

**Cours**  
**Programmation Orientée Objet 2**  
**Pour**  
**ING 2**

**Chap 00:**

**Rappels sur l'orienté objet**

MEKAHLIA Fatma Zohra LAKRID  
Maître de Conférences Classe B

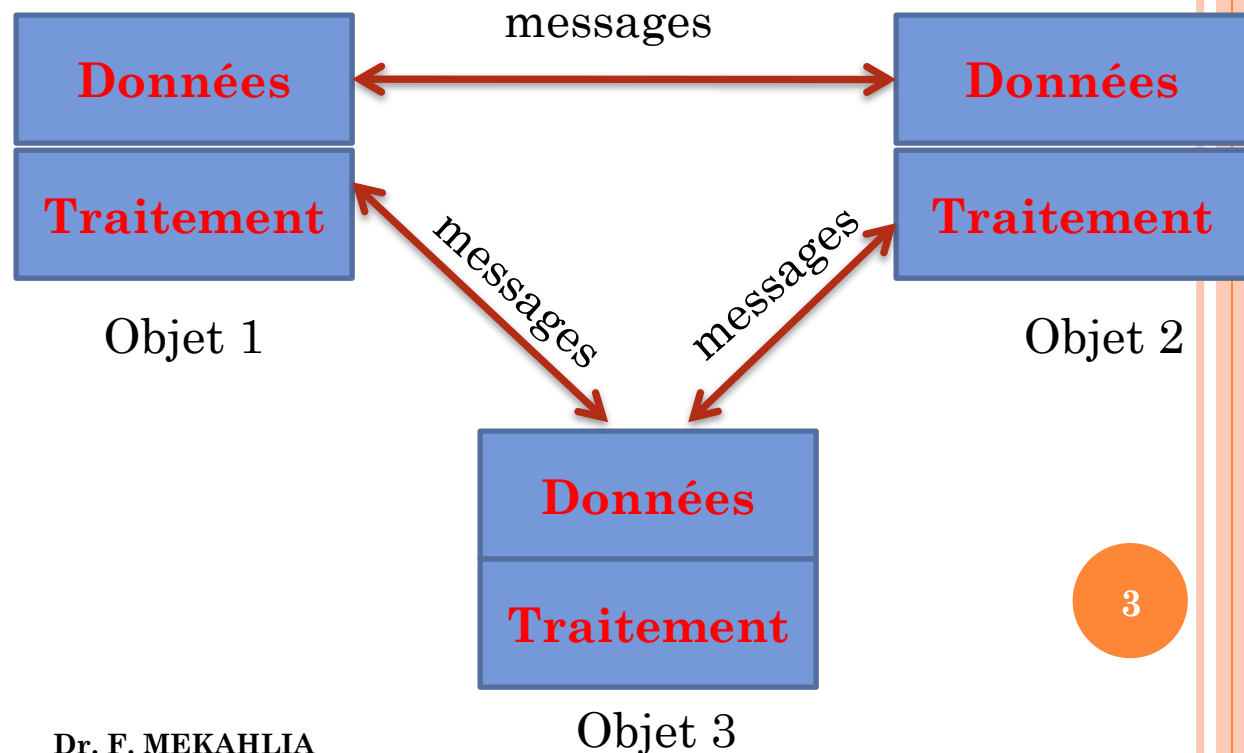
Laboratoire de Modélisation, Vérification et Evaluation des Performances des systèmes complexes (MOVEP)  
Bureau 123

# PLAN

- Introduction
- Object et classe
- Encapsulation
- Mot clé final
- Mot clé null
- Exercice
- Héritage
- Exercice
- Polymorphisme
- Exercice
- Entrées clavier
- Classe abstraite
- Exercices
- Classe interface
- Exercices
- Classe finale
- UML

# INTRODUCTION

- Un programme. O. O est constitué d'un ensemble d'objets chacun disposant d'une partie **fonction** (traitement) et d'une partie **données**.
- Les objets interagissent entre eux par l'envoi des **messages**.



# INTRODUCTION

- Un **objet** est une **variable** de type complexe appelé **Classe**.
- Une **classe** est la définition d'un **type**, alors que l'**objet** est une **déclaration** de variables.
- Après avoir créé une classe, on peut créer autant d'objets que l'on veut de type **classe**.
- Un **objet** est une instance de la classe càd une **valeur particulière de la classe**.

23

# OBJET

- Un objet est une entité :
  - Ayant une **identité**: valeur unique et invariante qui caractérise l'objet.
  - Ayant des **attributs** : capable de sauvegarder un **état** de l'objet c'est-à-dire l'ensemble des valeurs des attributs de cet objet.
  - Ayant des **méthodes**: répondant à des messages précis en déclenchant des activations internes appropriés qui **changent l'état de l'objet**. **Ce sont les traitements** que l'objet réalise.

# CLASSE

- Un objet est une **instance d'une classe**.
- Une classe consiste à créer **un nouveau type**. Une fois créée, la classe devient un type et des objets peuvent être associés à ce type.
- **Une classe** est la représentation de la structure d'une **famille d'objets** partageant des **propriétés** et **méthodes communes**.
- Elle permet la déclaration des attributs (propriétés) ainsi que la définition de l'ensemble de méthodes.

# CLASSE

La définition d'une classe consiste à lui attribuer :

1. Un **nom**,
2. des **attributs**,
3. des **méthodes**,
4. des **constructeurs**, qui permettent de créer des objets ;

”

# CLASSE

- Une classe est un ensemble d'objets qui ont en commun :
  - les mêmes méthodes.
  - les mêmes types d'attributs.
- Classe = attributs + méthodes
- Objet = état (attributs) + comportement (méthodes)



# CLASSE

## ○ Définition d'une classe en java

```
class Rectangle {  
  // définition des Attributs  
  // définition des Méthodes  
}
```

## ○ Conventions de nommage:

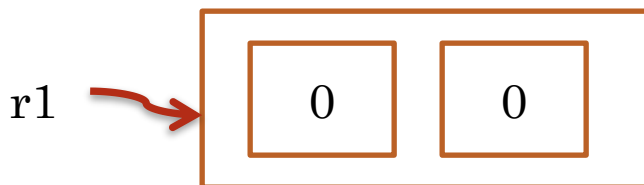
1. Le fichier .java doit avoir le même nom que la classe publique qu'il décrit,
2. le nom de la classe doit débuter par une **M**ajuscule **A**nsi **Q**ue **C**haque **P**remiere **L**ettre **D**e **C**haque **M**ot,
3. un fichier .java par classe, même pour celle contenant le main() .

# INSTANCES D'UNE CLASSE / CRÉATION D'OBJETS

- Pour créer une instance (ou objet) d'une classe, on utilise l'opérateur **new** suivi d'un constructeur de la classe. Bien évidemment, l'objet à créer doit être préalablement déclaré avec le type de la classe adéquate.
- **Rectangle** r1 ; // déclaration de l'objet r1



- r1 = **new** Rectangle( ) ; // création de l'objet r1



# ACCÈS AUX MEMBRES D'UN OBJET

- L'accès à un attribut (**variable d'instance**) ou méthode (**méthode d'instance**) d'un objet donné se fait à l'aide de la notation à point.
- *NomDeObjet.NomAttribut*  
*ou*  
○ *NomDeObjet.NomMethode()*

23

# SURCHARGE DE MÉTHODE

- La surcharge de méthode est un concept qui permet à une classe d'avoir plusieurs méthodes portant le **même nom** et le même **type de retour**, si leurs listes d'arguments sont différentes.

23

# SURCHARGE DE MÉTHODE

- **Nombre de paramètres:** cas valide de surcharge.
  - ❖ `somme(int, int) ;`
  - ❖ `somme(int, int, int);`
- **Type de données des paramètres:** cas valide de surcharge.
  - ❖ `somme(int, int);`
  - ❖ `somme(int, double);`
- **Cas non valide de surcharge de méthode:**
  - ❖ `int somme(int, int);`
  - ❖ `double somme(int, int);`

23

# ENCAPSULATION

- **L'encapsulation** consiste à cacher l'état interne d'un objet et d'imposer de passer par des méthodes permettant un accès sécurisé à l'état de l'objet.
- Pour cette raison, la déclaration d'une classe, d'une méthode ou d'un attribut peut être **précédée par un modificateur d'accès** (visibilité).

23

# MODIFICATEURS DE VISIBILITÉ ET ACCÈS

**public** : toutes les classes peuvent accéder à l'item. La déclaration de variables publiques est contraire au principe d'encapsulation.

- **protected** : seules les classes dérivées et les classes du même package peuvent accéder à l'item (attribut ou méthode) .
- **Private**: un item (attribut ou méthode) private (privé) est accessible uniquement au sein de la classe dans laquelle il est déclaré. Ces éléments ne peuvent être manipulés qu'à l'aide de méthode spécifiques appelés accesseur et mutateur
- **(par défaut)** : sans modificateur d'accès, seules les classes du même package peuvent accéder à l'item

# MODIFICATEURS DE VISIBILITÉ ET ACCÈS

Autres modificateurs :

- **static**: indique, **pour une méthode**, qu'elle peut être appelée sans instancier sa classe i.e. indépendamment de tout objet (méthode de la classe). **Pour un attribut**, qu'il s'agit d'un attribut de classe, et que sa valeur est partagée entre les différentes instances de sa classe (variable de classe).
- **final**: une variable déclarée final est en fait une constante, il n'est plus possible de la modifier. Les méthodes déclarées final ne peuvent pas être remplacée dans une sous classe. Les classes déclarées final ne peuvent pas avoir de sous-classe.



