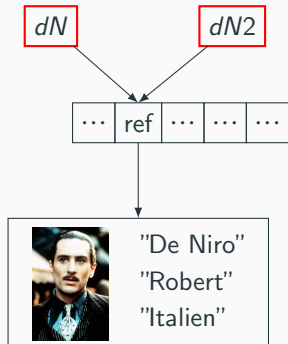
## Exemple d'affectation d'objet

```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur dN2 = dN ;
```



## Exemple d'affectation d'objet

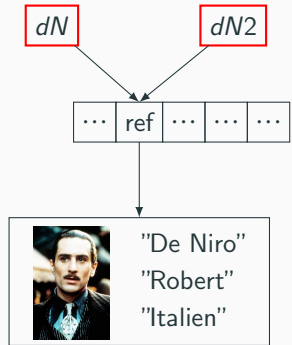
```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur dN2 = dN ;  
dN2.setNationalite("Italien") ;
```



## Exemple d'affectation d'objet

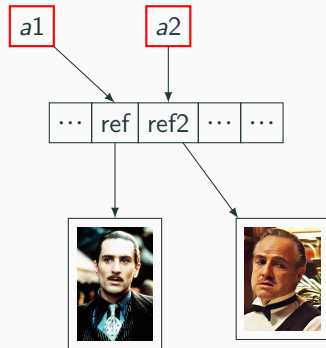
```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur dN2 = dN ;  
dN2.setNationalite("Italien") ;  
System.out.println(  
    dN.getNationalite()) ;
```

Affichage : "Italien"



## Exemple d'affectation d'objet

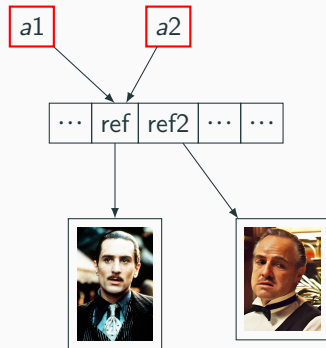
```
Acteur a1 = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur a2 = new Acteur("Brando",  
    "Marlon", "Americain");
```





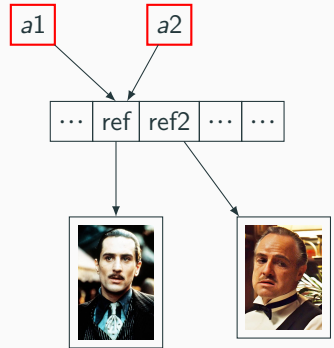
## Exemple d'affectation d'objet

```
Acteur a1 = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur a2 = new Acteur("Brando",  
    "Marlon", "Americain");  
a2 = a1 ;
```



## Exemple d'affectation d'objet

```
Acteur a1 = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur a2 = new Acteur("Brando",  
    "Marlon", "Americain");  
a2 = a1 ;
```

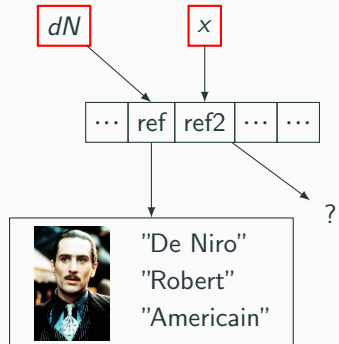


- Le *garbage collector* (GC, ou ramasse-miettes en français) est le sous-système de la JVM en charge de la mémoire
- Il libère l'espace correspondant à Marlon Brando dès que plus aucune référence à cet objet n'est faite dans le programme

# Null

- **null** signifie qu'on ne fait référence à aucun objet

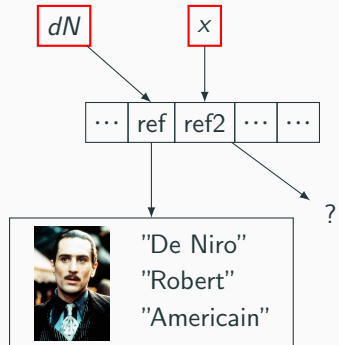
```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur x = null ;
```



# Null

- **null** signifie qu'on ne fait référence à aucun objet

```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur x = null ;  
System.out.println(dN.getNom());
```

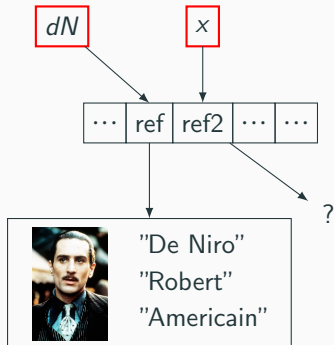


- Affichage : De Niro

# Null

- **null** signifie qu'on ne fait référence à aucun objet

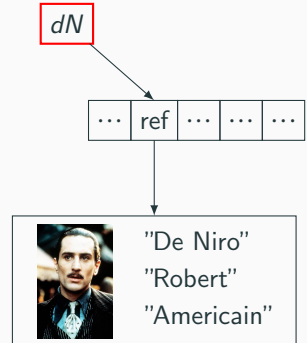
```
Acteur dN = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur x = null ;  
System.out.println(dN.getNom());  
System.out.println(x.getNom());
```



- Si on essaye d'accéder à un membre d'une référence **null**, on obtient une erreur : `java.lang.NullPointerException`

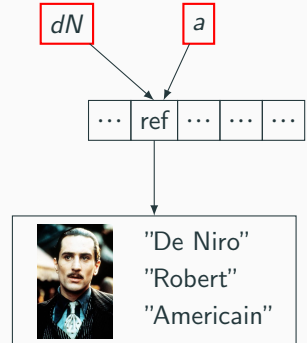
# Passage de paramètres pour les objets

```
public class Test {  
    private static void f(Acteur a){  
        a.setNationalite("Italien") ;  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(  
            dN.getNationalite());  
    }  
}
```



# Passage de paramètres pour les objets

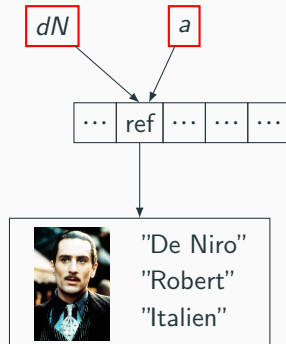
```
public class Test {  
    private static void f(Acteur a){  
        a.setNationalite("Italien") ;  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(  
            dN.getNationalite());  
    }  
}
```



- `f` : variable locale `a` qui correspond à la même référence

# Passage de paramètres pour les objets

```
public class Test {  
    private static void f(Acteur a){  
        a.setNationalite("Italien") ;  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(  
            dN.getNationalite());  
    }  
}
```

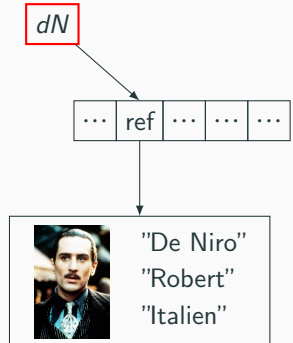


- `f` : modification de la nationalité de De Niro



# Passage de paramètres pour les objets

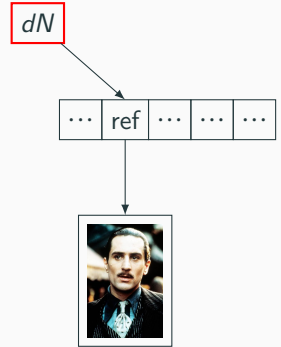
```
public class Test {  
    private static void f(Acteur a){  
        a.setNationalite("Italien") ;  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(  
            dN.getNationalite());  
    }  
}
```



