TP4 JAVA – Programmation Orientée Objets

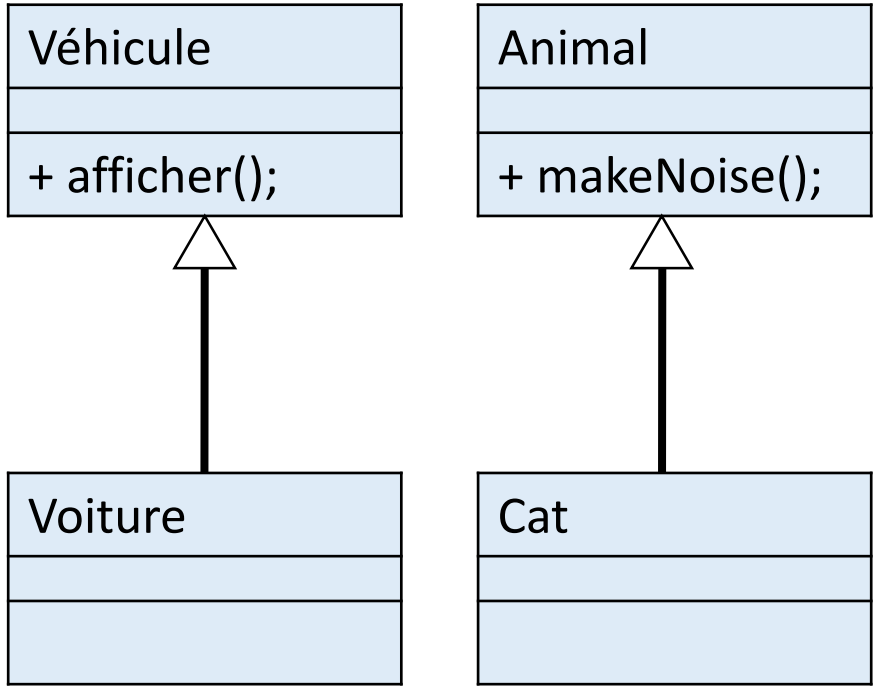
Rafael COLARES

Maitre de Conférences ISIMA

Bureau D104 – email:rafael.colares_borges@uca.fr

Leçons apprises du TP3

- Héritage
  - Modélisation des relations de type « est un »
    - Une `Voiture` est un `Véhicule`
    - Un `Chat` est un `Animal`



- La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe

```
Cat cat = new Cat();
cat.makeNoise();
```

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»;
        }
        public Vehicule(String immat) {
                this.immat = immat;
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {}
}
```

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»;
        }
        public Vehicule(String immat) {
                this.immat = immat;
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {}
}
```

```
public class App {
        public static void main(String[] args) {
                Vehicule v = new Vehicle();
                Vehicule v2 = new Vehicle(«ZZ2»);

                Voiture v3 = new Voiture(«ZZ2»);
                Voiture v4 = new Voiture();
}
```

v.immat = ?

v2.immat = ?

v3.immat = ?

v4.immat = ?

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»;
        }
        public Vehicule(String immat) {
                this.immat = immat;
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {}
}
```

```
public class App {
        public static void main(String[] args) {
                Vehicule v = new Vehicle();
                Vehicule v2 = new Vehicle(«ZZ2»);

                Voiture v3 = new Voiture(«ZZ2»);
                Voiture v4 = new Voiture();
}
```

v.immat = «Default»

v2.immat = ?

v3.immat = ?

v4.immat = ?

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»;
        }
        public Vehicule(String immat) {
                this.immat = immat;
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {}
}
```

```
public class App {
        public static void main(String[] args) {
                Vehicule v = new Vehicle();
                Vehicule v2 = new Vehicle(«ZZ2»);

                Voiture v3 = new Voiture(«ZZ2»);
                Voiture v4 = new Voiture();
}
```

v.immat = «Default»

v2.immat = « ZZ2 »

v3.immat = ?

v4.immat = ?