# LE MOT CLÉ FINAL

- Une variable qualifiée de **final** signifie que la variable est **constante**. Une variable déclarée final ne peut plus voir sa valeur modifiée.
- Une **méthode** final ne peut pas être redéfinie dans une sous classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous classée.
- Lorsque le modificateur final est ajouté à une **classe**, il est interdit de créer une classe qui en hérite.

23

# LE MOT CLÉ NULL

- Le mot clé **null** est utilisable partout. Il peut être utilisé à la place d'un objet de n'importe quelle classe ou comme paramètre.
- Il permet de représenter la référence qui ne référence rien.
- Le fait d'initialiser une variable référent un objet à null permet au ramasseur de miettes (garbage collector) de libérer la mémoire allouée.

# EXERCICE

- Définir une classe Rectangle ayant les attributs privées suivants : longueur et largeur et qui se trouve dans le dossier coursPOO.
- Ajouter deux constructeurs d'initialisation: Rectangle(double long, double largeur) et le constructeur d'initialisation à null.
- Ajouter les méthodes suivantes :
- isCarre ( ) : vérifie si le rectangle est un carré.

```
package coursPOO;
```

```
public class Rectangle {  
    private double longueur;  
    private double largeur;
```

```
        public Rectangle(double long, double largeur)  
    {  
        longueur = long;  
        this.largueur = largeur;  
    }
```

```
public Rectangle() { }
```

- public double getLongueur() {
- return longueur;
- }
- 
- public void setLongueur(double longueur) {
- this.longueur = longueur;
- }
- 
- public double getLargeur() {
- return largeur;
- }
- 
- public void setLargeur(double largeur) {
- this.largeur = largeur;
- }

23

```
○ public boolean isCarre() {  
○         if ( (longueur == largeur) &&  
            (longueur <> 0))  
○                 return true;  
○         else  
○                 return false;  
○  
○ }
```

22

# HÉRITAGE

- L'héritage est le troisième paradigme de la programmation orientée objet ( le 1er étant la structure de classe, le 2eme l'encapsulation).
- Une classe peut avoir plusieurs sous classes. Une classe ne peut avoir **qu'une seule classe mère** : il n'y a pas d'héritage multiple en java.

23

# HÉRITAGE

- Grâce à l'héritage, les **objets d'une classe fille** **ont accès** aux **données** et aux **méthodes** de la **classe mère** et peuvent les étendre.
- Les sous classes **peuvent redéfinir** les variables et les méthodes héritées.
- Pour les **variables**, il suffit de **les redéclarer sous le même nom avec un type différent**.
- Les **méthodes** sont **redéfinies** avec le même nom, le mêmes type de retour et le même nombre d'arguments, sinon il s'agit d'une surdéfinition.

23



# HÉRITAGE (ACCÈS AUX DONNÉES )

1. Une méthode d'une classe dérivée n'accède pas aux membres **private** de sa classe de base.
2. Une méthode d'une classe dérivée accède aux membres **protected** de sa classe de base.

23

# HÉRITAGE (REDÉFINITION)

- ❑ **On redéfinit** une méthode quand une nouvelle méthode de la classe fille **a la même signature** qu'une méthode héritée de la classe mère.
- ❑ Consiste à **substituer le corps** d'une méthode par un autre. même nom, même nombre et type d'argument, même type de retour et ne doit pas diminuer les droits d'accès à une méthode.  
**Exemple:** la méthode afficher() est public dans la classe mère ne doit pas devenir private dans la classe fille).

23

# HÉRITAGE (REDÉFINITION)

- **class A**
- { .....
- **double surface ( )** {return (0) ;}
- .....
- }
- **Class B extends A**
- { ....
- **double surface ( )** { return (rayon \* 2\* 3.14) ;}
- ...
- }

# HÉRITAGE (SURDÉFINITION )

- ❑ **La surdéfinition** cumule plusieurs méthodes de même nom avec des arguments (nombre ou type) différents.
- ❑ Si une méthode a le même nom qu'une méthode d'une classe ascendante avec des arguments (nombre ou type) différents, on est dans le cas d'une surdéfinition.
- ❑ La nouvelle méthode est utilisable par la classe dérivée et ses descendantes.

23

# HÉRITAGE (SURDÉFINITION )

- **class A**
- {public void f (int n) {...}}
- .....
- }
- **Class B extends A**
- { public void f (float x) {...}}
- ...
- }

# HÉRITAGE

- **Object** est la classe mère de toutes les classes en java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritage successif toutes les classes héritent d'Object.
- La classe **Object** comporte quelques méthodes qu'on peut soit utiliser telles quelles soit les redéfinir.
- **toString ()** : fournit une chaîne contenant le nom de la classe à laquelle appartient l'objet ainsi que l'adresse de l'objet en hexadecimal.

## Exemple:

- Rectangle r = new Rectangle (3,5);
- System.out.println(" r = " + r.toString() );
- Affiche: r = Rectangle @fc17kjlf

## SUITE DE L'EXERCICE

- toString ( ) : affiche **C'est un carré** ou **Ce n'est pas un carré**.
- Dans la méthode **main** de la classe **TestRectangle**, créer un tableau de type **Rectangle** et qui contient:
  - rectangle1 avec long =5 et larg=8
  - rectangle2 avec long =3.5 et larg=9.1
  - rectangle3 avec long =5 et larg=5
  - rectangle4 avec long =0 et larg=0
- En suite, le programme affiche pour chaque rectangle s'il s'agit d'un carré ou non.

## SUITE DE L'EXERCICE

```
○ public String toString() {  
○         String etat = null;  
○         if (this.isCarre())  
○             etat = "C'est un carré";  
○         else  
○             etat = "Ce n'est pas un carré";  
○         return etat;  
○     }
```



# SUITE DE L'EXERCICE

```
package coursPOO;
```

```
    public class TestRectangle {  
        public static void main(String[] args) {  
            Rectangle [] rectangles = new Rectangle[4];  
            rectangles [0] = new Rectangle(5, 8);  
            rectangles [1] = new Rectangle(3.5, 9.1);  
            rectangles [2] = new Rectangle(5, 5);  
            rectangles [3] = new Rectangle();  
  
            for(Rectangle e: rectangles)  
                System.out.println(e.toString());  
        }  
    }
```

# HÉRITAGE (APPEL DE CONSTRUCTEUR )

La première instruction d'un constructeur peut être un appel: à un constructeur de la classe mère par **super(...)** ou à un autre constructeur de la classe fille par **this(...)**.

Appeler le constructeur de la classe mère garantit que l'on peut initialiser les arguments de la classe mère.

Si l'on n'indique pas **super()**, il y a un appel du constructeur par défaut de la classe mère.

Si la première instruction d'un constructeur n'est ni **super(...)**, ni **this(...)**, le compilateur ajoute un appel implicite au constructeur sans paramètre de la classe mère (erreur s'il n'existe pas)

23

# EXERCICE

- Soit la classe RectangleColor qui hérite de la classe Rectangle déjà vue.

```
class RectangleColor extends Rectangle {  
    private String color;
```

```
    public Rectangle(double long, double largeur, String  
        color) {
```

```
        super (long, largeur); // Obligatoirement
```

```
        this. color= color;
```

```
    }
```

```
    public Rectangle(){
```

```
        this (5,6,bleu); // appel du constructeur de la classe même
```

```
    }
```

