

# POO en Java

## Héritage et **Polymorphisme**

**Plusieurs formes**

Poly == Plusieurs

Morph == Forme

# La classe **Object**

- Superclasse de toutes les classes Java
- Toute classe sans clause *extends* explicite est une sous-classe directe de la classe **Object**
- Les Méthodes de la classe **Object** incluent:
  - String toString()
  - boolean equals (Object other)
  - int hashCode()
  - Object clone()

# **La classe Object**

# La Méthode toString()

- Retourne une représentation textuelle de **Object**; décrit l'état de **Object** (et donc de tout autre objet qui l'utilise correctement)
- Automaticament appelée quand:
  - **Object** est concaténé avec une String
  - **Object** est imprimée utilisant print() ou println()
  - ...

# Exemple

```
Rectangle r = new Rectangle (0,0,20,40);  
System.out.println(r);
```

Affiche:

```
java.awt.Rectangle[x=0,y=0,width=20,height=40]
```

# toString()

- La méthode toString() par défaut imprime juste (full) nom de la classe & le *hash code* de l'objet
- Pas toutes les classes de l'API redéfinissent toString()
- Bonne idée de l'implémenter pour besoins de *debugging*:
  - Devrait retourner une String contenant les valeurs des champs importants avec leurs noms
  - Devrait aussi retourner le résultat de `getClass().getName()` que le nom de la classe *hard-coded*

# toString()

- Invoquée, par défaut, dans des contextes requierant une String à la place de la référence d'un objet.

```
class Object {  
    public String toString() {  
        return getClass().getName()+"@"+  
            Integer.toHexString(hashCode());  
    }  
    ...  
}
```

# Representation de Object sur System.out

- Qu'est ce qui se passe lorsqu'un objet est imprimé?
  - La méthode toString() de l'objet fournit la string à être imprimée. **Rectangle@a122c09**
  - Toutes les classes ont une méthode toString( ) par défaut, celle qui est définie par la classe Object (pas très descriptive)

```
public String toString() {  
    return getClass().getName()+"@"+Integer.toHexString(hashCode());  
}
```

- On peut fournir une version "taillée" de toString() dans n'importe quelle classe très facilement



# Overriding toString(): exemple

```
public class Personne
{
    public String toString()
    {
        return getClass().getName()
            + "[nom=" + nom
            + ",NoCIN=" + noCIN
            + "]";
    }
    ...
}
```

String typique produite: Personne[nom=Anas Ali,noCIN =AB309]

# Overriding toString dans une sous-classe

- Formater la superclasse en premeir
- Ajouter les détails unique à la sous-classe
- Example:

```
public class Etudiant extends Personne
{
    public String toString()
    {
        return super.toString()
            + "[CNE=" + cne+ "];"
    }
    ...
}
```

# Exemple (suite)

- String Typique produite:  
Etudiant[nom=Omar Ahmed,noCIN=A313858][CNE=201398765]
- Noter que la superclasse rapporte le nom de la classe actuelle

# Test d'égalité

- La méthode equals() teste si deux objets ont le même contenu
- Par contraste, l'opérateur == teste 2 références pour voir s'ils font référence au même objet (ou teste les valeurs primitives pour l'égalité)
- Nécessité de définir pour chaque classe ce que signifie “égalité” :
  - Comparer tous les champs
  - Comparer 1 ou 2 champs clés

# Test d'égalité

- Object.equals teste pour l'identité:

```
public class Object
{
    public boolean equals(Object obj)
    {
        return this == obj;
    }
    ...
}
```

- Override (redéfinir) equals si on ne veut pas hériter ce comportement

# Overriding equals()

## Redéfinition de equals()

- Bonne pratique de override, puisque beaucoup de méthodes de l'API Java présument que les objets ont une notion bien définie de l'égalité
- Quand on redéfinit equals() dans une sous classe, on peut appeler la version de sa superclasse en utilisant `super.equals()`

# Exigences pour la méthode equals()

- Doit être ***reflexive***: pour toute référence x, x.equals(x) est true
- Doit être ***symmetric***: pour toute référence x et y, x.equals(y) est true si et seulement si y.equals(x) est true
- Doit être ***transitive***: si x.equals(y) et y.equals(z), alors x.equals(z)
- Si x est non null, alors x.equals(null) doit être false

```

class Point {
    ...
    public String toString(){
        return "["+getX()+" , "+getY()+" ] ";
    }

    public boolean equals(Object o){
        if(o == this){
            return true;
        }
        else if(o == null){
            return false;
        }
        else if(o instanceof Point){
            Point p = (Point) o;
            return getX() == p.getX() && getY() == p.getY();
        }
        else{
            return false;
        }
    }
    ...
}

```



# Ou encore

```
public boolean equals(Object otherObject)
{
    if (this == otherObject) return true;
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
    ...
}
```

# Hashage

- Technique utilisée pour trouver rapidement les éléments dans une structure de données, sans faire une recherche linéaire complète
- Concepts importants :
  - Hash code: valeur entière utilisée pour trouver indice d'un tableau pour la lecture/écriture de données
  - Hash table: tableau d'éléments arrangés suivant le *hash code*
  - Hash function: calcule le *hash code* pour un élément; utilise un algorithme qui produit probablement différents *hash codes* pour différents objets afin de minimiser les collisions

# Hashage en Java

- La librairie Java contient les classes HashSet et HashMap
  - Utilise les *hash tables* pour le stockage de données
  - Puisque Object a une méthode hashCode (fonction de hashage ), n'importe quel type d'objet peut être stocké dans une *hash table*

# hashCode() par défaut

- Hashe l'adresse mémoire d'un objet; consistante avec la méthode equals() par défaut
- Si on redéfinit equals(), on devrait aussi redéfinir hashCode()
- Pour les classes qu'on définit, utiliser le produit de hashage de chaque champ et un nombre premier, puis additionner tous ensemble – donne le hash code en résultat

# Exemple

```
public class Employee
{
    public int hashCode()
    {
        return 11 * name.hashCode()
            + 13 * new Double(salary).hashCode();
    }
    ...
}
```

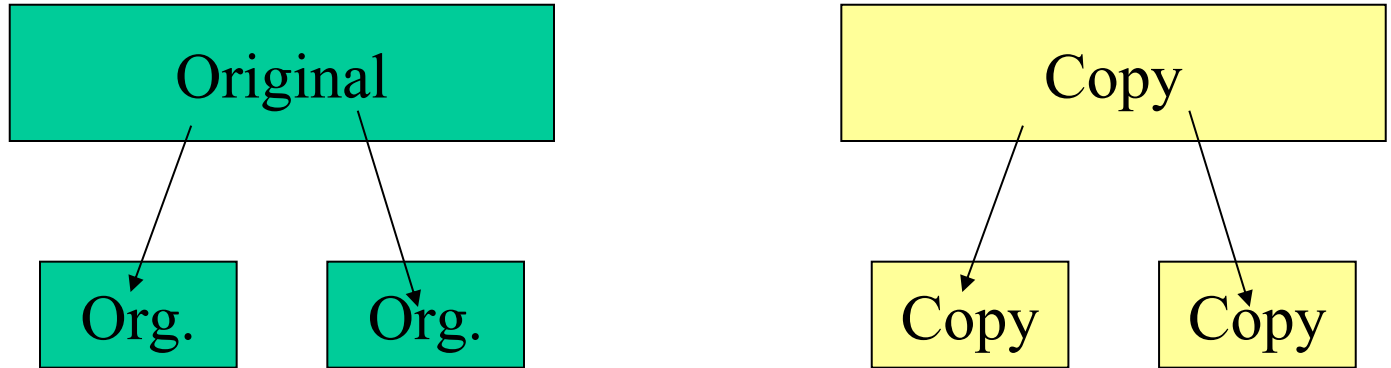
Peut être générée automatiquement...

# Copie des objets

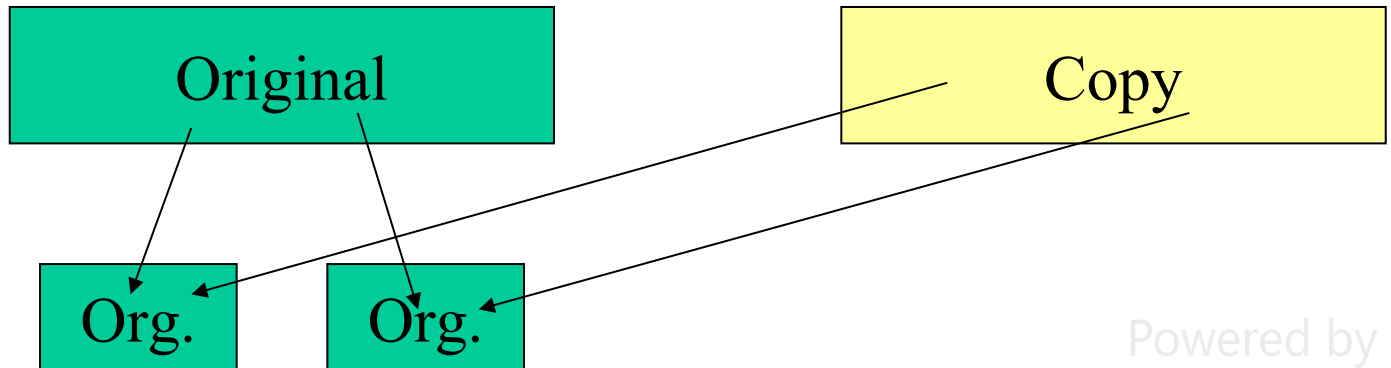
- Shallow copy (superficielle)
  - Copie de la référence d'un objet est une autre référence au même objet
  - Réalisée avec l'opérateur d'affectation
- Deep copy (profonde)
  - Nouvel objet réellement créé, identique à l'original
  - A.K.A clonage
  - Constructeur de recopie ou clone()