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»;
        }
        public Vehicule(String immat) {
                this.immat = immat;
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {}
}
```

```
public class App {
        public static void main(String[] args) {
                Vehicule v = new Vehicle();
                Vehicule v2 = new Vehicle(«ZZ2»);

                Voiture v3 = new Voiture(«ZZ2»);
                Voiture v4 = new Voiture();
}
```

v.immat = «Default»

v2.immat = « ZZ2 »

v3.immat = «Default»

v4.immat = ?

Dans tous les constructeurs de Voiture:

- Appel implicite de super() avant toute autre instruction

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»;
        }
        public Vehicule(String immat) {
                this.immat = immat;
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {
                super(immat);
        }
}
```

```
public class App {
        public static void main(String[] args) {
                Vehicule v = new Vehicle();
                Vehicule v2 = new Vehicle(«ZZ2»);

                Voiture v3 = new Voiture(«ZZ2»);
                Voiture v4 = new Voiture();
}
```

v.immat = «Default»

v2.immat = « ZZ2 »

v3.immat = «Default»

v4.immat = ?

Dans tous les constructeurs de Voiture:
- Appel implicite de super() avant toute autre instruction
- **Sauf** si appel explicite à un constructeur de la super-classe

Leçons apprises du TP3

- Héritage
  - La sous-classe a accès direct à tous les attributs et méthodes **protégés** et **publics** de la super-classe **sauf** les constructeurs

Vehicule.java

```
public class Vehicule {
        private String immat;
        public Vehicule() {
                this.immat = «Default»
        }
        public Vehicule(String immat) {
                this.immat = imm
        }
}
```

Voiture.java

```
public class Voiture extends Vehicule{
        public Voiture(String immat) {}
}
```

```
public class App {
        public static void main(String[] args) {
                Vehicule v = new Vehicle();
                Vehicule v2 = new Vehicle(«ZZ2»);

                Voiture v3 = new Voiture(«ZZ2»);
                Voiture v4 = new Voiture();
```

v.immat = «Default»

v2.immat = « ZZ2 »

v3.immat = «Default»

error: constructor Voiture in class Voiture cannot be applied to given types;

- Voiture n'a pas de constructeur sans argument
- Le constructeur sans argument de Vehicule n'est pas hérité!

Leçons apprises du TP3

- Polymorphisme

In Biology:
“Polymorphism is when there are two or more possibilities of a trait on a gene.” (Wikipedia)

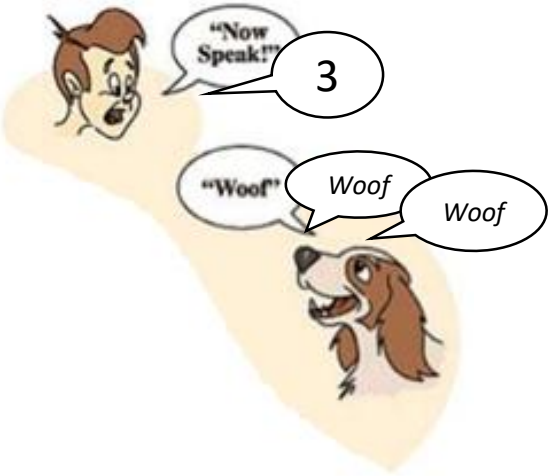
In Computer Science ?

Leçons apprises du TP3

- Polymorphisme

Plusieurs Formes

In Biology:
"Polymorphism is when there are two or more possibilities of a trait on a gene." (Wikipedia)



In Computer Science (Java):
Capacité d'exécuter différents comportements à partir d'une même instruction

- Selon l'objet qui l'exécute
- Selon ce qu'on donne à ces objets

Leçons apprises du TP3

- Polymorphisme
  - 2 types:

Leçons apprises du TP3

- Polymorphisme
  - 2 types:

Surcharge de méthode

(*Overloading*)

Redéfinition

(*Overriding*)

Leçons apprises du TP3

- Polymorphisme faible : overloading
  - Méthodes avec signatures différentes: même nom, arguments différents

```
public class ZZ2 {
	private int javaKnowledge;
	public void study(int nbHoursOfCoding) {
		javaKnowledge += nbHoursOfCoding;
	}
	public void study(Book book) {
		javaKnowledge += book.getKnowledge();
	}
}
```

Leçons apprises du TP3

- Polymorphisme faible : overloading
  - Méthodes avec signatures différentes

```
public class ZZ2 {
        private int javaKnowledge;
        public void study(int nbHoursOfCoding) {
                javaKnowledge += nbHoursOfCoding;
        }
        public void study(Book book) {
                study(book.getKnowledge());
        }
}
```

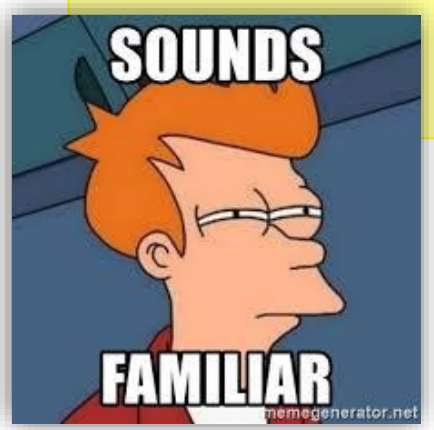
Méthodes différentes ➔ Pas de récursivité!

