

# Programmation Objet – Cours 1

## Introduction

License 3 Informatique

# Présentation de l'UE

- Cours Intégrés (CM+TD) : 20h (10 séances)
- Travaux pratiques (TP) : 24h (6 séances)

## Evaluation

- 2 contrôles continus
- Evaluation des TP

## Contact

[paul-emile.augusseau@etu.univ-guyane.fr](mailto:paul-emile.augusseau@etu.univ-guyane.fr)

## Pour reprendre depuis le début

<https://openclassrooms.com/en/courses/6173501-debutez-la-programmation-avec-java>

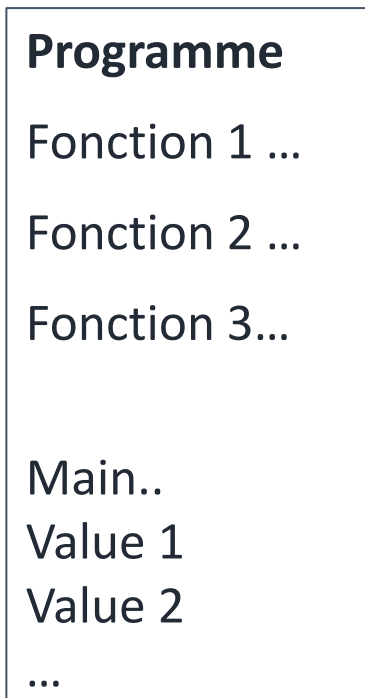
[https://gayerie.dev/epsi-b3-java/langage\\_java/](https://gayerie.dev/epsi-b3-java/langage_java/)



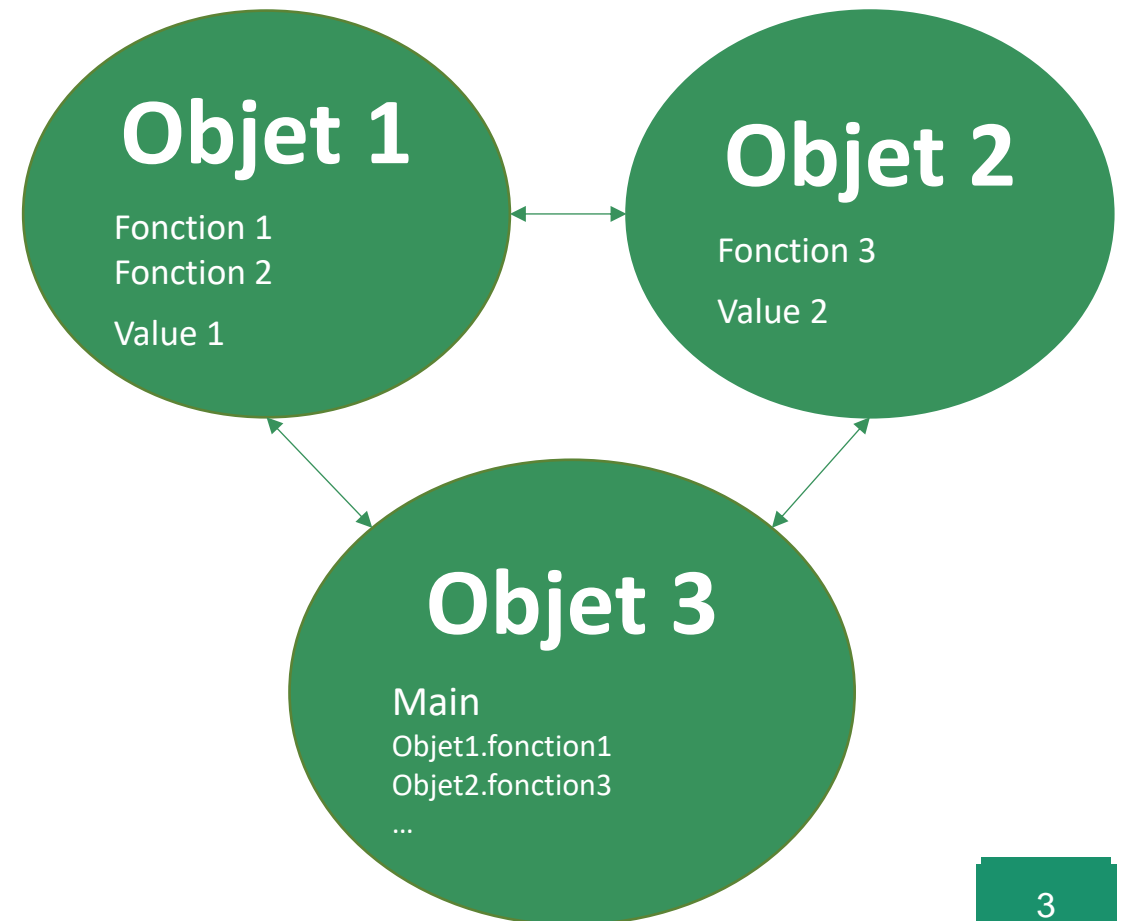
Présence obligatoire à chaque  
séance (Appel systématique)

# Principe général : Définir des objets en regroupant les données et comportements qui les caractérisent

## Programmation impérative classique



## Programmation orientée objet (POO)



# Les objets

# Les objets

La **programmation orientée objet (POO)**, ou programmation par objet, est un paradigme de programmation informatique basé sur la définition des propriétés et des interactions de briques logicielles de base appelées **Objets**.

