esprit
Se former autrement

Conception Orientée Objet & Programmation JAVA

Chapitre 5 : Polymorphisme

PLAN

- Introduction
- Classe et objet
- Encapsulation
- Héritage
- **Polymorphisme**
- Exceptions
- Interfaces
- Collection
- Interface Fonctionnelle
- Expression Lambda
- Stream

Objectifs

- Redéfinition de méthodes dans les sous-classes :

  POLYMORPHISME

- Sur-classement et substitution.

- Le transtypage (conversion de type ou cast en anglais)

Polymorphisme

Polymorphisme : Définition

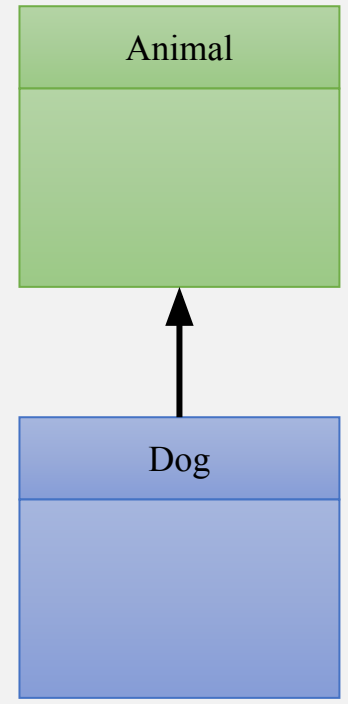
- Le polymorphisme est le fait d'instancier un objet d'une classe ***fille*** avec une référence déclarée de type de classe ***mère***.

Exemple

Animal myDog = new Dog();

Référence déclaré en tant qu'Animal().

Objet crée en tant que Dog().



Polymorphisme : Exemple

- Le polymorphisme permet au développeur d'utiliser une méthode ou un attribut selon plusieurs manières, en fonction du besoin. D'ailleurs, le mot polymorphisme est apparu dans la Grèce antique.
- Il signifie quelque chose qui peut prendre plusieurs formes.

```
class Nationality {
    public String showNationality() {
        return "";
    }
}
class Polish extends Nationality {
    public String showNationality() {
        return "Polish citizen";
    }
}
class French extends Nationality {
    public String showNationality() {
        return "French citizen";
    }
}
```

Tableau polymorphique

- Avec le polymorphisme : le type de la référence peut être la classe mère de l'objet instancié.

0 1 2

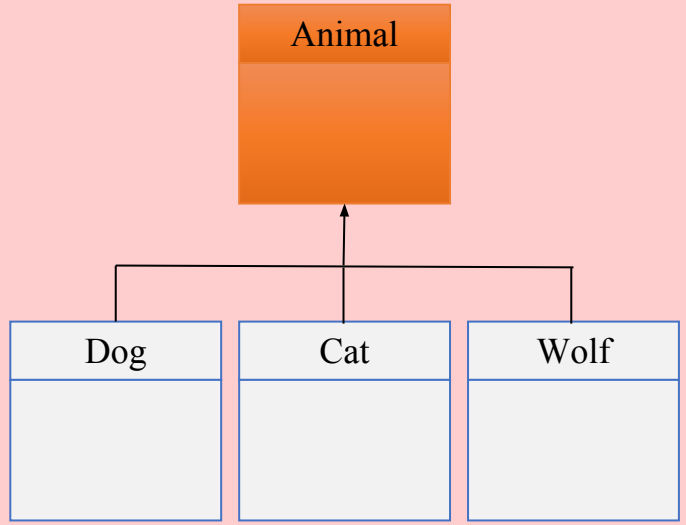
 un tableau polymorphique.

Soit : Animal [ ] animals = new Animal [3];

✔ Déclarer un tableau de type Animal.

