Programmation orientée objet

Héritage et polymorphisme

Agence nationale pour la formation professionnelle des adultes

1. Principe d’héritage
2. Exemple de classe
3. Héritage en Java
4. Constructeur de sous-classe
5. Classe abstraite
6. Interfaces

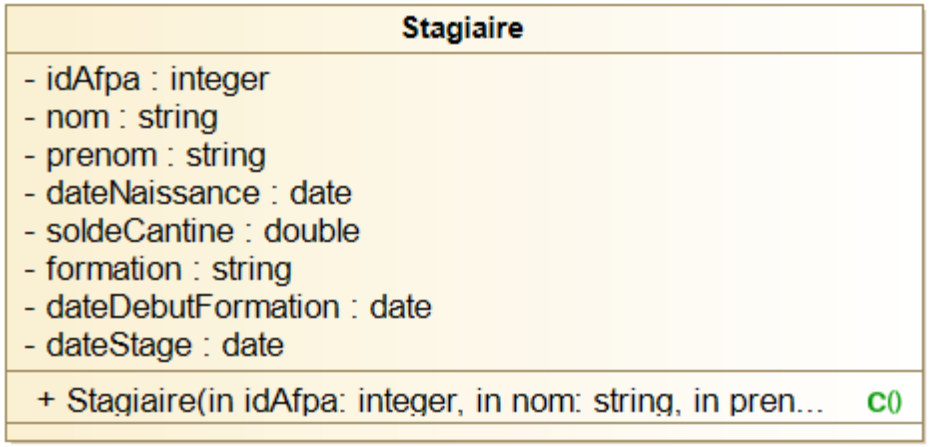
- # Principe :
- L'héritage désigne le principe selon lequel une classe peut hériter de caractéristiques (attributs et méthodes) d'autres classes

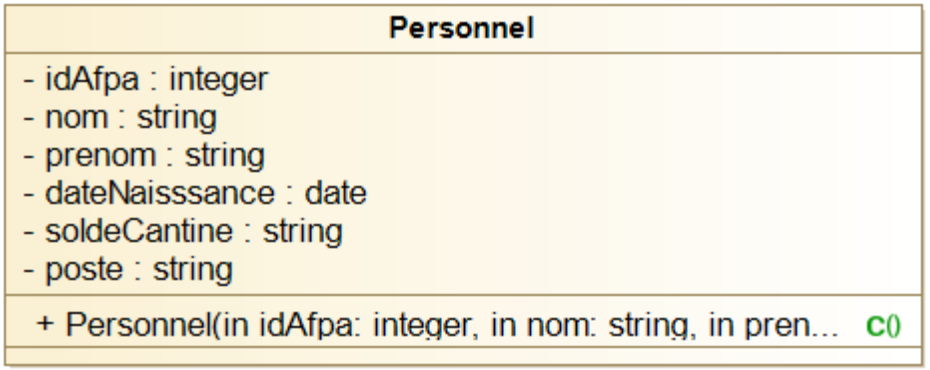
- # Relation d'héritage :
- La classe B « EST UNE/UN » classe A
- A est la **classe mère** de B
- B est une **sous-classe** de A
- La classe B étend les caractéristiques de la class A

- # Exemples :
- « Voiture » est sous-classe de « Véhicules »
- « Piano » est sous-classe de « Instrument »
- « Chien » est sous-classe de « Animal »
- « Chaise de bureau » est sous-classe de « Chaise »