# Copie d'objet

Deep  
copy



Shallow  
copy



# Polymorphisme



# Les Classes Shape

- Soit la classe **Shape**
  - supposons tous les shapes ont des coordonnées x et y
  - redéfinissons la version **toString** de la classe **Object**
- Deux sous-classes de la classe Shape
  - **Rectangle** définit une **nouvelle** méthode **changeWidth**
  - **Circle**

# Une Classe *Shape*

```
public class Shape
{
    private double dMyX, dMyY;

    public Shape() { this(0,0);}

    public Shape (double x, double y)
    {
        dMyX = x;
        dMyY = y;
    }

    public String toString()
    {
        return "x: " + dMyX + " y: " + dMyY;
    }

    public double getArea() {
        return 0;
    }
}
```

# Une Classe *Rectangle*

```
public class Rect extends Shape
{
    private double dMyWidth, dMyHeight;

    public Rect(double width, double height)
    {
        dMyWidth = width;
        dMyHeight = height;
    }
    public String toString()
    {
        return
            " width " + dMyWidth
            + " height " + dMyHeight;
    }
    public double getArea() {
        return dMyWidth * dMyHeight;
    }
    public void changeWidth(double width){
        dMyWidth = width;
    }
}
```

# Une Classe *Circle*

```
class Circle extends Shape {  
    private double radius;  
    private static final double PI = 3.14159;  
  
    public Circle(double rad) {  
        radius = rad;  
    }  
  
    public double getArea() {  
        return PI * radius * radius;  
    }  
  
    public String toString() {  
        return " radius " + radius;  
    }  
}
```

# Polymorphisme

- Si la classe *Rect* est **derivée** de la classe *Shape*, une operation qui peut être exécutée sur un objet de la classe *Shape* peut aussi être exécutée sur un objet de la classe *Rect*
- Quand une requête d'utiliser une **méthode** est effectuée à travers une référence de la **superclasse**, Java choisit la méthode redéfinie correcte "**polymorphiquement**" dans la **sous-classe appropriée** associée à l'objet
- **Polymorphisme** signifie “différentes formes”

# Références aux objets

```
Rect r = new Rect(10, 20);  
Shape s = r;  
System.out.println("Area is " + s.getArea());
```

- Etant donné que la classe **Rect** est une **sous-classe** de la classe **Shape**, et elle **redéfinit** la méthode *getArea()*.
- Cela va t-il fonctionner?

# Références aux objets

```
Rect r = new Rect(10, 20);  
Shape s = r;  
System.out.println("Area is " + s.getArea());
```

- Le code fonctionne si **Rect extends Shape**
- Une référence à un objet peut faire référence à un objet de son **type de base** ou un **descendant** dans la chaîne d'héritage
  - La relation *est-un* est vérifiée. Un **Rect est-un shape** alors **s** peut faire référence à **Rect**
- C'est une forme de **polymorphisme** et est utilisée extensivement dans le JCF "Java *Collection* Framework"
  - Vector, ArrayList, LinkedList sont des listes d'objets

# Polymorphisme

```
Circle c = new Circle(5);  
Rect r = new Rect(5, 3);  
Shape s = null;  
if( Math.random(100) % 2 == 0 )  
    s = c;  
else  
    s = r;  
System.out.println( "Shape is "  
    + s.toString() );
```

- Supposons maintenant que **Circle** et **Rect** sont toutes les deux des **sous-classes** de **Shape**, et toutes les deux ont **redéfinis** *toString()*, quelle version sera appelée?



# Polymorphisme

```
Circle c = new Circle(5);  
Rect r = new Rect(5, 3);  
Shape s = null;  
if( Math.random(100) % 2 == 0 )  
    s = c;  
else  
    s = r;  
System.out.println( "Shape is "  
    + s.toString() );
```

- **Circle** et **Rect** ont **redéfinis** toString quelle version sera appelée?
  - Le code fonctionne parce que **s** est **polymorphique**
  - L'appel de méthode déterminé à l'exécution par le **dynamic binding (liaison dynamique)**

# Compatibilité de Type

```
Rect r = new Rect(5, 10);  
Shape s = r;  
s.changeWidth(20);
```

- Etant donné la classe **Rect** qui est une sous-classe de la classe **Shape**, et qui a une **nouvelle** méthode *changeWidth(double width)*, que sa superclasse n'a pas.
- Cela va t-il fonctionner?

# Compatibilité de Type

```
Rect r = new Rect(5, 10);  
Shape s = r;  
s.changeWidth(20); // erreur de syntaxe
```

- Le **polymorphisme** permet à **s** de faire référence à un objet **Rect**, mais il y a des limitations
- Le code en haut ne fonctionne pas
- Comment le modifier un peu pour le faire fonctionner sans changer les définitions des classes ?

# Compatibilité de Type

```
Rect r = new Rect(5, 10);  
Shape s = r;  
s.changeWidth(20); // erreur de syntaxe
```

- Le **polymorphisme** permet à **s** de faire référence à un objet **Rect**, mais il y a des limitations
- Le code en haut ne fonctionne pas
- Statiquement **s** est déclarée pour être un **shape**
  - pas de méthode **changeWidth** dans la classe **Shape**
  - doit faire un **cast** de **s** à un *rectangle*;

```
Rect r = new Rect(5, 10);  
Shape s = r;  
((Rect) s).changeWidth(20); // Okay
```

# Problèmes avec le Casting

```
Rect r = new Rect(5, 10);  
Circle c = new Circle(5);  
Shape s = c;  
((Rect) s).changeWidth(4);
```

- Est ce que cela fonctionne?

# Problèmes avec le Casting

- Le code suivant compile mais une **exception** est lancée à l'**exécution**

```
Rect r = new Rect(5, 10);  
Circle c = new Circle(5);  
Shape s = c;  
(Rect) s).changeWidth(4);
```

- Le Casting** doit être fait soigneusement et correctement
- Si l'on est pas sûr de quel type l'objet sera alors utilise l'opérateur **instanceof**

# L'opérateur instanceof

```
Rect r = new Rect(5, 10);  
Circle c = new Circle(5);  
Shape s = c;  
if (s instanceof Rect)  
    ((Rect) s).changeWidth(4);
```

- syntaxe: **expression instanceof NomClasse**

# Casting

- Il est toujours possible de **convertir une sous-classe à une superclasse**. pour cette raison, le casting explicite peut être omis. Par exemple,
  - **Circle c1 = new Circle(5) ;**
  - **Shape s = c1 ;**

est équivalent à

- **Shape s = (Shape) c1 ;**
- Le casting **explicite** doit être utilisé quand on fait le casting d'un objet d'une **superclasse à une sous-classe**. Ce type de casting peut ne pas aboutir.
  - **Circle c2 = (Circle) s ;**



```

class Point {
    ...
    public String toString(){
        return "["+getX()+" , "+getY()+" ] ";
    }

    public boolean equals(Object o){
        if(o == this){
            return true;
        }
        else if(o == null){
            return false;
        }
        else if(o instanceof Point){
            Point p = (Point) o;
            return getX() == p.getX() && getY() == p.getY();
        }
        else{
            return false;
        }
    }
    ...
}

```

# Polymorphisme et collections

# Liskov's Substitution Principle

**If for each **object O1** of type S there is an **object O2** of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.**

- Barbara Liskov, "Data Abstraction and Hierarchy",  
SIGPLAN Notices, 23, 5 (May, 1988)

# Polymorphisme : Principe général

Le terme de *polymorphisme* décrit la caractéristique d'un élément qui peut prendre plusieurs formes, comme l'eau qui se trouve à l'état solide, liquide ou gazeux.

## Principe de substitution défini par Liskov :

Il doit être possible de substituer n'importe quel objet instance d'une sous-classe à n'importe quel objet instance d'une superclasse sans que la sémantique du programme écrit dans les termes de la superclasse ne soit affectée.

# une sous-classe ne peut pas diminuer l'accessibilité

- *Une sous-classe ne peut pas diminuer l'accessibilité d'une méthode définie dans la superclasse.*

```
public class Base {  
    public void method() {...}  
}  
public class Sub extends Base {  
    private void method() {...}  
}
```



Si ci-dessus est admis,  
alors...**conflit** (compile  
correctement, mais  
erreur d'exécution,  
**private** ne peut pas être  
accessible)

```
Base base=new Sub();  
base.method();
```

Powered by

WPS Office

# Hiérarchies de classes

- Les classes en Java forment des **hierarchies**. Except é la classe **Object** qui se trouve au sommet de la hierarchie, toute classe en Java est une **sous-classe** d'une autre classe. Une classe peut avoir plusieurs sous-classes, mais chaque classe a seulement une superclasse.
- Une classe représente une spécialisation de sa superclasse. Si vous créez un objet qui est une instance d'une classe, cet objet est aussi une instance de toutes les autres classes au-dessus de lui dans la hiérarchie (dans chaîne des superclasses).
- Lorsqu'on d é finit une nouvelle classe en Java, cette classe **h érite** automatiquement le comportement de sa superclasse.<sup>46</sup>