  - Interprétation à la compilation (polymorphisme statique)

Leçons apprises du TP3

- Polymorphisme faible : overloading
  - Méthodes avec signatures différentes

```
public class ZZ2 {
        private int javaKnowledge;
        public void study(int nbHoursOfCoding) {
                javaKnowledge += nbHoursOfCoding;
        }
        public void study(Book book) {
                study(book.getKnowledge());
```

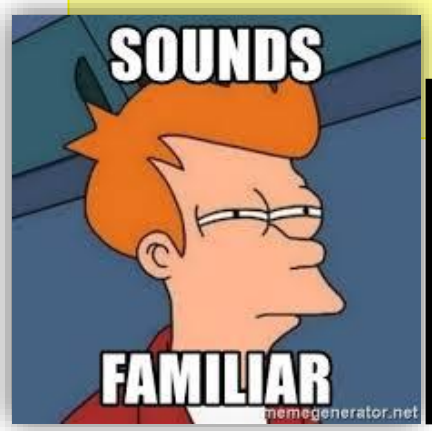


Même chose, plusieurs formes ➔ POLYMORPHISME

Leçons apprises du TP3

- Polymorphisme faible : overloading
  - Méthodes avec signatures différentes

```
public class ZZ2 {
        private int javaKnowledge;
        public void study(int nbHoursOfCoding) {
                javaKnowledge += nbHoursOfCoding;
        }
        public void study(Book book) {
                study(book.getKnowledge());
```




```
public ZZ2(int javaKnowledge) {
        this.javaKnowledge = javaKnowledge;
}
public ZZ2() {
        this();
}
```

Leçons apprises du TP3

- Polymorphisme fort : overriding
  - Méthodes avec signatures identiques

Animal.java

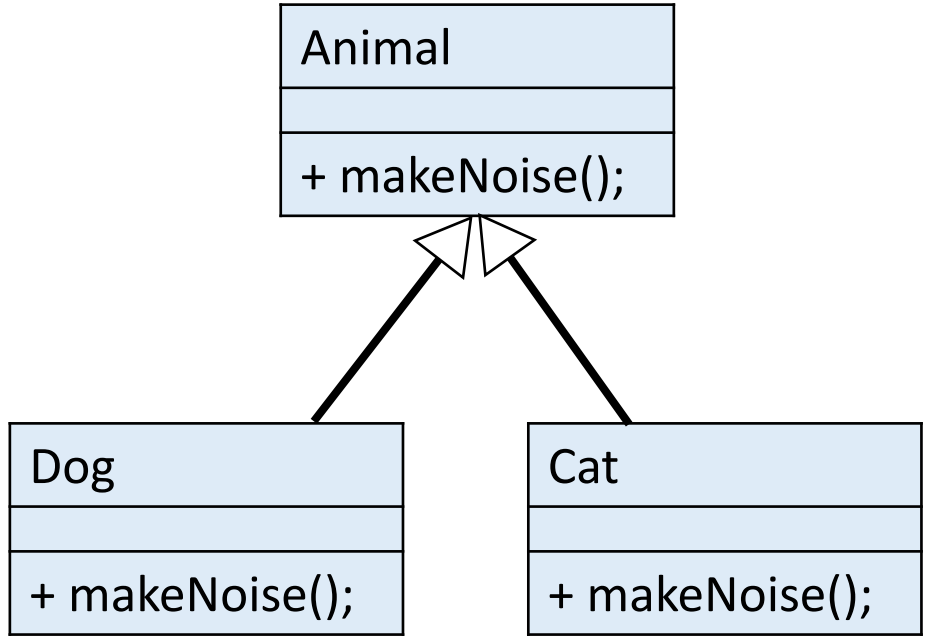
```
public class Animal {
        public void makeNoise() {
                System.out.println(« ... »);
        }
}
```

Dog.java

```
public class Dog extends Animal{
        public void makeNoise() {
                System.out.println(« Woof woof »);
        }
}
```