Un objet est composé de deux types de membres:

- Attributs : Données caractéristiques
- Méthodes : Actions réalisables sur/avec cet objet (fonctions et procédures)

On fait référence aux membres d'un objet de la manière suivante:

`objet.membre`

# Les objets : exemple



# Les classes

# Les classes

On appelle **classe** la construction qui permet de décrire les caractéristiques et les comportement d'objets similaires.

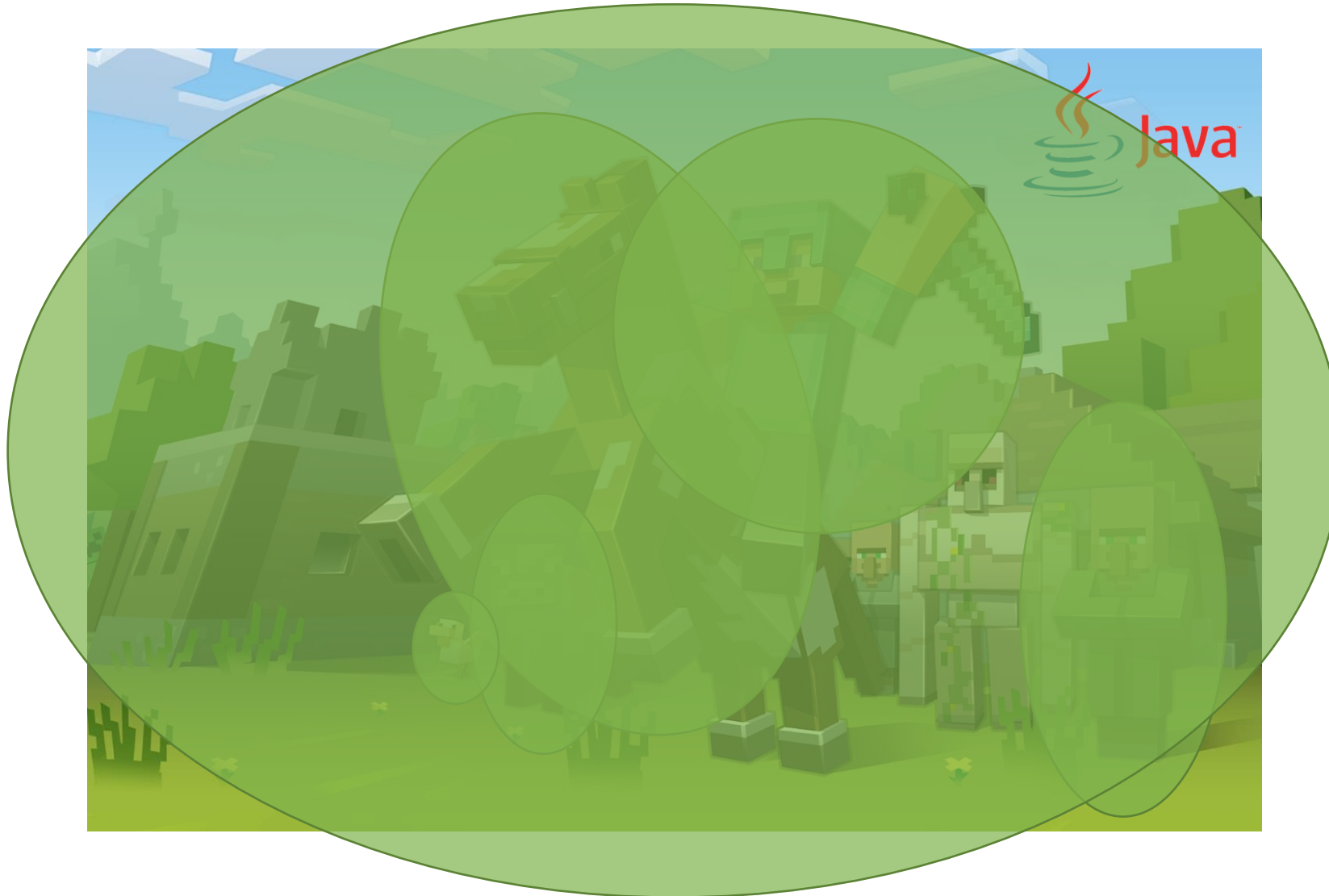
En programmation la classe est un type qui permet de définir des objets ayant les mêmes attributs et méthodes.

On définit un membre de classe de la manière suivante:

Classe.membre



# Les classes : exemple



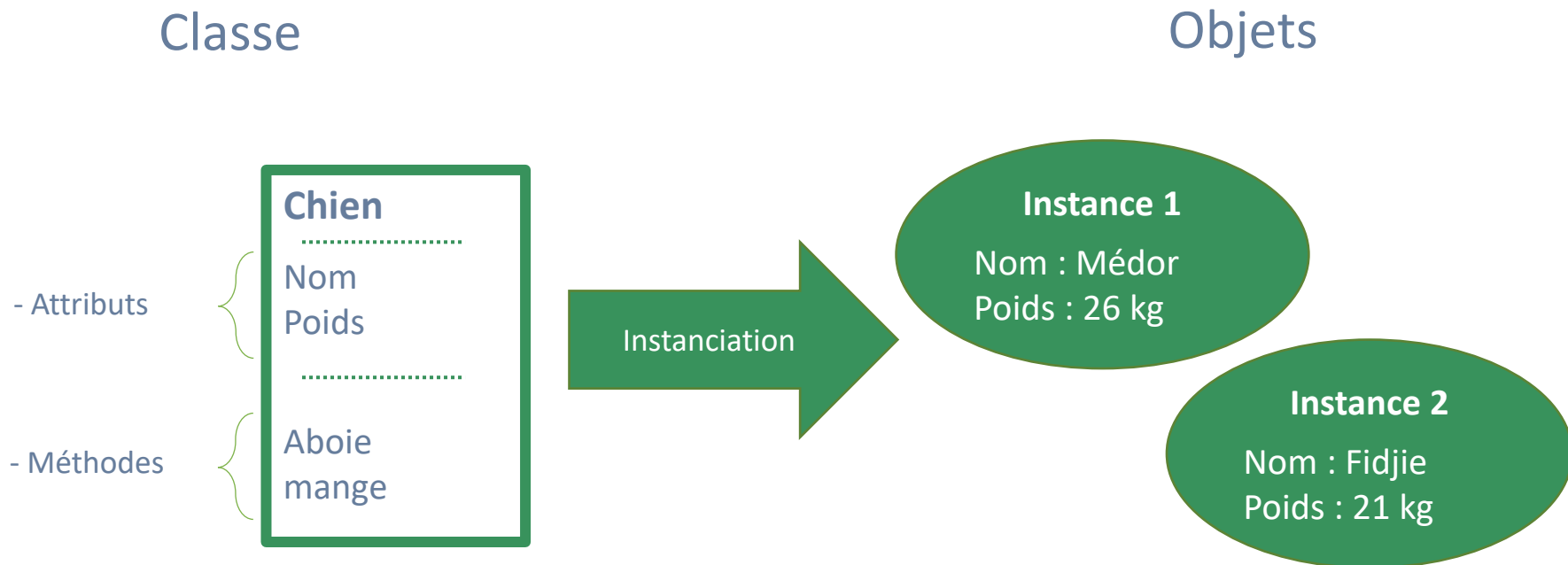
- Cheval
  - 4 pattes
  - Couleur
  - Vitesse
  - Saute
- Joueur
  - Inventaire
  - Skin
  - Se déplace
  - Saute
  - Détruit
  - Construit
- Personnage
- Vache
- Poulet
- Monde !

# Classe et objet

- ▶ La classe représente le type d'objet, les caractéristiques du concept qu'elle modélise
- ▶ L'objet est la représentation concrète du concept, il peut y avoir autant d'objets qu'on désire d'une seule classe.

La classe permet de créer des objets par instanciation.

Objet = instance de classe



# Instanciación

Dans la plupart des langages, Il existe deux manières d'instancier :

- **Statiquement:** l'espace est réservé à la compilation / ou au début de l'exécution du programme, comme pour les variables classiques.

*Exemple en C++ :*  
`MaClasse objet();`

- **Dynamiquement:** L'espace mémoire est réservé durant l'exécution du programme et l'objet est créé avec un mot clé.

*Exemple en C++ :*  
`MaClasse objet = new MaClasse();`

En Java, les allocations d'objets sont systématiquement dynamiques et se font par le mot clé **new**.

# Cas concret : Déclaration d'une classe en Java

- ▶ *Le langage Java se base en grande partie sur le langage C.*
- ▶ *Le nom de la classe commence toujours par une majuscule (NomClasse), le nom des variables commencent toujours par une minuscule.*
- ▶ Une classe est définie (en général) dans un fichier portant son nom (NomClasse.java)
- ▶ Le mot clé pour définir une classe est **class** (**class** NomClasse)

## ▶ NomClasse.java

```
class NomClasse {  
    [public/private/protected] int attribut1;  
    [public/private/protected] float attribut2;  
    [public/private/protected] String attribut3;  
  
    [public/private/protected] void methode2{...};  
    [public/private/protected] int methode1{...};  
  
    [public static void main(String[] args) {...}]  
}
```

## ▶ Chien.java

```
class Chien {  
    String nom;  
    float poids;  
  
    [public/private/protected] void aboie(boolean b){...};  
    [public/private/protected] int mange{...};  
}
```

# Comparaison avec le C

En programmation structurée, on définit la structure

En C :

```
struct Chien {  
    char* nom ;  
    int poids ;  
    int positionX ;  
    int positionY ;  
}
```

Puis on définit ses actions avec des fonctions

```
void aboie(struct Chien c) { printf(«%s : ouaf ! », c.nom) ; };
```

On peut déplacer le chien en manipulant directement les membres positionX, positionY de la structure.

# Comparaison avec le C

On peut ensuite manipuler une variable de type chien:

```
int main{  
  
    struct Chien medor ;  
  
    medor.poids = 26;  
    medor.nom = "medor" ;  
    medor.positionX = xPortail ;  
    medor.positionY = yPortail ;  
  
    aboie(medor) ;  
    return 0;  
}
```

# Comparaison avec le C

En Java:

## ► Chien.Java

```
class Chien {  
    String nom ;  
    float poids ;  
    int x ;  
    int y ;  
  
    void aboie(boolean ouaf) {  
        if(ouaf){  
            System.out.println(this.nom + " aboie")  
        }  
    }  
    ...  
}
```

```
...  
public static void main(String[] args) {  
    Chien medor = new Chien() ;  
    medor.poids = 5 ;  
    medor.nom = new String("Medor") ;  
    medor.x = xPortail ;  
    medor.y = yPortail ;  
  
    medor.aboie(true) ;  
}  
}
```