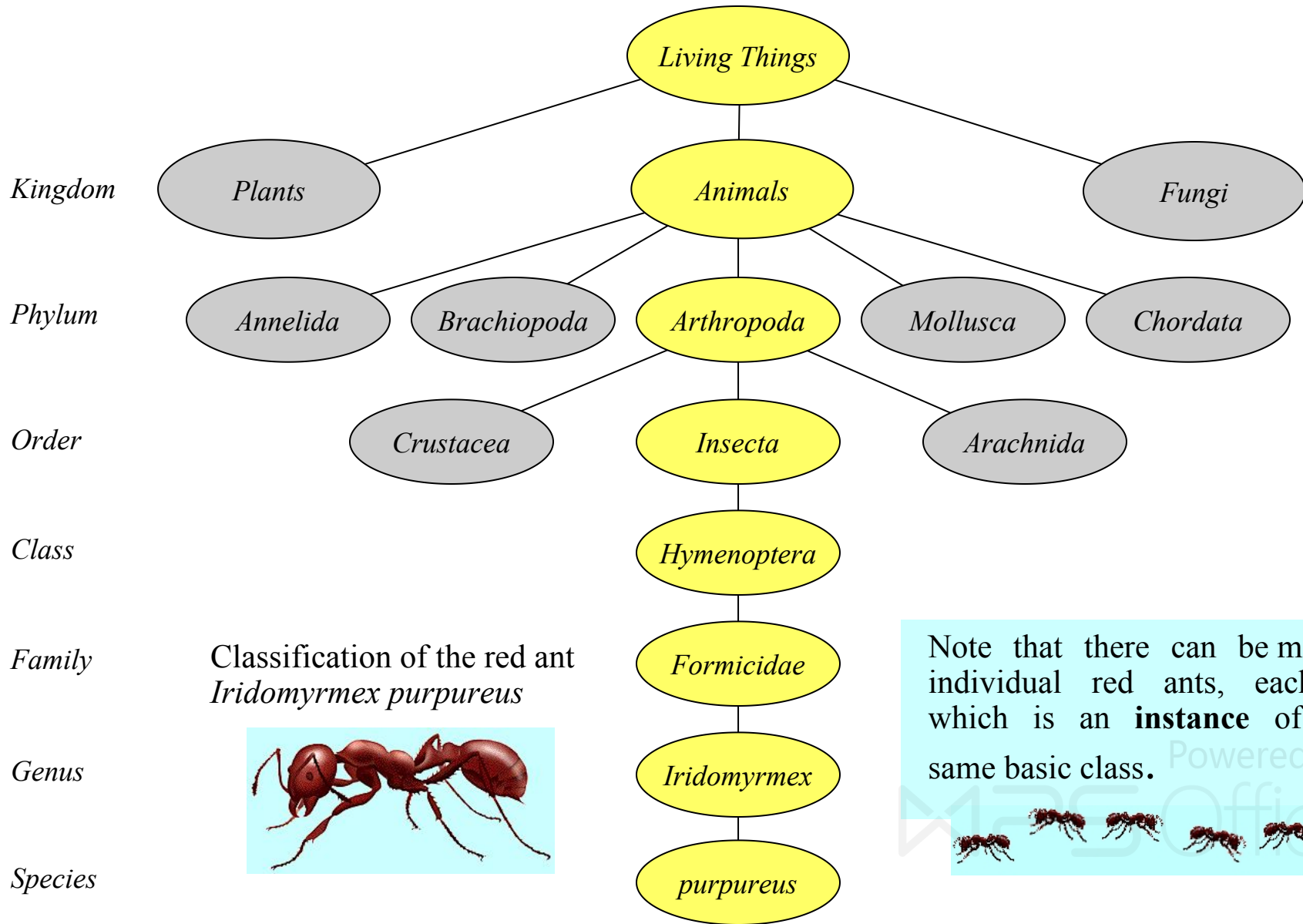
# Modèles Biologiques de la Structure de Classe

The structure of Java's class hierarchy resembles the biological classification scheme introduced by Scandinavian botanist Carl Linnaeus in the 18th century. Linnaeus's contribution was to recognize that organisms fit into a hierarchical classification scheme in which the placement of individual species reflects anatomical similarities.



**Carl Linnaeus (1707–1778)**

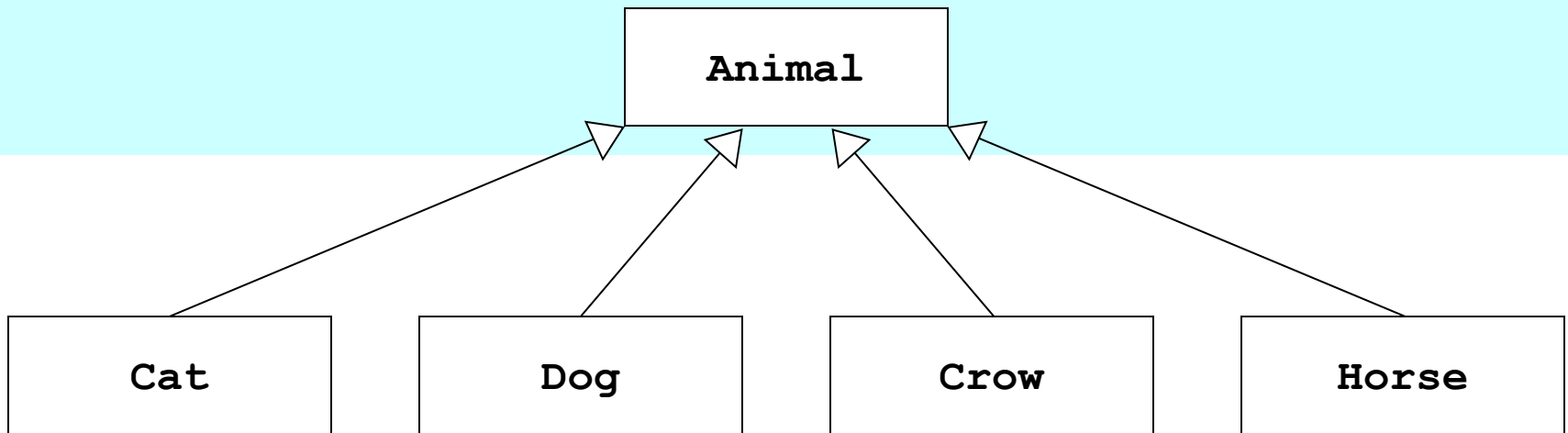
# Hiérarchies de classes





# La Hiérarchie **Animal**

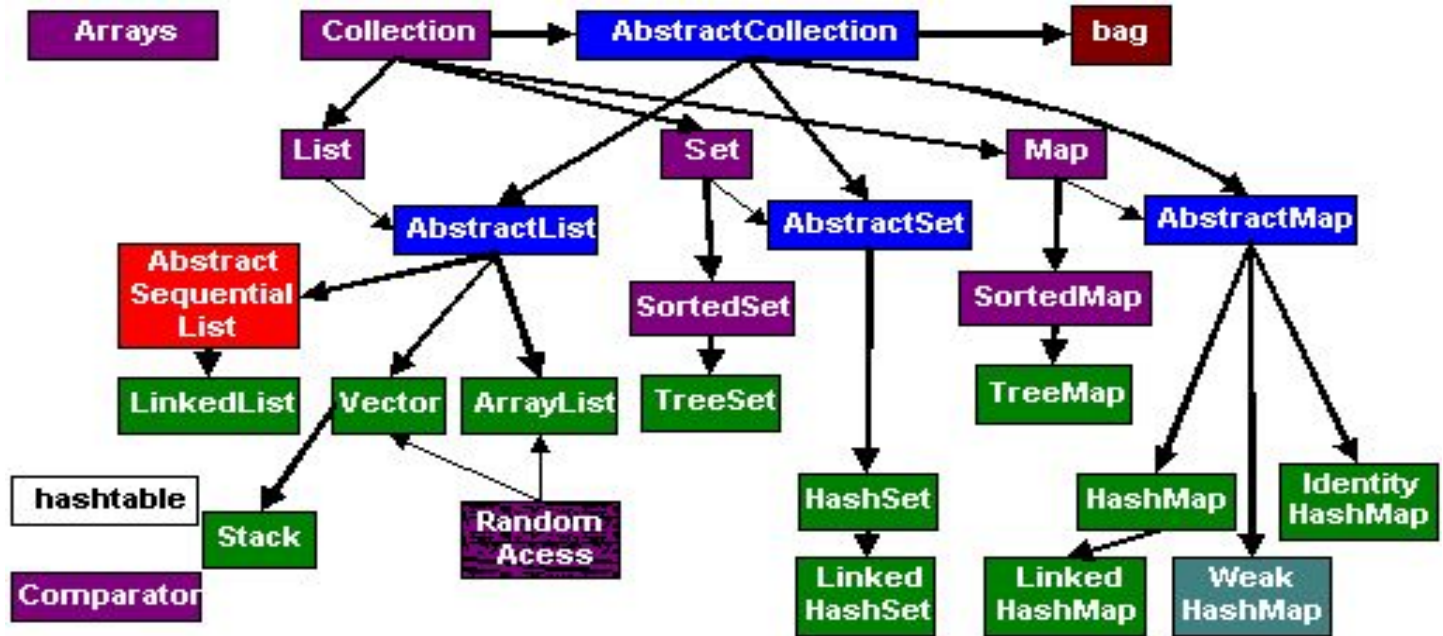
Les classes qui représentent des objets de type animal forment une hiérarchie, dont une partie ressemble à ceci:



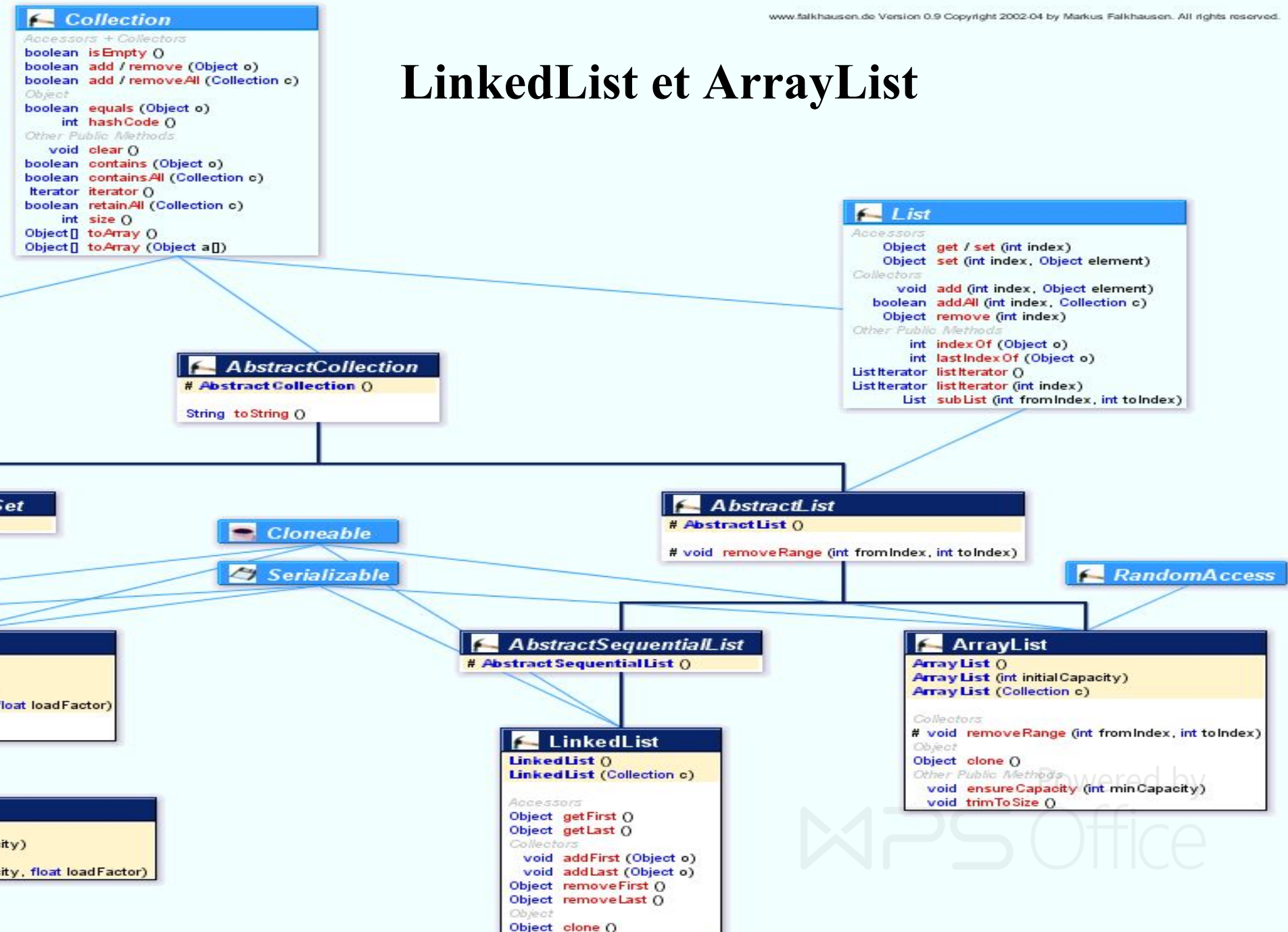
La classe **Animal** représente tous les objets d'origine animale. Les quatre sous-classes indiquées dans ce schéma correspondent à des types particuliers d'objets: chats, chiens, chevaux, les corneilles. Il est clair dans le diagramme de classe, que tout **Cat**, **Dog**, **Crow**, or **Horse** est aussi un **Animal**. Mais l'inverse n'est PAS vrai. c'est-à-dire, tout **Animal** n'est pas un **Cat**; tout **Animal** n'est pas un **Dog**, etc.

# Java Collections

- LinkedList
- ArrayList
- Vector
- Stack
- HashSet
- TreeSet
- HashTable
- Plus beaucoup d'autres développées par vous ...
- Ils manipulent tous des **Object**



# LinkedList et ArrayList



# Operations sur LinkedList

## quelques méthodes

- Constructeur

LinkedList() construit une liste vide

- insertion au début de la liste

void addFirst(Object o)

- insertion à la fin de la liste

void addLast(Object o)

- suppression du début de la liste

Object removeFirst()

- suppression de la fin de la liste

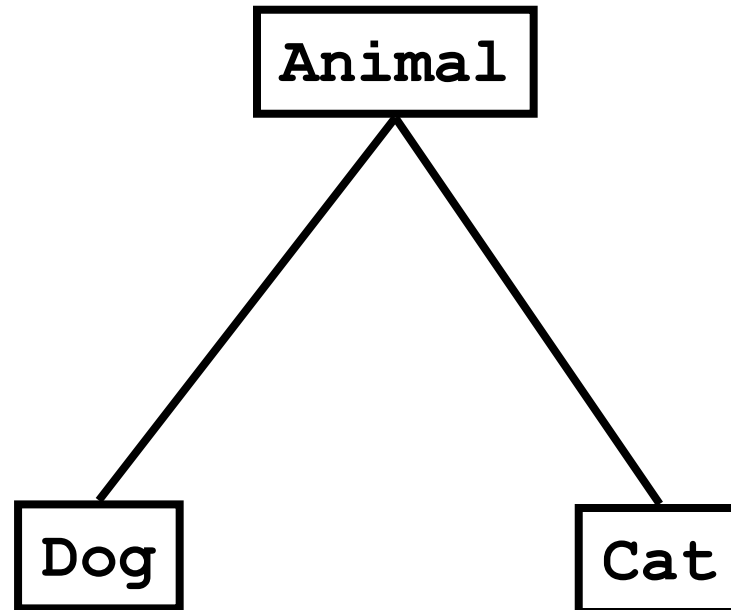
Object removeLast()

- accès à l'objet d'indice *index*

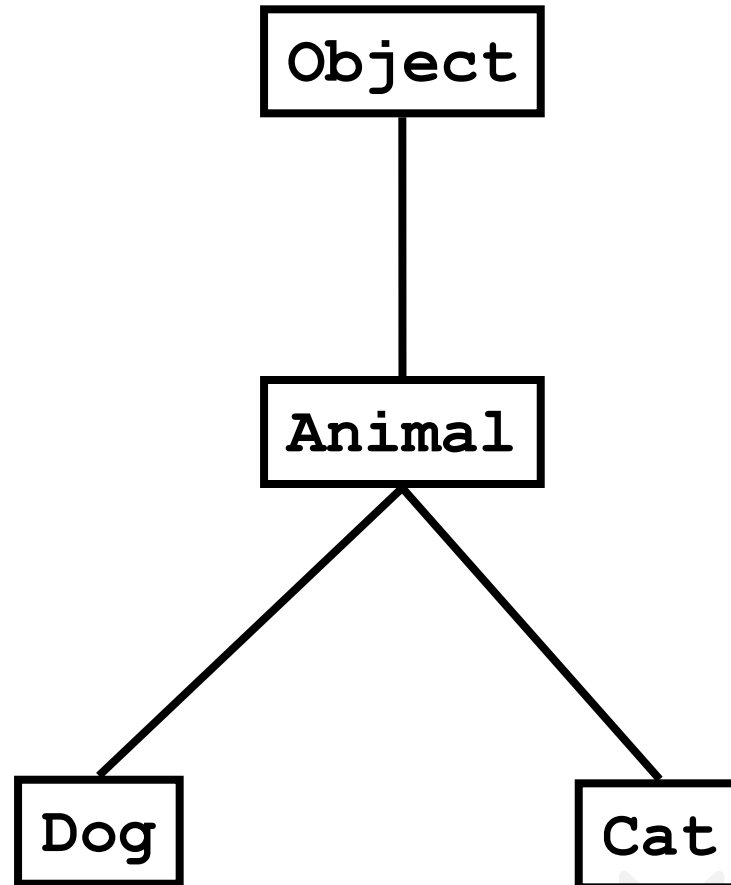
Object get(int index)

- int **size()** Retourne le nombre d'éléments de cette liste.

# Polymorphisme, Collections et casting



# Polymorphisme, Collections et casting



Powered by

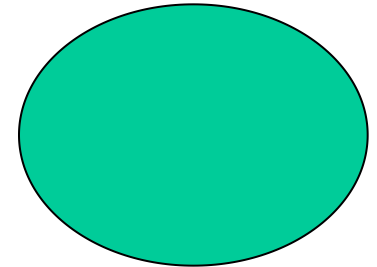
WPS Office

# Polymorphisme, Collections et casting

- La classe **Object** existe
- Elle définit beaucoup de méthodes utiles
  - e.g. toString
  - voir l'API
- Ainsi, chaque classe est soit
  - une sous-classe directe de **Object** (pas d'*extends*)
  - ou
  - une sous-classe d'un descendant de **Object** (extends)
  - Et alors?

