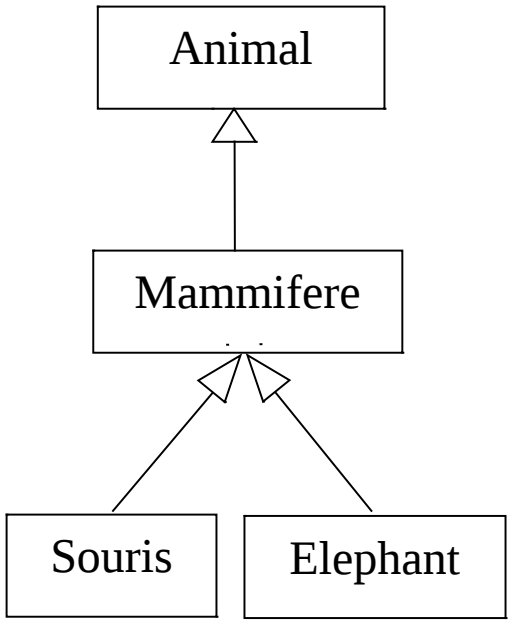
Programmation objet – Cours 6
Héritage (2ème partie)

1. Hiérarchie des classes
2. Compatibilité entre classes
3. Conversion de types (cast)
4. Polymorphisme
5. La classe *Object*

1. Hiérarchie des classes

L'héritage définit un ordre partiel entre les classes



Elephant hérite de *Mammifère*

et

Mammifère hérite de *Animal*

⇩

Elephant hérite (implicitement) de *Animal*

(par transitivité)

Souris et *Elephant* ne peuvent être comparées :

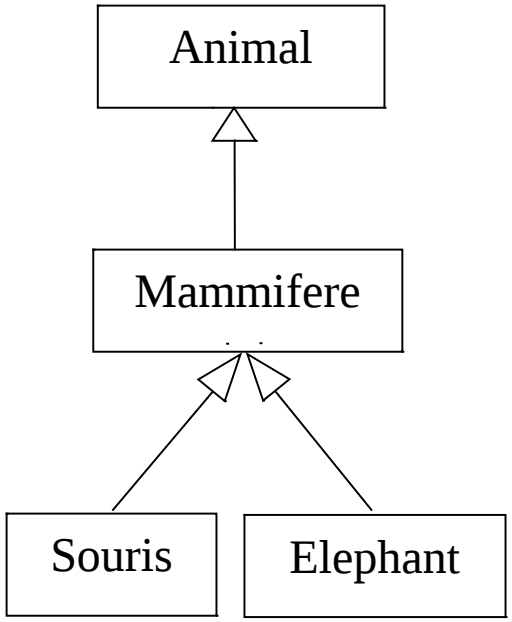
aucune n'hérite de l'autre (même par transitivité)

Vocabulaire

une classe *A* hérite (directement ou par transitivité) d'une classe *B*

⇔ *A* est descendant de *B*

⇔ *B* est ascendant (ancêtre) de *A*



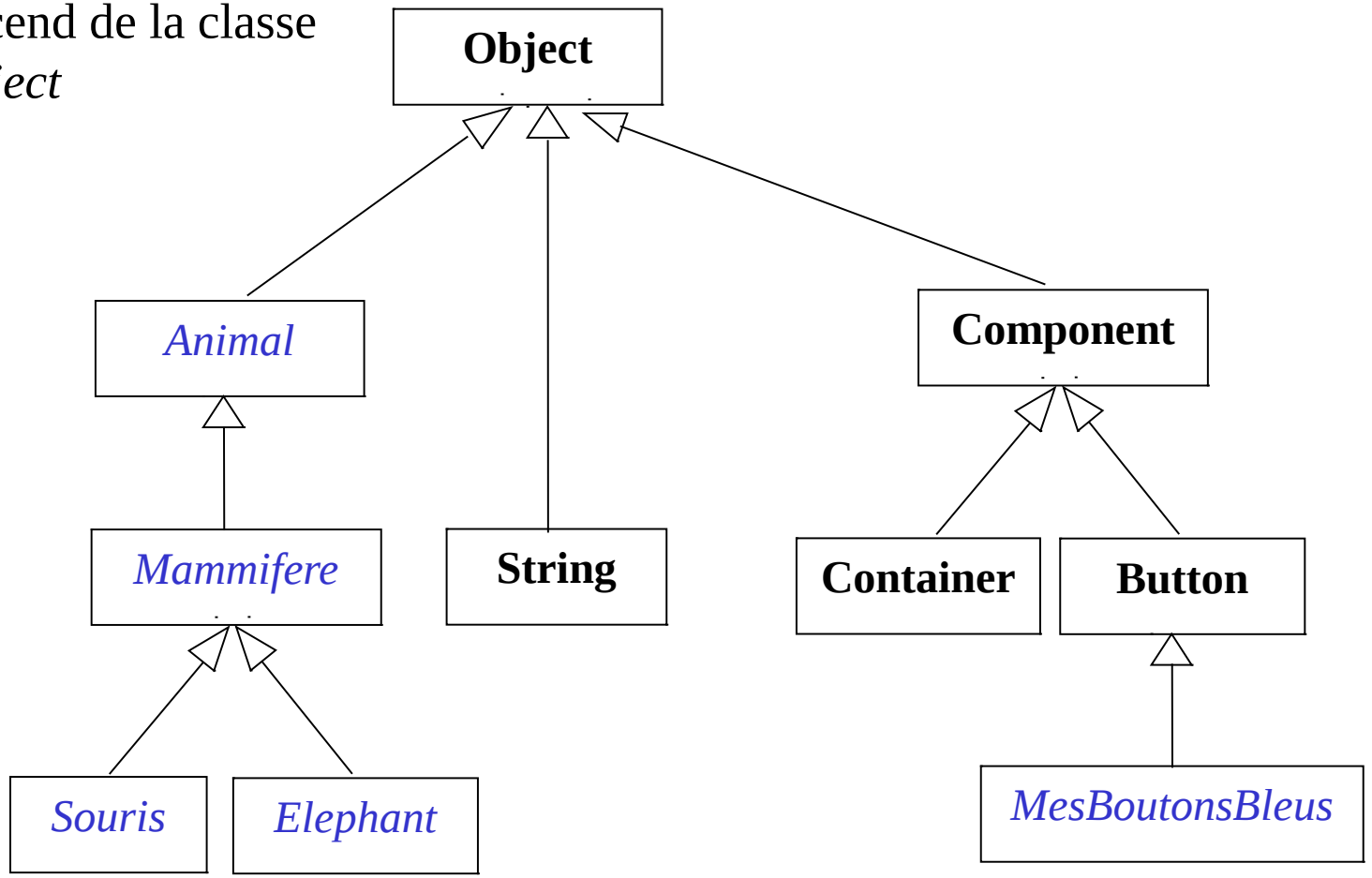
Mammifere, *Souris* et *Elephant* sont des descendants de *Animal*.