# Principes fondamentaux de la POO

- ▶ Encapsulation
- ▶ Héritage
- ▶ Polymorphisme



# L'encapsulation

# L'encapsulation

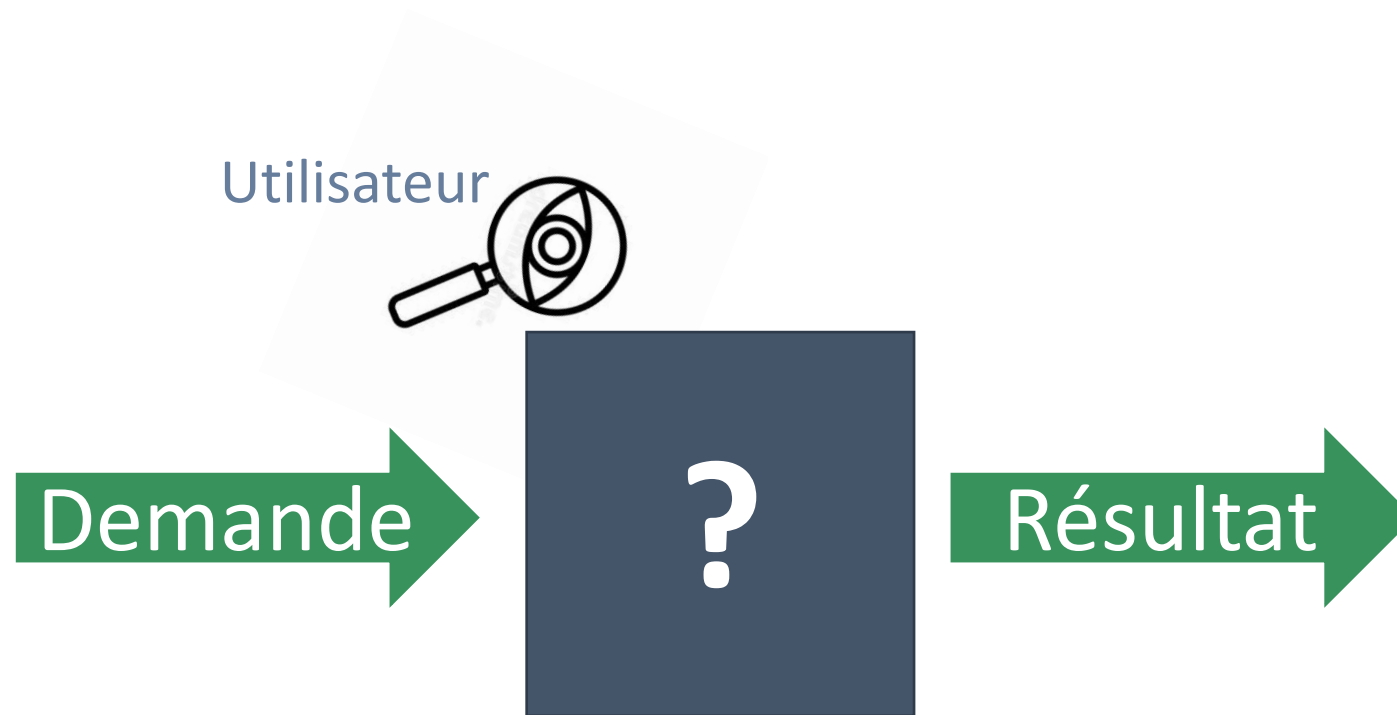
**L'encapsulation** désigne le principe de regrouper les données avec les traitements qui les concernent directement. En POO on regroupe les attributs et les méthodes.

```
class Chien {  
    String nom ;  
    float poids ;  
    int x ;  
    int y ;  
  
    void aboie(boolean ouaf) {  
        if(ouaf){  
            System.out.println(this.nom + " aboie")  
        };  
        ...  
    }  
}
```



# Le masquage

- ▶ Le masquage consiste à **cacher ou limiter l'usage des membres en dehors des méthodes de l'objet**.
- ▶ Présenter à l'utilisateur uniquement ce que le concepteur désire mettre à sa disposition.



# Le masquage

- ▶ Certains langages ne donnent accès qu'en lecture aux attributs d'un objet, imposant l'usage d'une méthode pour en modifier la valeur (Smalltalk, Eiffel).
- ▶ Les méthodes donnant accès en lecture ou écriture à des attributs s'appellent des *accesseurs*.
- ▶ Java laisse le choix au concepteur d'autoriser ou non l'usage des membres de la classe à l'utilisateur, via des mots clés :
  - + *public* : accès partout
  - ± *protected* : accès uniquement dans le package et les sous-classes (= classes filles)
  - - *private* : accès uniquement dans la classe

Mot clef	Classe	Package	Classe(s) fille(s)	Partout
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
par défaut	✓	✓	✗	✗
private	✓	✗	✗	✗

# Le masquage

- ▶ Nous ne voulons pas que Médor ait le droit de sortir du terrain dont les coordonnées vont de (10,20) à (50,60).

