INFO-H-200
Programmation orientée objet

Séance d'exercices 4
Classes abstraites et polymorphisme

Université libre de Bruxelles
École polytechnique de Bruxelles

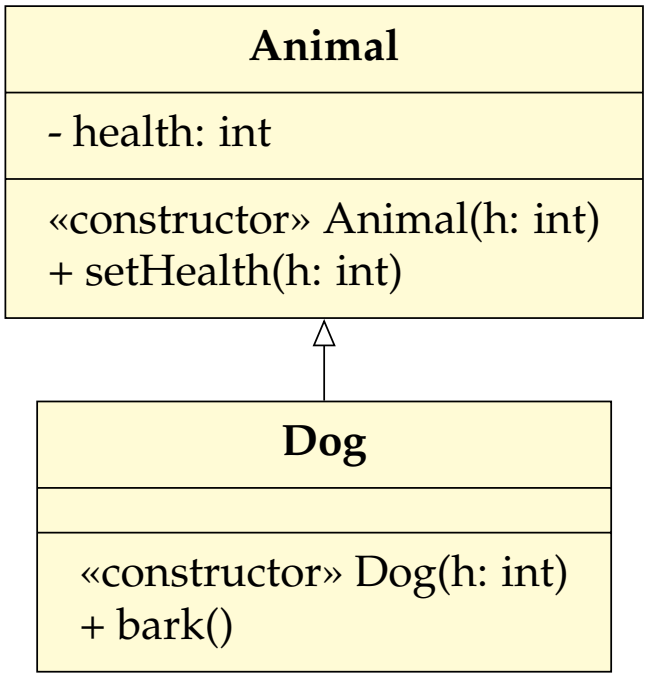
Professeur : Hugues Bersini

2017-2018

Héritage

L'héritage permet la réutiliation de code. Ce mécanisme permet de développer de nouvelles classes à partir de classes existances en étandant celles-ci.

On appele la classe existante "mère" alors que la nouvelles, étandant la classe mère est appellée la classe "fille".




```
Dog medor = new Dog(5);
medor.setHealth(10);
```

Héritage

- Une classe ne peut pas hériter de plusieurs classes.
- Les méthodes que l'on veut redéfinir dans la classe fille doivent être réécrites ainsi que ses méthodes et attributs complémentaires.
- Si vous redéfinissez une méthode de la classe mère dans la classe fille, celle-ci ne peut voir sa visibilité restreinte. Ce principe d'héritage n'est pas limité à un seul niveau, mère - fille, mais peut s'étendre à plusieurs niveaux.
- Il est possible d'empêcher l'héritage d'une classe, dans ce cas, elle doit être déclarée "final".
- On peut empêcher la redéfinitnion d'un méthode dans une sous classe en la spécifiant "final".