Cat.java

```
public class Cat extends Animal{
        public void makeNoise() {
                System.out.println(« Meow »);
        }
}
```



Même chose, plusieurs formes ➔ POLYMORPHISME

Leçons apprises du TP3

- Polymorphisme fort : overriding
  - Méthodes avec signatures identiques

Animal.java

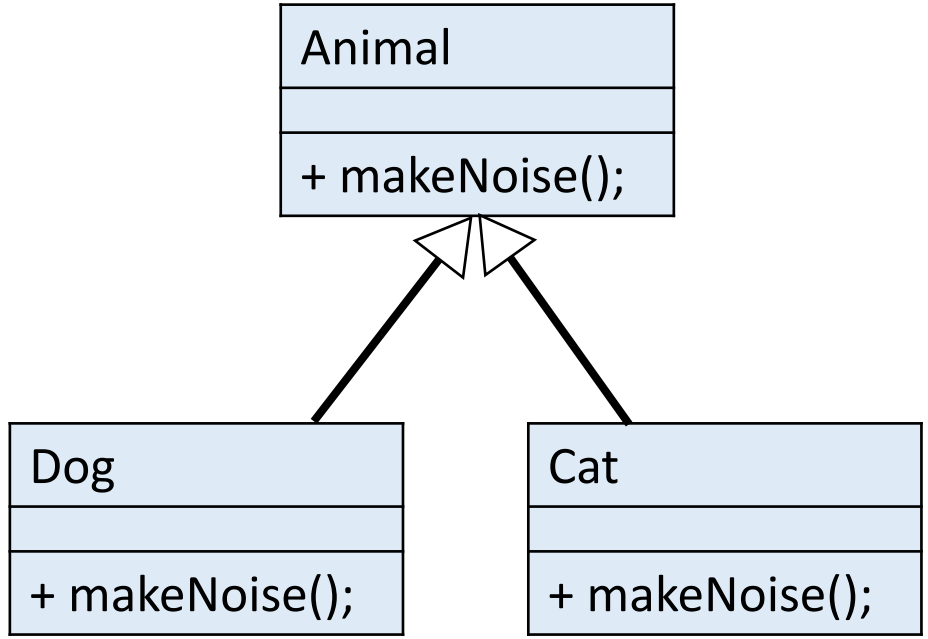
```
public class Animal {
        public void makeNoise() {
                System.out.println(« ... »);
        }
}
```

Dog.java

```
public class Dog extends Animal{
        @Override
        public void makeNoise() {
                System.out.println(« Woof woof »);
        }
}
```

Cat.java

```
public class Cat extends Animal{
        @Override
        public void makeNoise() {
                System.out.println(« Woof woof »);
        }
}
```



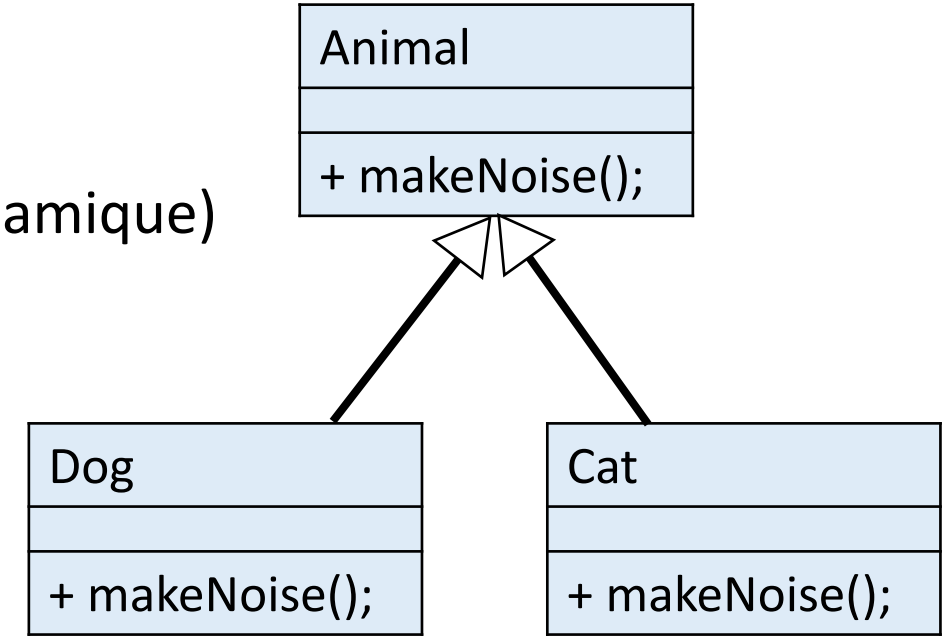
BONNE PRATIQUE : utilisation de *@Override*