⇔

Animal est un ascendant de *Mammifere*, *Souris* et *Elephant*

L'ensemble des classes forme une hiérarchie

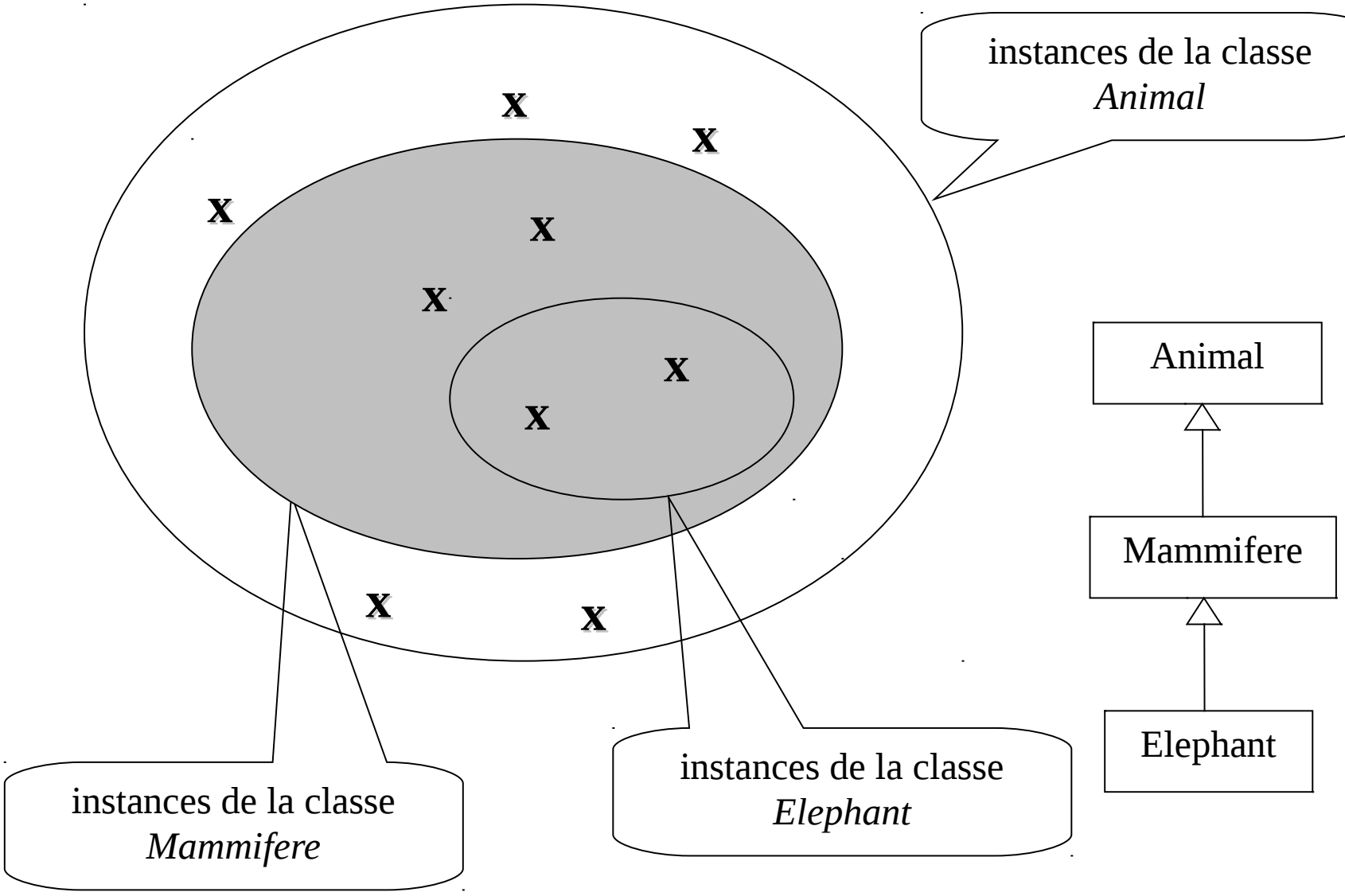
toute classe descend de la classe *Object*



On peut étendre les classes standards (non finales) fournies par Java

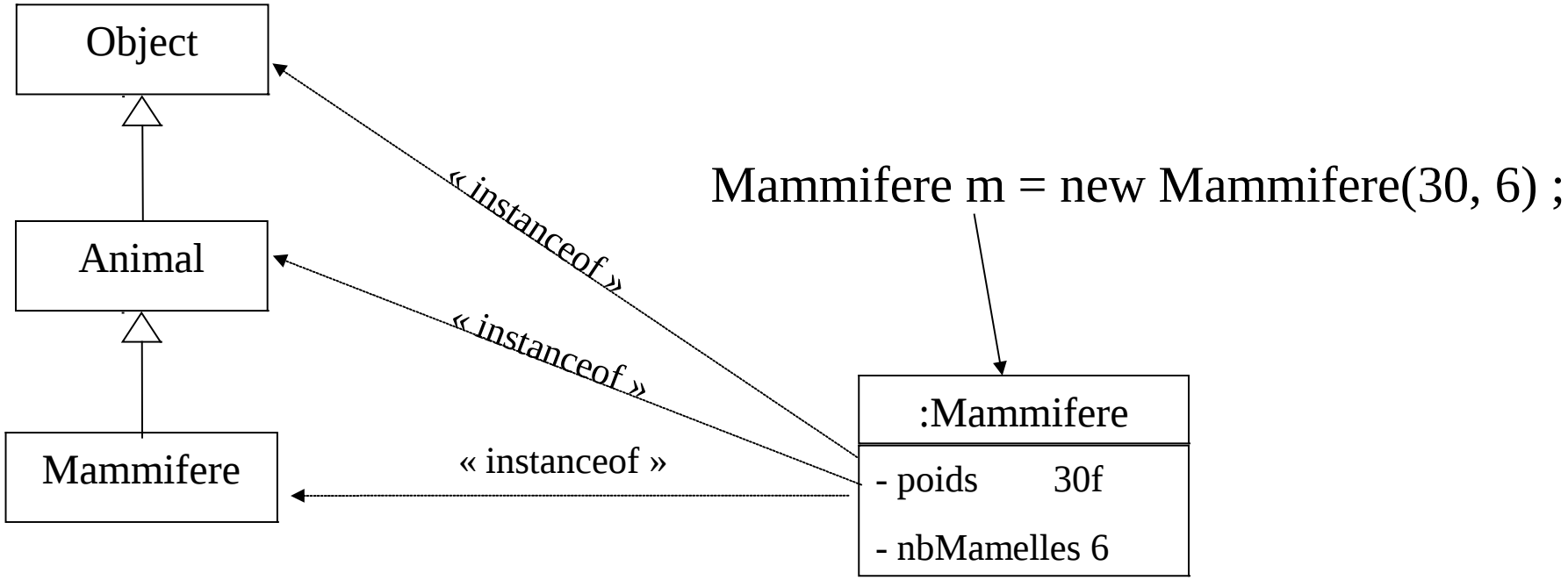
2. Compatibilité entre classes

Une instance d'une classe est aussi instance de tous ses ascendants



instanceof

m ***instanceof*** *Animal* retourne *true* si la variable *m* référence une instance d'*Animal*, *false* sinon.