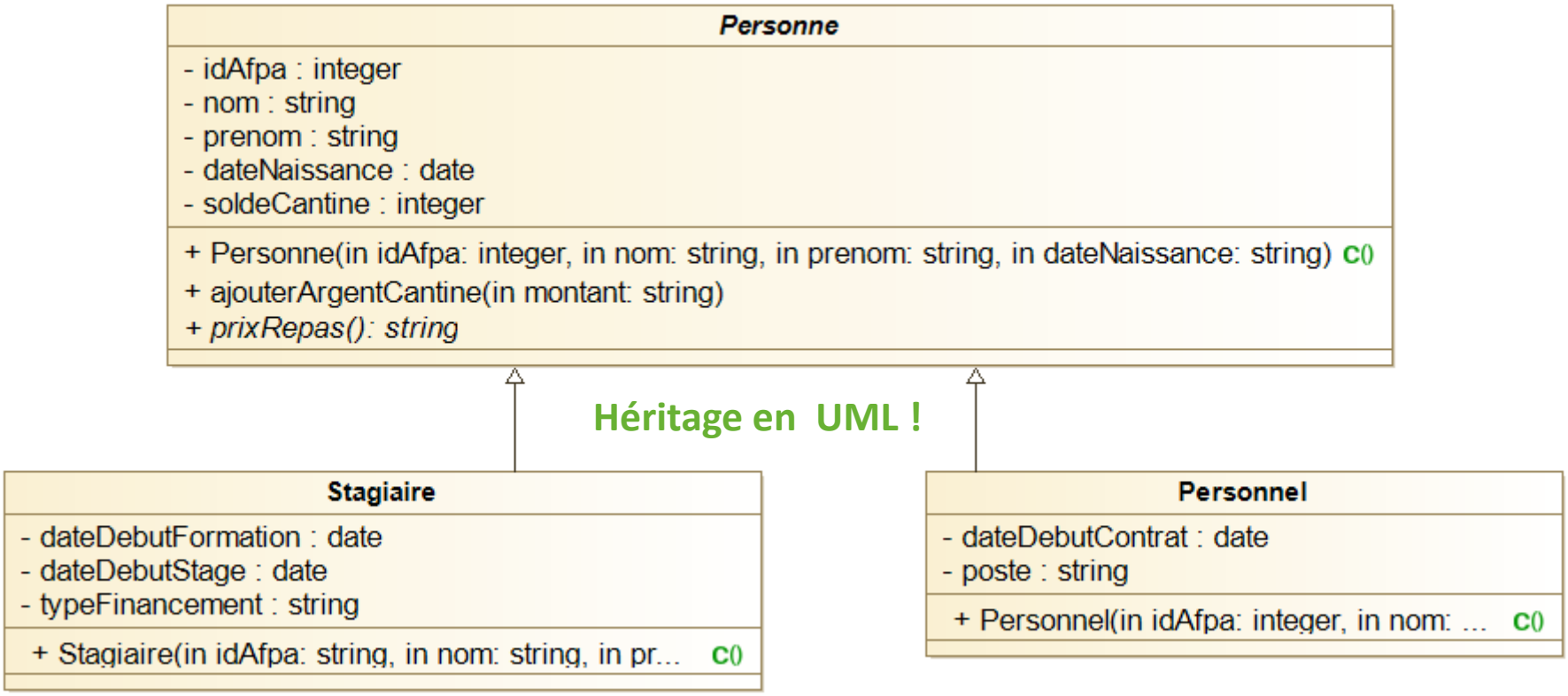
- Exemple d'application : Gestion des comptes cantine de l'Afpa
  - Gestion de plusieurs profils de personnes (stagiaire, employé, visiteur…)
  - Chaque profil a des règles tarifaires différentes

- Représentation UML :
  - Approche sans héritage → **plusieurs classes**





→ **FACTORISATION SOUHAITEE**


Personne
- idAfpa : integer
- nom : string
- prenom : string
- dateNaissance : date
- soldeCantine : integer
+ Personne(in idAfpa: integer, in nom: string, in prenom: string, in dateNaissance: string) C0
+ ajouterArgentCantine(in montant: string)
+ prixRepas(): string
Héritage en UML !
Stagiaire
- dateDebutFormation : date
- dateDebutStage : date
- typeFinancement : string
+ Stagiaire(in idAfpa: string, in nom: string, in pr... C0
Personnel
- dateDebutContrat : date
- poste : string
+ Personnel(in idAfpa: integer, in nom: ... C0