- ▶ Actuellement, notre classe Chien nous permet le code suivant :

```
medor.x = 0 ;  
medor.y = -12 ;
```

- ▶ On peut faire une méthode position(entier, entier) qui permet de ne changer la position du chien que s'il se trouve dans les limites du terrain (et retourne *true* si le déplacement est possible) :

```
boolean position(int nx, int ny) {  
    boolean dansTerrain = false;  
    if (nx <= 50 && nx >= 10) {  
        if (ny <= 60 && ny >= 20) {  
            this.x = nx;  
            this.y = ny;  
            dansTerrain = true ;  
        }  
    }  
    return dansTerrain;  
}
```

# Le masquage

- ▶ On peut demander à l'utilisateur, via la documentation, de ne déplacer un chien qu'avec cette nouvelle méthode :

```
Chien medor = new Chien();  
boolean deplacement;  
deplacement = medor.position(15,35); // médor est déplacé  
deplacement = medor.position(5,35); // médor ne bouge pas
```

- ▶ Mais rien ne nous empêche d'aller modifier les données directement :

```
medor.x = -5 ;
```

- ▶ L'utilisateur **peut** être fortement tenté d'**aller modifier directement les données** plutôt que d'utiliser la méthode que nous mettons à sa disposition.

# Le masquage

- Pour *autoriser ou non l'accès aux attributs et aux méthodes* on peut utiliser le mot clé private :

```
class Chien {  
    int taille;  
    int poids;  
    private int x;  
    private int y;  
  
    void aboie(boolean ouaf) {  
        if(ouaf){  
            System.out.println(this.nom + " aboie")  
        };  
  
    boolean position(int nx, int ny) {  
        boolean dansTerrain = false;  
        if (nx <= 50 && nx >= 10) {  
            if (ny <= 60 && ny >= 20) {  
                x = nx;  
                y = ny;  
                dansTerrain = true ;  
            }  
        }  
        return dansTerrain;  
    }  
}
```

# Le masquage

- Nous pouvons accéder aux membres private *au sein de la classe* Chien mais on ne peut plus y accéder en dehors de cette classe.

Le code suivant devient alors impossible :

```
Chien medor = new Chien();  
medor.x = -5;
```

Le compilateur va remonter l'erreur suivante :

*"The field Chien.x is not visible"*

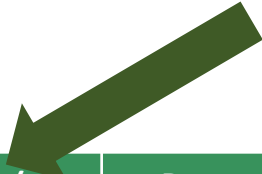


- L'utilisateur de notre classe *Chien* ne pourra plus le faire sortir du terrain.
- De plus, si nous changeons la manière de gérer la position du *Chien*, nous pouvons créer une nouvelle fonction *position(int, int)* avec la même signature que l'ancienne. Les utilisateurs de notre classe Chien ne seront pas affectés par le changement.



# À retenir

- ▶ **l'encapsulation** est le fait de grouper les attributs et méthodes au sein d'un objet
- ▶ le **masquage** est le fait de limiter **la portée** (l'accessibilité) des membres d'une classe
- ▶ la portée des membres d'une classe selon le mot clé donné est la suivante :



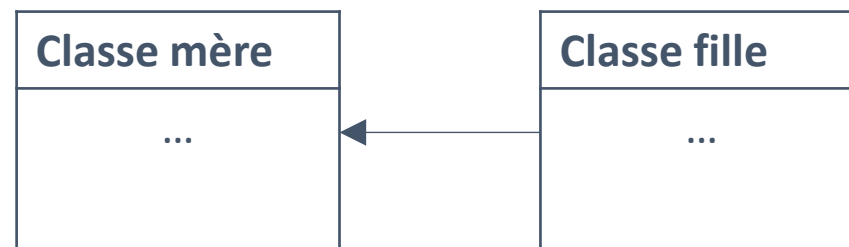
Mot clef	Classe	Package	Classe(s) fille(s)	Partout
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
par défaut	✓	✓	✗	✗
private	✓	✗	✗	✗

# L'héritage

# L'héritage

Définition de wikipedia :

- ▶ **L'héritage** est un mécanisme qui permet, lors de la déclaration d'une nouvelle classe, d'y inclure les caractéristiques d'une autre classe.
- ▶ On dit alors que la classe ainsi définie **hérite** ou **dérive** de la classe d'origine.
- ▶ La classe qui hérite est appelée **classe fille** ou **classe dérivée**, la classe dont elle hérite est appelée **classe mère** ou **superclasse**.
- ▶ La classe fille **hérite de tous les membres** d'une classe mère.
- ▶ L'ensemble des classes d'un programme et des relations d'héritage entre elles est appelé **hiérarchie de classe**.



# L'héritage

- ▶ On veut pouvoir avoir *différents types* de chiens dans notre simulateur, des gardiens, des bergers, des chasseurs.
- ▶ Tous sont des chiens et ont les mêmes caractéristiques (poids, nom, position) et comportement (ils se déplacent et aboient) de base. On identifie bien là une classe *Chien*.
- ▶ Toutefois, un gardien garde une zone ou un objet, un berger surveille un troupeau et un chasseur piste un éventuel gibier. On pourrait tout coller dans la même classe

On ajoute un type de Chien (g,b,c)