- Affichage : "Italien"

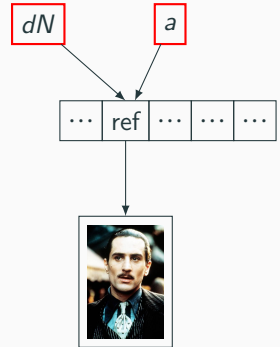
## Passage de paramètres pour les objets

```
public class Test {
    private static void f(Acteur a){
        a = new Acteur("Brando","Marlon",
                        "Americain");
    }
    public static void main(
        String[] args){
        Acteur dN = new Acteur(
            "De_Niro", "Robert",
            "Americain");
        f(dN);
        System.out.println(dN.getNom());
    }
}
```



# Passage de paramètres pour les objets

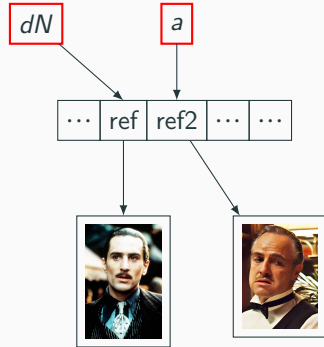
```
public class Test {  
    private static void f(Acteur a){  
        a = new Acteur("Brando", "Marlon",  
                        "Americain");  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(dN.getNom());  
    }  
}
```



- `f` : variable locale `a` qui correspond à la même référence

# Passage de paramètres pour les objets

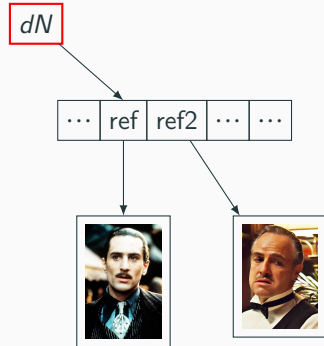
```
public class Test {  
    private static void f(Acteur a){  
        a = new Acteur("Brando", "Marlon",  
                        "Americain");  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(dN.getNom());  
    }  
}
```



- `f` : création nouvel objet et nouvelle référence pour Marlon Brando

# Passage de paramètres pour les objets

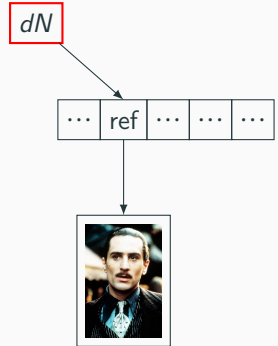
```
public class Test {  
    private static void f(Acteur a){  
        a = new Acteur("Brando", "Marlon",  
                        "Americain");  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(dN.getNom());  
    }  
}
```



- Sortie de `f` : suppression de la variable locale `a`

# Passage de paramètres pour les objets

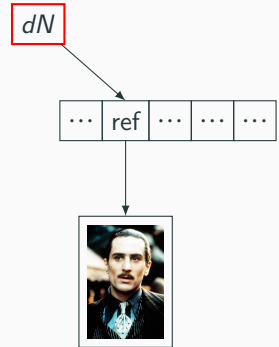
```
public class Test {  
    private static void f(Acteur a){  
        a = new Acteur("Brando", "Marlon",  
                        "Americain");  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(dN.getNom());  
    }  
}
```



- Sortie de `f` : le GC libère l'espace occupé par Marlon Brando

# Passage de paramètres pour les objets

```
public class Test {  
    private static void f(Acteur a){  
        a = new Acteur("Brando", "Marlon",  
                        "Americain");  
    }  
    public static void main(  
        String[] args){  
        Acteur dN = new Acteur(  
            "De_Niro", "Robert",  
            "Americain");  
        f(dN);  
        System.out.println(dN.getNom());  
    }  
}
```



- Affichage : "De Niro"

## Complément d'information sur les expression « new »

- L'opérateur **new** retourne un objet qui peut être directement utilisé dans une expression plus complexe
- Exemple : appel d'une méthode sur le résultat du **new**

```
System.out.println(new Acteur("De_Niro", "Robert",  
                             "Américain").getNom());
```

- Exemple 2 : utilisation comme paramètre d'une méthode

```
Film film = new Film();  
film.ajouterActeur(new Acteur("De_Niro", "Robert",  
                              "Américain"));
```



- Le GC détecte les objets inutilisables et les supprime de la mémoire
- Il fonctionne en tâche de fond et ne supprime pas forcément les objets *immédiatement* après qu'ils soient devenus inutilisables
- Objet inutilisable : objet qui n'est associé à aucune référence
- Le programmeur n'a pas à gérer manuellement la mémoire (pas de `free()` comme en C)
- En cas de besoin, on peut activer le GC manuellement avec « `System.gc()` » ; », même si ce n'est pas courant

# Tableaux, objets et classes (1/2)

- Les tableaux sont des objets : pour chaque type d'éléments de tableaux, une classe correspondante est créée
- Exemple :

```
public class Test {  
    public static void main(String[] args){  
        String[] tab = {"Hello_", "World!"};  
        System.out.println(tab[0].getClass());  
        System.out.println(tab.getClass());  
    }  
}
```

- Affichage :

```
class java.lang.String  
class [Ljava.lang.String;
```

## Tableaux, objets et classes (2/2)

- Comme les tableaux sont des objets, les affectations et passages de paramètres se font par référence
- La longueur du tableau correspond donc à un attribut public et constant : `tab.length`
  - Nous verrons plus tard la définition d'attributs constants
- Les éléments d'un tableaux peuvent être de n'importe quel type, y compris des tableaux !
- Une matrice d'entiers de dimension  $3 \times 3$

```
int [][] t = new int [3][] ;  
t[0] = {1,2,3} ;  
t[1] = {4,5,6} ;  
t[2] = {7,8,9} ;
```

# Principe d'encapsulation

- *Principe d'encapsulation* : les données (→ les attributs) ne doivent pas être accessibles et modifiables librement par n'importe qui
- Pour cela, il est recommandé que les attributs soient **private**, et ne puissent être utilisés ou modifiés que via des méthodes
  - Cela permet au développeur de s'assurer que ses classes seront utilisées d'une manière cohérente par la suite

## Une classe sans encapsulation

```
public class Produit {  
    public String nom ;  
    public double prix ;  
    public Produit(String nom, double prix){  
        this.nom = nom ;  
        this.prix = prix ;  
    }  
}
```

# Une classe sans encapsulation

```
public class Produit {  
    public String nom ;  
    public double prix ;  
    public Produit(String nom, double prix){  
        this.nom = nom ;  
        this.prix = prix ;  
    }  
}
```

- L'utilisation de cette classe peut mener à des incohérences :

```
Produit prod = new Produit(  
    "CD_Greatest_Hits", 16.99);  
// Ailleurs dans le code...  
prod.prix = -10 ;
```



16<sup>99</sup> €

✓prime

Queen Greatest Hits I, II &  
III - Platinum Collection

★★★★★ 357

## La même classe avec encapsulation

```
public class Produit {  
    private String nom ;  
    private double prix ;  
  
    public Produit(String nom, double prix){  
        this.nom = nom ;  
        this.prix = prix ;  
    }  
    public void setPrix(double prix){  
        if (prix >= 0){  
            this.prix = prix ;  
        }  
    }  
}
```

- Impossible de mettre un prix incohérent maintenant !
- Remarque : si nécessaire, on peut faire le même genre de contrôle de cohérence dans le constructeur

# Les buts de l'encapsulation

- On assure dans les méthodes le respect de certaines contraintes (voir l'exemple du prix d'un produit)
- Les éventuelles modifications de la classe d'une version à l'autre sont transparentes pour les utilisateurs, qui ne voient que son interface (= les méthodes publiques) sans voir les détails d'implémentation