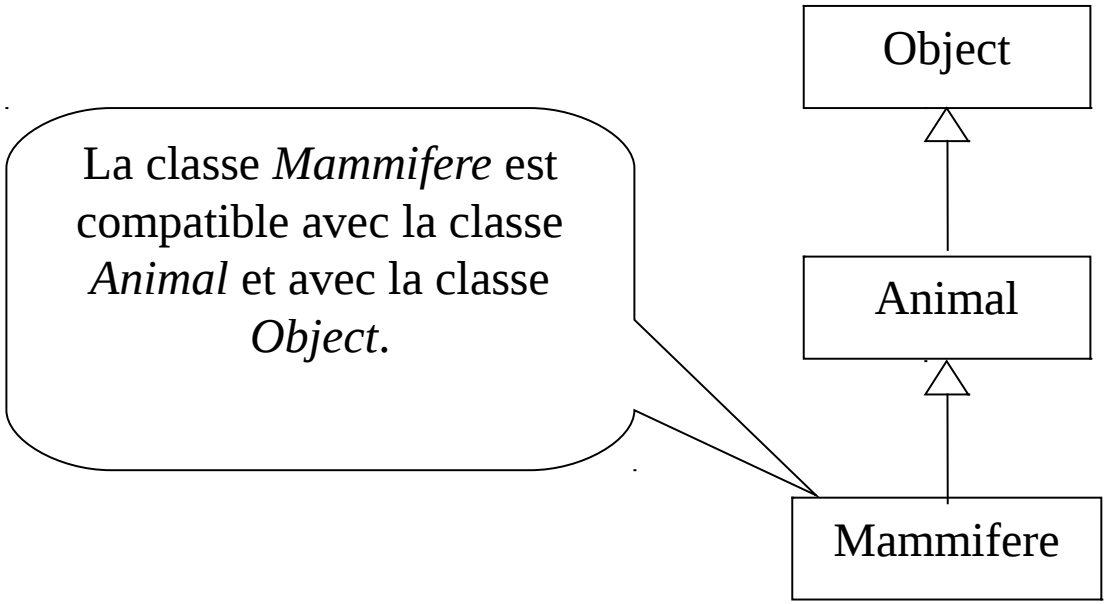
m **instanceof** *Object* retourne *true*

m **instanceof** *Animal* retourne *true*

m **instanceof** *Mammifere* retourne *true*

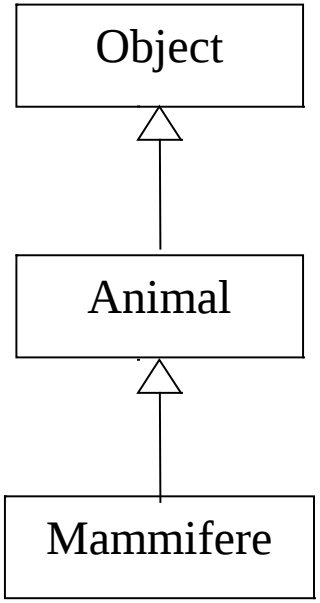
Compatibilité entre classes (1/2)

Une classe est compatible avec tous ses ascendants



Compatibilité entre classes (2/2)

Règle : si la classe *A* est descendante d'une classe *B*, on peut toujours fournir une référence sur *A* quand on attend une référence sur *B*

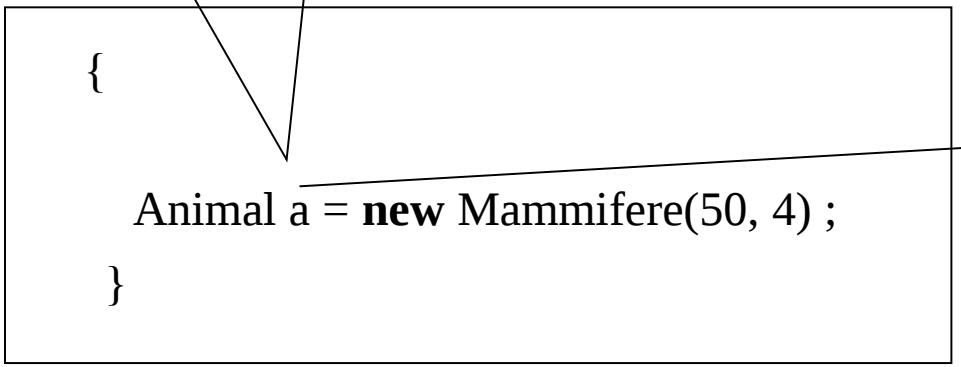


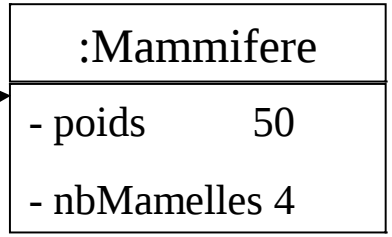
Un *Mammifere* étant un *Object (*spécifique) on peut fournir un *Mammifère* partout où l'on attend un *Object*

Un *Mammifere* étant un *Animal (*spécifique) on peut fournir un *Mammifère* partout où l'on attend un *Animal*

Exemple 1

a est une référence sur un *Animal*.
On lui fournit l'adresse d'un *Mammifère* qui est un *Animal* spécifique.

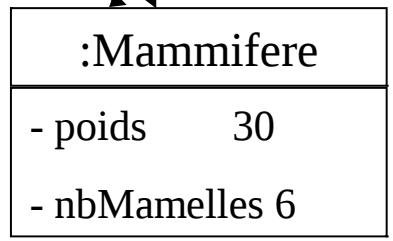




Exemple 2

```
public class Animal
{

    private float poids ;

    ...

    public static void affichePoids(Animal a )
    {
        System.out.println (a.poids) ;
    }


}   // fin classe Animal
```

```
public class Test
{

    public static void main(String[] args )
    {

        Mammifere m = new Mammifere(30, 6) ;

        Animal.affichePoids(m) ;

    }


}   // fin classe Test
```



Le paramètre *a* est une référence sur un *Animal*.
On lui fournit l'adresse d'un *Mammifère* qui est un *Animal* (spécifique).