# Polymorphisme, Collections et casting

- Tâches répétitives
- Collections de choses (objets)
  - Library items
  - Shapes
  - Animals
  - Vehicles
  - Students

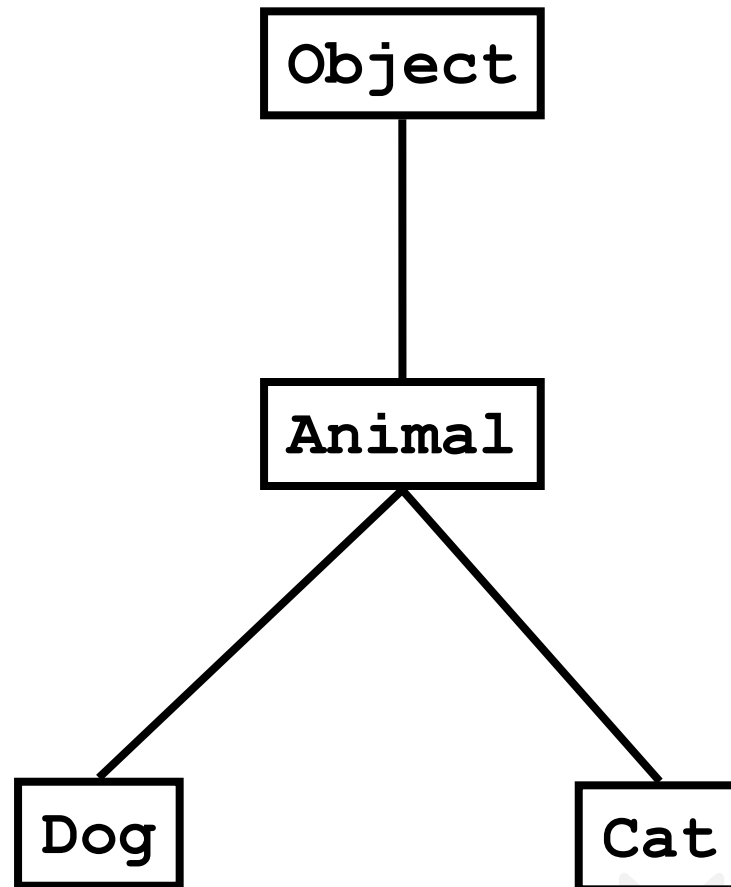




# Polymorphisme, Collections et casting

- Les collections sont rarement uniforme
- Désire d'une méthode pour tenir une collection d'éléments dissemblables
- Besoin de changer les règles de *type mismatch*

# Polymorphisme, Collections et casting

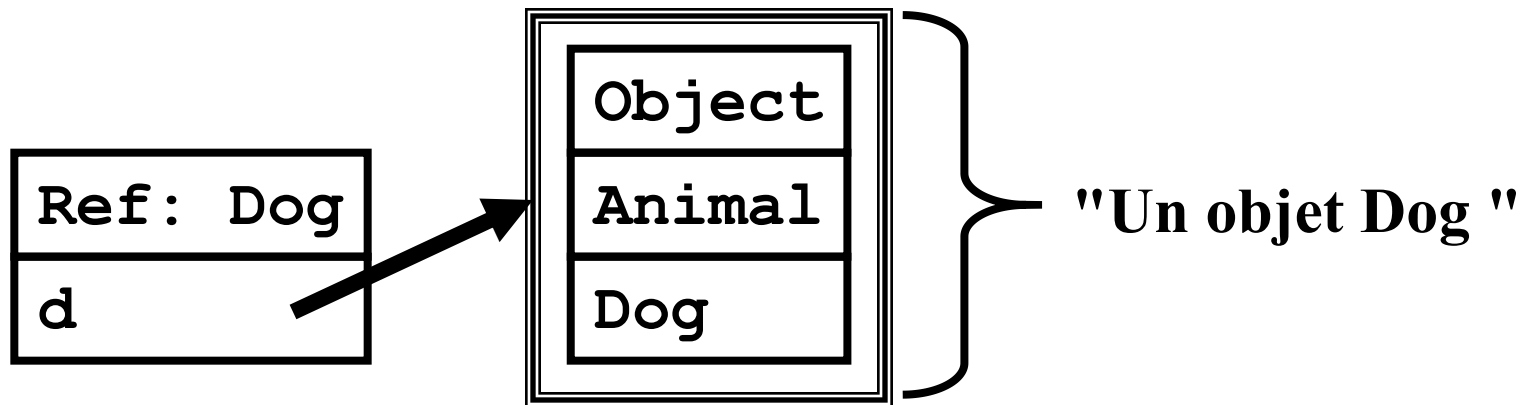


# Polymorphisme, Collections et casting

```
Object o;  
Animal a;  
Dog d = new Dog();  
Cat c = new Cat();  
d = c;           // Illégal  
a = c;           // OK, un Cat est un Animal  
o = c;           // OK, un Cat est un Object  
o = a;           // OK, un Animal est un Object  
a = o;           // Illégal, pas tous les Object sont  
                  // des Animal  
d = a;           // Illégal, pas tous les Animal sont des  
                  //Dogs
```

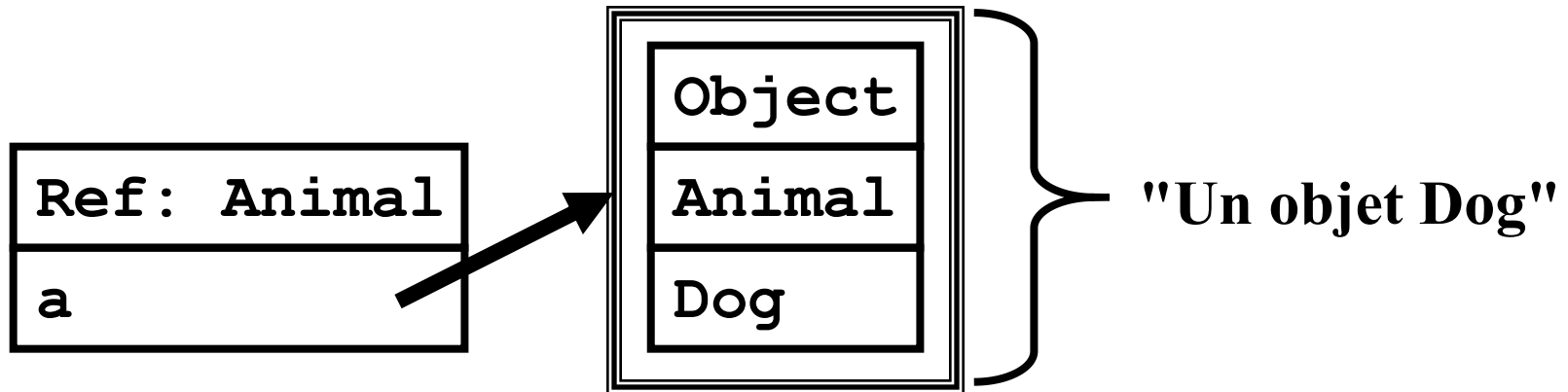
# Polymorphisme, Collections et casting

```
Dog d = new Dog();
```



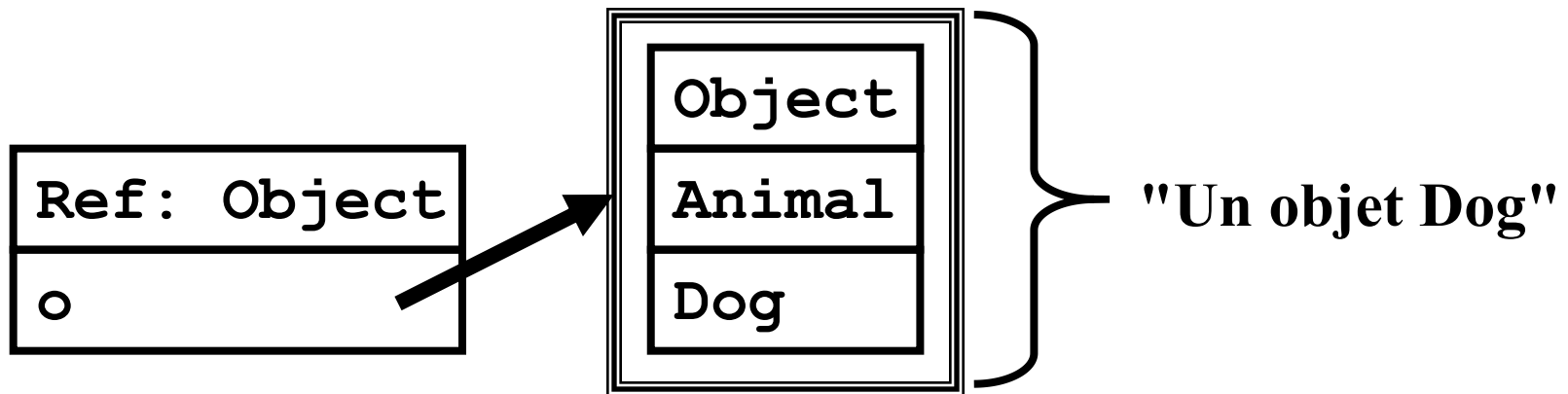
# Polymorphisme, Collections et casting

```
Animal a = new Dog();
```



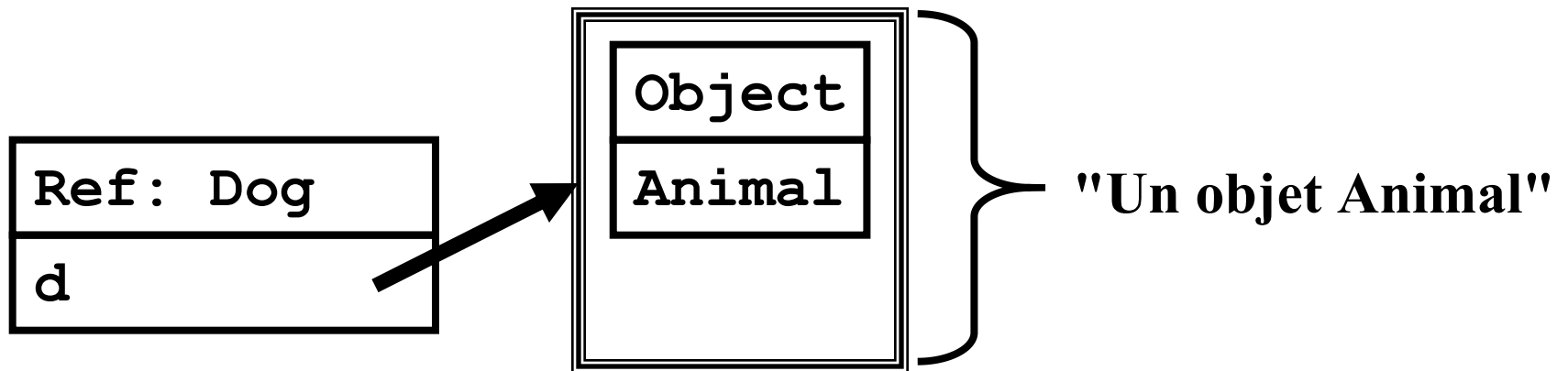
# Polymorphisme, Collections et casting

```
Object o = new Dog();
```