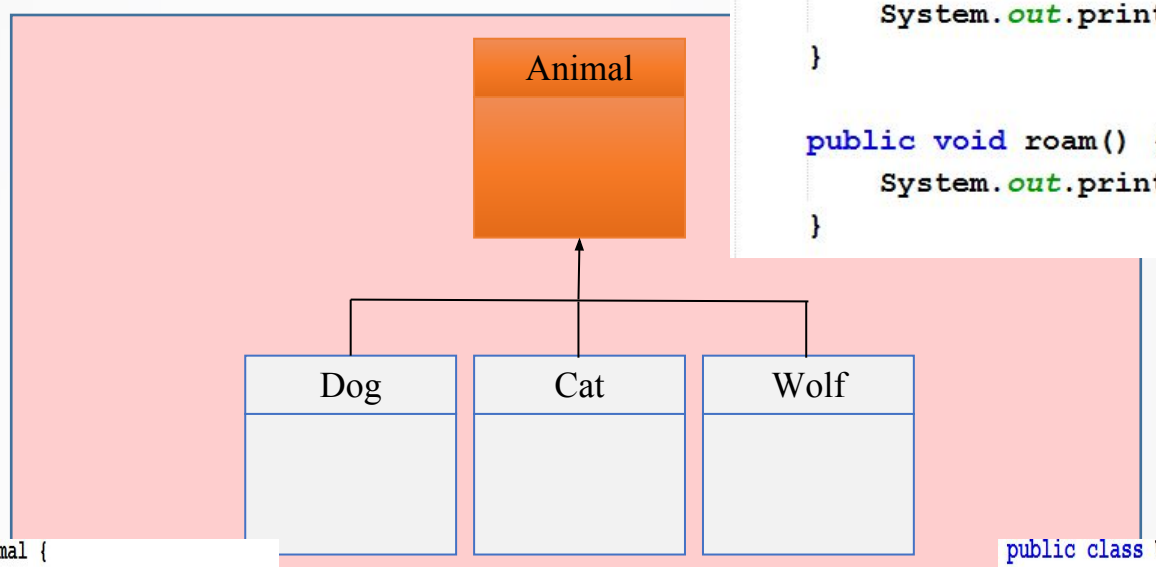
✔ Un tableau qui contiendra des objets de type Animal.

```
animals [0] = new Dog();

animals [1] = new Cat();

animals [2] = new Wolf();
```



On peut affecter les classes « filles » de la classe **Animal** dans le tableau (de type Animal).

Tableau polymorphique




```java
public class Animal {

    public void eat() {
        System.out.println("Un animal peut manger !!");
    }

    public void roam() {
        System.out.println("Un animal peut voyager !!");
    }
```

```java
public class Dog extends Animal {

    @Override
    public void eat() {

        System.out.println("Un chien peut manger !!");
    }
    @Override
    public void roam() {
        System.out.println("Un chiens peut vyager !!");
    }
}
```

```java
public class Cat extends Animal {

    @Override
    public void eat() {

        System.out.println("Un chat peut manger !!");
    }
    @Override
    public void roam() {
        System.out.println("Un chat peut vyager !!");
    }
}
```

```java
public class Wolf extends Animal {

    @Override
    public void eat() {

        System.out.println("Un loup peut manger !!");
    }
    @Override
    public void roam() {
        System.out.println("Un loup peut vyager !!");
    }
}
```

Tableau polymorphique

```
public class Test {

    public static void main(String[] args) {
        Animal[] animals = new Animal[3];
        animals[0] = new Cat();
        animals[1] = new Dog();
        animals[2] = new Wolf();

        for (int i = 0; i < animals.length; i++) {
            System.out.println("**********");
            animals[i].eat();
            animals[i].roam();
            System.out.println("**********");
        }

    }
```

```
run:
**********
Un chat peut manger !!
Un chat peut vyager !!
**********
**********
Un chien peut manger !!
Un chiens peut vyager !!
**********
**********
Un loup peut manger !!
Un loup peut vyager !!
**********
BUILD SUCCESSFUL (total time: 0 seconds)
```