# Les buts de l'encapsulation

- On assure dans les méthodes le respect de certaines contraintes (voir l'exemple du prix d'un produit)
- Les éventuelles modifications de la classe d'une version à l'autre sont transparentes pour les utilisateurs, qui ne voient que son interface (= les méthodes publiques) sans voir les détails d'implémentation

→ autant que possible, un attribut doit être **private**, et on y accède ou on le modifie via des méthodes

Seules exceptions : (certaines) constantes, et l'utilisation de **protected** pour l'héritage

## Les constantes (1/2)

- Le modificateur **final** permet de définir des constantes : un attribut **final** ne peut plus être modifié après son initialisation
- Pour un attribut de type primitif, la valeur est fixée. Exemple :  
**final int** N = 5 ; → N vaudra toujours 5, il est impossible par la suite de lui affecter une autre valeur
- Pour un attribut de type objet, la référence est fixée, mais l'état de l'objet peut être modifié via des méthodes. Exemple :  
**final** Acteur DE\_NIRO = **new** Acteur(...);  
et ailleurs dans le code : DE\_NIRO.setNationalite(" Italien ") ;
- Même fonctionnement pour une variable locale ; on parlera plus tard de l'utilisation de **final** pour les méthodes et classes
- Convention : noms des constantes en majuscules, mots séparés par un underscore \_

## Les constantes (2/2)

- Par défaut, une constante est relative à une instance de la classe, elle peut (par exemple) être initialisée dans le constructeur

```
public class Personne {  
    private final String NUM_SECU ;  
    public Personne(String numSecu){  
        NUM_SECU = numSecu ;  
    }  
}
```

## Les constantes (2/2)

- Par défaut, une constante est relative à une instance de la classe, elle peut (par exemple) être initialisée dans le constructeur

```
public class Personne {  
    private final String NUM_SECU ;  
    public Personne(String numSecu){  
        NUM_SECU = numSecu ;  
    }  
}
```

- On peut également définir une constante « absolue » :

```
public class Math {  
    public static final double PI = 3.14 ;  
}
```

→ PI est accessible depuis n'importe quel endroit du code (**public**), sans devoir créer d'instance de Math (**static**), et sans possibilité de modifier sa valeur (**final**)

# Accès et modification des attributs

- Par convention, on accède à un attribut XXX via une méthode getXXX et on le modifie avec setXXX (on parle de getters et setters)

```
public class Acteur {  
    private String nationalite ;  
  
    public void setNationalite(String nationalite){  
        this.nationalite = nationalite ;  
    }  
  
    public String getNationalite(){  
        return nationalite ;  
    }  
}
```

- La documentation du code est essentielle : facilite la réutilisation et les modifications du code (par soi-même, ou par d'autres développeurs)
- Deux formes classiques de commentaires :

*// Un commentaire en une ligne*

*/\*  
Un commentaire sur  
plusieurs lignes  
\*/*

- Ces formes de commentaires doivent être utilisées avec parcimonie, pour expliquer uniquement des points précis du code qui peuvent être complexes

## Il y a le bon commentaire, et il y a le mauvais commentaire...

⚠ Eviter de commenter les parties triviales du code !

```
int i = 0 ; // on initialise le compteur
while(i < 100) { // on boucle 100 fois
    if((i % 2) == 0){ // on verifie si i est pair
        System.out.println(i + "_est_pair.");
        /*
        On affiche un message si
        i est pair
        */
    }
    i++; // on incremente le compteur
}
```

- On privilégie la documentation des classes, méthodes et attributs via des commentaires Javadoc
- Un commentaire Javadoc commence par `/**` et se termine par `*/`, et commence par une description textuelle, et comporte un certain nombre de tags qui permettent d'indiquer le rôle des éléments du code
- L'outil javadoc du JDK génère un ensemble de pages web correspondant à la documentation du code, en se basant sur les commentaires Javadoc



# Les tags Javadoc (1/2)

Tag	Localisation	Rôle
@author	Classe	Nom de l'auteur
@version	Classe	Version de la classe
@deprecated	Classe Constructeur Méthode Attribut	Marquer comme obsolète, <sup>2</sup> décrire pourquoi, et par quoi remplacer
@see	Classe Constructeur Méthode Attribut	Ajoute un lien « Voir aussi »

---

2. Provoque un warning du compilateur si utilisé(e)

## Les tags Javadoc (2/2)

Tag	Localisation	Rôle
@param	Constructeur Méthode	Décrire un paramètre
@return	Méthode	Décrire la valeur retournée

- Cette liste n'est pas exhaustive
- Les commentaires Javadoc peuvent utiliser HTML pour leur mise en forme

- Idéalement, tous les attributs, toutes les méthodes et tous les constructeurs doivent être commentés : permet de comprendre le fonctionnement de la classe sans même avoir besoin de regarder le code, la Javadoc suffit
- Une phrase doit expliquer clairement le rôle de l'entité commentée
- Le rôle de chaque paramètre doit être expliqué, ainsi que la signification de la valeur de retour des méthodes
- Les éventuelles restrictions et contraintes doivent être données (par exemple, le prix d'un produit ne peut pas être négatif)
- Par défaut, seule la Javadoc des membres **public** est générée ; cela peut être modifié via la ligne de commande de l'outil javadoc ou l'interface de l'IDE

## Exemple de Javadoc (1/9)

```
package up.mi.jgm.util;  
  
/**  
 * Classe utilitaire qui permet la  
 * manipulation de tableaux  
 *  
 * @author Jean-Guy Mailly  
 * @version 1.0  
 *  
 */  
public class UtilTab {  
    // ...  
  
}
```

## Exemple de Javadoc (2/9)

```
/**
 * Permet l'affichage des elements d'un tableau ,
 * en ligne ou en colonne
 *
 * @param tab      le tableau a afficher
 * @param enLigne  boolean qui vaut true si le tableau
 *                  doit etre affiche en ligne
 */
public static void affichageTableau(double[] tab,
                                     boolean enLigne) {
    // ...
}
```

## Exemple de Javadoc (3/9)

```
/**
 * Methode qui permet de determine si une valeur
 * appartient au tableau
 *
 * @param val la valeur dont on teste l'appartenance
 * @param tab le tableau
 * @return true si et seulement si val appartient a tab
 */
public static boolean appartient(double val,
                                double[] tab) {
    // ...
}
```

## Exemple de Javadoc (4/9)

```
/**  
 * Methode qui determine l'element minimal d'un tableau  
 *  
 * @param tab le tableau dont on cherche le minimum  
 * @return le minimum de tab  
 */  
public static double min(double[] tab) {  
    // ...  
}
```

## Exemple de Javadoc (5/9)

```
/**  
 * Methode qui determine l'element maximal d'un tableau  
 *  
 * @param tab le tableau dont on cherche le maximum  
 * @return le maximum de tab  
 */  
public static double max(double[] tab) {  
    // ...  
}
```



## Exemple de Javadoc (6/9)

```
/**  
 * Methode qui calcule la somme des elements  
 * d'un tableau  
 *  
 * @param tab le tableau dont on calcule la somme  
 * @return la somme des elements de tab  
 */  
public static double somme(double[] tab) {  
    // ...  
}
```