# Polymorphisme, Collections et casting

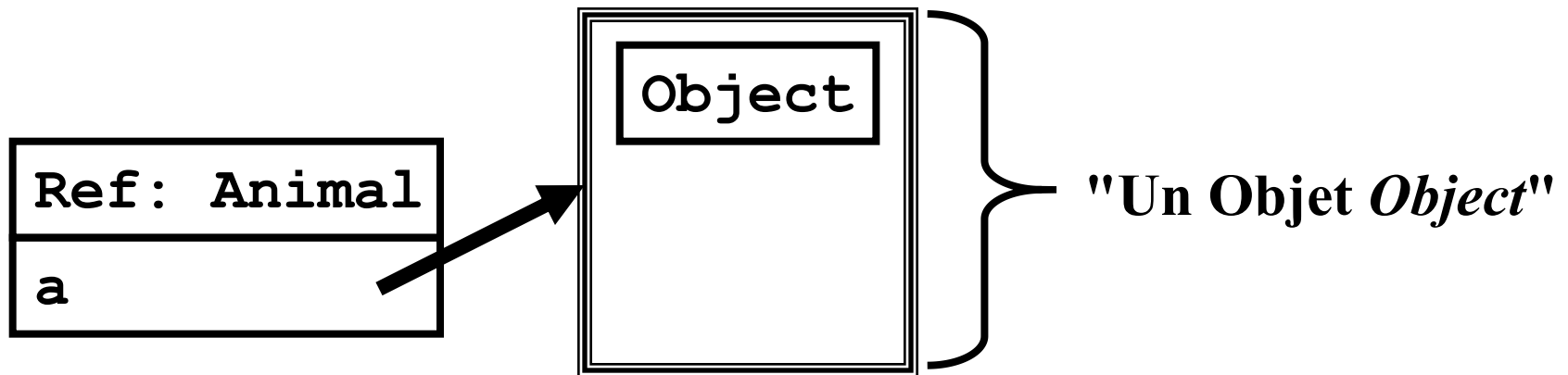
```
Dog d = new Animal();
```



**ILLEGAL**

# Polymorphisme, Collections et casting

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



**ILLEGAL**



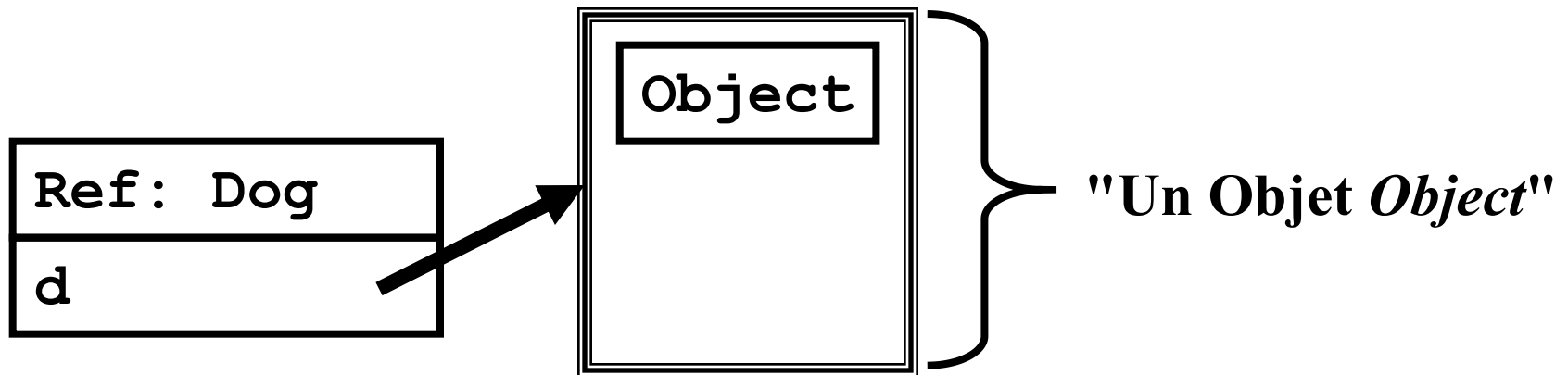
# Polymorphisme, Collections et casting

```
Dog d = new Object();
```

?

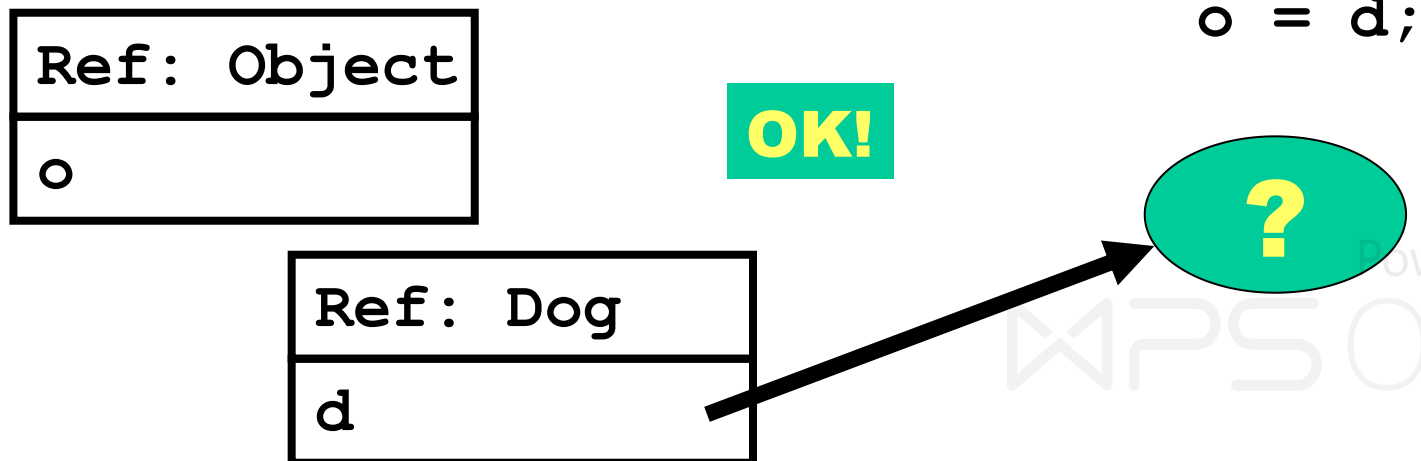
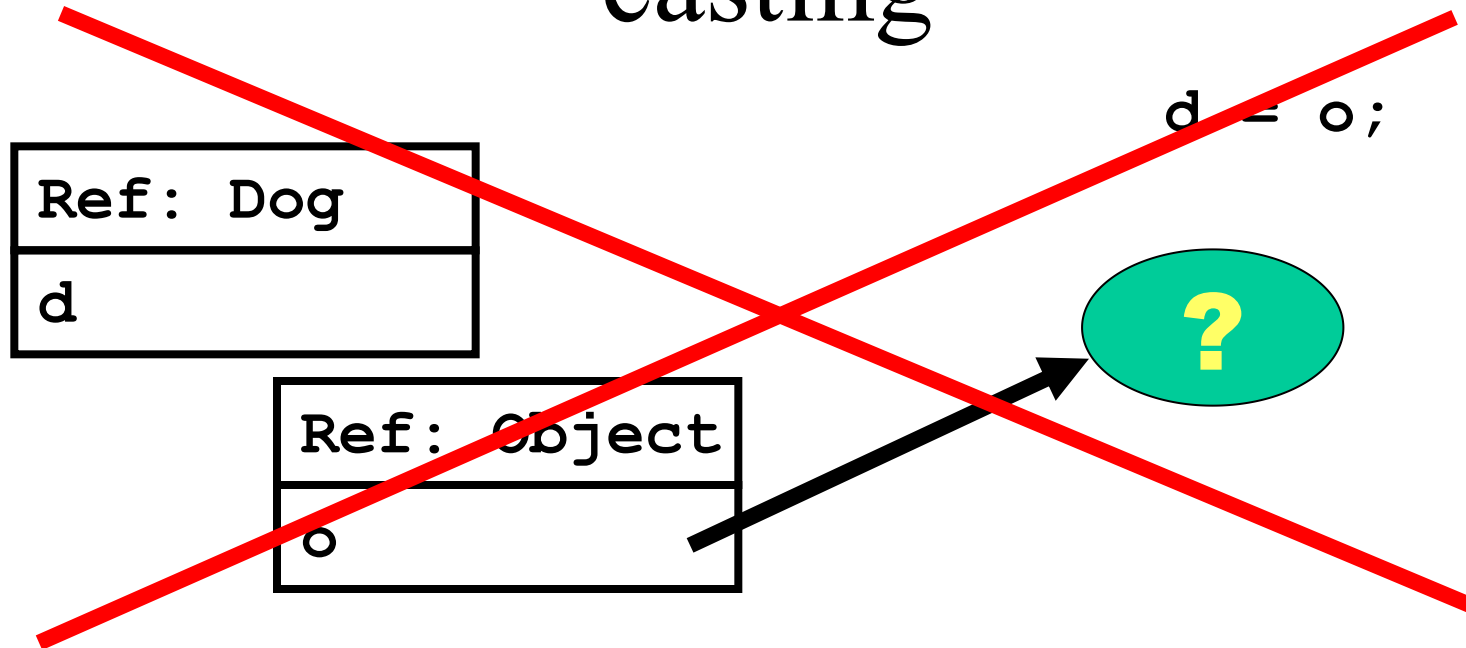
# Polymorphisme, Collections et casting

```
Dog d = new Object();
```