23

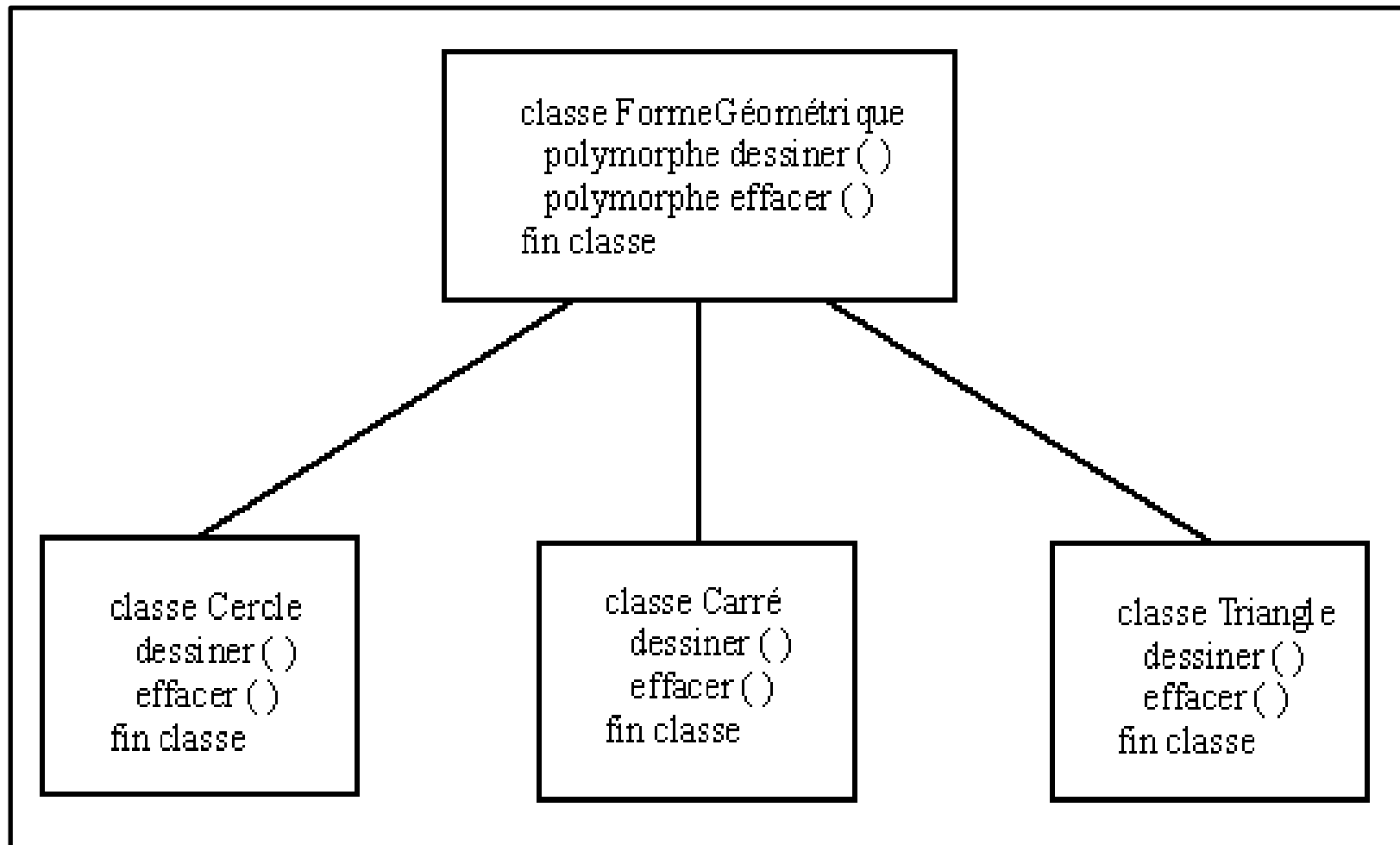
# POLYMORPHISME

- PolyMorphisme ?
  - Poly: plusieurs.
  - Morphisme: forme.

# POLYMORPHISME

- La première catégorie de polymorphisme, que vous avez déjà vu c'est **la surcharge du constructeur**
- vous connaissez également: le **polymorphisme par héritage de méthode**, lié à la redéfinition: Il explique comment une méthode peut se comporter suivant l'objet **sur lequel elle s'applique**, ie, quand une même méthode est définie à la fois dans la classe **mère** et dans la classe **fille**, son exécution est réalisée **en fonction de l'objet associé à l'appel**.

# POLYMORPHISME



# POLYMORPHISME

- **Exemple:** les méthodes *dessiner()* et *effacer()* de la classe géométrique sont polymorphes. Leur nom est similaire dans les trois classes dérivées (Cercle, Carré et Triangle), mais dessiner un cercle est différent de dessiner un carré ou un triangle.
- Ainsi le **polymorphisme** se résume par : un même nom et plusieurs implémentations.

# POLYMORPHISME

- Nous avons une autre catégorie de polymorphisme, appelé **polymorphisme d'objets**.
- C'est un concept très puissant en P.O.O, qui complète l'**héritage**. Il se base sur le concept d'héritage et la redéfinition des méthodes.
- Le polymorphisme d'objet se base sur cette affirmation : un objet a comme type non seulement sa classe mais aussi n'importe quelle classe dérivée



# EXERCICE

- Créer un tableau d'objets, les uns étant de type **Point** et les autres de type **PointCol** et appeler la méthode `afficher()`.
- Chaque objet réagira selon son type.
- Les objets points colorés se sont aussi des points et peuvent donc être traités comme des points,

23

# EXERCICE

Soient les classes Point et PointCol suivantes:

```
class Point {  
    private double x;  
    private double y;  
    Public Point (double x, double y)  
    {  
        this.x = x ;  
        this.y = y ;  
    }  
    public void afficher()  
    {  
        System.out.println("je suis un point de  
        coordonnées:" + x + " " + y);  
    }  
}
```

```
class PointCol extends Point {  
    private String couleur;  
  
    Public PointCol (double x, double y,  
        String couleur) {  
        super(x,y);  
        this.couleur = couleur;  
    }  
    public void afficher() {  
        super.afficher();  
        System.out.println("couleur:" + couleur);  
    }  
}
```

# EXERCICE

```
class TestPolymorphisme {  
    Public static main (String args[])  
    {  
        Point p;  
        p = new Point(3,5);  
        p.afficher();  
        p = new PointCol (4,9,"Vert");  
        p.afficher();  
    }  
}
```

appelle la méthode afficher de la classe Point

*je suis un point de coordonnées: 3 5*

appelle la méthode afficher de la classe PointCol

*je suis un point de coordonnées: 4 9  
vert*

L'instruction p. afficher() se base non pas sur le type de la variable p mais sur le type effectif de l'objet référencé par p au moment de l'appel car celui-ci peut évoluer dans le temps.

**Règle** : Le choix de la méthode appelée se fait selon le type effectif de l'objet référencé au moment de l'exécution

Ce choix se fait au moment de l'exécution et non pas au moment de la compilation.

# EXERCICE

- La méthode **afficher()** est décrite dans la classe Point et dans la classe PointCol.
- Les deux méthodes **afficher()** sont définies sans aucun paramètre.
- Le choix de la méthode ne peut donc s'effectuer sur la différence de paramètres. Il est effectuée par rapport à l'objet sur lequel la méthode est appliquée

23

# EXERCICE

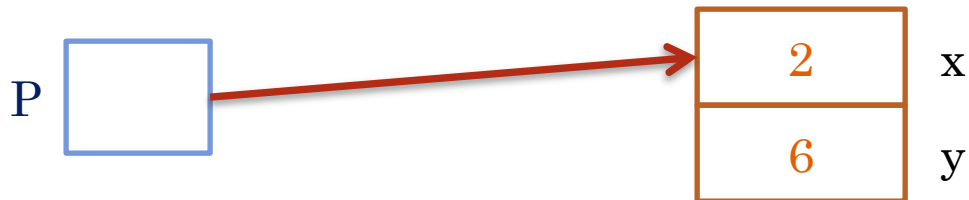
- Exemple:

*/\*\* même référence \*/*

Point p;

p = new Point(2,6);

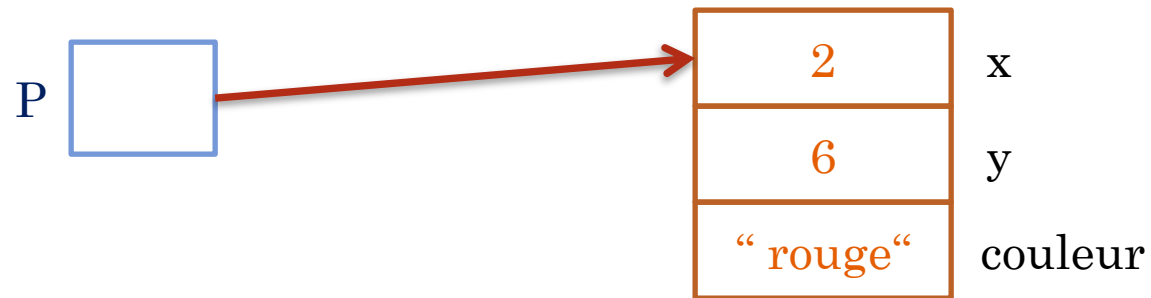
- On aboutit tout naturellement à cette situation



# EXERCICE

/\*\* changement de référence \*/

- Point p;
- p = new PointCol (2,6,"rouge");



**NB:** Malgré que **p** est de type **Point**, Java autorise cette affectation !!

**Règle :** Java permet à une variable objet l'affectation d'une référence à objet d'un type dérivé.

# POLYMORPHISME ET TABLEAUX

- Le polymorphisme peut s'appliquer à des tableaux d'objets.

```
1 package polymorphisme;
2
3 public class Personne {
4
5     public void parler () {
6         System.out.println("cette personne parle");
7     }
8 }
-
```

# POLYMORPHISME ET TABLEAUX

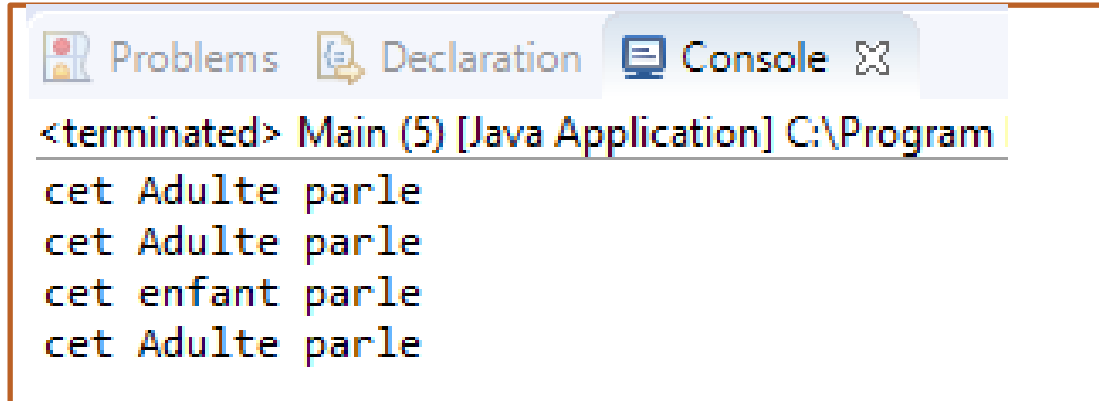
```
1 package polymorphisme;
2
3 public class Adulte extends Personne{
4
5     @Override
6     public void parler() {
7         System.out.println("cet Adulte parle");
8     }
9 }
```

```
1 package polymorphisme;
2
3 public class Enfant extends Personne{
4
5     @Override
6     public void parler() {
7         System.out.println("cet enfant parle");
8     }
9 }
```

```
1 package polymorphisme;
2
3 import java.util.ArrayList;
4
5 public class Main {
6     public static void main(String[] args) {
7         ArrayList <Personne> P = new ArrayList<> ();
8         P.add(new Adulte());
9         P.add(new Adulte());
10        P.add(new Enfant());
11        P.add(new Adulte());
12
13        for(Personne personne:P) {
14            personne.parler();
15        }
16    }
17 }
```