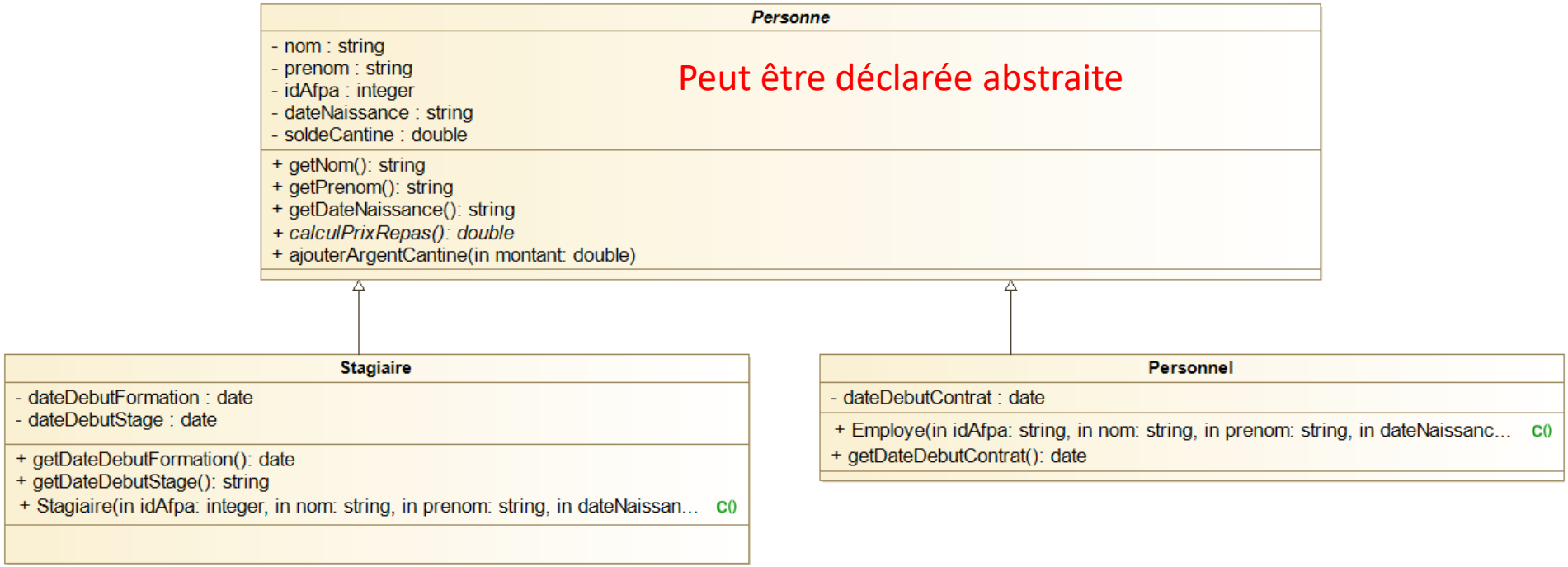
- Mots clef :
- **extends** : indique que la classe hérite d'une autre
- Code java :

```
// fichier « Personne.java »
public class Personne {
    // code de la classe Personne
}


// fichier « Stagiaire.java »
public class Stagiaire extends Personne {
    // code de la classe Stagiaire à écrire + Personne
// (hérité)
}
```

- Code java :

```
// fichier « Personne.java »
public class Personne {
     public Personne(int idAfpa, String nom, String prenom, LocalDate
dateNaissance) {
          // code d'initialisation
     }
}


// fichier « Stagiaire.java »
public class Stagiaire extends Personne {
     public Stagiaire(int idAfpa, String nom, String prenom,
                    LocalDate dateNaissance, LocalDate dateDebutFormation) {
          super(idAfpa, nom, prenom, dateNaissance);
          // suite du code
     }
}
```

- Classe **abstraite** :
- Classe qui **ne peut pas** être instanciée



Impossible de faire →

- Code java :

```
// fichier « Personne.java »
abstract public class Personne {
     // code de la classe Personne
}

// fichier « Stagiaire.java »
public class Stagiaire extends Personne {
     // code de la classe Stagiaire
}



// il est impossible d'écrire le code suivant (erreur de compilation) :
Personne pers1 = new Personne(150363, "Tobet", "Nicolas", "18/04/1995");
// il est possible d'écrire le code suivant (compilateur content) :
Stagiaire stag1 = new Stagiaire(150363, "Tobet", "Nicolas", "18/04/1995");
```