**ILLEGAL**

# Polymorphisme, Collections et casting



# Polymorphisme, Collections et casting

- Lorsqu'on enfreint les règles mentionnées ci-avant ...
- Java nous informe parfois que le *cast* est nécessaire
- Même si c'est vraiment une mauvaise idée

```
Pearls p;
```

```
Sugar s;
```

```
p = (Pearls) s;
```

# Polymorphisme, Collections et casting

- Philosophie Java: Capture des erreurs lors de la compilation.
- Conduisant à concept délicat :  
Dynamic Binding (Liaison dynamique)
- A l'exécution (dynamique) lorsqu'une méthode est invoquée sur une référence **l'objet réel** est examiné et la version "bas" ou plus proche de la méthode est réellement exécuté.

# Polymorphisme, Collections et casting

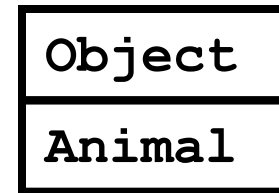
## Dynamic Binding (liaison dynamique)

- *The heart of polymorphism*
- Supposons que les classes Animal et Dog ont une méthode toString

```
Object o = new Dog();
```



```
Animal a = new Dog();
```



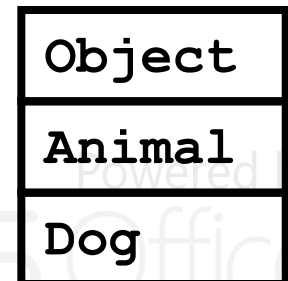
```
Dog d = new Dog();
```

```
o.toString();
```

```
a.toString();
```

```
d.toString();
```

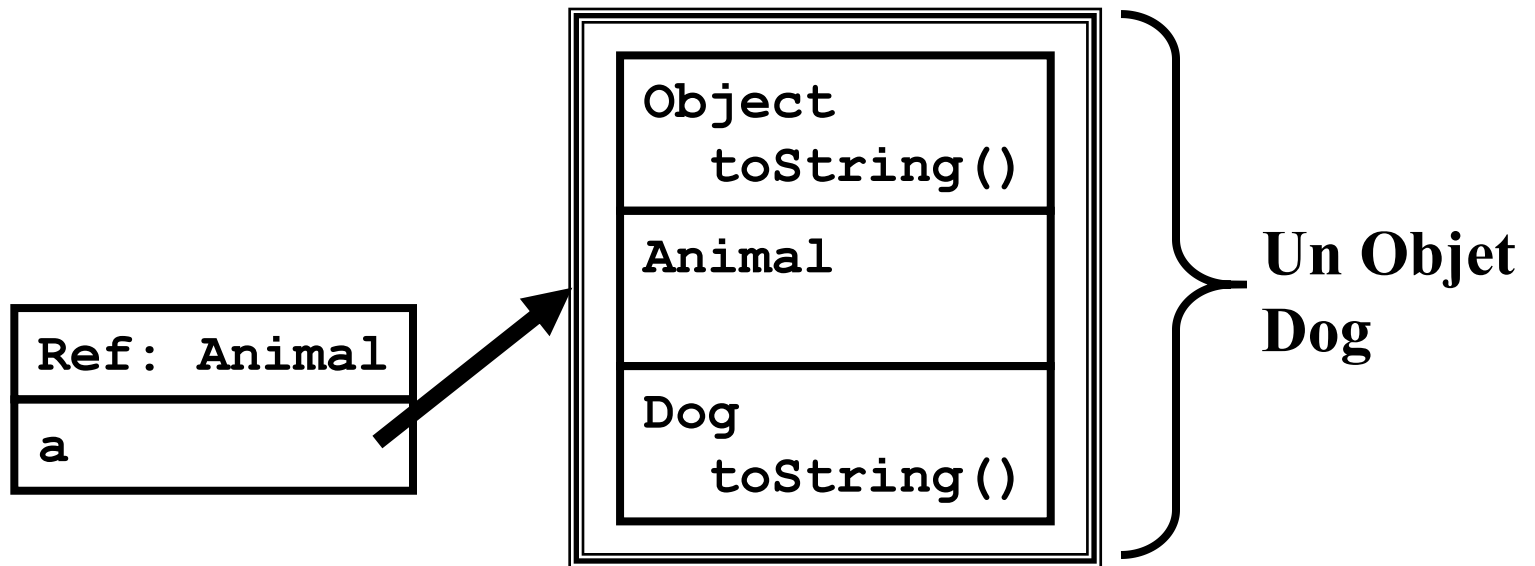
```
((Object)o).toString();
```



# Polymorphisme, Collections et casting

## Dynamic Binding (liaison dynamique)

- Ça fonctionne même comme ça...



```
Animal a = new Dog();  
a.toString();
```

# Polymorphisme, Collections et casting

## Dynamic Binding (liaison dynamique)

- Java vérifie les types lors de la compilation lors de l'affectation des références (la vérification au moment de l'exécution est également effectuée).
- Java décide toujours la méthode à être invoquée en regardant l'objet à l'exécution.
- Au moment de la compilation Java vérifie les invocations des méthodes pour s'assurer que le type de référence aura la bonne méthode. Cela peut paraître contradictoire à la liaison dynamique.



# Polymorphisme, Collections et casting

## Dynamic Binding (liaison dynamique)

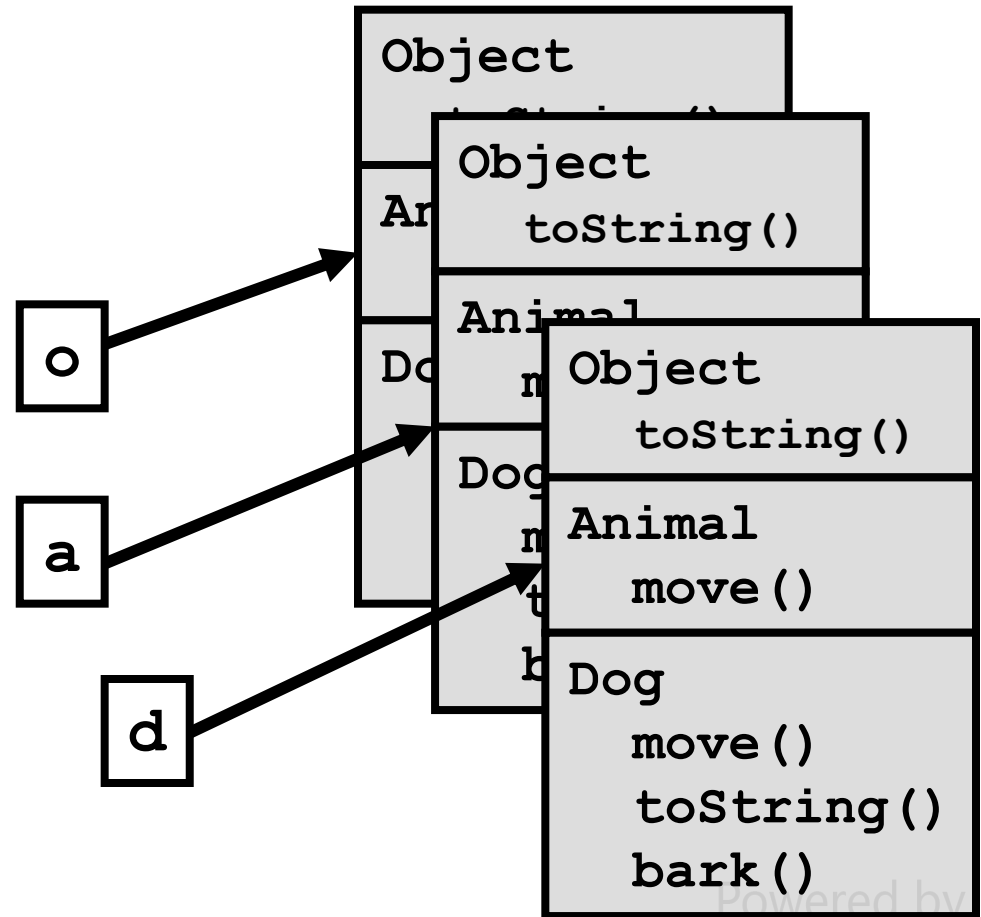
**$x.y()$  ;**

- $x$  est une référence qui a un type qui est sa classe
- Cette classe (ou une superclasse) doivent avoir une méthode  $y$  ou une erreur de compilation se produira.