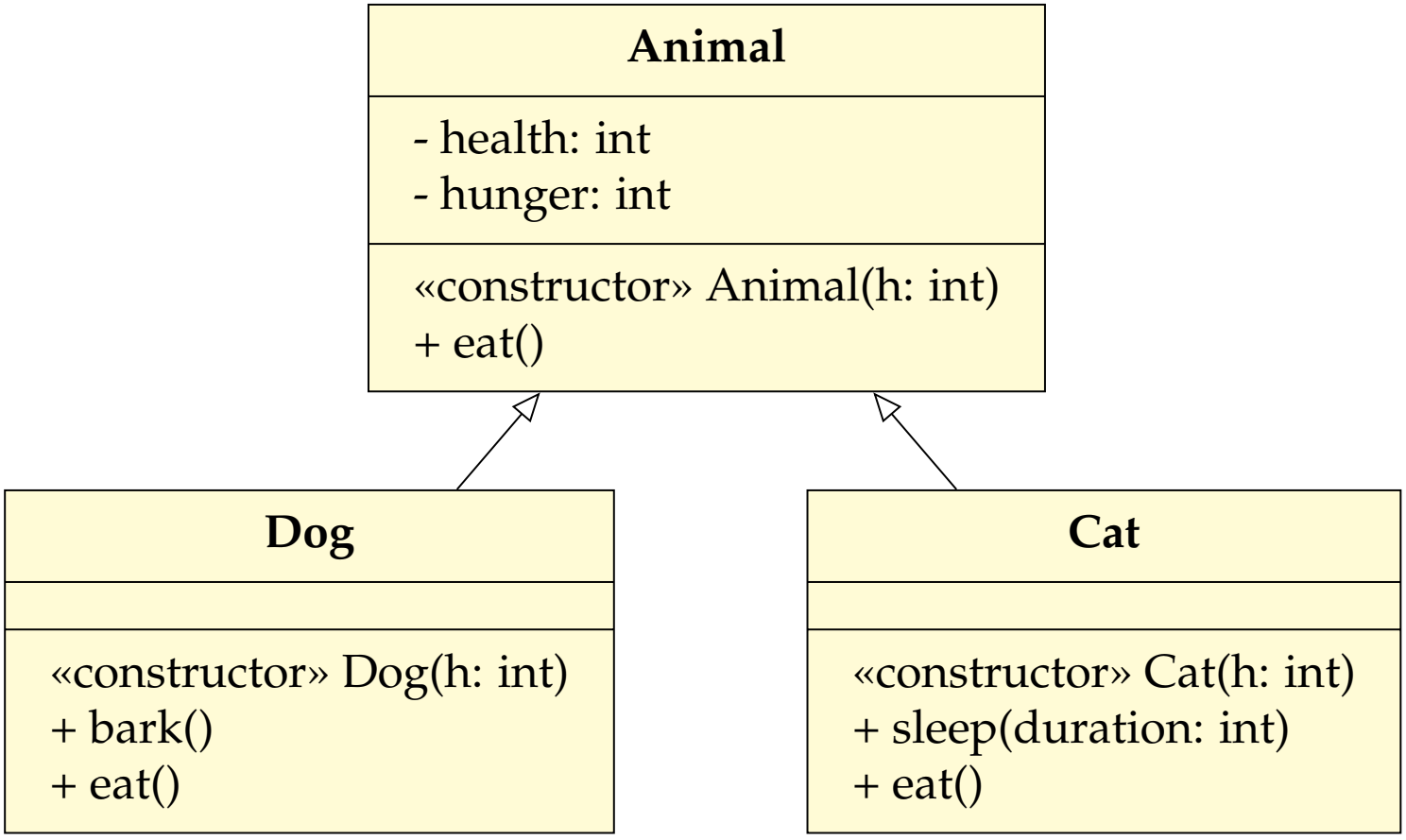
Une classe fille possède donc l'ensemble des propriétés de la classe mère ainsi que ses méthodes.

Héritage

- super() : Fonctionne selon le même principe que this mais fait référence à la classe mère.

```
public class Animal {
    private int age;

    public Animal(int age){
        this.age = age;
    }
}

public class Dog extends Animal{
    String name;

    public Dog(int age, String name) {
        super(age);
        this.name = name;
    }

}
```

Héritage



- + : public
- − : private
- # : protected

Transtypage

En Java, un objet peut être *casté* en un autre objet (typecasting ou transtypage):

- soit en un objet d'une sous classe (downcasting)
- soit en un objet d'une super-classe (upcasting)

Une erreur de compilation aura lieu dans le cas d'un transtypage impossible (entre objets non liés par héritage)

Une exception sera déclenchée lors de l'éxécution si l'objet *casté* n'est pas compatible avec le type dans lequel il a été *casté*.

L'upcasting est automatique. Le downcasting nécessite un opérateur :

```
Animal a = new Cat(...);
Cat c = (Cat) a;
```

Transtypage

Attention ! "Caster" n'implique pas de changer l'objet ! On l'identifie juste différemment. Un objet Cat upcasté en Animal restera un Chat. Néanmoins, votre chat sera désormais traité comme n'importe quel animal. Les attributs et méthodes spécifiques au Cat seront donc cachées juste qu'à ce que l'on fasse un downcasting.

```
Animal a = new Cat(...);
a.eat()      // 'a' est un Cat -> on exécute la
             // méthode ecrite dans la classe Cat

a.sleep()    // Erreur : 'a' est considéré
             // par le compilateur comme un animal
```

Classes abstraites

Une **classe abstraite** ne peut être instanciée mais peut être sous-classée. Elle peut contenir ou non des méthodes abstraites.

Une **méthode abstraite** est une méthode déclarée mais non implémentée.

Si une classe inclut des méthodes abstraites, elle doit être abstraite.

Une classe qui hérite d'une classe abstraite doit implémenter toutes les méthodes abstraites de la classe parente. Sinon, elle doit être déclarée abstraite.

```
public abstract class Animal {
    private int hunger;

    ...
    public void eat();
        // hunger = 0;
}
```

Utilité ? Créer un modèle de classe ! Par exemple, la classe Animal n'a pas à être instanciée. C'est un type d'objet générique. Ce que vous désirez instancier, ce sont des chats, des chiens, etc... des animaux bien spécifiques.