# POLYMORPHISME ET TABLEAUX



The screenshot shows a Java IDE console window with three tabs: 'Problems', 'Declaration', and 'Console'. The 'Console' tab is active, displaying the output of a Java application. The output consists of four lines of text: '<terminated> Main (5) [Java Application] C:\Program', 'cet Adulte parle', 'cet Adulte parle', 'cet enfant parle', and 'cet Adulte parle'. The text is in a monospaced font, typical of a terminal or console window.

```
<terminated> Main (5) [Java Application] C:\Program  
cet Adulte parle  
cet Adulte parle  
cet enfant parle  
cet Adulte parle
```

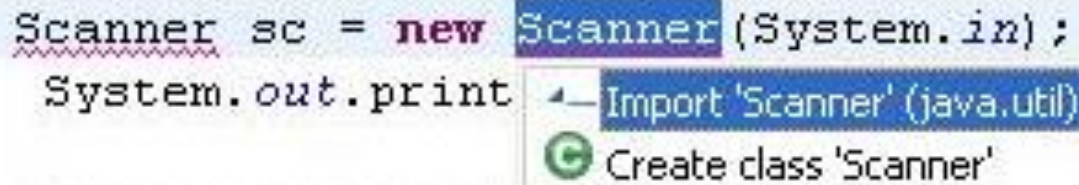
23

# LIRE LES ENTRÉES CLAVIER

- Pour que Java puisse lire ce que vous tapez au clavier, vous allez utiliser un objet de type **Scanner**.
- Lorsque vous faites **System.out.println()**, je vous rappelle que vous appliquez la méthode **println()** sur la sortie standard ; or ici, nous allons utiliser l'entrée standard **System.in**.
- Donc, avant de dire à Java de lire ce que nous allons taper au clavier, nous devons instancier un objet **Scanner**.
- **Scanner** sc = **new Scanner** (System.in);

# LIRE LES ENTRÉES CLAVIER

```
// ...  
  
Scanner sc = new Scanner(System.in);  
System.out.print
```



# LIRE LES ENTRÉES CLAVIER

- l'instruction **nextLine()** renvoie une chaîne de caractères.

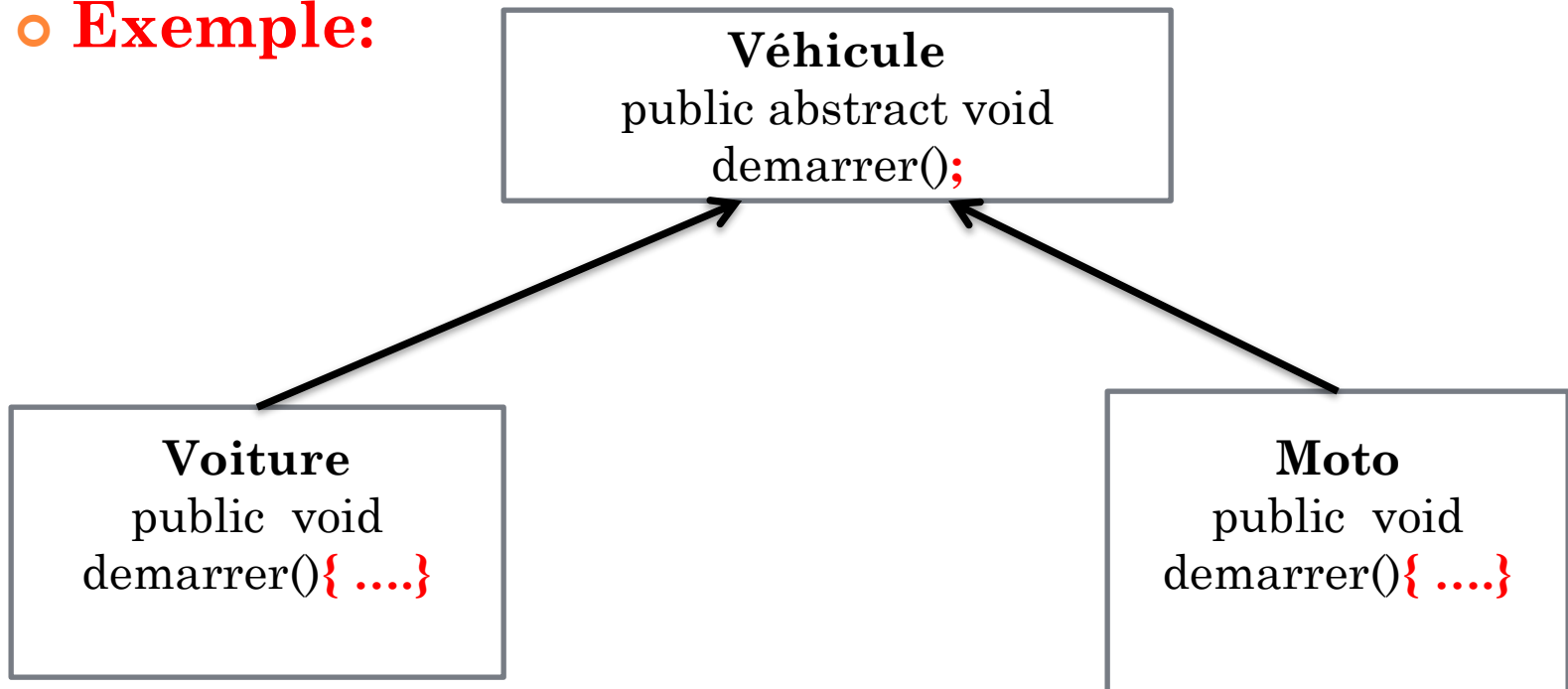
```
1 import java.util.Scanner;
2 public class Sdz {
3
4     public static void main(String[] args){
5         Scanner sc = new Scanner(System.in);
6         System.out.println("Veuillez saisir un mot :");
7         String str = sc.nextLine();
8         System.out.println("Vous avez saisie : " + str);
9     }
```

```
1 Scanner sc = new Scanner(System.in);
2 int i = sc.nextInt();
3 double d = sc.nextDouble();
4 long l = sc.nextLong();
5 byte b = sc.nextByte();
6 //etc
```

# Classe abstraite

# MÉTHODE ABSTRAITE

- Les méthodes abstraites sont principalement déclarées dans la classe de base lorsque deux ou plusieurs sous-classes font également la même méthode de différentes manières à travers des implémentations différentes.
- Exemple:**



# CLASSE ABSTRAITE

- Une classe qui comporte une ou plusieurs méthodes abstraites (des méthodes dont on fournit uniquement la signature ainsi que le type de retour.) est **abstraite**, même si on n'indique pas le mot clé **abstract** à sa déclaration. Et de ce fait, elle ne sera pas **instanciable** et elle ne peut servir que de classe de base pour être dérivée.
- Exemple:

```
abstract class Vehicule {  
    // variables et autre méthodes  
    public abstract void demarrer () ;  
}
```

# CLASSE ABSTRAITE

- L'intérêt des classes abstraites est de regrouper plusieurs classes sous un même nom de classe ainsi que de **décrire partiellement des attributs et méthodes communs à plusieurs classes.**
- Une méthode abstraite doit obligatoirement être déclarée publique (puisque'elle est destinée à être redéfinie dans une classe dérivée),

23



# CLASSE ABSTRAITE

- Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites de sa classe de base et peut même n'en redéfinir aucune et restera abstraite elle-même,
- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et / ou contenir des méthodes abstraites.

23

# EXERCICE

- Créer une classe *Vehicule* qui peut retourner son nombre de roues à l'aide d'une méthode. Chaque véhicule possède une marque et un nombre max de vitesse.
- Créer deux classes concrètes (*Voiture* et *Moto*) qui héritent de *Vehicule* et qui doivent, maintenant, fournir une implémentation de la méthode *getNbRoues* pour pouvoir compiler.