Exemple 3

```
public class PointPlan
{
    private float abscisse ;
    private float ordonnee ;

    public PointPlan(float x, float y)
    {
        this.abscisse = x ;
        this.ordonnee = y ;
    }

    public PointPlan(PointPlan p)
    {
        this.abscisse = p.abscisse ;
        this.ordonnee = p.ordonnee ;
    }
}
```

```
public class Point3D extends PointPlan
{
    private float azimut ;

    public Point3D(float x, float y, float z)
    {
        super(x, y) ;
        this.azimut = z ;
    }

    public Point3D(Point3D p3)
    {
        super(p3) ;
        this.azimut = p3.azimut ;
    }
}
```

:Point3D
- abscisse 10
-ordonnee 16
- azimut 20

Dans *Point3D* **this** réfère une instance de *Point3D*, dans *PointPlan* **this** réfère la même instance vu comme un *PointPlan*

:Point3D
- abscisse 10
-ordonnee 16
- azimut 20

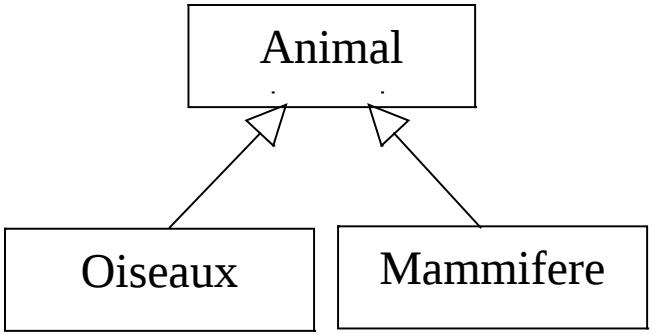
Un Point3D est un *PointPlan* (spécifique).

Une classe n’est compatible avec aucune de ses descendantes

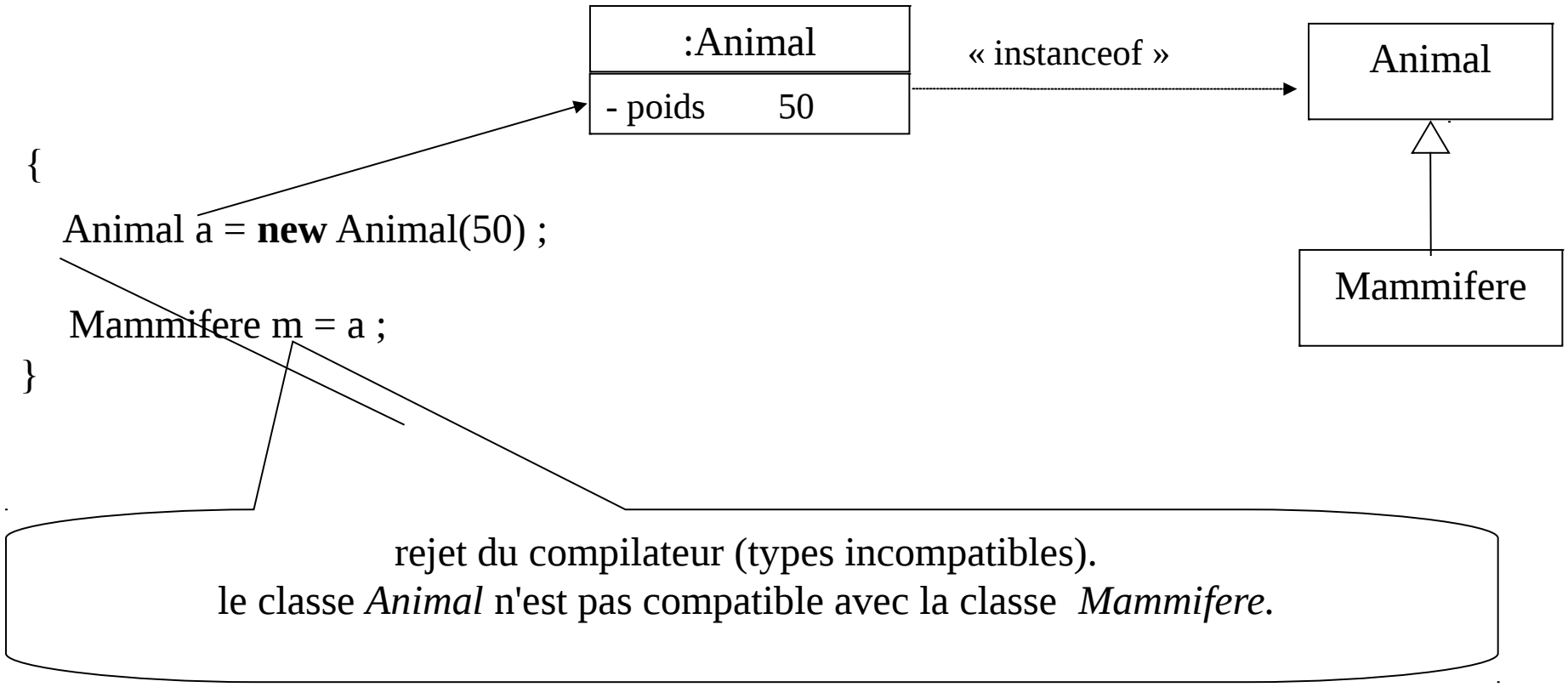
Règle : si la classe A est descendante de la classe *B* on ne peut jamais fournir une référence sur *B* quand on attend une référence sur *A*