- Prévient le compilateur de la présence de polymorphisme fort

Leçons apprises du TP3

- Polymorphisme fort : overriding
  - Interprétation à l'exécution (polymorphisme dynamique)

```
public class App {
	public static void main(String[] args) {
		Animal animal = new Animal();
		animal.makeNoise();
		Dog dog = new Dog();
		dog.makeNoise();

		Cat cat = new Animal();
		cat.makeNoise();
		Animal felin = new Cat();
		felin.makeNoise();
	}
}
```

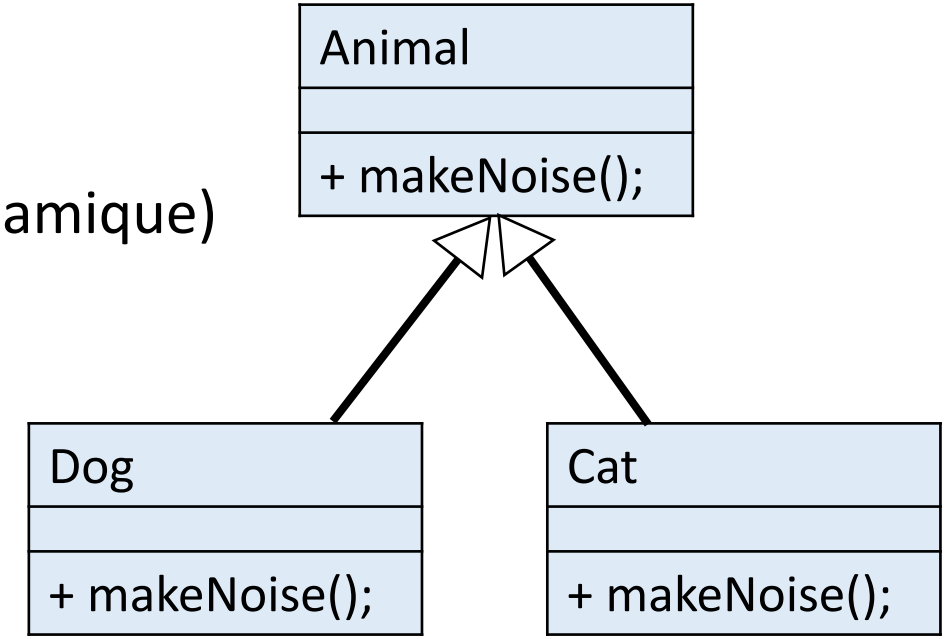


Leçons apprises du TP3

- Polymorphisme fort : overriding
  - Interprétation à l'exécution (polymorphisme dynamique)

```
public class App {
	public static void main(String[] args) {
		Animal animal = new Animal();
		animal.makeNoise();
		Dog dog = new Dog();
		dog.makeNoise();

		Cat cat = new Animal();
		cat.makeNoise();
		Animal felin = new Cat();
		felin.makeNoise();
	}
}
```