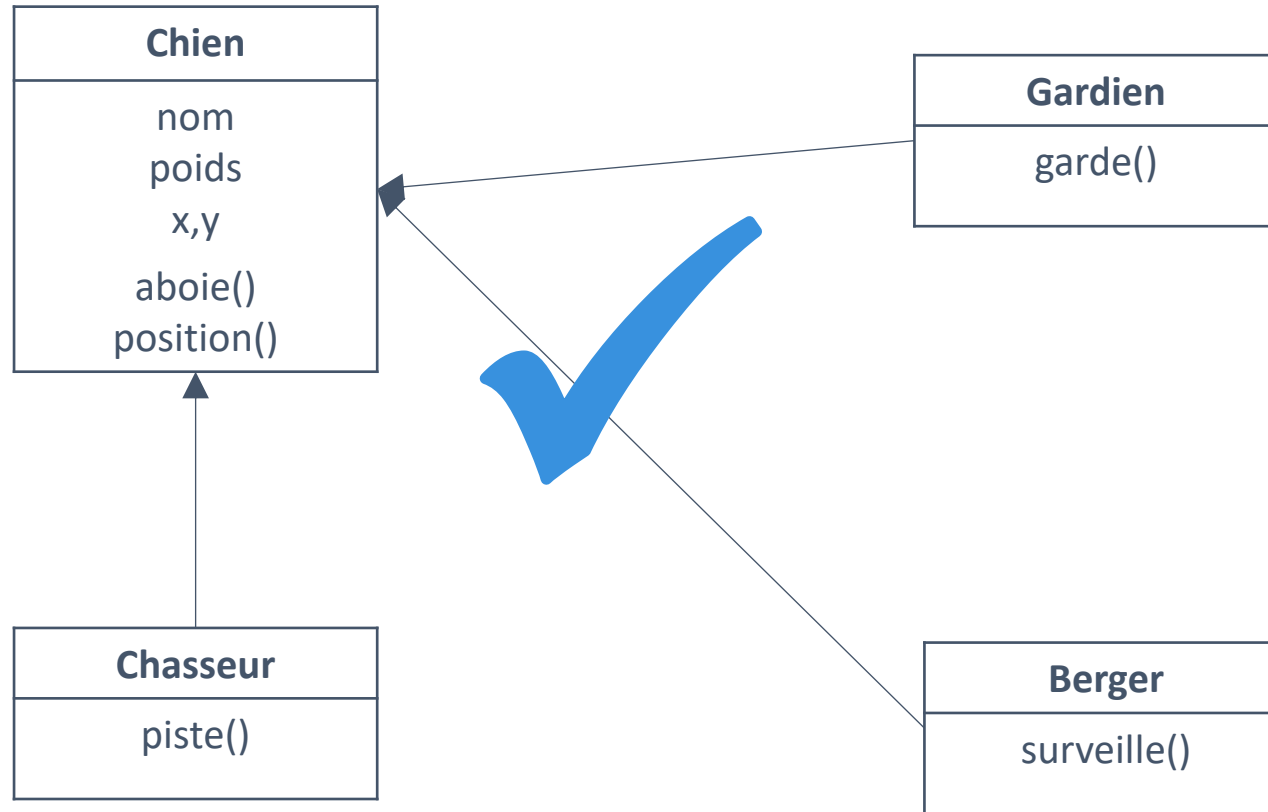
On ajoute des tests :

- si Chien est un gardien alors garde() a un effet
  - si Chien est un berger alors surveille() a un effet
  - si Chien est un chasseur alors piste() a un effet
- ▶ Mais si on a 500 sortes de Chien ou si les relations sont plus compliquées, ça devient ingérable....

Chien
nom
poids
x,y
type
aboie()
position()
garde()
surveille()
piste()

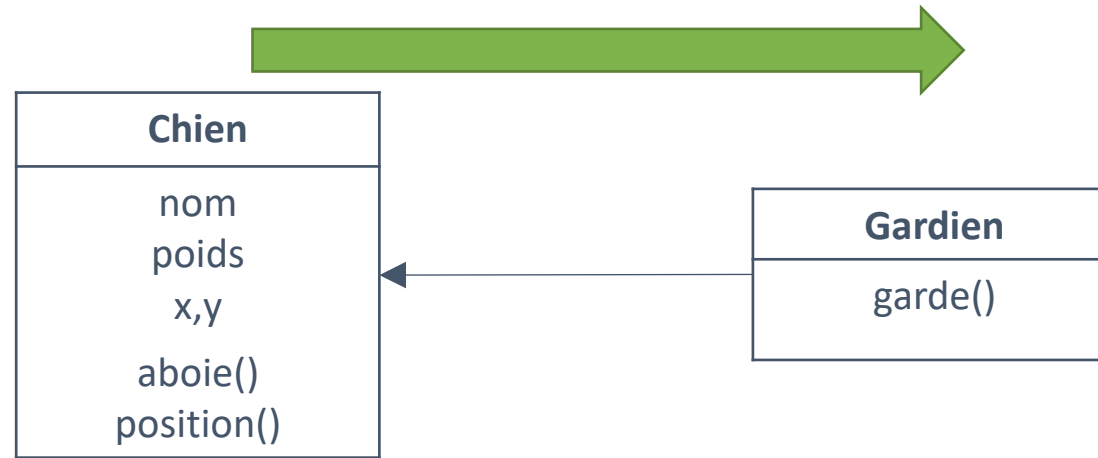
# L'héritage

- Il vaut mieux représenter chaque type de chien comme suit:

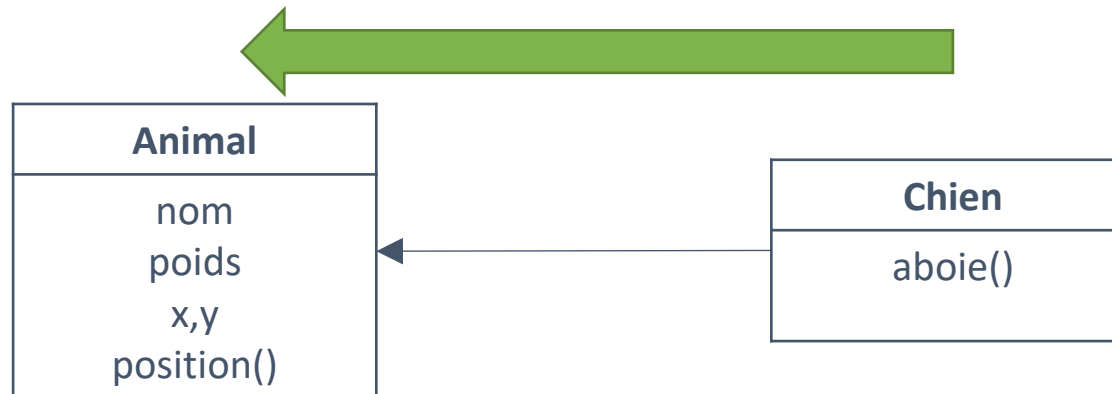


# L'héritage

- Quand on dérive une classe dans le but de préciser un concept plus général, on dit qu'on a une relation de **spécialisation** entre la classe mère et la classe fille.



- Quand à l'inverse, on extrait des membres d'une classe pour créer une classe mère plus généraliste que la classe d'origine, on parle d'une relation de **généralisation**.



# L'héritage

- ▶ En code Java, la relation d'héritage s'exprime par le mot clé *extends*.
- ▶ Nos classes s'écriront comme suit :

```
public class Gardien extends Chien {  
    void garde() { /* code */ };  
}
```

```
public class Berger extends Chien {  
    void surveille() { /* code */ };  
}
```

```
public class Chasseur extends Chien {  
    void piste() { /* code */ };  
}
```

- ▶ On pourra alors écrire dans la fonction main:

```
Gardien medor = new Gardien();  
medor.poids = 12;  
medor.garde();  
medor.aboie(true);
```

# L'héritage

- ▶ Ce mécanisme permet de réutiliser le code des superclasses et de
  - l'enrichir de nouvelles fonctions
  - découper les concepts selon des niveaux de spécialisation
- ▶ L'héritage permet à un objet d'utiliser les membres de leurs ancêtres en plus de leurs membres:

```
Gardien medor = new Gardien();
```

```
✓ medor.aboie(true);
```

```
✓ medor.garder();
```

```
✗ medor.surveiller();
```

- ▶ Mais une classe mère ignore tout de ses classes filles :

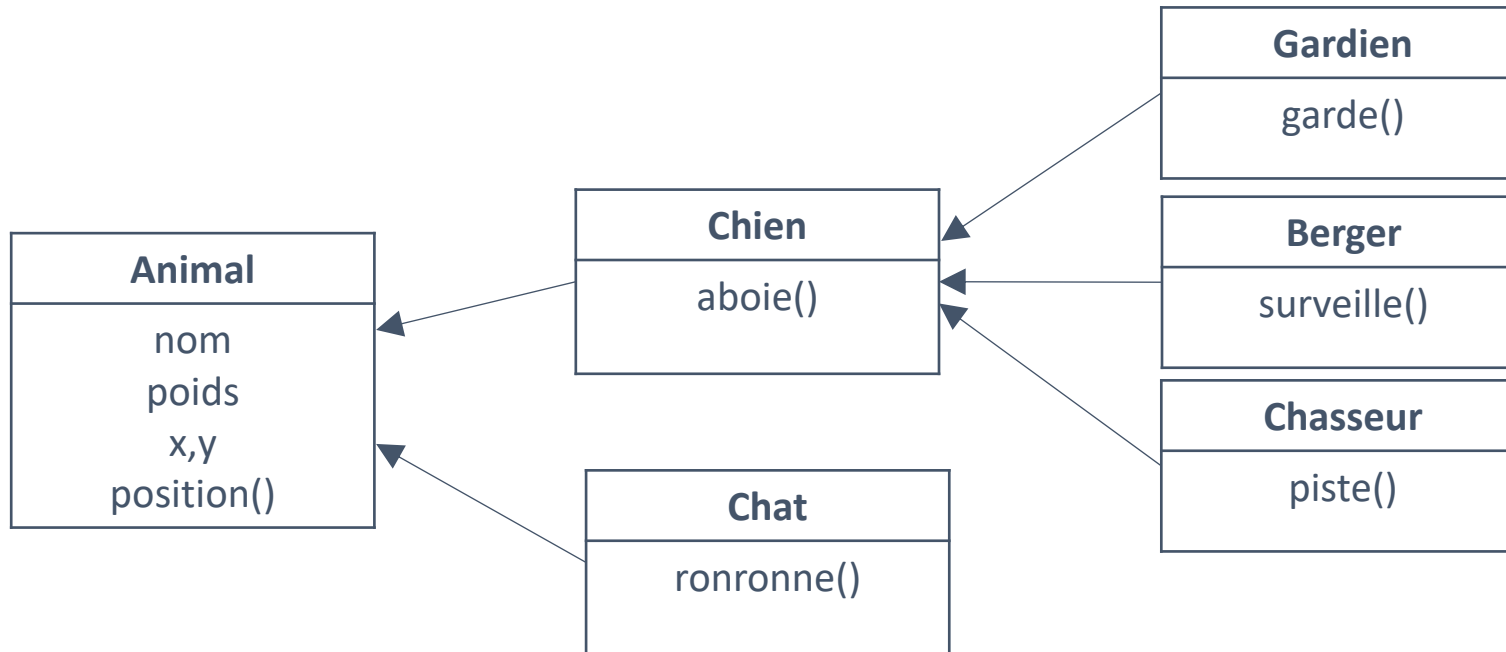
```
Chien chipie = new Chien();
```

```
✗ chipie.garder();
```