Ex : La classe *Animal* n’est pas compatible avec la classe *Mammifère :*

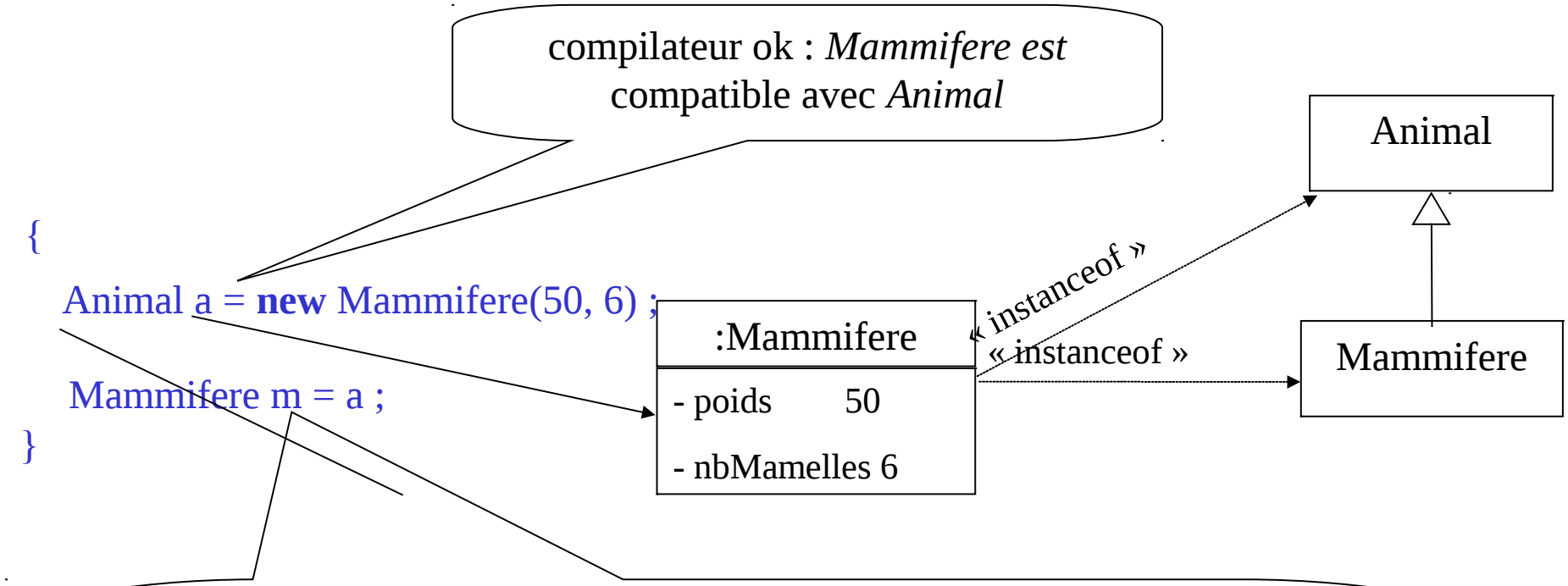
on ne peut pas fournir une référence sur un *Animal* là où on attend une référence sur un *Mammifere* (l'*Animal* en question peut ne pas être un *Mammifere*)



Exemple 1


:Animal
- poids 50
« instanceof »
Animal
Mammifere
{
Animal a = new Animal(50) ;
Mammifere m = a ;
}
rejet du compilateur (types incompatibles).
le classe *Animal* n'est pas compatible avec la classe *Mammifere*.

Exemple 2



rejet du compilateur : *Animal* n'est pas compatible avec *Mammifere.*
Le compilateur refuse car la référence *a* annonce un *Animal* :
Le compilateur possède une vue statique (et non dynamique) il ne considère que la classe annoncée par la référence. Il ne "voit pas" l'instance de type *Mammifere* référencée par *a* car cette instance ne sera créée qu'à l'exécution.

3. Conversion de types (cast)

"caster" une expression revient à signaler au compilateur qu'elle est d'un type différent par rapport au type déclaré

syntaxe d'un cast : *(autre_type) expression*

signale au compilateur que *expression* est du type *autre_type*

```
{
    Animal a = new Mammifere(50, 6) ;

    Mammifere m = (Mammifere) a ;
}
```

compilateur ok : le cast signale au compilateur que dans cette instruction la référence *a* sur un *Animal* doit être considérée comme une référence sur un *Mammifere*.

Priorité des opérateurs cast et .

L'opérateur de cast *(autre_type)* et moins prioritaire que l'opérateur .

```
{
    Object a = new Animal (50) ;

    (Animal) a.init( ) ;


     ((Animal) a).init( ) ;

}
```

rejeté par le compilateur car interprété comme un cast de l'expression *a.init( )* (qui est du type void).

compilateur ok :
cast de l'expression *a*.
On applique la méthode *init* à l'expression "castée" *(Mammifere) a*.

4. Polymorphisme

1. possibilité pour une variable de référencer des instances de types différents

2. ligature statique (vision du compilateur)

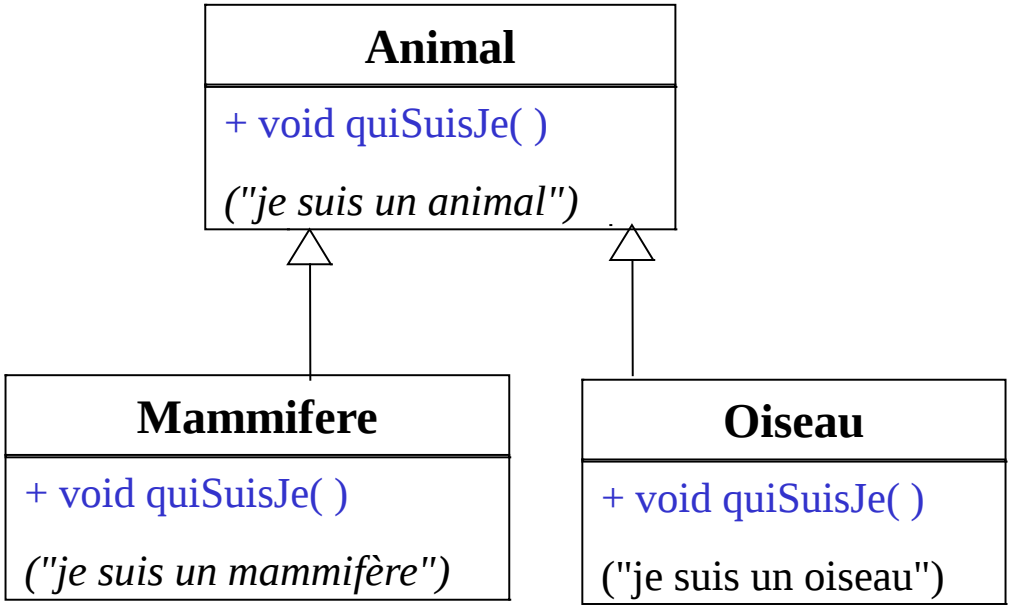
3. ligature dynamique (vision de la JVM)

Méthode polymorphe

une méthode est *polymorphe* (plusieurs formes) lorsqu'elle est redéfinie (il en existe plusieurs versions de mêmes signatures dans la hiérarchie des classes)

ex) La méthode *quiSuisJe( )* est définie dans *Animal*,

redéfinie dans *Mammifere( )* et redéfinie dans Oiseau.