## Exemple de Javadoc (7/9)

```
/**
 * Methode qui trie un tableau par selection. Le tri
 * par selection consiste a chercher l'element le
 * plus petit du tableau, et a l'echanger avec le
 * premier element. Le processus est reitere pour le
 * deuxieme plus petit qui est echange avec le deuxieme
 * element du tableau, etc, jusqu'a ce que le tableau
 * soit trie. Cette methode modifie le tableau.
 *
 * @param tab le tableau a trier.
 */
public static void triParSelection(double[] tab) {
    // ...
}
```

## Exemple de Javadoc (8/9)

```
/**
 * Methode qui permet d'obtenir l'indice du plus petit
 * element d'un tableau a partir d'une position donnee
 *
 * @param tab      le tableau dont on cherche le plus
 *                  petit element
 * @param indiceMin la position a partir de laquelle on
 *                  recherche le plus petit element
 * @return l'indice du plus petit element de tab situe
 *         apres la position indiceMin
 */
private static int rechercheIndicePlusPetit(double[]
                                             tab, int indiceMin) {
    // ...
}
```

## Exemple de Javadoc (9/9)

```
/**
 * Methode qui echange deux elements d'un tableau ,
 * donnees par leur position. Modifie le tableau.
 * @param tab le tableau dans lequel on echange
 *                                deux elements
 * @param i l'indice du premier element a echanger
 * @param j l'indice du second element a echanger
 */
private static void echanger(double[] tab ,
                              int i, int j) {
    // ...
}
```

# Aperçu de la Javadoc générée (1/3)

## Method Summary

All Methods	Static Methods	Concrete Methods
Modifier and Type	Method and Description	
static void	<b>affichageTableau</b> (double[] tab, boolean enLigne)	Permet l'affichage des elements d'un tableau, en ligne ou en colonne
static boolean	<b>appartient</b> (double val, double[] tab)	Methode qui permet de determine si une valeur appartient au tableau
private static void	<b>echanger</b> (double[] tab, int i, int j)	Methode qui echange deux elements d'un tableau, donnees par leur position
static double	<b>max</b> (double[] tab)	Methode qui determine l'element maximal d'un tableau
static double	<b>min</b> (double[] tab)	Methode qui determine l'element minimal d'un tableau
private static int	<b>rechercheIndicePlusPetit</b> (double[] tab, int indiceMin)	Methode qui permet d'obtenir l'indice du plus petit element d'un tableau a partir d'une position donnee
static double	<b>somme</b> (double[] tab)	Methode qui calcule la somme des elements d'un tableau
static void	<b>triParSelection</b> (double[] tab)	Methode qui trie un tableau par selection.

# Aperçu de la Javadoc générée (2/3)

## Method Detail

### affichageTableau

```
public static void affichageTableau(double[] tab,  
                                   boolean enLigne)
```

Permet l'affichage des elements d'un tableau, en ligne ou en colonne

Parameters:

tab - le tableau a afficher

enLigne - boolean qui vaut true si et seulement si le tableau doit etre affiche en ligne

### appartient

```
public static boolean appartient(double val,  
                                double[] tab)
```

Methode qui permet de determine si une valeur appartient au tableau

Parameters:

val - la valeur dont on teste l'appartenance

tab - le tableau

Returns:

true si et seulement si val appartient a tab

# Aperçu de la Javadoc générée (3/3)

## triParSelection

```
public static void triParSelection(double[] tab)
```

Methode qui trie un tableau par selection. Le tri par selection consiste a chercher l'element le plus petit du tableau, et a l'echanger avec le premier element. Le processus est reitere pour le deuxieme plus petit qui est echange avec le deuxieme element du tableau, etc, jusqu'a ce que le tableau soit trie.

Parameters:

tab - le tableau a trier.

## rechercheIndicePlusPetit

```
private static int rechercheIndicePlusPetit(double[] tab,  
                                             int indiceMin)
```

Methode qui permet d'obtenir l'indice du plus petit element d'un tableau a partir d'une position donnee

Parameters:

tab - le tableau dont on cherche le plus petit element

indiceMin - la position a partir de laquelle on recherche le plus petit element

Returns:

l'indice du plus petit element de tab situe apres la position indiceMin

## echanger

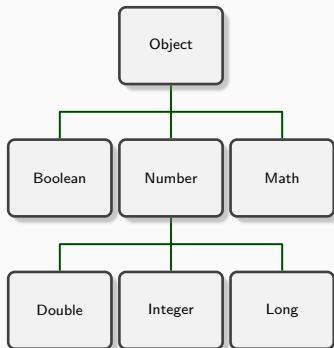
# Héritage

---



# Héritage

- Définition d'une classe B à partir d'une classe A, de telle façon que B reprend les membres de A, avec la possibilité d'en ajouter de nouveaux ou de modifier certains des membres existants
  - B est appelée une classe *filles*, ou sous-classe, de A (et inversement, A est une classe *mère*, ou super-classe, de B)
  - On peut généraliser ces notions en parlant de classes ancêtres et descendantes, et on peut représenter des classes sous forme d'un arbre généalogique



- Spécialisation : on ajoute des caractéristiques spécifiques en définissant une classe fille, par exemple une Voiture est un Vehicule qui dispose d'un moteur
- Redéfinition : on modifie le comportement d'une classe en redéfinissant certaines méthodes (par exemple, la méthode avancer() de la classe Voiture nécessite de vérifier si le réservoir n'est pas vide, ce qui n'est pas le cas pour d'autres véhicules)
- Réutilisation de code : on peut faire appel au code de la classe A même sans avoir son code source à disposition, en définissant la classe B qui hérite de A

## Exemple d'héritage avec le mot clé extends

- On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre classe

```
public class Voiture extends Vehicule {  
    private String typeCarburant ;  
  
    public Voiture(String typeCarburant){  
        this.typeCarburant = typeCarburant ;  
    }  
}
```

- Problème : les attributs hérités de la classe-mère Vehicule ne sont pas initialisés

## Le mot-clé **super** : constructeurs

- On peut utiliser les constructeurs de la classe mère avec le mot-clé **super**, pour initialiser les attributs hérités. Exemple :

```
public class Vehicule {  
    private String proprietaire ;  
    public Vehicule(String proprietaire){  
        this.proprietaire = proprietaire ;  
    }  
}
```

Et dans Voiture :

```
public Voiture(String proprietaire ,  
                String typeCarburant){  
    super(proprietaire) ;  
    this.typeCarburant = typeCarburant ;  
}
```

## Le mot-clé **super** : constructeurs

- Le mot-clé **super** doit être la première instruction du constructeur, avant un éventuel constructeur **this** et les autres instructions :

```
public class Voiture extends Vehicule {  
    private String typeCarburant ;  
    private String immatriculation ;  
    public Voiture(String typeCarburant){  
        this.typeCarburant = typeCarburant ;  
    }  
  
    public Voiture(String proprietaire ,  
        String typeCarburant , String immatriculation){  
        super(proprietaire) ;  
        this(typeCarburant) ;  
        this.immatriculation = immatriculation ;  
    }  
}
```

- Une classe fille peut accéder aux membres hérités de la classe mère s'ils sont **public**
  - Rappel : le principe d'encapsulation recommande de garder les attributs privés autant que possible
  - Dans certains cas ce n'est pas un problème : utilisation des méthodes `getXXX()` et `setXXX()` pour utiliser l'attribut `XXX`
- S'il y a besoin d'accéder directement aux attributs de la classe mère dans la classe fille, il faut que ces attributs soient **protected**

## Exemple d'utilisation de protected

```
public class Animal {  
    protected int nbPattes ;  
    protected int age ;  
}  
  
public class Chien extends Animal {  
    public Chien(int age){  
        nbPattes = 4 ;  
        this.age = age ;  
    }  
  
    public void unAnDePlus(){  
        age++ ;  
    }  
}
```