Polymorphisme

Le polymorphisme est un concept OO dont l'idée est d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

Exemple :

```
ArrayList<Animal> animalFarm = new
    ArrayList<Animal>();
animalFarm.add(new Cat(10));
animalFarm.add(new Dog(13));
...
for(Animal a: animalFarm){
   a.eat();          // les différentes méthodes
                     // eat sont appelées
}
```

Les interfaces

Une interface est un cas particulier d'une classe abstraite qui ne peut contenir que des constantes et signatures de méthodes.

Il n'y a pas de niveau de protection à préciser :

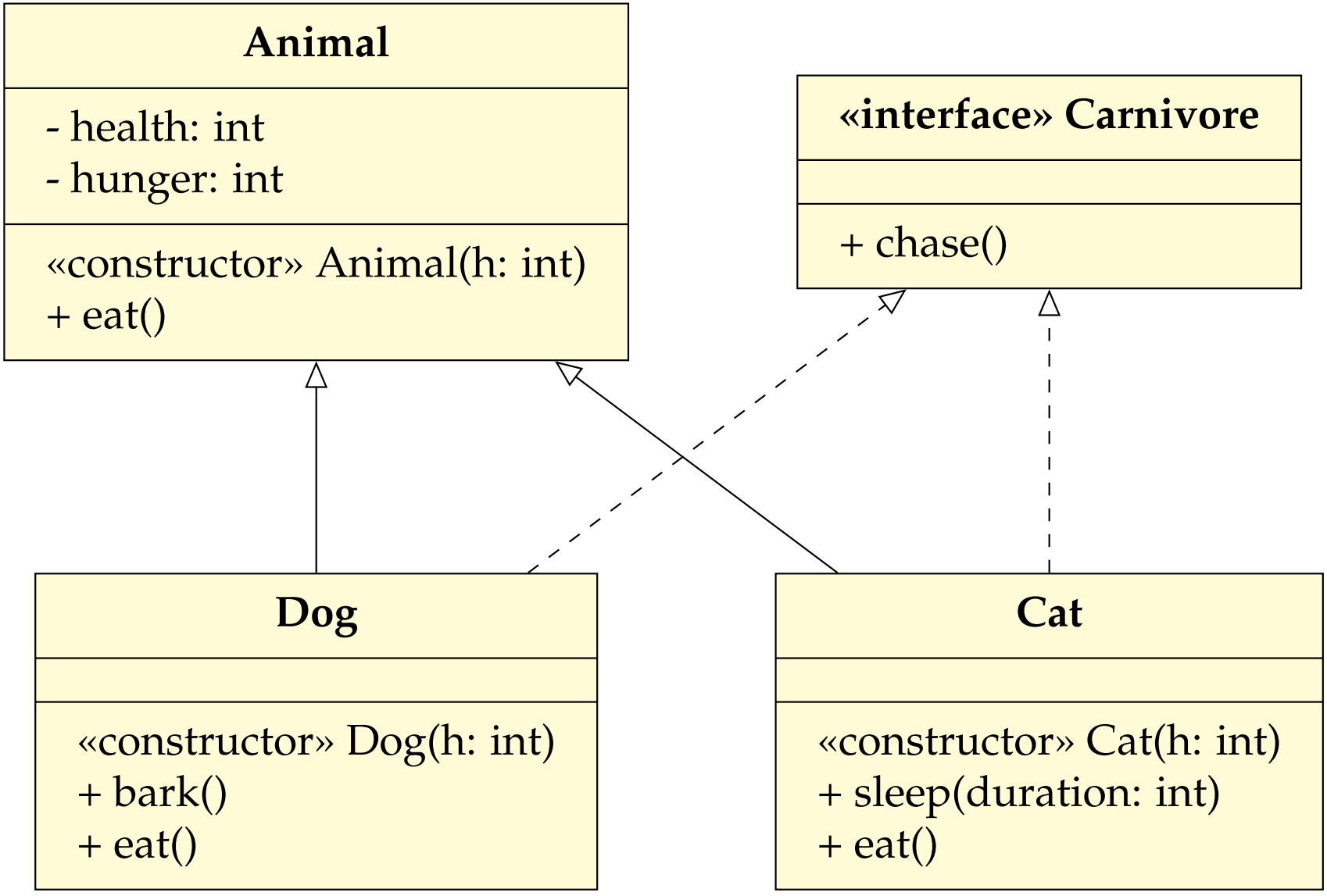
- Les constantes sont implicitement 'public static' et 'final'
- Les méthodes sont implicitement 'public'

Elle ne peut être qu'implémentée par des classes ou étendue par d'autres interfaces.