Il y a 2 méthodes *quiSuisJe* de même signature dans les classes *Mammifere* et *Oiseau* : 1 héritée de *Animal* + 1 spécifique

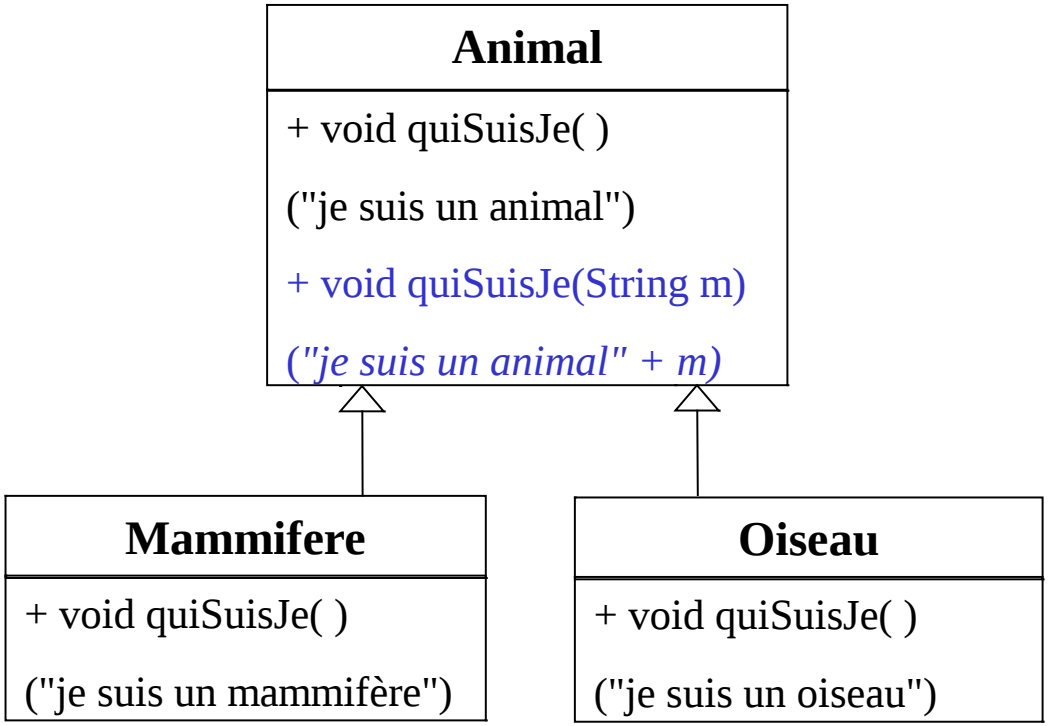
Redéfinition ≠ surcharge

Une méthode est surchargée quand il en existe plusieurs versions de signatures différentes au sein d'une même classe

ex) dans *Animal* : public void quiSuisJe()

{System.out.println("je suis un animal : " + m) };

quiSuisJe(String m) n'est pas polymorphe car non redéfinie

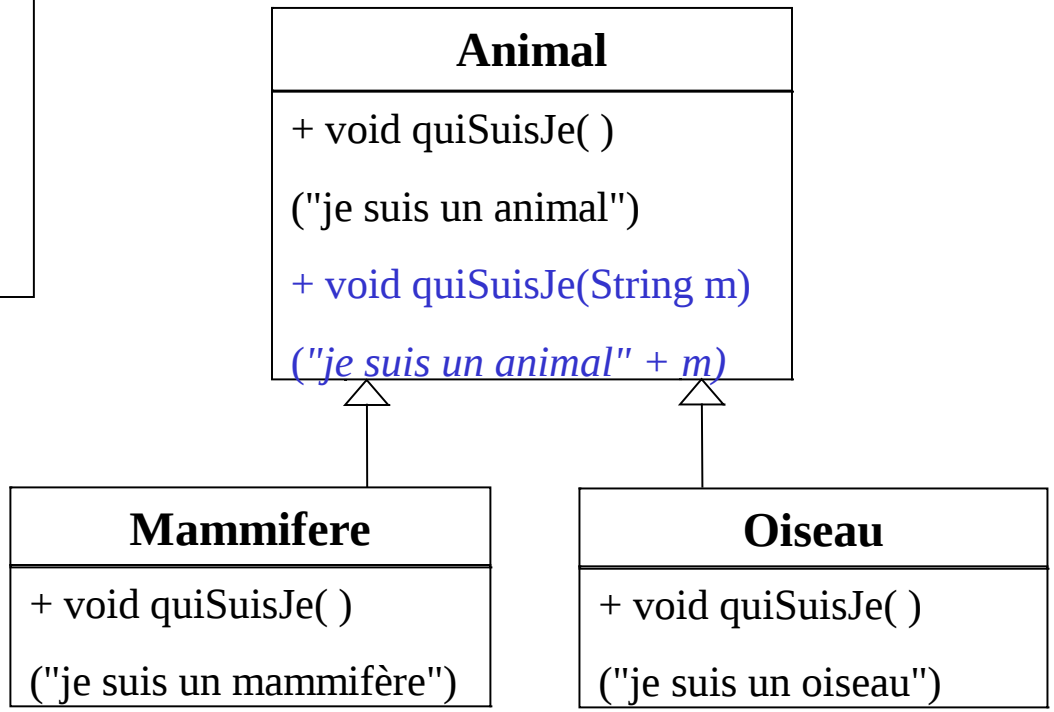


Compilation : liaison statique

Le compilateur vérifie l'existence d'une méthode dans la classe annoncée par la référence (et non par l'instance)

```
{
    Animal a = new Oiseau(1.2, 0.8 ) ;
    a.quiSuisJe( ) ;
}
```

le compilateur recherche l'existence d'une méthode de signature *quiSuisJe( )* dans la classe *Animal* (car *a* est une référence sur un *Animal)*

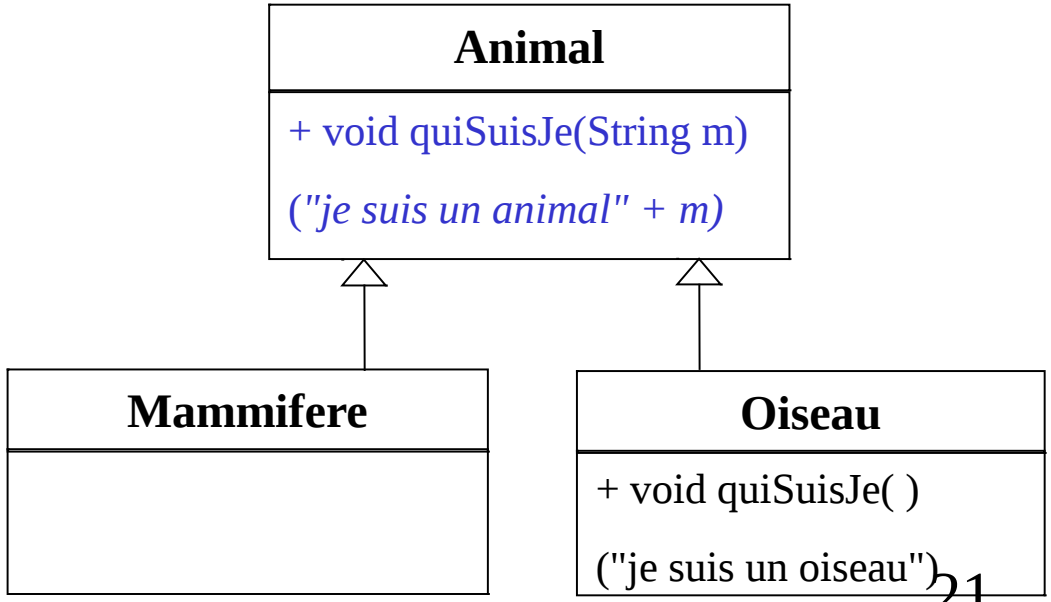


liaison dynamique - Invocation d'une méthode

toute méthode invoquée est recherchée dans la classe de l'instance impliquée dans l'appel (puis en remontant dans ses ascendants si elle n'est pas trouvée)

```
{
        Animal a = new Oiseau(1.2, 0.8 ) ;

        a.quiSuisJe( ) ;
}
```

la machine virtuelle recherche (pour invocation) une méthode compatible avec la signature *quiSuisJe( )* dans la classe *Oiseau* puis s'il n'y en a pas dans la classe mère *Animal* (et ainsi de suite en remontant).
Affiche *"je suis un oiseau"*