# Redéfinition et surcharge

- Une classe fille hérite des membres (méthode et attributs) **public** et **protected** de sa mère
- Il est possible de surcharger une méthode héritée de la mère
  - Par exemple, `Vehicule` possède une méthode **void** `avancer(double distance)`, tandis que `Voiture` possède une méthode **void** `avancer(double distance, boolean carburantOK)` qui avance uniquement s'il y a du carburant dans le réservoir
- Il est également possible de redéfinir la méthode de la classe mère :
  - Par exemple, `Voiture` peut redéfinir **void** `avancer(double distance)` pour tenir compte de la quantité de carburant sans avoir besoin d'un booléen en paramètre
  - On indique que la méthode est une redéfinition avec `@Override`



## Le mot-clé super : membres (1/3)

- La redéfinition « efface » la méthode héritée de la classe mère

```
public class A {  
    public void methode(){  
        System.out.println("Dans_la_classe_A");  
    }  
}
```

```
public class B extends A {  
    @Override  
    public void methode(){  
        System.out.println("Dans_la_classe_B");  
    }  
}
```

## Le mot-clé super : membres (2/3)

```
public class Test {  
    public static void main(String[] args){  
        B b = new B();  
        b.methode();  
    }  
}
```

- Affichage : Dans la classe B
- Si on a quand même besoin d'accéder à la méthode de la mère, on peut le faire avec **super**

## Le mot-clé super : membres (3/3)

```
public class A {  
    public void methode(){  
        System.out.println("Dans_la_classe_A");  
    }  
}
```

```
public class B extends A {  
    @Override  
    public void methode(){  
        super.methode();  
        System.out.println("Dans_la_classe_B");  
    }  
}
```

- Affichage (avec la même classe Test) :

Dans la classe A

Dans la classe B

# Héritage multiple en Java

- En Java, il est impossible d'avoir une classe qui hérite de plusieurs classes mères différentes. Cela évite d'avoir à gérer des conflits d'héritage si deux classes mères possèdent une même méthode :

```
public class A {  
    public void f(){ System.out.println("A_!");  
}  
public class B {  
    public void f(){ System.out.println("B_!");  
}  
public class C extends A, B { }
```

- On ne peut pas dire ce qui se passe si on fait :  
C c = **new** C();  
c.f();  
C'est donc interdit en Java

# La mère de toutes les classes : Object

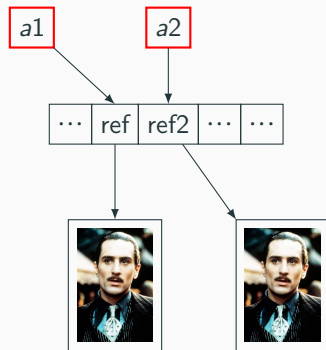
- Toutes les classes héritent de `java.lang.Object`
  - `java.lang.Object` est la seule classe qui n'a pas d'ancêtres en Java
- Implicitement, **`public class A { ... }`** est équivalent à **`public class A extends Object { ... }`** donc `Object` est une classe mère dans ce cas
- Si **`public class A extends B { ... }`**, alors `Object` est un ancêtre de `A`
- Un certain nombre de méthodes sont définies dans `Object`, et peuvent nécessiter d'être redéfinies dans les classes filles

# La méthode equals d'Object

- **boolean** equals(Object obj) : indique si un autre objet est « égal » à celui-ci
- Pourquoi ne pas utiliser == pour vérifier si deux objets sont identiques ?
  - o1 == o2 vaut vrai si o1 et o2 correspondent à la même référence
  - On peut implémenter la méthode equals pour assurer que le résultat est **true** si les objets sont conceptuellement identiques, même si la référence est différente
- Remarque : il est recommandé de redéfinir également la méthode **public int** hashCode() lorsque equals est redéfinie. Eclipse est capable de générer automatiquement les méthodes hashCode et equals

## equals versus == (1/3)

```
Acteur a1 = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur a2 = new Acteur("De_Niro",  
    "Robert", "Americain");  
if(a1 == a2){  
    System.out.println("Identiques");  
}else{  
    System.out.println("Différents");  
}
```



- Affichage : Différents

## equals versus == (2/3)

- On ajoute une méthode equals dans la classe Acteur :

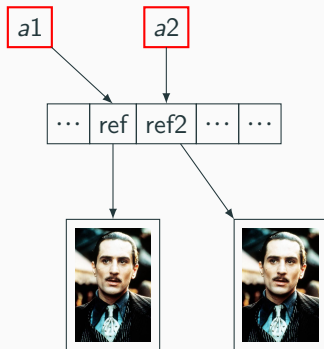
```
public boolean equals(Acteur a){  
    return (this.nom.equals(a.nom))  
        && (this.prenom.equals(a.prenom))  
        && (this.nationalite.equals(a.nationalite)) ;  
}
```

- Remarque : la méthode equals doit normalement prendre en paramètre un Object. On verra dans la suite comment faire cela



## equals versus == (3/3)

```
Acteur a1 = new Acteur("De_Niro",  
    "Robert", "Americain");  
Acteur a2 = new Acteur("De_Niro",  
    "Robert", "Americain");  
if(a1.equals(a2)){  
    System.out.println("Identiques");  
}else{  
    System.out.println("Différents");  
}
```



- Affichage : Identiques

# La méthode toString d'Object

- **public** String toString () : retourne une représentation de l'objet sous forme de String
- Dans la classe Object, cette méthode correspond à `getClass().getName() + '@' + Integer.toHexString(hashCode())`
- Il est recommandé de redéfinir la méthode afin d'avoir une information qui peut être affichée
- `toString` est appelée implicitement lors d'un `System.out.println ()` ou une concaténation

## Exemple de toString (1/2)

```
Acteur dN = new Acteur("De_Niro", "Robert",  
                        "Americain");  
System.out.println(dN);
```

- Affichage : up.mi.jgm.cinema.Acteur@670018c6

## Exemple de toString (2/2)

```
public class Acteur {  
    // ...  
  
    @Override  
    public String toString() {  
        return prenom + " " + nom + ", " + nationalite ;  
    }  
}
```

- On exécute le même test :

```
Acteur dN = new Acteur("De_Niro", "Robert",  
                        "Américain");  
System.out.println(dN);
```

- Affichage : Robert De Niro, Américain

# instanceof

- Lorsqu'on fait `A obj = new A();`, l'objet est une *instance directe* de la classe A
- Si B est une classe fille de A, alors `B obj = new B();` crée une *instance indirecte* de la classe A
- On peut savoir si un objet est une instance d'une classe avec le mot clé **instanceof**

```
System.out.println(dN instanceof Acteur);  
System.out.println(dN instanceof Object);  
  
Object obj = new Object();  
System.out.println(obj instanceof Object);  
System.out.println(obj instanceof Acteur);
```

- Affichage :  
true  
true  
true  
false

- Le transtypage (ou *cast*) consiste à utiliser un type moins élevé dans la hiérarchie que celui attendu :
  - `Object o = new String("toto")` ; → cela fonctionne car un `String` est un `Object` particulier
- On ne peut pas transtyper s'il n'y a pas de relation d'héritage (direct ou indirect) entre l'objet qu'on veut transtyper, et le type cible
  - `Integer i = new String("toto");` → Impossible : un `String` n'est pas un `Integer` !
  - `String s = new Object();` → Impossible : un `Object` n'est pas un `String` !