# EXERCICE

```
public abstract class Vehicule {  
    private String marque;  
    private int vitesse;  
  
    public Vehicule() ;  
    public Vehicule(String marque, int vitesse) {  
        this.marque = marque;  
        this.vitesse = vitesse;}  
  
    public abstract int getNbRoues();  
  
}
```

# EXERCICE

```
public class Voiture extends Vehicule {
```

```
    @Override
```

```
    public int getNbRoues() {
```

```
        return 4; }
```

```
}
```

23

# EXERCICE

```
public class Moto extends Vehicule {
```

```
    @Override
```

```
    public int getNbRoues() {
```

```
        return 2; }
```

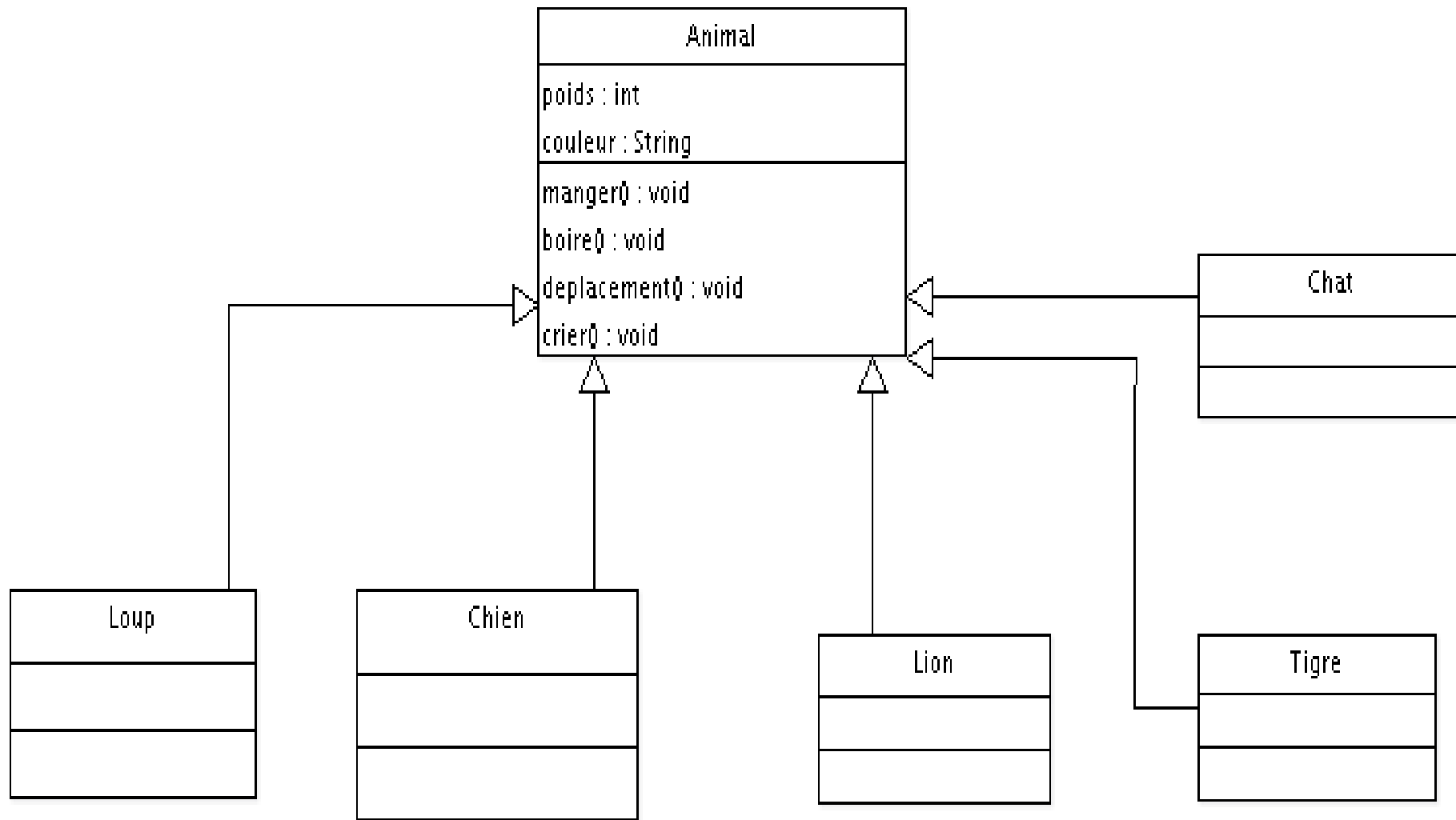
```
}
```

23

# EXEMPLE D'APPLICATION

- Notre programme intéresse à la gestion de différents types d'animaux tel que: **des loups, des chiens, des chats, des lions, des tigres.**
- Que pouvons-nous définir de commun à tous ces animaux ? : **une couleur, un poids, qu'ils crient, qu'ils se déplacent, qu'ils mangent, qu'ils boivent.**
- Nous pouvons donc faire une classe mère, appelons-la **Animal !!**
- Voici donc à quoi pourraient ressembler nos classes:

# EXEMPLE D'APLICATION



Nous avons bien notre classe mère **Animal** et nos animaux qui en héritent.

# EXEMPLE D'APLICACION

- Il est possible de créer un objet **Animal** ? Quel est son poids, sa couleur, que mange-t-il ?

```
1  public class Test{
2      public static void main(String[] args){
3          Animal ani = new Animal();
4          ani.manger();//que doit-il faire ? ?
5      }
6  }
```

- On ne sais pas comment mange un objet **Animal**... Donc il faut empêcher la classe **Animale** d'être instanciable !!
- Pour répondre à ce besoin: la classe **Animal** doit être une classe **Abstraite**.

```
1  public class Test{
2      public static void main(String[] args){
3          Animal ani = new Animal();//Erreur de compilation ! !
```



# EXEMPLE D'APPLICATION

```
1  abstract class Animal{
2      abstract void manger(); //une méthode abstraite
3  }
```

- Pour pouvoir utiliser les méthodes abstraites de animal, nos classes enfants seront **OBLIGÉES** de redéfinir ces méthodes !

```
1  public class Test{
2
3      public static void main(String args[]){
4
5          Animal loup = new Loup();
6          Animal chien = new Chien();
7          loup.manger();
8          chien.crier();
9
10     }
11 }
```

Nous avons instancié un objet **Loup** que nous avons mis dans un objet de type **Animal** qui n'a pas été instancié !!!

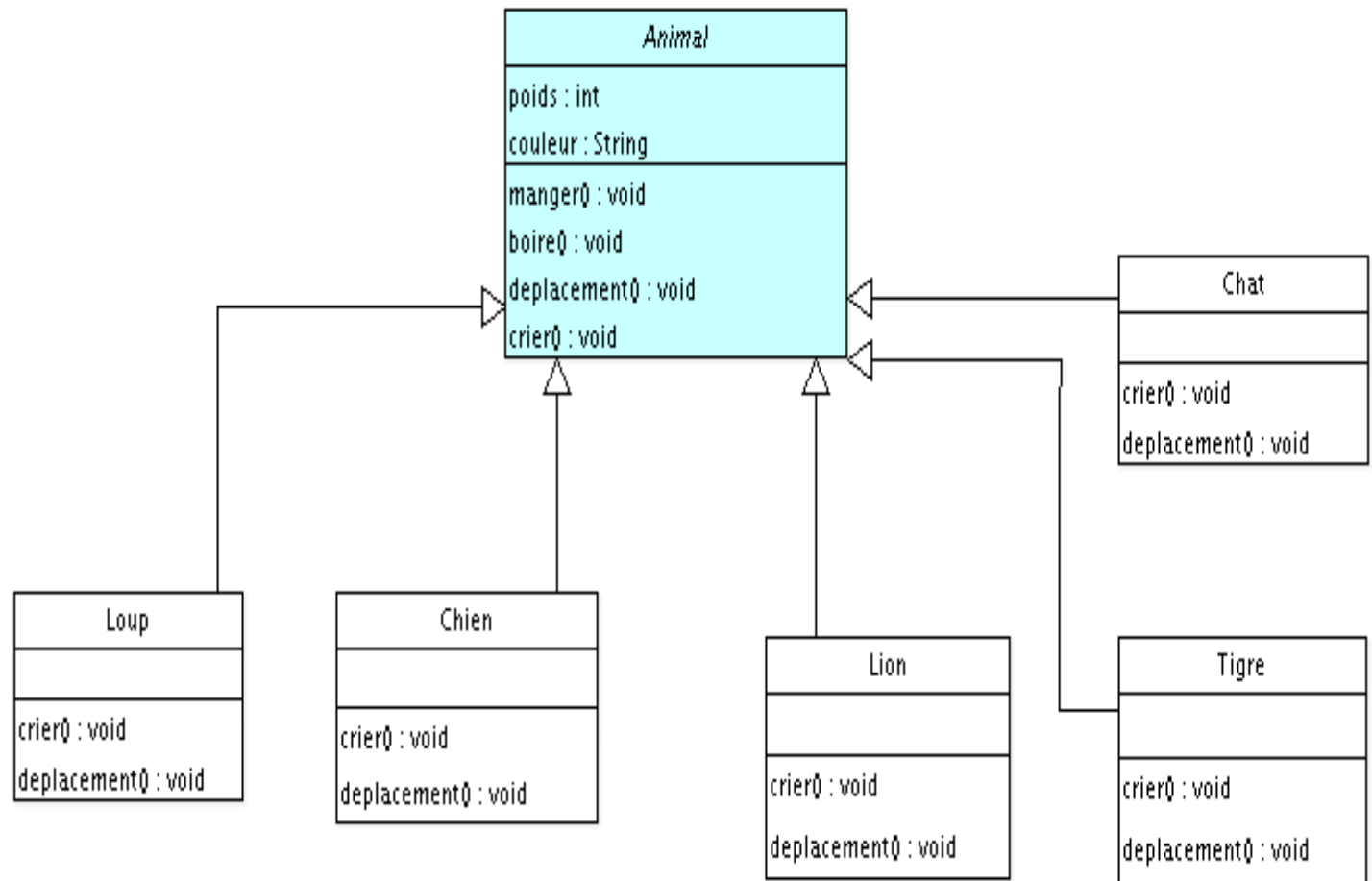
## EXEMPLE D'APLICACION

- Dans ce programme, nos objets auront tous une couleur et un poids différents. Nos classes auront donc le droit de modifier ceux-ci.
- tous nos animaux mangeront de la viande. Donc, la méthode **manger()** sera définie dans la classe **Animal**.
- Idem pour la méthode **boire()**. Ils boiront tous de l'eau .
- Par contre, ils ne crient pas et ne se déplaceront pas de la même manière. Nous ferons donc des méthodes polymorphes et déclarerons les méthodes **crier()** et **deplacement()** abstraites dans la classe **Animal**.