En Java, une classe ne peut hériter que d'une autre classe mais peut implémenter plusieurs interfaces. De fait, un objet peut avoir plusieurs types : le type de sa classe et les types des interfaces que cette classe implémente.

Documentation

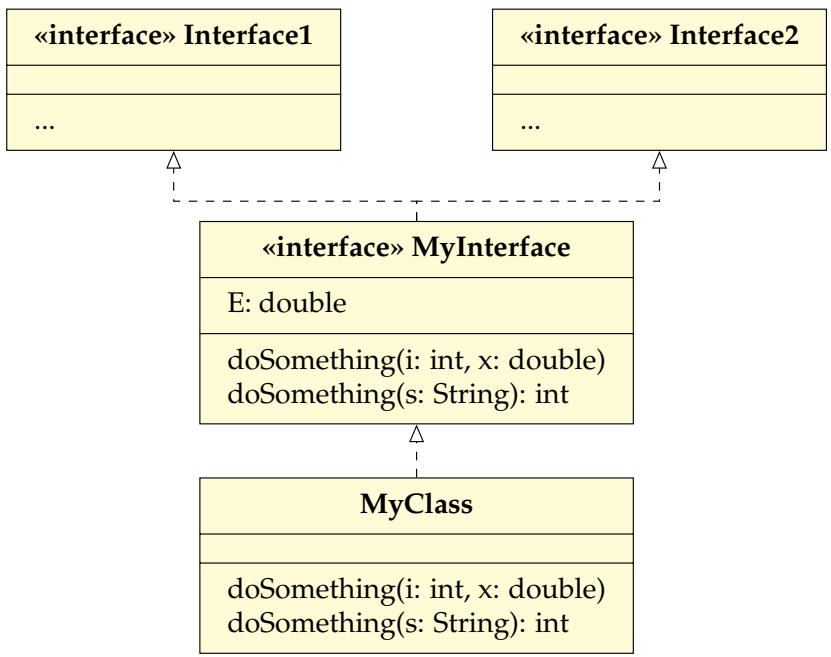
Les interfaces



Les interfaces

```
public interface MyInterface extends Interface1, Interface2{
    double E = 2.7182;
    void doSomething(int i, double x);
    int doSomething(String s);
}

public class MyClass implements MyInterface{
    public void doSomething(int i, double x){
        ...
    }

    public int doSomething(String s){
    ...
    }
}
```

```
MyInterface m = new MyClass();  // Interface utilisée comme type
m.doSomething(4,2.1);
```



Récapitulatif

- **Classes abstraites:** Toutes les méthodes ne sont pas définies. Elles ne peuvent donc pas être instanciés. Les classes filles implémentant toutes les méthodes abstraites peuvent être instanciées.
- **Interfaces:** Cas particulier de classes 100% abstraites. Un classe peut "implémenter" plusieurs interfaces.
- **Polymorphisme:** Mécanisme de redéfinition des méthodes. Une méthode redéfinie dans une classe fille peut avoir un comportement différent en fonction de l'objet sur lequel elle est appelée.
- **Static:** méthodes, attributs et classes qui ne sont pas liés à une instance d'un objet.

Construction d'un objet

```
public class A {
  public A(){
    …
  }
}
public class B extends A {
 public B(){
    …
  }
}
public class C extends B {
 public C(){
    …
  }
}
```

Lors de la construction d'un objet de la classe C :

- le constructeur de C commence par appeler le construteur de B
- le constructeur de B commence par appeler le constructeur de A
- le constructeur de A appelle le constructeur d'Object
- le constructeur d'Object s'exécute
- le constructeur de A se termine
- le constructeur de B se termine
- le constructeur de C se termine

Garbage collector

En Java, quand un objet n'est plus référence, il est détruit par un ramasse-miettes ou garbage collector.

Avant la destruction d'un objet, sa méthode 'void finalize();' est appellée.

Une bonne pratique est d'utiliser *try-catch-finaly* dans les *finalize* pour être certain de ne pas oublier de fermer les ressources réservées par les super-classes.

```
protected void finalize() throws Throwable {
    try{
        close();  // Close open files
    } finally{
        super.finalize();
    }
}
```

Si une exception est déclenchée dans un *finalize*, la finalisation est arrêtée mais l'exception est ignorée.

La méthode *finalize* n'est jamais appellée qu'une seule fois sur un objet.