- Il peut y avoir une différence entre le type « réel » d'un objet (c'est-à-dire la structure représentée en mémoire) et le type « apparent » (c'est-à-dire le type manipulé par le programmeur)

- Il peut y avoir une différence entre le type « réel » d'un objet (c'est-à-dire la structure représentée en mémoire) et le type « apparent » (c'est-à-dire le type manipulé par le programmeur)
  - `Object o = new String("toto");` → on a un `String` en mémoire, mais on manipule un `Object`



- Il peut y avoir une différence entre le type « réel » d'un objet (c'est-à-dire la structure représentée en mémoire) et le type « apparent » (c'est-à-dire le type manipulé par le programmeur)
  - `Object o = new String("toto");` → on a un `String` en mémoire, mais on manipule un `Object`
  - `String s = o ;` → on ne peut pas affecter un `Object` à un `String` ! ❌

- Il peut y avoir une différence entre le type « réel » d'un objet (c'est-à-dire la structure représentée en mémoire) et le type « apparent » (c'est-à-dire le type manipulé par le programmeur)
  - `Object o = new String("toto");` → on a un `String` en mémoire, mais on manipule un `Object`
  - `String s = o;` → on ne peut pas affecter un `Object` à un `String` ! ❌
  - `String s = (String) o;` → on prévient le compilateur que `o` peut être manipulé comme un `String` : ça marche ✅

## Methode equals avec instanceof et transtypage

- Grâce à **instanceof** et au transtypage, on peut écrire une « vraie » méthode equals qui prend en paramètre un Object

```
public class Acteur {  
    // ...  
    @Override  
    public boolean equals(Object o){  
        if((o == null) || !(o instanceof Acteur)){  
            return false ;  
        }  
        if(o == this){ return true ;}  
        Acteur autre = (Acteur) o ;  
        return (this.nom.equals(autre.nom))  
            && (this.prenom.equals(autre.prenom))  
            && (this.nationalite.equals(autre.nationalite)) ;  
    }  
}
```

- Une classe abstraite est une classe partiellement implémentée qui ne peut pas être utilisée directement, mais seulement par l'intermédiaire de classes filles qui implémentent les parties abstraites de la classe
- Utilisation du mot-clé **abstract** dans la déclaration de la classe et la signature des méthodes abstraites
- On ne peut pas instancier directement une classe abstraite, seulement indirectement via les classes filles
- Une classe fille d'une classe abstraite doit implémenter ses méthodes abstraites, sinon elle est elle-même déclarée abstraite

## Exemple de classe abstraite

```
public abstract class Animal {  
    private int age ;  
  
    public Animal(){  
        this.age = 0 ;  
    }  
  
    public void unAnDePlus(){  
        age++ ;  
    }  
  
    public abstract void crier();  
}
```

- Impossible de faire **new** Animal(), le constructeur sera utilisé dans les classes filles (avec **super()**)
- Les classes filles de Animal doivent implémenter la méthode crier () ou être déclarées abstraites

## Héritage d'une classe abstraite (1/2)

```
public class Mammifere
    extends Animal {
    private boolean terrestre ;

    public Mammifere(boolean t){
        super() ;
        terrestre = t ;
    }
}
```

✗ Incorrect : Mammifere  
n'implémente pas la méthode  
abstraite crier () de sa classe  
mère, alors elle doit être  
déclarée abstraite

## Héritage d'une classe abstraite (1/2)

```
public abstract class Mammifere
    extends Animal {
    private boolean terrestre ;

    public Mammifere(boolean t){
        super() ;
        terrestre = t ;
    }
}
```

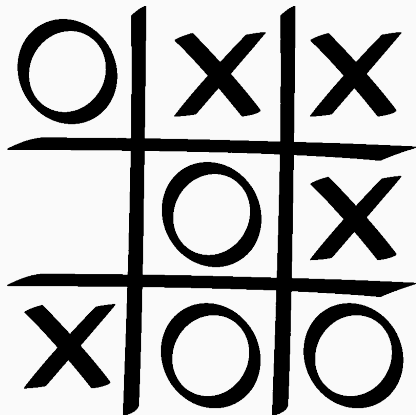
✓ Correct : une classe fille de Mammifere peut être instanciée si elle n'est pas abstraite et implémente la méthode crier ()

## Héritage d'une classe abstraite (2/2)

```
public class Chien extends Mammifere {  
    private String race ;  
  
    public Chien(String race){  
        super(true) ;  
        this.race = race ;  
    }  
  
    public void crier(){  
        System.out.println("Wouaf_wouaf_!") ;  
    }  
}
```



- Une interface définit un type d'objets en fonction d'une liste de méthodes qui doivent être fournies par ces objets
  - Une interface peut être vue comme « une classe entièrement abstraite »
- On dit qu'une classe implémente une interface, utilisation du mot-clé **implements**
- Une classe peut implémenter plusieurs interfaces :  
**public class A implements B, C { ... }**
- Si toutes les méthodes de l'interface ne sont pas implémentées, la classe doit être déclarée abstraite
- Une interface n'a pas de constructeur, ni d'attributs (sauf éventuellement des attributs **static**)
- Une interface peut être utilisée pour mieux séparer les différentes parties de la conception d'un programme



- On définit une interface pour représenter une grille de jeu, et une interface pour les interactions entre les joueurs et le jeu

## Interface pour la grille de jeu (1/3)

```
public interface ITicTacToe {  
    /**  
     * Symbole correspondant au premier joueur  
     */  
    public static final char X = 'X';  
  
    /**  
     * Symbole correspondant au second joueur  
     */  
    public static final char O = 'O';  
  
    /**  
     * Symbole correspondant a une case vide  
     */  
    public static final char VIDE = '␣';  
}
```

## Interface pour la grille de jeu (2/3)

```
/**  
 * Permet d'obtenir le symbole place a une certaine  
 * position dans la grille  
 * @param x l'abscisse de la position demandee  
 * @param y l'ordonnee de la position demandee  
 * @return le symbole contenu a la position (x,y)  
 */  
public char getSymbole(int x, int y);
```

```
/**  
 * Place un symbole a une position donnee  
 * @param symbole le symbole a placer  
 * @param x l'abscisse de la position  
 * @param y l'ordonnee de la position  
 */  
public void putSymbole(char symbole, int x, int y);
```

## Interface pour la grille de jeu (3/3)

```
/**
 * Determine si un joueur a gagne
 * @param symbole le symbole correspondant a un joueur
 * @return true si et seulement si la joueur
 * correspondant au symbole a gagne
 */
public boolean aGagne(char symbole);

/**
 * Determine si la grille de jeu est pleine
 *
 * @return true si et seulement si la grille est pleine
 */
public boolean grillePleine();
}
```

## Interface pour le moteur de jeu (1/3)

```
public interface IJeu {  
    /**  
     * Indique si le jeu est fini  
     *  
     * @return true si et seulement si le jeu est fini  
     */  
    public boolean estFini();  
  
    /**  
     * Determine quel est le prochain joueur  
     *  
     * @return le symbole correspondant au prochain joueur  
     */  
    public char prochainJoueur();  
}
```

## Interface pour le moteur de jeu (2/3)

```
/**
 * Effectue le tour d'un joueur
 *
 * @param x le symbole correspondant au joueur dont
 * c'est le tour
 */
public void tourJoueur(char x);

/**
 * Indique si un joueur a gagne le jeu
 *
 * @param x le joueur qui doit etre teste
 * @return true si et seulement si le joueur x a gagne
 */
public boolean aGagne(char x);
```

## Interface pour le moteur de jeu (3/3)

```
/**
 * Permet de feliciter le joueur gagnant
 *
 * @param x le gagnant de la partie
 */
public void feliciter(char x);

/**
 * Informe les joueurs qu'ils ont tous les deux perdu
 */
public void perdu();
}
```