…



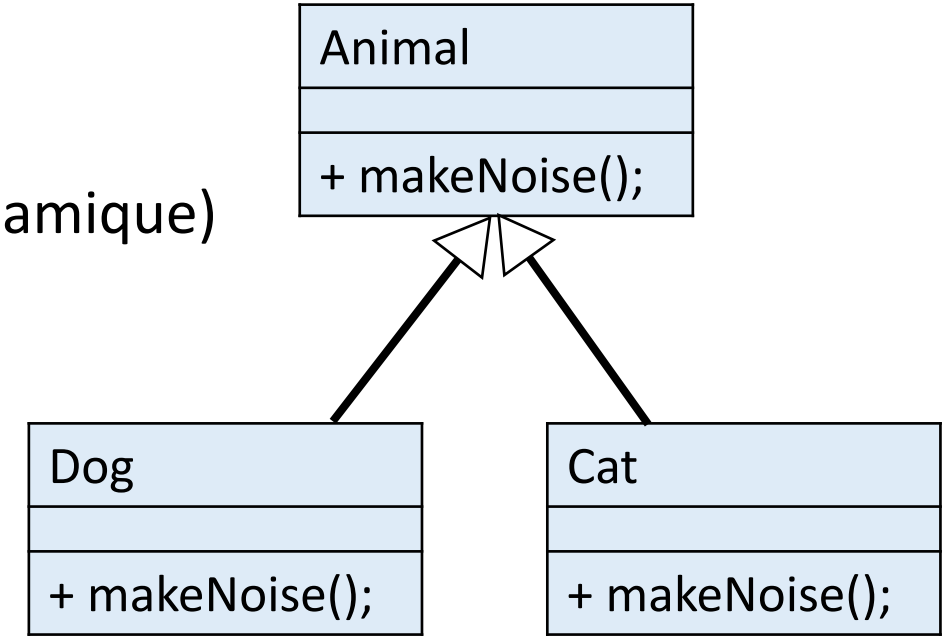
Leçons apprises du TP3

- Polymorphisme fort : overriding
  - Interprétation à l'exécution (polymorphisme dynamique)

```
public class App {
        public static void main(String[] args) {
                Animal animal = new Animal();
                animal.makeNoise();
                Dog dog = new Dog();
                dog.makeNoise();

                Cat cat = new Animal();
                cat.makeNoise();
                Animal felin = new Cat();
                felin.makeNoise();
        }
}
```

…

Woof woof



Leçons apprises du TP3

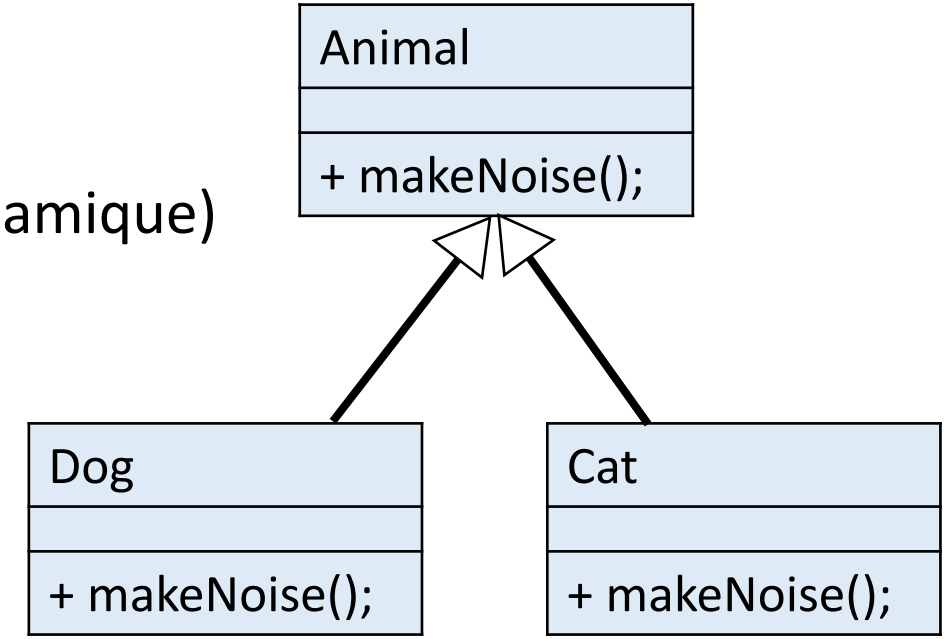
- Polymorphisme fort : overriding
  - Interprétation à l'exécution (polymorphisme dynamique)

```
public class App {
        public static void main(String[] args) {
                Animal animal = new Animal();
                animal.makeNoise();
                Dog dog = new Dog();
                dog.makeNoise();

                Cat cat = new Animal();
                cat.makeNoise();
                Animal felin = new Cat();
                felin.makeNoise();
        }
}
```

…

Woof woof

error: incompatible types: Animal cannot be converted to Cat



Leçons apprises du TP3

- Polymorphisme fort : overriding
  - Interprétation à l'exécution (polymorphisme dynamique)