23

# EXEMPLE D'APLICATON

Voici ce que donneraient nos classes :

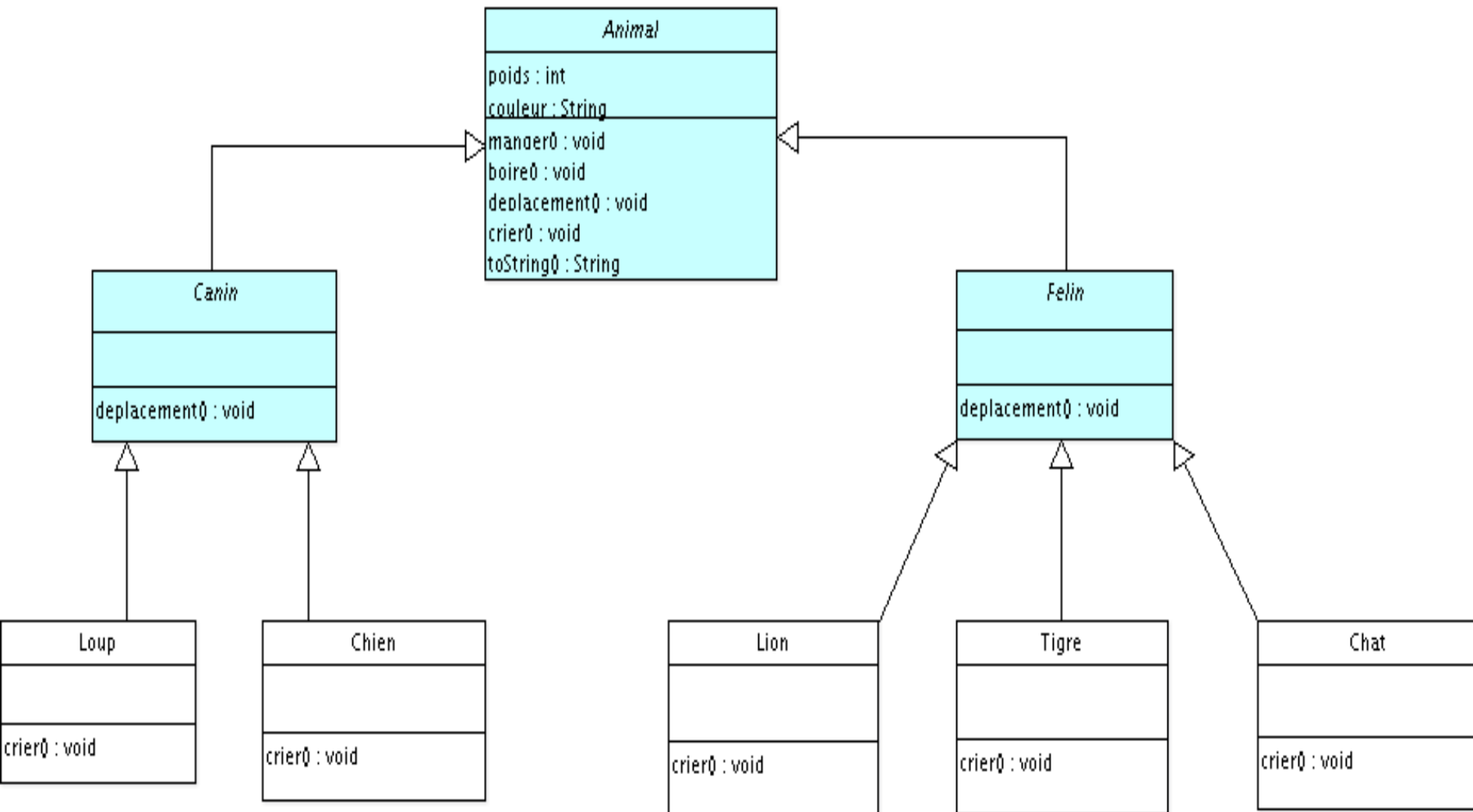


# EXEMPLE D'APLICATON

- Maintenant, nous voulons créer deux sous classes qui permettent de préciser la manière de déplacement de chaque animal !! les **félins** se déplacent d'une certaine façon, et les **canins** d'une autre.
- **Implémenter les classes de diagramme de classe suivant. Créer une classe de Test qui affiche le résultat suivant:**

```
Je bois de l'eau !  
Je mange de la viande  
Je me déplace en meute !  
J'hurle à la lune ...  
Je suis un objet de la class Loup, je suis Gris bleuté, je pèse 20
```

# EXEMPLE D'APLICATION



# Classe Interface

# CONCEPT D'INTERFACE

- Si on considère une classe abstraite **n'implantant aucune méthode et aucun champ** (sauf les constantes), on aboutit à la notion **d'interface**.
- Une interface c'est une collections de méthodes abstraites.
- Une interface représente une classe abstraite a 100%.
- Une classe X peut **implémenter** plusieurs interfaces.
- Les interfaces peuvent se dériver.

23

# DÉFINITION D'UNE INTERFACE

- La définition d'une interface est identique à la définition des classes, en remplaçant le mot clé **class** par **interface**.
- Les méthodes d'une interface sont toutes **abstraites** et **publiques**, il n'est donc pas obligatoire de le mentionner comme abstraites. **Exemple:**

**public interface** NomInterface {...}

```
public interface I
{
    final int MAX = 100;
    void f(int n); // public et abstract sont facultatifs
    void g();
}
```



# CONCEPT D'INTERFACE

- Maintenant, nous voulons utiliser l'architecture précédente de classe abstraite dans une autre application où les chiens vont devoir apprendre à faire de nouvelles choses comme : faire des câlins.
- Donc, on vas ajouter cette méthode dans la classe **Animal !!**
- **Mais NON**, car on auras des lions qui vont faire des câlins !!
- Dans ce cas, on n'a qu'à mettre cette méthode dans la classe **Chien !**
- **Mais avec cette solution**, vous ne pourrez pas appeler vos objets Chien par le biais d'un super type. Pour pouvoir accéder à cette méthode, vous devrez obligatoirement passer par une référence à un objet **Chien. ADIEU LE POLYMORPHISME !**

# CONCEPT D'INTERFACE

- Pour utiliser au mieux le **polymorphisme**, nous devons définir les méthodes au plus haut niveau de la hiérarchie. Et comme l'héritage multiple est interdit en Java !!
- Il faudrait pouvoir développer un nouveau super type qui est **les interfaces**. Donc, nous allons créer l'interface **Cal** pour ensuite l'implémenter dans notre objet **Chien**, comme suit:

```
public interface Cal{  
    public void faireCalin(); }
```

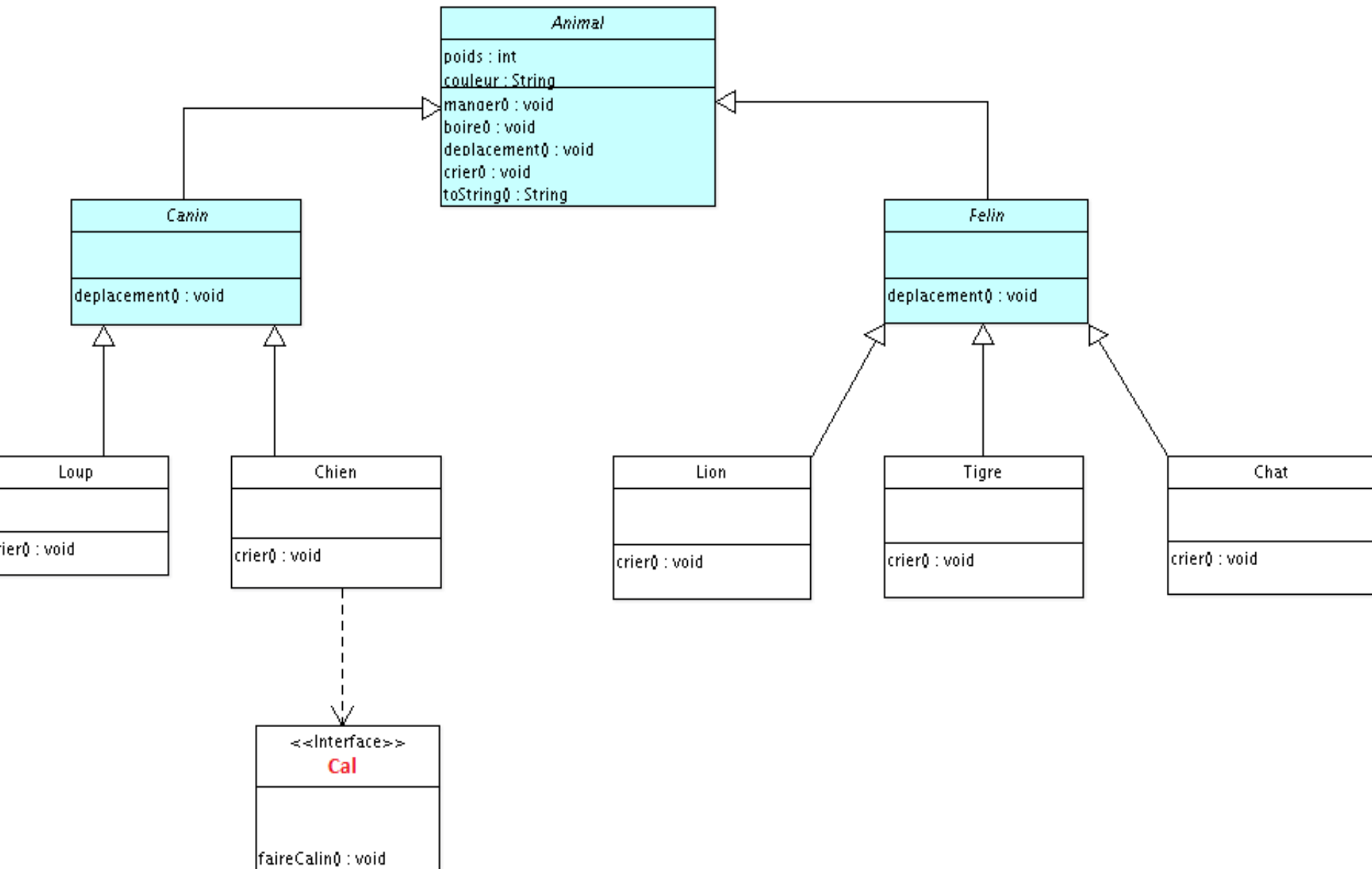
- il ne nous reste plus qu'à implémenter l'interface dans notre classe **Chien**. Ce qui nous donne :