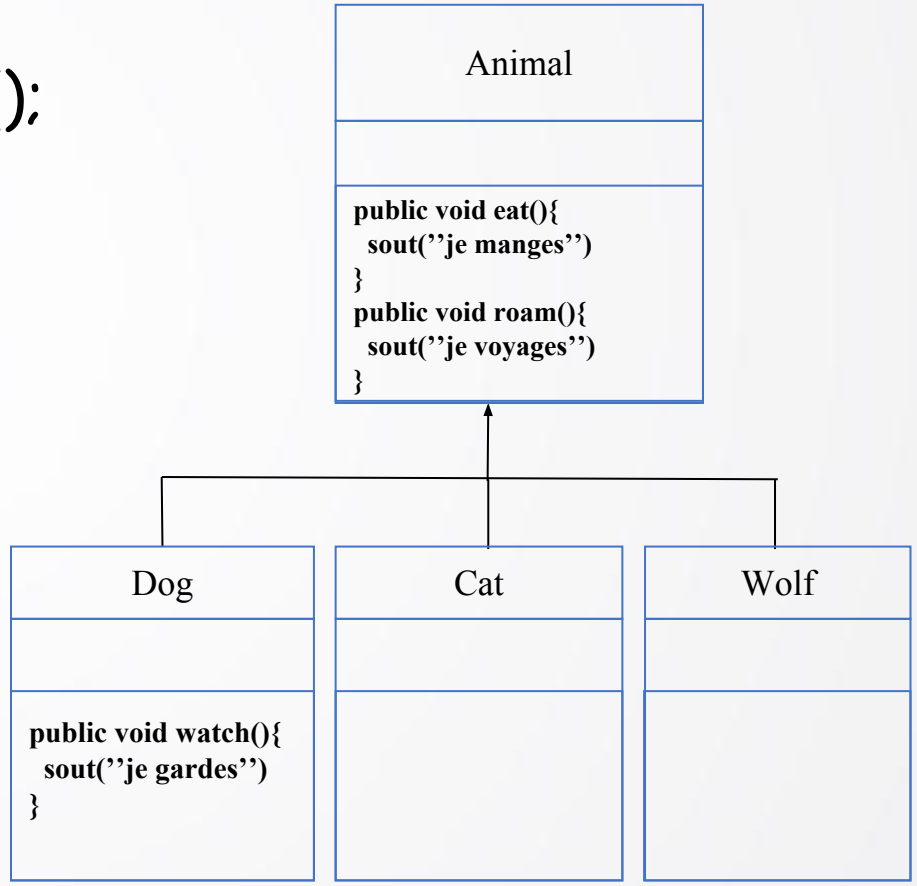
Surclassement et substitution

Surclassement et Substitution

```
Animal animal;
Dog myDog = new Dog();
animal = myDog;
```

myDog.eat()	OK
myDogl.roam()	OK
myDog.watch()	OK

animal.eat()	OK
animal.roam()	OK
animal.~~watch()~~	NO



Surclassement et Substitution

- l'objet Dog crée est « ***surclassé*** » il est vu de type Animal référence déclaré (animal).

- Les fonctionnalités de Dog sont restreintes à celles de Animal
- Le chien **ne** pourra **pas** être un chien de garde (il peut pas appelé **watch()**).

Il faut donc ***substituer*** animal par :