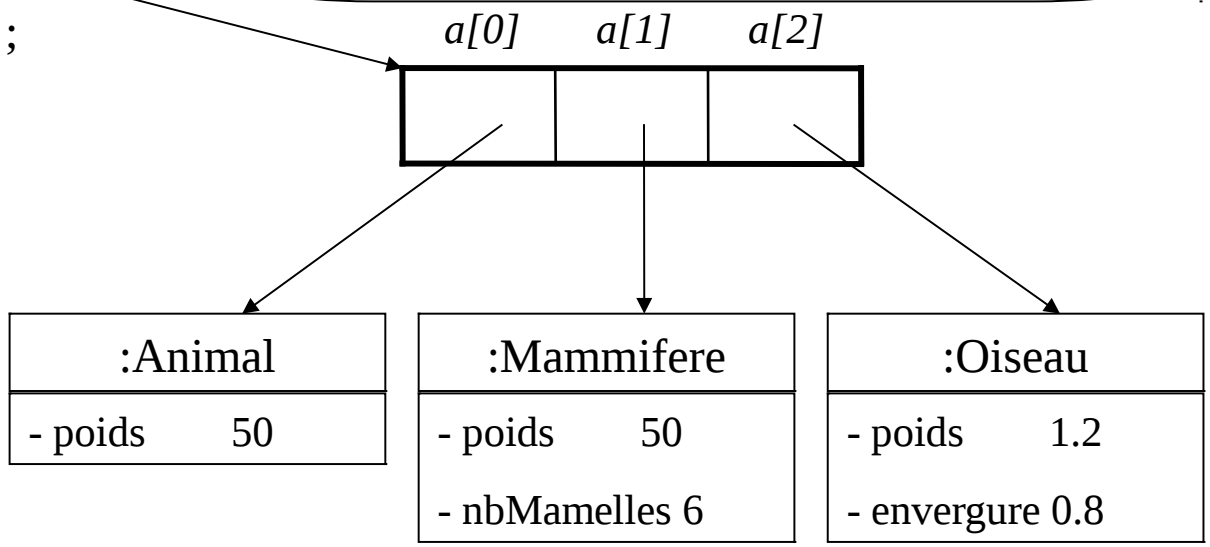
Exemple d'utilisation du polymorphisme

Un même code est utilisé pour traiter des instances de différentes classes ayant entre elles des relations d'héritage

{

```
Animal [] a ;

a[0] = new Animal(50) ;
a[1] = new Mammifere(30, 6) ;
a[2] = new Oiseau(1.2, 0.8) ;

fot(int i=0;  i < a.length; i++)
      a.quiSuisJe( ) ;
```

On référence toutes les instances comme étant de la classe la plus générale (Animal)



}

La méthode *quiSuisJe( ) est* recherchée dans la classe de l'instance.
Affiche respectivement :
(1) "je suis un animal"
(2) "je suis un mammifère"
(3) "je suis un oiseau"

Exemple d'utilisation de références trop générales

```
{
    Object []a;

    a[0] = new Animal(50) ;
    a[1] = new Mammifere(30, 6) ;
    a[2] = new Oiseau(1.2, 0.8) ;

    fot(int i=0;  i < a.length; i++)
        a.quiSuisJe( ) ;

}
```

On référence toutes les instances avec la classe *Object*.

Compilateur ok
(les classes *Animal*, *Mammifere* et Oiseau sont compatibles avec la classe *Object*)

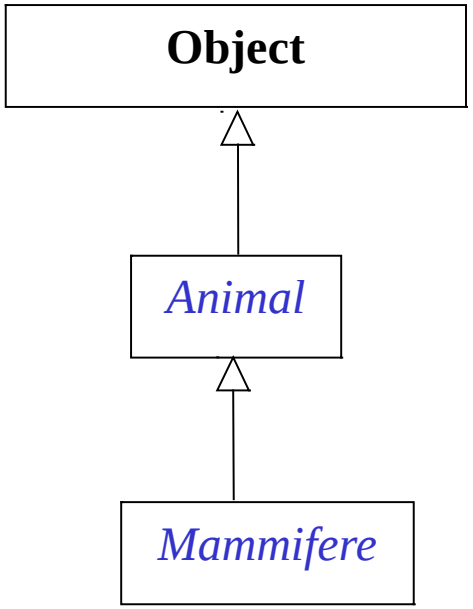
Rejet du compilateur :
la méthode *quiSuisJe* n'est pas dans la classe *Object*.

5. La classe Object

toute classe descend de la classe Object

```
public class Animal
{
    private float poids ;

    public Animal(float p)
    {
        this.poids = y ;
    }

    public float getPoids( )
    {
      return this.poids ;
    }

    public void affiche( )
    {
        System.out.println (this.poids) ;
    }
}// fin classe Animal
```

extends *Object* est implicite
(on peut le mettre explicitement)



Méthodes de la classe Object

public String toString( )

La chaîne retournée est constituée du nom de la classe de l'instance référencée par **this**, suivi de @, suivi de la valeur de la référence de l'instance