# CONCEPT D'INTERFACE

```
public class Chien extends Canin implements Cal {  
    public Chien(){ }  
    public Chien(String couleur, int poids){  
        this.couleur = couleur;  
        this.poids = poids; }  
    void crier() {  
        System.out.println("J'aboie sans raison ! "); }  
    public void faireCalin() {  
        System.out.println("Je te fais un GROS CÂLIN"); }  
}
```

L'ordre des déclarations est PRIMORDIAL. Vous DEVEZ mettre **l'expression d'héritage AVANT l'expression d'implémentation**, SINON votre code ne compilera pas !

# CONCEPT D'INTERFACE



# IMPLÉMENTATION D'UNE INTERFACE

- Une classe peut **implémenter** une ou plusieurs interfaces

## Exemple

class A implements I

```
{ /** A doit redéfinir toutes les méthodes de I  
    sinon erreur à la compilation */ }
```

## Exemple

```
public interface I1  
{ void f();  
}  
public interface I2  
{void g()  
}  
class A implements I1,I2  
{// A doit définir les méthodes f et g  
}
```

# IMPLÉMENTATION D'UNE INTERFACE

- Supposons que la méthode « g » est présente de 2 façons différentes (`void g()` et `int g()` ) dans I1 et I2.

## Exemple:

```
class A implements I1, I2 {
```

```
    /** Erreur car void g() et int g() ne peuvent pas  
        coexister au sein de la même classe d'après les  
        règles de la surcharge des méthodes***/
```

# INTERFACE ET CLASSE DÉRIVÉE

- Le mot clé **implements** est une garantie de la part d'une classe d'implémenter toutes les méthodes proposées dans une interface. Une classe dérivée peut implémenter une ou plusieurs interfaces.

## Exemple 1:

```
interface I
{
    void f(int n);
    void g();
}
class A {...}
class B extends A implements I
{
    // les méthodes f et g doivent
    être soit déjà définies dans A
    soit définies dans B
}
```

## Exemple 2:

```
interface I1{...}
interface I2{...}
class A implements I1{...}
class B extends A implements I2
{...}
```

# DÉRIVATION D'UNE INTERFACE

- On peut définir une interface comme étant **dérivée d'une autre interface** (mais pas d'une classe) en utilisant le mot clé **extends**. La dérivation d'interfaces revient simplement à **concaténer les déclarations** et **n'est pas aussi riche que celles des classes**.

```
interface I1
{
    static final MAXI = 100;
    void f (int n);
}
interface I2 extends I1
{
    static final MINI = 10;
    void g();
}
```



```
interface I2
{
    static final MINI = 10;
    static final MAXI = 100;
    void f (int n);
    void g();
}
```



# DÉRIVATION D'UNE INTERFACE

- Une interface peut dériver de plusieurs interfaces. L'héritage multiple est autorisé pour les interfaces.

```
interface I1
{ void f();
}
interface I2
{ void f1();
}
interface I3 extends I1,I2
{ void f2();
}
```

# AVANTAGE DE CLASSE INTERFACE

- Soit les classes définies comme suit:
- Public **interface** Animal {.....}
- Public **class** Chat **implements** Animal {...}
- On ne peut créer un objet de la classe **Animal**,
- On peut créer des objets de la classe **Chat** ainsi:
  - **Chat** cat = new Chat();
  - ou
  - **Animal** cat=new Chat();
- La conversion est implicite en Java.
- En utilisant la liaisons dynamiques de Java.

```

public interface Vehicule {
    int nbr_roues=4; // automatiquement final
    public void klaxonner() ; // automatiquement abstract
}
Class Camion implements Vehicule{
    @Override
    public String klaxonner() { return « BomBom » ;}
}
Class Voiture implements Vehicule{
    @Override
    public String klaxonner() { return « TitTit» ;}
}

```

Dans main, nous pouvons créer les objets enfants ainsi:

```
Voiture v = new Voiture();
```

Ou

```
Vehicule v = new Voiture();
```

# EXEMPLE DE L'INTERFACE COMPARABLE DE JAVA

- L'interface **Comparable** de Java consiste en une seule méthode (et pas de constantes) **int compareTo(T obj)** qui compare l'objet à un objet de type T .
- **A.compareTo(B)** retourne un entier négatif si l'objet A est plus petit que B, zéro si les deux objets sont égaux et un entier positif si l'objet A est plus grand que l'objet B.

## Exemple:

- "Bonjour ".compareTo ("Bonsoir ") renvoie un entier négatif.

# EXEMPLE DE L'INTERFACE COMPARABLE DE JAVA

- Une classe implémente l'interface **Comparable** si ses objets peuvent être ordonnés selon un ordre particulier.
- Par exemple la classe **String** implémente **Comparable** parce que les chaînes de caractères peuvent être ordonnées selon l'ordre alphabétique.
- Les classes numériques comme **Integer** et **Double** implémentent **Comparable** parce que les nombres peuvent être ordonnés selon l'ordre numérique.

23

# Exercices

# EXERCICE

- Créez une classe `CompteEnBanque` munie d'un seul attribut `solde`, et définissez l'égalité de deux objets `CompteEnBanque`, de telle manière qu'elle soit vérifiée dès que les deux soldes sont égaux.

23

# SOLUTION

```
class CompteEnBanque implements Comparable{

    private float solde;

    public float getSolde(){ return solde;}
    public void setSolde(float s){ solde=s;}

    public int compareTo(Object cmp) {

        float solde2 = ((CompteEnBanque)cmp).getSolde();

        if(this.solde>solde2)
            return 1;
        else if(this.solde <solde2)
            return -1;
        else
            return 0;

    }
}
```

33



# EXERCICE: INTERFACE

```
interface Affichable{  
    void afficher() ;  
}
```

```
class Entier implements  
Affichable{  
  
    private int val ;  
    public Entier (int n) {val = n ;}  
  
    public void afficher() {  
        System.out.println (« Je suis  
un entier de valeur » + val) ;  
    }  
}
```

```
class Flottant implements  
Affichable{  
  
    private float val ;  
    public Flottant (float n) {val = n ;}  
  
    public void afficher() {  
        System.out.println (« Je suis un  
flottant de valeur » + val) ; }  
}
```

# EXERCICE: INTERFACE

```
public class TestInterface {  
    public static void main (String [] args) {  
        Affichable [] tab = new Affichable [2];  
        tab[0] = new Entier (25);  
        tab[1] = new Flottant (1.25);  
        tab[0].afficher();  
        tab[1].afficher(); } }
```

## Résultat de l'exécution

- Je suis un entier de valeur 25
- Je suis un flottant de valeur 1.25

# Classe Finale

# CLASSE FINALE

- Ne peut être hérité par aucune autre classe ni de redéfinir ses méthodes.
- L'intérêt est de protéger les classes et leurs implémentations.
- Définie avec le mot clé **final**.

## Syntaxe:

```
public final class NomClasse{...}
```

23

# CLASSE FINALE

- La classe **String** de Java est une classe final.
- Il est impossible de créer des sous classes de String, par mesure de sécurité Java.
- La classe **Math** de Java également.