# Polymorphisme, Collections et casting

## Dynamic Binding (liaison dynamique)

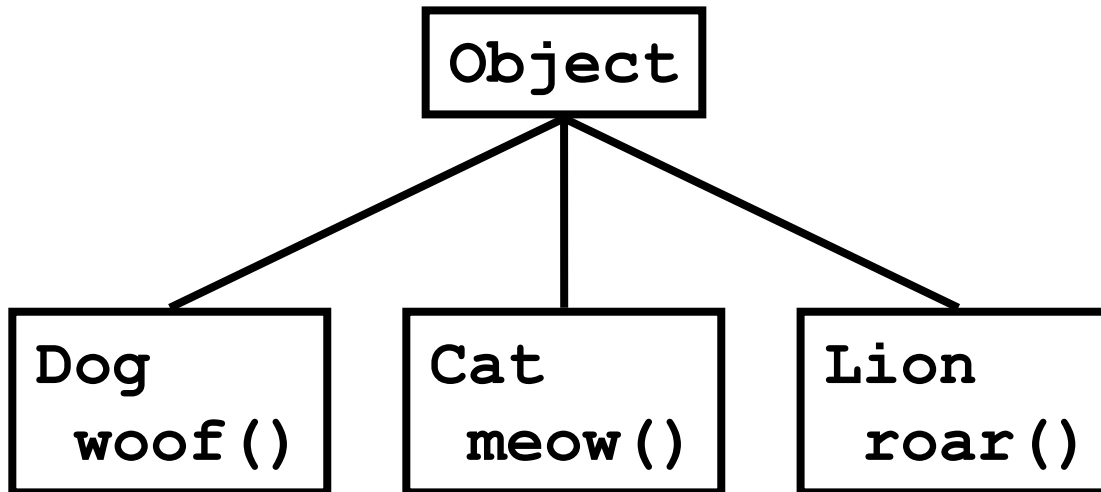
```
Object o = new Dog();  
Animal a = new Dog();  
Dog d = new Dog();  
o.toString();  
o.move();  
o.bark(); //aboyer  
a.toString();  
a.move();  
a.bark();  
d.toString();  
d.move();  
d.bark();
```



# Polymorphisme, Collections et casting

## Dynamic Binding (liaison dynamique)

- L'approche simple



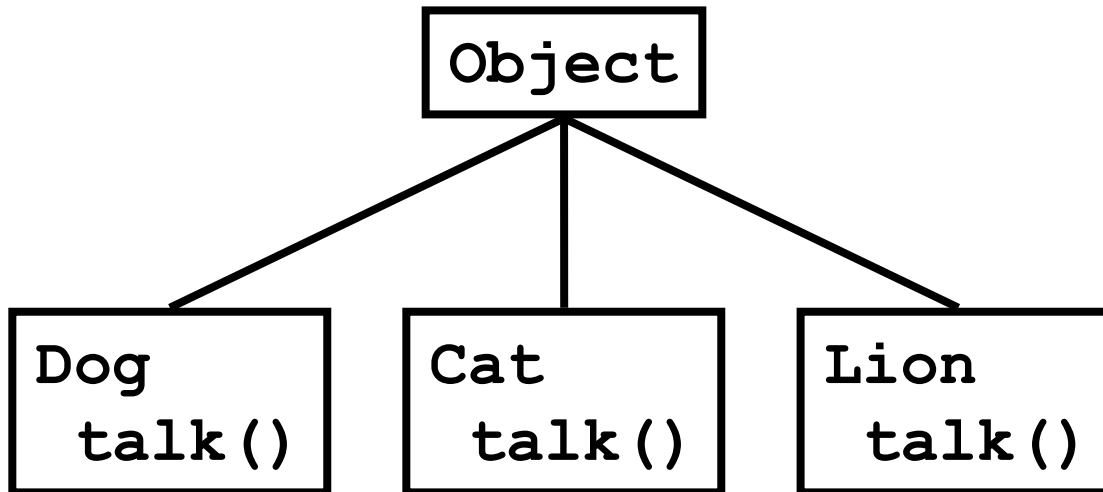
# Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();  
Object o = new Dog();  
Lion p = new Lion();  
zoo.add(o);  
zoo.add(new Cat());  
zoo.add(p);  
while(zoo.size() > 0) {  
    o = zoo.removeFirst();  
    if(o instanceof Dog)  
        ((Dog)o).bark();  
    if(o instanceof Cat)  
        ((Cat)o).meow();  
    if(o instanceof Lion)  
        ((Lion)o).roar();  
}
```

Utilisation médiocre

# Can We Do Better?

- Un premier essai



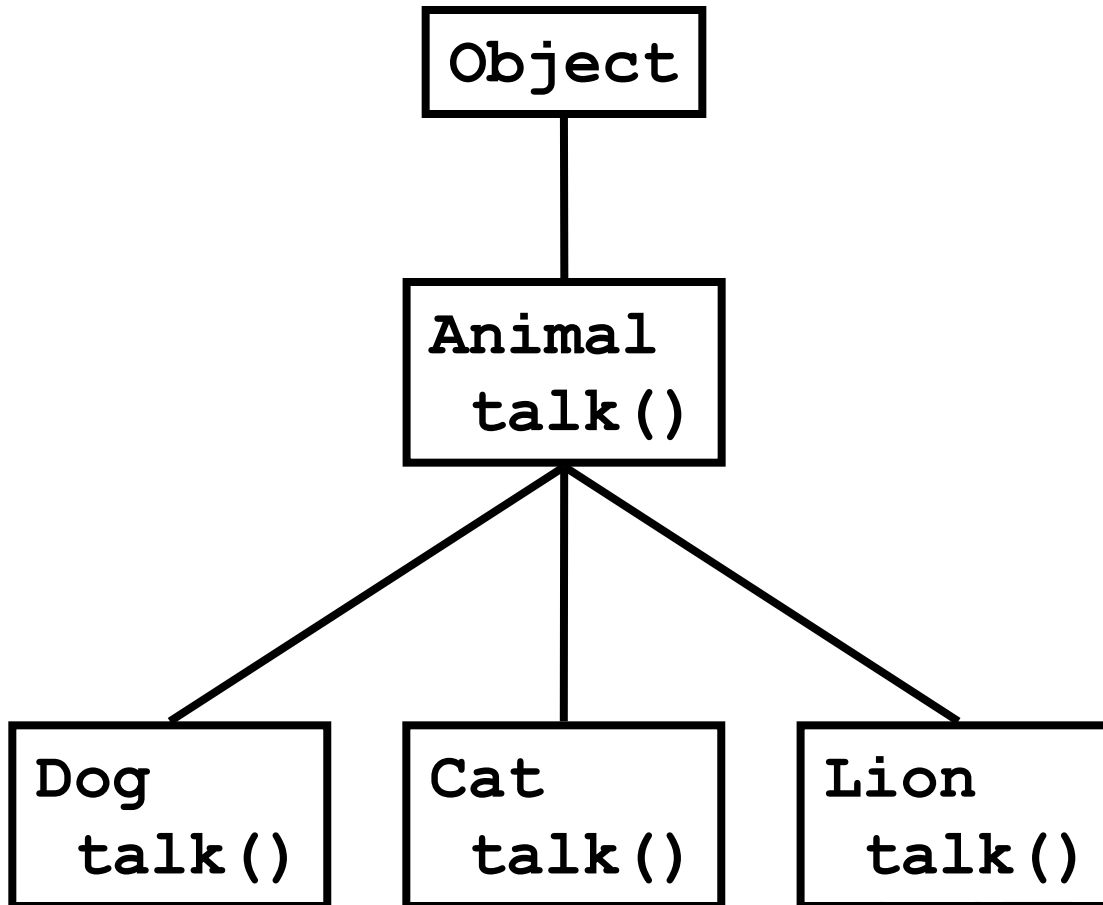
# Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();  
Object o = new Dog();  
Lion p = new Lion();  
zoo.add(o);  
zoo.add(new Cat());  
zoo.add(p);  
while(zoo.size() > 0) {  
    o = zoo.removeFirst();  
    o.talk();    // Est ce que ça fonctionne???  
}
```

# Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();  
Object o = new Dog();  
Lion p = new Lion();  
zoo.add(o);  
zoo.add(new Cat());  
zoo.add(p);  
while(zoo.size() > 0) {  
    o = zoo.removeFirst();  
    ((???)o).talk(); // Est ce que ça fonctionne??  
}
```

# Polymorphisme, Collections et casting





# Polymorphisme, Collections et casting

```
LinkedList zoo = new LinkedList();  
Animal a = new Dog();  
Object o = new Dog();  
Lion p = new Lion();  
zoo.add(a);  
zoo.add(o);  
zoo.add(new Cat());  
zoo.add(p);  
while(zoo.size() > 0) {  
    o = zoo.removeFirst();  
    ((Animal)o).talk();  
}
```

Utilisation correcte.

# Polymorphisme

- instructions **switch** ou **if...else...elseif**
  - Peut être utilisé pour manipuler de nombreux objets de types différents
    - Les actions appropriées sont basées sur (en fonction) du type
- Problèmes
  - Programmeur peut oublier d'inclure un type
  - Pourrait oublier de tester tous les cas possibles
  - Chaque ajout / suppression d'une classe exige que toutes les instructions **switch** soient modifiés
    - Le suivi de tous ces changements consomme du temps et est source d'erreurs
  - Programmation polymorphe peut éliminer le besoin de la **logique** switch
    - Evite tous ces problèmes automatiquement

# Support de présentation

<http://www.cc.gatech.edu/~bleahy/>  
<http://www.kirkwood.edu/pdf/uploaded/262/horstch7pt2.ppt>  
<http://srl.ozyegin.edu.tr/cs102/resources.php>  
<http://cs.nyu.edu/courses/spring04/G22.2110-001/java/>  
<https://sites.google.com/site/sureshdevang/java-collections>  
<https://prashantgaurav1.wordpress.com/>  
<http://jpkc.fudan.edu.cn/picture/article/80/0b36f94e-289e-42e6-b77c-a6faa6313740/206020be-4d55-4194-9f37-1ca84af55256.ppt>

# POO en Java

## Héritage et **Polymorphisme**