# Classe principale

```
public class Main {  
    public static void main(String[] args) {  
        IJeu jeu = new Jeu(new TicTacToe());  
  
        while(! jeu.estFini()) {  
            char x = jeu.prochainJoueur();  
            jeu.tourJoueur(x);  
        }  
  
        if(jeu.aGagne(ITicTacToe.O)) {  
            jeu.feliciter(ITicTacToe.O);  
        } else if(jeu.aGagne(ITicTacToe.X)) {  
            jeu.feliciter(ITicTacToe.X);  
        } else { jeu.perdu(); }  
    }  
}
```

# Implémenter les interfaces

- Il reste à implémenter les classes Jeu et TicTacToe pour avoir un jeu fonctionnel
- Intérêts du découpage avec les interfaces
  - Séparer le développement des différentes parties : le développeur de la classe Jeu n'a pas besoin de connaître les détails d'implémentation de la classe TicTacToe, seulement de savoir que cette classe implémente les méthodes de l'interface ITicTacToe (et vice-versa)
  - Faire évoluer les différentes parties séparément :
    - si une nouvelle version du logiciel propose meilleure classe TicTacToe2, il suffit de remplacer la première ligne par `IJeu = new Jeu(new TicTacToe2());` pour en bénéficier.
    - si on veut proposer un jeu avec une interface graphique au lieu d'une interface textuelle, il faut remplacer par `IJeu = new JeuGraphique(new TicTacToe());`

## Classe TicTacToe (1/7)

```
public class TicTacToe implements ITicTacToe {  
    char [][] grille;  
  
    public TicTacToe() {  
        grille = new char[3][3];  
        for (int i = 0; i < 3; i++) {  
            for (int j = 0; j < 3; j++) {  
                grille[i][j] = ITicTacToe.VIDE;  
            }  
        }  
    }  
}
```

```
@Override  
public char getSymbole(int x, int y) {  
    return grille[x][y];  
}
```

```
@Override  
public void putSymbole(char symbole, int x, int y) {  
    if (grille[x][y] == ITicTacToe.VIDE)  
        grille[x][y] = symbole;  
}
```

## Classe TicTacToe (3/7)

```
@Override
public boolean aGagne(char symbole) {
    for (int ligne = 0; ligne < 3; ligne++) {
        if (aGagneLigne(symbole, ligne))
            return true;
    }
    for (int colonne = 0; colonne < 3; colonne++) {
        if (aGagneColonne(symbole, colonne))
            return true;
    }
    if (aGagneDiagonale1(symbole)
        || aGagneDiagonale2(symbole))
        return true;

    return false;
}
```

```
private boolean aGagneColonne(char symbole ,  
                                int colonne) {  
    return ( grille[0][colonne] == symbole)  
        && ( grille[1][colonne] == symbole)  
        && ( grille[2][colonne] == symbole);  
}
```

```
private boolean aGagneLigne(char symbole ,  
                              int ligne) {  
    return ( grille[ligne][0] == symbole)  
        && ( grille[ligne][1] == symbole)  
        && ( grille[ligne][2] == symbole);  
}
```

```
private boolean aGagneDiagonale1(char symbole) {  
    return ( grille[0][0] == symbole)  
        && ( grille[1][1] == symbole)  
        && ( grille[2][2] == symbole);  
}
```

```
private boolean aGagneDiagonale2(char symbole) {  
    return ( grille[0][2] == symbole)  
        && ( grille[1][1] == symbole)  
        && ( grille[2][0] == symbole);  
}
```

## Classe TicTacToe (6/7)

```
@Override
public boolean grillePleine() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grille[i][j] == ITicTacToe.VIDE) {
                return false;
            }
        }
    }
    return true;
}
```



```
@Override
public String toString() {
    return ligneToString(0) + ligneToString(1)
        + ligneToString(2);
}

private String ligneToString(int ligne) {
    return grille[ligne][0]
        + "|" + grille[ligne][1]
        + "|" + grille[ligne][2] + "\n";
}
}
```

```
public class Jeu implements IJeu {  
    private ITicTacToe ticTacToe;  
    private char prochainJoueur;  
    private Scanner sc;  
  
    public Jeu(ITicTacToe ticTacToe) {  
        this.ticTacToe = ticTacToe;  
        prochainJoueur = ITicTacToe.X;  
        sc = new Scanner(System.in);  
    }  
}
```

```
@Override
public boolean estFini() {
    boolean estFini = (ticTacToe.aGagne(ITicTacToe.O))
        || (ticTacToe.aGagne(ITicTacToe.X))
        || (ticTacToe.grillePleine());

    if (estFini)
        sc.close();
    return estFini;
}
```

```
@Override
public char prochainJoueur() {
    return prochainJoueur;
}
```

## Classe Jeu (3/4)

```
@Override
public void tourJoueur(char x) {
    System.out.println("Joueur_" + x +
        ",_quelle_case_voulez_vous_jouer_?");
    System.out.println("Abscisse_:");
    int abs = sc.nextInt();
    System.out.println("Ordonnee_:");
    int ord = sc.nextInt();
    ticTacToe.putSymbole(x, abs, ord);
    System.out.println(ticTacToe);

    if (prochainJoueur == ITicTacToe.X) {
        prochainJoueur = ITicTacToe.O;
    } else {
        prochainJoueur = ITicTacToe.X;
    }
}
```

## Classe Jeu (4/4)

```
@Override  
public boolean aGagne(char x) {  
    return ticTacToe.aGagne(x);  
}
```

```
@Override  
public void feliciter(char x) {  
    System.out.println("Félicitations_"  
        + "au_joueur_" + x + "_!");  
}
```

```
@Override  
public void perdu() {  
    System.out.println("Le_jeu_est_bloque_"  
        + "_Tout_le_monde_a_perdu_");  
}
```

```
}
```

# Héritage d'interfaces

- Une interface peut hériter d'une autre interface, y compris l'héritage multiple :

```
public interface A {  
    public void f();  
}
```

```
public interface B {  
    public void g();  
}
```

```
public interface C extends A, B {  
    public void h();  
}
```

- Une classe (non abstraite) qui implémente l'interface C doit avoir les méthodes f(), g() et h()

# Interfaces en tant que marqueurs

- Une interface peut ne posséder aucune méthode, et n'être utilisée que comme marqueur du fait que les classes qui implémentent cette interface peuvent être utilisées à un certain endroit
- Exemple : l'interface A est utilisée pour indiquer si un objet peut être passé en paramètre de la méthode MaClasse.f

```
public interface A { }
```

```
public class MaClasse {  
    public static void f(A a){  
        // ...  
    }  
}
```

- Un cas concret : l'interface `java.io.Serializable` dont nous reparlerons plus tard

- Polymorphisme : une même méthode peut avoir un comportement différent selon les situations, sans que le programmeur n'ait à anticiper quelle situation concrète se présente
- Se manifeste de différentes manières
  - lors de la redéfinition d'une méthode dans les classes filles
  - lorsque différentes classes filles implémentent différemment une méthode abstraite d'une classe abstraite, ou une méthode d'une interface



## Exemple : le polymorphisme à la ferme (1/4)

```
public abstract class Animal {  
    public abstract void crier();  
}  
  
public class Vache extends Animal {  
    @Override  
    public void crier() {  
        System.out.println("Meuh_!");  
    }  
}  
  
public class Canard extends Animal {  
    @Override  
    public void crier() {  
        System.out.println("Coin_coin_!");  
    }  
}
```