# L'héritage

- Si on veut gérer des chats qui ronronnent, on peut **généraliser** Chien en classe Animal et **spécialiser** Animal en Chat qui ronronne(), ce qui donne la **hiérarchie de classe** suivante :



- Exercice : écrire le code correspondant à cette hiérarchie de classe (séparée en fichiers)

# Le polymorphisme

# Le polymorphisme

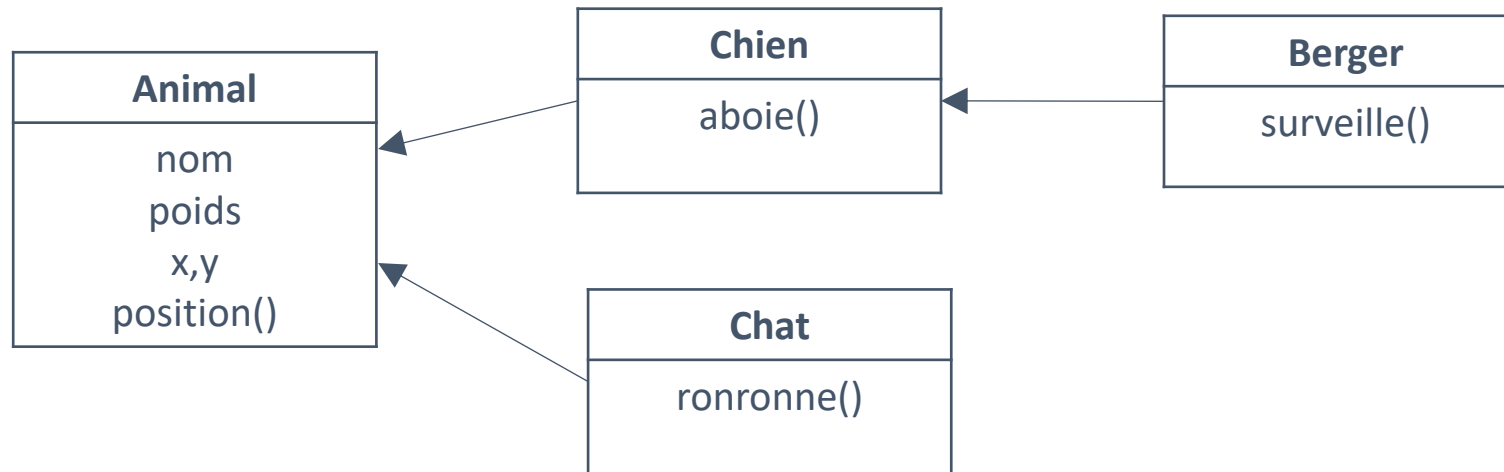
Définition de wikipedia :

- ▶ Le **polymorphisme** est le mécanisme qui permet d'offrir une interface unique pour des entités de type différents.
- ▶ Il existe plusieurs types de polymorphisme, celui qui nous intéresse est celui appelé **polymorphisme par héritage**.
- ▶ Il se traduit par la possibilité d'utiliser la même interface (les mêmes membres) sur des types différents.
- ▶ Exemple:

```
Berger fidjie = new Berger();  
Gardien medor = new Gardien();  
nourrir(fidjie);  
nourrir(medor);
```

# Le polymorphisme

- Si nous reprenons notre hiérarchie de classe précédente :



- Nous utilisons le polymorphisme quand on utilise indifféremment les Chats, les Chiens ou les Gardiens dans notre programme.
- Par exemple, le vétérinaire soigne indifféremment les chats, les chiens, quelque soit leur fonction (Gardiens, Chasseurs ou Bergers).

# Le polymorphisme

- Considérons une nouvelle classe :

```
class Veterinaire {  
    void soigne(Animal a) {  
        System.out.println(a.nom + « est guéri ») ;  
    }  
}
```

- Ainsi, le programme principal suivant est correct :

```
Gardien medor = new Gardien();  
medor.nom = "Médor";  
medor.aboie(true);  
Chat felix = new Chat();  
  
Veterinaire paul = new Veterinaire();  
paul.soigne(medor);  
paul.soigne(felix);
```

- Car medor est un chien, qui est un animal, de même que felix est un chat, qui est aussi un animal.

# Conclusion

- ▶ L'approche Orientée Objet nous apporte certains avantages :
  - Mécanismes de protection clairs vis à vis des variables internes aux classes grâce à l'*encapsulation* et au *masquage*
  - Réutilisation et maintenance du code facilité par l'*héritage*
  - Généralisation de certains traitements grâce au *polymorphisme*
  - Représentation via des concepts « naturels » directement traduits dans le code
  
- ▶ Les cours suivant seront dédiés aux autres mécanismes de la POO