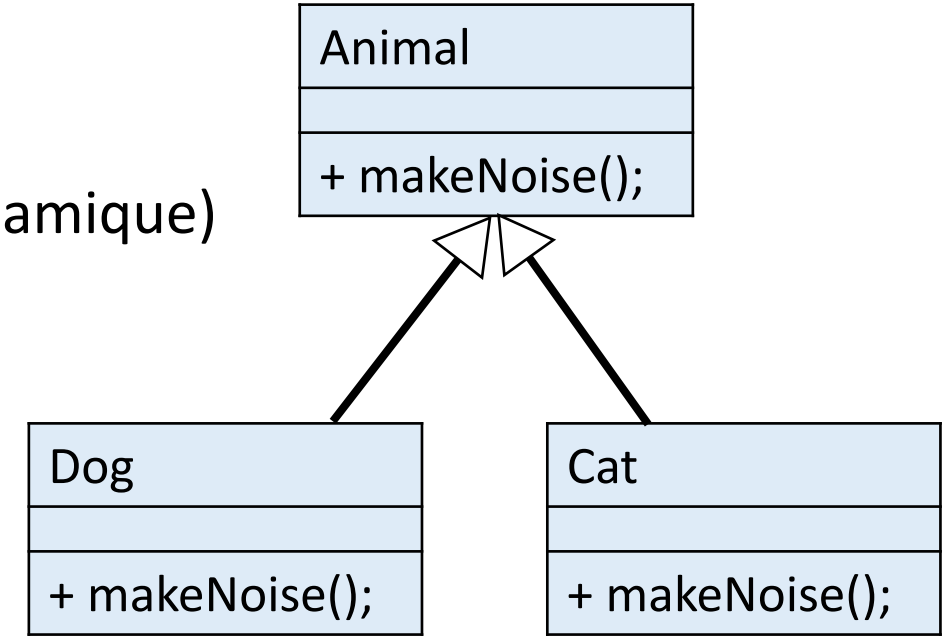
```
public class App {
        public static void main(String[] args) {
                Animal animal = new Animal();
                animal.makeNoise();
                Dog dog = new Dog();
                dog.makeNoise();

                Cat cat = new Animal();
                cat.makeNoise();
                Animal felin = new Cat();
                felin.makeNoise();
        }
}
```

…

Woof woof

error: incompatible types: Animal cannot be converted to Cat

Meow



Leçons apprises du TP3

- La classe Object
  - La racine de toute arborescence d'héritage en Java est la classe Object
  - Fournit des méthodes communes à toutes les classes
    - Important de connaître ces méthodes

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

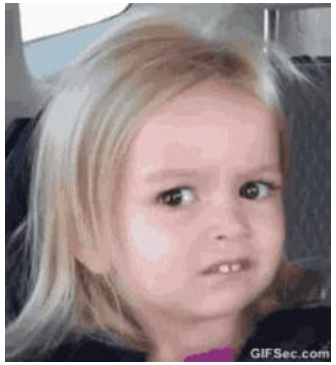
  - l'opérateur **==** sert à comparer des références

Leçons apprises du TP3

- La classe Object
  - La racine de toute arborescence d'héritage en Java est la classe Object
  - Fournit des méthodes communes à toutes les classes
    - Important de connaître ces méthodes

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

  - l'opérateur **==** sert à comparer des références
  - Pour comparer des objets en Java, il ne faut donc **jamais** utiliser ==

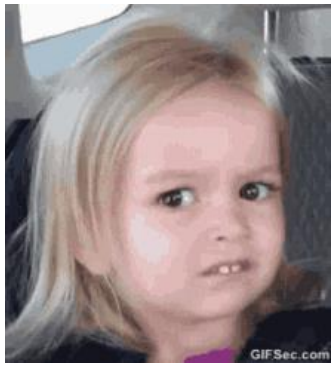


Leçons apprises du TP3

- La classe Object
  - La racine de toute arborescence d'héritage en Java est la classe Object
  - Fournit des méthodes communes à toutes les classes
    - Important de connaître ces méthodes

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

  - l'opérateur **==** sert à comparer des références
  - Pour comparer des objets en Java, il ne faut donc **jamais** utiliser ==



N'importe quelle classe héritant de la classe `Object` peut redéfinir le comportement de la méthode *equals*
- C'est typiquement le cas pour la classe `String`.

Leçons apprises du TP3

- Les tests unitaires
  - Importance +++ pour votre vie professionnelle de développeur
  - Permet de s’assurer que votre code fonctionne comme prévu

TP 4

- Contenu:
  - ✓ Classes abstraites
  - ✓ Interfaces

- Aller voir *https://perso.isima.fr/loic/java/tp_04.php*
  - Après avoir redéfini la méthode *toString* dans la section 2, essayez

```
Voiture v = new Voiture(«ISI063_C»);
System.out.println(v);
```