De la même façon : Poule et Coq

## Exemple : le polymorphisme à la ferme (2/4)

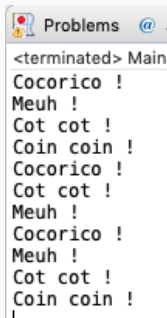
```
public class Ferme {  
    private List<Animal> animaux ;  
  
    public Ferme() {  
        animaux = new ArrayList<Animal>();  
    }  
  
    public void ajouteAnimal(Animal a) {  
        animaux.add(a);  
    }  
  
    public void crier() {  
        for(Animal a : animaux)  
            a.crier();  
    }  
}
```

## Exemple : le polymorphisme à la ferme (3/4)

```
public class MainFerme {  
    public static void main(String[] args) {  
        Ferme f = new Ferme();  
        f.ajouteAnimal(new Coq());  
        f.ajouteAnimal(new Vache());  
        f.ajouteAnimal(new Poule());  
        f.ajouteAnimal(new Canard());  
        f.ajouteAnimal(new Coq());  
        f.ajouteAnimal(new Poule());  
        f.ajouteAnimal(new Vache());  
        f.ajouteAnimal(new Coq());  
        f.ajouteAnimal(new Vache());  
        f.ajouteAnimal(new Poule());  
        f.ajouteAnimal(new Canard());  
        f.crier();  
    }  
}
```

## Exemple : le polymorphisme à la ferme (4/4)

- La méthode `f.crier()` manipule uniquement des objets de type `Animal`, mais les méthodes appelées sont les différentes méthodes des sous-classes :



```
<terminated> Mainl  
Cocorico !  
Meuh !  
Cot cot !  
Coin coin !  
Cocorico !  
Cot cot !  
Meuh !  
Cocorico !  
Meuh !  
Cot cot !  
Coin coin !  
|
```

## **Classes de l'API standard**

---

- On a déjà vu que les String sont des objets immuables : lors d'une opération sur l'objet, il n'est pas modifié, mais un autre objet est créé
  - Peu efficace : allocation d'une zone mémoire avec une référence pour y accéder, travail en plus pour le garbage collector lorsque les objets deviennent inutilisables
- Exemple : "Bon" + "jo" + "ur" crée cinq objets
  - Trois objets pour les chaînes "Bon", "jo" et "ur"
  - Un objet pour le résultat de la première concaténation : "Bonjo"
  - Un objet pour le résultat de la seconde concaténation : "Bonjour"

# StringBuffer et StringBuilder (1/2)

- `java.lang.StringBuffer` et `java.lang.StringBuilder` : une chaîne de caractères modifiable avec différentes méthodes
  - `append(...)` pour ajouter quelque chose à la fin de la chaîne
  - `delete(...)` pour retirer une partie de la chaîne
  - `insert(...)` pour insérer quelque chose dans la chaîne
  - `replace(...)` pour remplacer une partie de la chaîne
  - `reverse(...)` pour retourner la chaîne
- On peut éviter de nombreuses créations inutiles d'objets `String` en utilisant ces méthodes. Si on doit récupérer un objet `String` à la fin des manipulations de la chaîne, il suffit d'utiliser la méthode `toString()` du `StringBuffer` ou `StringBuilder`

## StringBuffer et StringBuilder (2/2)

- Différence entre les deux : `StringBuffer` est synchronisé, ce qui signifie qu'elle est sécurisée dans un contexte multi-threads, contrairement à `StringBuilder`
  - On préfère `StringBuilder` dans une situation mono-thread pour gagner en efficacité
  - On préfère `StringBuffer` dans une situation multi-threads pour gagner en sécurité
- Voir la Javadoc pour plus de détails sur ces classes et leurs méthodes



- Il peut être utile d'avoir une représentation sous forme d'objet des types primitifs. Ces classes sont présentes dans le package `java.lang`
  - Boolean
  - Character
  - La classe abstraite `Number` représente les nombres de type quelconque. Classes filles :
    - Byte
    - Short
    - Integer
    - Long
    - Float
    - Double
- Ces classes fournissent des méthodes utiles pour manipuler et convertir les données de types primitifs, on parle de classes enveloppes (*wrappers*)

- Autoboxing : le compilateur peut directement convertir une valeur de type primitif en un objet correspondant
  - `Integer i = 2 ;` : créer un objet de type `Integer` correspondant à la valeur 2 de type **int**
- Le processus inverse est l'unboxing
  - **double** d = **new** Double(2.3) ;
- Cela permet de manipuler plus simplement les classes wrappers et d'écrire du code plus propre
  - par exemple, on n'a pas besoin de convertir soi même un `Integer` en **int** pour le passer en paramètre d'une méthode qui attend un **int**

- Les collections en Java sont un ensemble de classes qui implémentent l'interface générique<sup>3</sup> `Collection <E>` ou une de ses filles
- Une collection est un *groupe d'objets* appelés éléments. Il y a plusieurs interfaces filles selon la sémantique du groupe, notamment :
  - `Set<E>` : ensemble, collection sans doublons
  - `List<E>` : liste, collection ordonnée
  - ...
- Les éléments doivent être du même type `E` (ou un sous-type de `E`)
- On peut parcourir une collection avec une boucle `for each` :

```
Collection<String> col = ... ;  
for(String s : col){  
    System.out.println(s);  
}
```

---

3. Nous reviendrons sur les types génériques plus tard.

- Comme pour un tableau, les listes sont indexées à partir de 0. Les méthodes les plus habituelles sont :
  - **boolean** add(E e) : ajoute l'élément e à la fin de la liste
  - **boolean** contains(Object o) : détermine si l'objet o est dans la liste
  - E get(**int** i) : retourne l'objet à la position i
  - E remove(**int** i) : supprime et retourne l'objet à la position i
  - **boolean** remove(Object o) : supprime la première occurrence de l'objet o
  - **boolean** isEmpty() : détermine si la liste est vide
  - **int** size() : retourne le nombre d'éléments de la liste
- ArrayList<E> : liste basée sur une structure de tableau, idéale si on doit souvent accéder aux éléments en fonction de leur indice
- LinkedList<E> : liste doublement chaînée, idéale si on doit faire de nombreux ajouts/suppressions/déplacements d'éléments

- Un ensemble ne contient pas de doublon, et les éléments ne sont pas indexés (donc pas de méthode `get(int i)`).
  - **boolean** `add(E e)` : ajoute l'élément `e` s'il n'est pas déjà présent
  - **boolean** `contains(Object o)` : détermine si l'objet `o` est dans l'ensemble
  - **boolean** `isEmpty()` : détermine si l'ensemble est vide
  - **int** `size()` : retourne le nombre d'éléments de l'ensemble
- `HashSet<E>` : ensemble basé sur une structure de table de hachage

- Interface `Map<K,V>` : association d'objets de type `K` appelés *clés* à des *valeurs* de type `V`
- On parle de tableau associatif : au lieu d'être associés à un index entier, les éléments sont associés à une clé qui est un objet
  - **boolean** `containsKey(Object key)` : détermine si la clé spécifiée est dans la Map
  - **boolean** `containsValue(Object val)` : détermine si la valeur spécifiée est dans la Map
  - `Set<K> keySet()` : retourne l'ensemble des clés
  - `Collection<V> values()` : retourne les valeurs sous forme de Collection
  - `V get(Object key)` : retourne la valeur associée à la clé, ou **null** s'il n'y en a pas
  - `V put(K key, V val)` : associe la valeur `val` à la clé `key`
- `HashMap<K,V>` : implémentation de Map basée sur une table de hachage