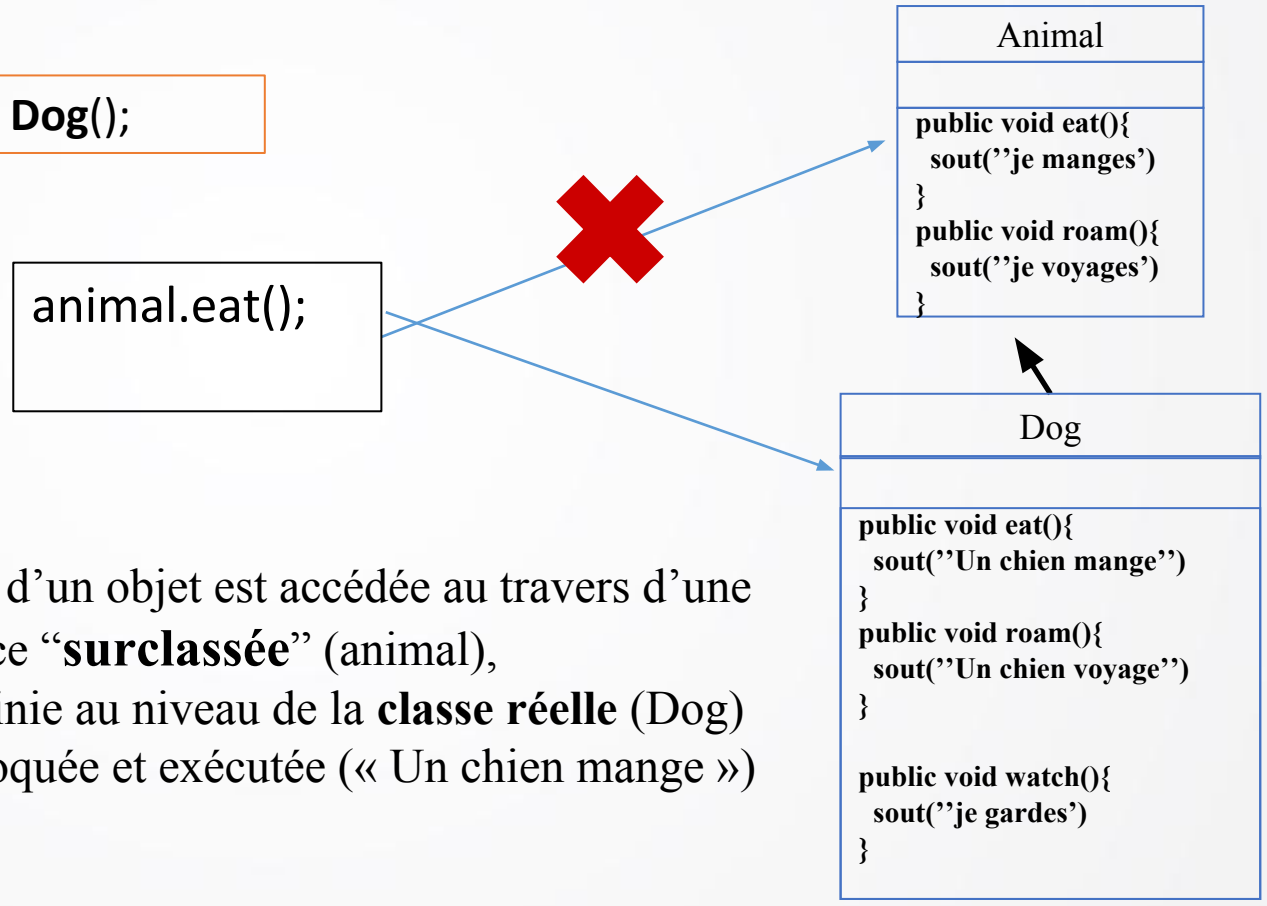
((Dog) animal).watch();

animal.eat()	**OK**
animal.roam()	OK
((Dog) animal).watch()	OK

Le polymorphisme au runtime

Animal animal = new **Dog**();

animal.eat();



Lorsqu'une méthode d'un objet est accédée au travers d'une référence "**surclassée**" (animal),
c'est la méthode définie au niveau de la **classe réelle** (Dog)
de l'objet qui est invoquée et exécutée (« Un chien mange »)

Le polymorphisme au runtime

le choix du code à exécuter (pour une méthode polymorphe) ne se fait pas **statiquement à la compilation mais dynamiquement à l'exécution.**

Le polymorphisme qui est fondamental en programmation OO est rendu possible par le fait que les messages sont résolus dynamiquement (**message binding**).

Le polymorphisme au runtime

- Quand vous appelez une méthode d'un objet référencé, vous appelez au fait la méthode la plus spécifique du type de cet objet.

- **Le type le plus *inférieur* gagne !**

(inférieur : dans l'arbre d'héritage.)