23

# UML

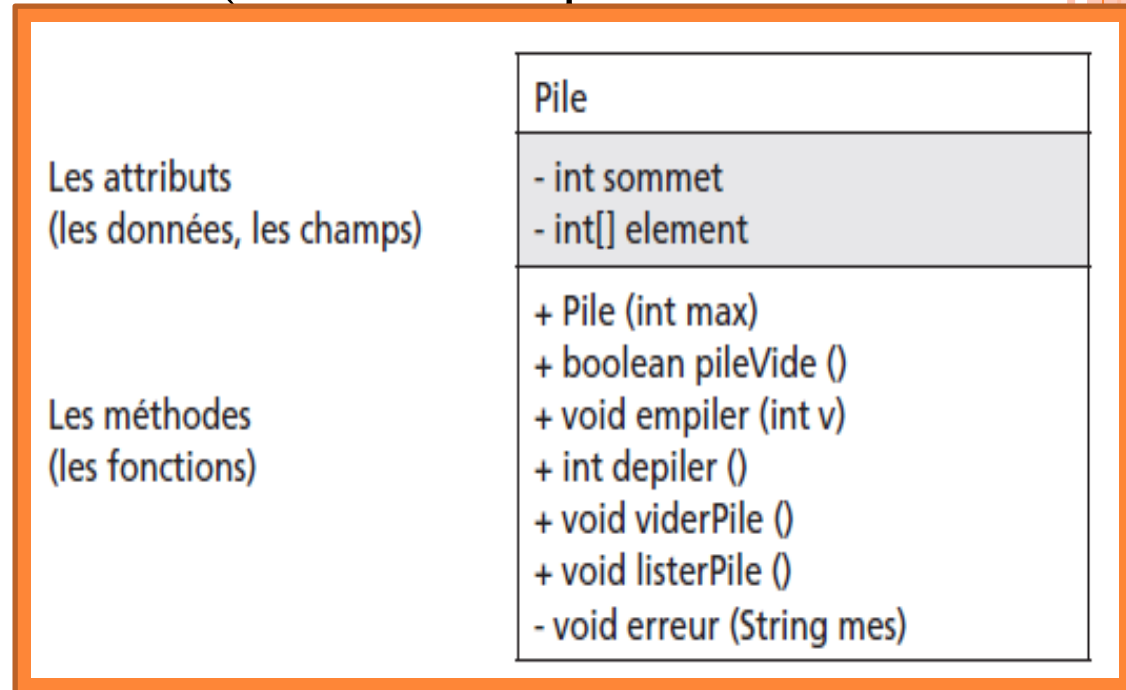
# REPRÉSENTATION D'UNE CLASSE EN UML

- UML, c'est l'acronyme anglais pour « Unified Modeling Language ». On le traduit par « Langage de modélisation unifié ». La notation UML est un langage visuel constitué d'un ensemble des diagrammes qui donnent chacun une vision différente du projet à traiter. UML nous fournit donc des diagrammes pour représenter le logiciel à développer : son fonctionnement, sa mise en route, les actions susceptibles d'être effectuées par le logiciel, etc.
- Dans ce module, nous utiliserons le diagramme de classe.

# REPRÉSENTATION D'UNE CLASSE EN UML

Rectangle composé de quatre parties:

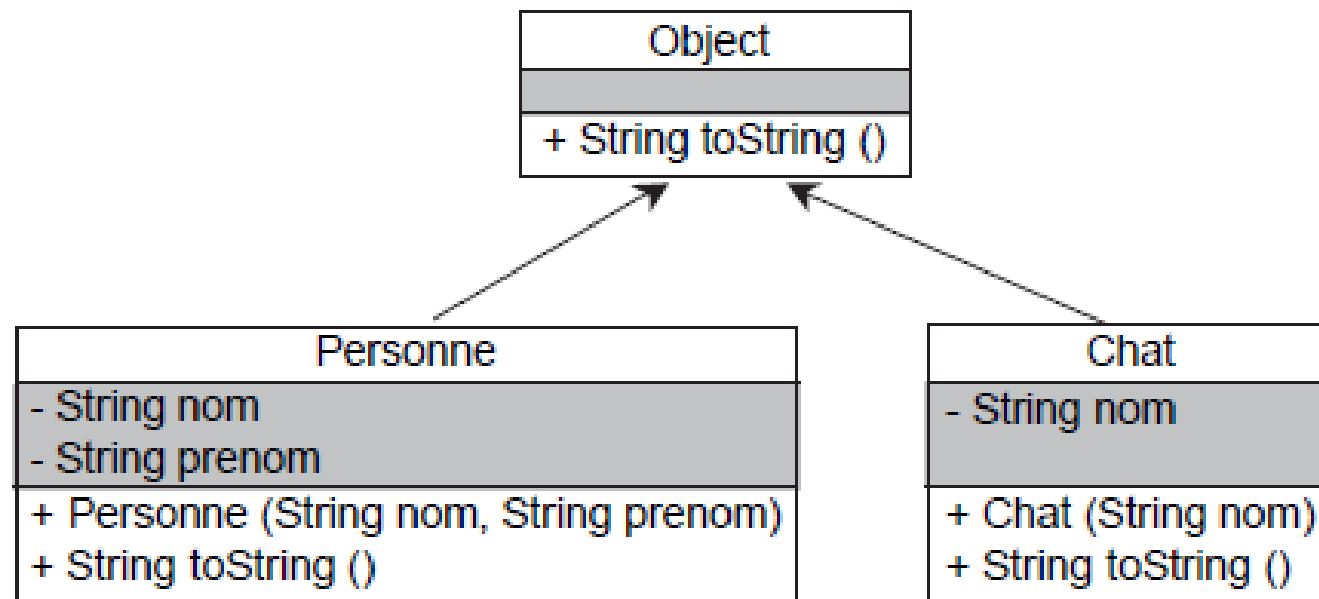
- **Partie 1:** Nom de la classe (commence par une majuscule, en gras)
- **Partie 2:** Attributs
- **Partie 3:** Méthodes



- indique un attribut / méthode **private**  
+ indique un attribut / méthode **public**  
# indique un attribut / méthode **protected**  
**Rien** indique un attribut / méthode **par défaut**

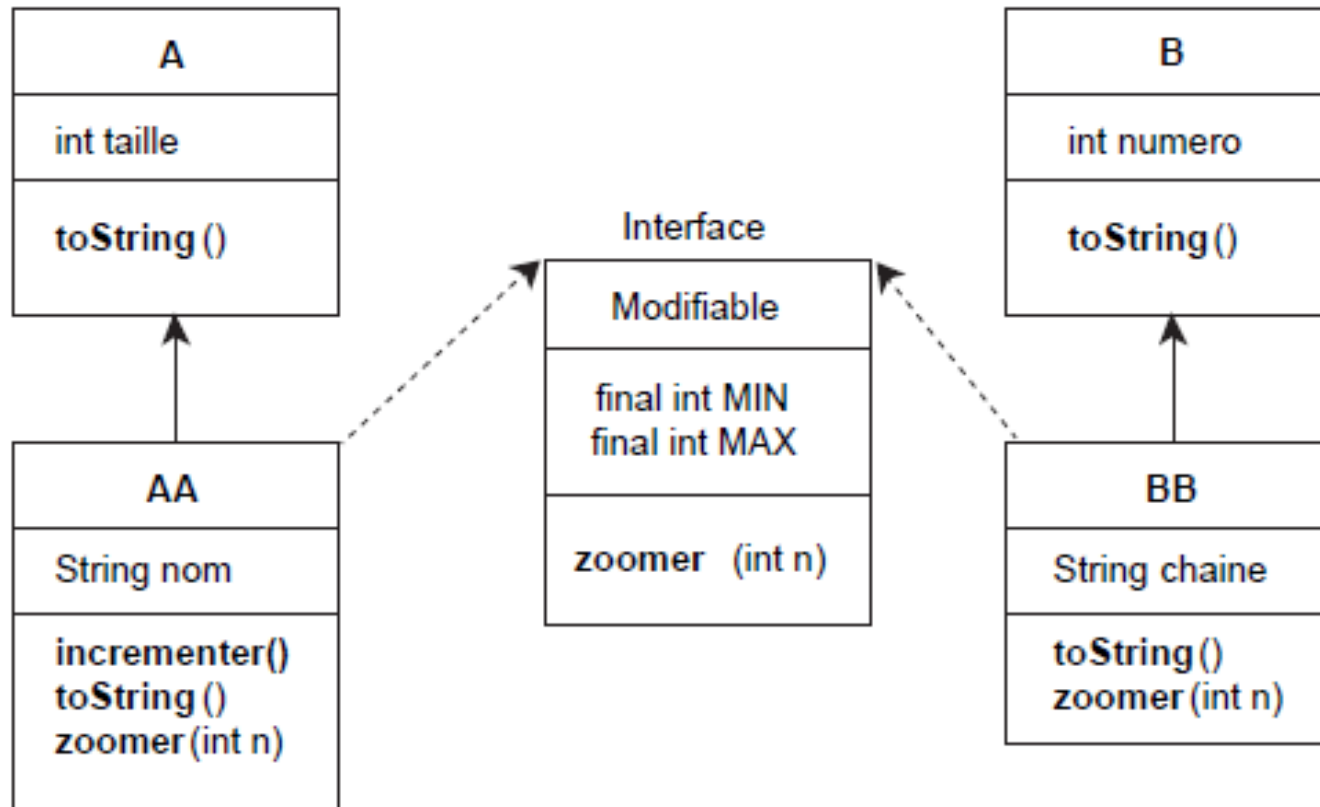


# EXEMPLE DE MODÉLISATION: HÉRITAGE



Les classes **Personne** et **Chat** héritent par défaut de la classe **Object**.

# EXEMPLE DE MODÉLISATION: HÉRITAGE & INTERFACE



Héritage et interface.

Merci !!