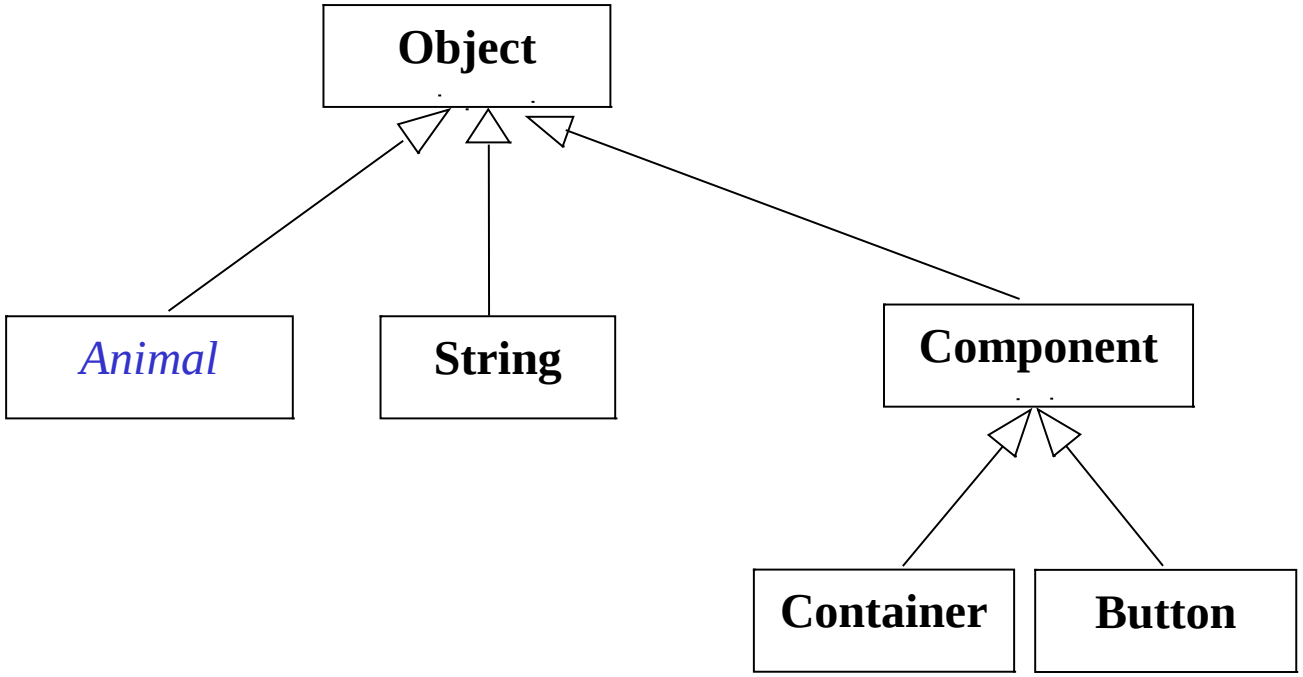
Ex : Animal a = new Animal(50) ;
System.out.println(a) ;

affiche
«Animal@13721b2»
(si la méthode *toString* n'est pas redéfinie dans *Animal*)

public boolean equals(Object o)

(compare l'instance référencée par **this** avec celle référencée par **o**)
retourne *true* si *this* et *o* référencent la même instance (*this == o*).

Chaque classe hérite des méthodes *toString* et *equals* de la classe *Object*



Chaque classe doit donc redéfinir ces 2 méthodes si elle veut en changer le comportement

Redéfinition de la méthode *toString*

Exemple

```
public class Object
{

    public String toString( )
    {
        ...
    }

    ...

} // fin classe Object
```

toString est redéfinie car les signatures sont identiques

```
public class Animal
{

    private float poids ;

    ...

    public String toString( )
    {
        return ("poids = " + this.poids) ;
    }

} // fin classe Animal
```

```
{
    Animal a = new Animal(50) ;

    System.out.println(a) ;
}
```

c'est la méthode *toString* spécifique (et non *toString* héritée) qui est appelée.
Affichage "poids = 50".
La méthode *toString* héritée est masquée par la méthode *toString* spécifique

Redéfinition de la méthode *equals* (1/2)

Exemple de ce qu'il ne faut pas faire :

```
public class Object
{

    public boolean equals(Object o )
    {
        return this == o ;
    }

     ...

} // fin classe Object
```

equals n'est pas redéfinie car les signatures ne sont pas identiques !

```
{

    private float poids ;

     ...

    public boolean equals(Animal a)
    {
        return (this.poids = = a.poids) ;
    }

} // fin classe Animal
```

```
{
    Animal a = new Animal(50) ;

    Object o  = new Animal(50) ;

     if (a.equals(o))

        System.out.println("égalité") ;
     else
         System.out.println("pas égalité") ;
}
```

L'argument *o* étant une référence sur un *Object*, c'est la méthode de signature *equals(Object)* que recherche la machine virtuelle dans la classe *Animal*. Celle-ci n'y étant pas, la méthode est recherchée dans la classe *Object*.
Affichage : *"pas égalité"*

Redéfinition de la méthode *equals* (2/2)

Exemple de ce qu'il faut faire :

```
public class Object
{

    public boolean equals(Object o )
    {
        return this == o ;
    }

    ...

} // fin classe Object
```

equals(Object o) est redéfinie

```
public class Animal
{
    public boolean equals(Object o)
    {
        if (o = = null)  return false ;

        if (!(o instanceof Animal))  return false ;

        return (this.poids = = ((Animal) a).poids) ;

    }
} // fin classe Animal
```

```
{
    Animal a = new Animal(50) ;

    Object o  = new Animal(50) ;

    if (a.equals(o))

        System.out.println("égalité") ;
    else
        System.out.println("pas égalité") ;
}
```

La machine virtuelle recherche la méthode *equals(Object)* dans la classe *Animal*. Elle la trouve et l'invoque.
Affichage *"égalité"*

Sens de la rédéfinition de la méthode equals

equals retourne *true* si :

1) l'instance référencée par *o* existe,
2) les instances référencées par *o* et *this* sont de la même classe,
3) et ont les mêmes valeurs des variables d'instances

```
public class Animal
{

    public boolean equals(Object o)
    {
        if (o = = null)  return false ;

        if (!(o instanceof Animal))  return false ;


        return (this.poids = = ((Animal) o).poids) ;

    }

}   // fin classe Animal
```

si o = = null il n'y a pas d'instance référencée par *o*

retourne *false* si o ne réfère pas une instance d'*Animal*

On compare les valeurs des variables d'instance *poids*. Il faut faire un *cast* : pour le compilateur *o* est une référence sur *un Object* et ne reconnaît pas la variable d'instance *poids*.

Exemple de rédéfinition de *equals*

```
public class Mammifere extends Animal
{

  public boolean equals(Object o)
  {
      if (o = = null)  return false ;

       if (!(o instanceof Mammifere))  return false ;

      return (
              super.equals(o) &&

              this.nbMamelles = = ((Mammifere) o).nbMamelles
             ) ;
   }
```

appelle la méthode *equals(Object o)* de *Animal* pour comparer les variables d'instances propres à *Animal*

```
{
  { Mammifere m1 = new Mammifere(50, 6) ;
    Mammifere m2 = new Mammifere(50, 4) ;

   if (m1.equals(m2)) System.out.println("égales") ;
   else System.out.println("pas égales") ;
}
```

affiche *"pas égales"*

:Mammifere
- poids 50
- nbMamelles 6

:Mammifere
- poids 50
- nbMamelles 4

```
}     // fin classe Mammifere
```