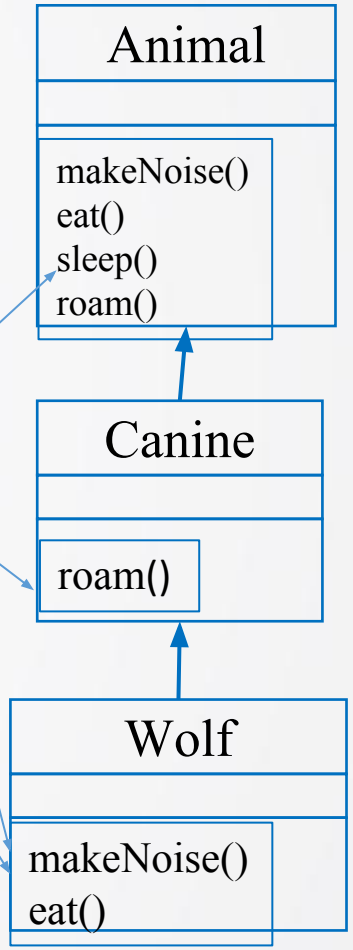
Animal w = new **Wolf();**

w.makeNoise();

w.roam();

w.eat();

w.sleep();



- La machine virtuelle *JVM* commence tout d'abord à voir dans la classe *Wolf* . Si elle ne trouve pas une correspondance de la version de la méthode,
- elle commence à grimper l'hiérarchie de l'héritage jusqu'à trouver la bonne méthode.

Upcast & downcast

Upcast

- Permet de convertir le type d'une référence vers un type parent.

```
1 Object obj1 = new String(); // UpCasting implicite
2
3 Object obj2 = (Object) new String(); // UpCasting explicite
```

- La conversion (implicite ou explicite) vers un type parent est toujours acceptée par le compilateur et elle ne posera aucun problème à l'exécution du programme
- une référence d'un type parent peut, sans risque, lire/modifier les attributs ou invoquer les méthodes dont elle a accès, indépendamment si l'instance référée et du même type ou un sous-type du type de la référence.

Downcast

- Permet de convertir le type d'une référence vers un sous type.

```
String str1 = new Object(); // DownCasting implicite (erreur de compilation : Type mismatch: cannot convert from Object to String)
```

```
String str2 = (String) new Object(); // DownCasting explicite (erreur à l'exécution : java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.String)
```

- La conversion implicite *(String str1 = new Object();)* vers un sous type est toujours refusée par le compilateur.

- Par contre, la conversion explicite *(String str2 = (String) new Object();)* vers un sous type est toujours acceptée par le compilateur. Cependant à l'exécution du programme, la JVM va vérifier si le type de l'instance (Object) est le même ou un sous type du type qu'on a spécifiée pour la conversion (String) : si ce n'est pas le cas, la JVM déclenche une exception.

instanceof: Définition

- L'opérateur **instanceof** (également appelé opérateur de comparaison de type) permet de vérifier si l'objet est une instance du type spécifié (classe ou sous-classe ou interface).

- Il retourne :
  - **true** - si la variable est une instance de la classe spécifiée, ou de la classe parente
  - **false** - si la variable n'est pas une instance de la classe ou la variable est nulle

instanceof: Exemple

```
public class B {
}
public class A {
}

public class Test {

public static void main(String... ar) {

        A ob= new A();
        System.out.println(ob instanceof A); //checking if ob is an object of A class
        System.out.println(ob instanceof B); //checking if ob is an object of B class.
   }

}
```

Output

```
true
false
```

Downcasting: exemple

```
Animal a = new Dog();
Dog d = (Dog) a;
d.watch()
```

- L'objet **a** est de type **Animal**
- Il pointe vers un objet de type **Dog**, donc la conversion (cast) dans ce cas là est permise
- Lors de l'exécution, et après avoir effectué la conversion, l'objet **d** exécute la méthode watch()

Downcasting: cas d'erreur

- L'upcasting est toujours permis, par contre le downcasting implique une vérification de type et peut générer une exception de type « **ClassCastException** »

```
Animal a = new Animal();
Dog d = (Dog) a;
d.watch()
```

- Cet exemple nous génère une erreur d'exécution (**ClassCastException**), parce que le type de la variable a est Animal, et au moment de la conversion (cast), il s'avère que la variable a ne pointe pas, vraiment, vers un objet de type Dog!

Downcasting: cas d'erreur (solution)

- Pour remédier à ce problème, on peut utiliser l'opérateur **instanceof** avant de faire l'opération du cast, afin de tester le type de l'objet à convertir.

```
Animal a = new Animal();

    If (a instanceof Dog){
        Dog d = (Dog) a;
        d.watch();
    }
    else
    {
        System.out.println(''Conversion de type impossible!'');
    }
```

Transtypage : Définition

- L'opérateur de transtypage (opérateur de "cast") permet de modifier explicitement le type d'une valeur avant de l'affecter à une variable ou de l'utiliser.
- On écrit entre parenthèses le nom du nouveau type voulu, suivi de la valeur à transtyper, du nom de la variable contenant cette valeur, de l'opération fournissant cette valeur...

```
package initial;

public class Transtyp1 {

        public static void main(String[] args) {
                ConsoleTexte mc=new ConsoleTexte();
                mc.ecritln(12.5/3);
                mc.ecritln((int)12.5/3);

        }

}
```

Transtypage : Quand?

- **S'il risque d'y avoir perte d'information**

Par exemple lorsqu'on fait l'affectation d'un entier long (64 bits) à un entier int (32 bits), ou d'un réel double (Virgule flottante double précision) vers un réel float. On force alors le type, indiquant ainsi à la machine Java qu'on est conscient du problème de risque de perte d'information. (La première variable peut contenir plus d'information que la seconde).

Transtypage : Quand?

- **Si on veut prendre la partie entière d'un réel (ou double)**

Par exemple si on utilise la méthode random() de la class Math cette fonction retourne par défaut un double entre 0 et 1,alors qu'on a besoin d'un int. Par exemple, dans ce cas on peut procéder comme suite :

```
int a = (int) Math.random()*100 ; //on ne prend que la partie entière
```

Cacher une méthode static

- On ne redéfinit pas une méthode static, on la cache (comme les variables)

- Si la méthode static m de Classe1 est cachée par une méthode m d'une classe fille, la différence est que :

  – On peut désigner la méthode cachée de Classe1 en préfixant par le nom de la classe : Classe1.m()

  – ou par un cast(x est une instance d'une classe fille de Classe1 : ((Classe1)x).m()

  - mais on ne peut pas la désigner en la préfixant par « super. »

Final : classe/méthode

- **Classe final** : ne peut pas avoir de classes filles

```
public final class Vehicule {

    void m(){

    }
}
```

```
public class Voiture extends Vehicule{

}
```

- ***Méthode final :*** *ne peut pas être redéfinie*

```
public class Vehicule {

  public final void  m(){

    }
}
```

```
public class Voiture extends Vehicule{

   public void m(){
    }
}
```

La méthode equals

Définition

- La méthode **equals** permet à Java de comparer deux objets et de déterminer l'égalité entre ces deux objets, ce qui parait logique.

- Par défaut, la méthode retourne vrai s'il s'agit du même objet en mémoire. Par conséquent, si deux objets distincts instancié à partir de la classe Personne, par exemple, ont tout les deux le même id, nom, prénom et date de naissance, la méthode ne retourne pas vrai mais faux, car il s'agit de **deux objets distincts en mémoire**, c'est pourquoi il faut parfois **la surcharger**.

Exemple 1

```
@Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }

        if (o instanceof Personne) {
            return (this.getId() == ((Personne) o).getId());
        } else {
            return false;
        }
    }
```

Exemple 2

```
public class Animal
{
String name;

 @Override
   public boolean equals(Object obj) {
      if (obj == null) {
         return false;
      }
      if (getClass() != obj.getClass()) { // if(!(obj instanceof Animal))
         return false;
      }
      final Animal animal = (Animal) obj;

      if (this.name != animal.name) {
         return false;
      }
      return true;
   }
}
```

Merci pour votre attention