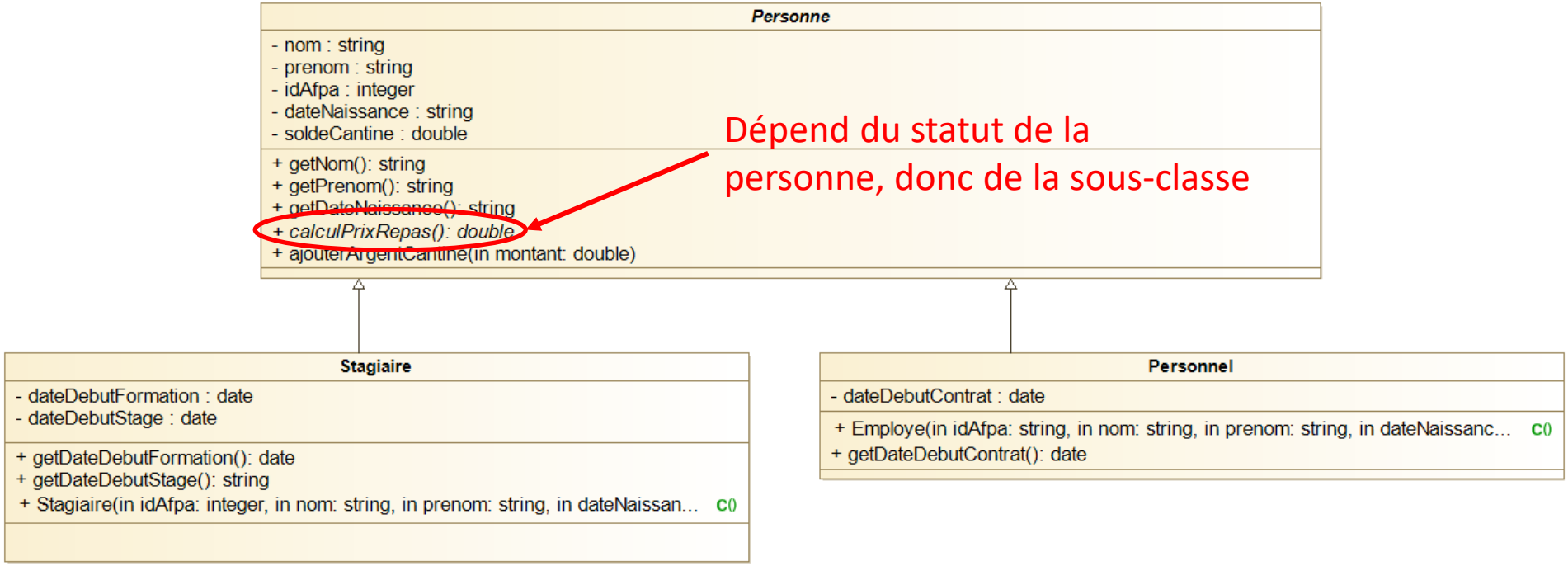
- Différences d'implémentation :
- Objectif : **adapter le comportement** en fonction de la classe

- Exemple :
- Méthode retournant le prix des repas
- **Règles de calcul** : prix fixe pour un stagiaire, en fonction d'un taux pour une employé

- Méthode **abstraite** :
- Méthode qui **doit être implémentée par une sous-classe**

- Méthode **abstraite** :
- Méthode qui **doit être** implémentée par une sous-classe

- En UML : méthode notée *en italique* !



- Code Java :

```
abstract public class Personne {

	private int idAfpa;
	private String nom;
	private String prenom;
	private LocalDate dateNaissance;
	private double soldeCantine;

	public Personne(int idAfpa, String nom, String prenom, LocalDate dateNaissance) {
		this.idAfpa = idAfpa;
		this.nom = nom;
		this.prenom = prenom;
		this.dateNaissance = dateNaissance;
	}

	abstract public double prixRepas();

	public void ajouterArgentCantine(double montant) {
		this.soldeCantine += montant;
	}
	// suite de getters et setters
}
```

Pas d'implémentation (pas de code !)

- Code Java :

```
public class Stagiaire extends Personne {

	private LocalDate dateDebutFormation;
	private LocalDate dateExamen;
	private LocalDate dateDebutStage;

	public Stagiaire(int idAfpa, String nom, String prenom,
			LocalDate dateNaissance,
			LocalDate dateDebutFormation) {
		super(idAfpa, nom, prenom, dateNaissance);
		this.dateDebutFormation = dateDebutFormation;
	}

	@Override
	public double prixRepas() {
		// Le prix d'un repas stagiaire est fixe
		return 1.5;
	}
}
```

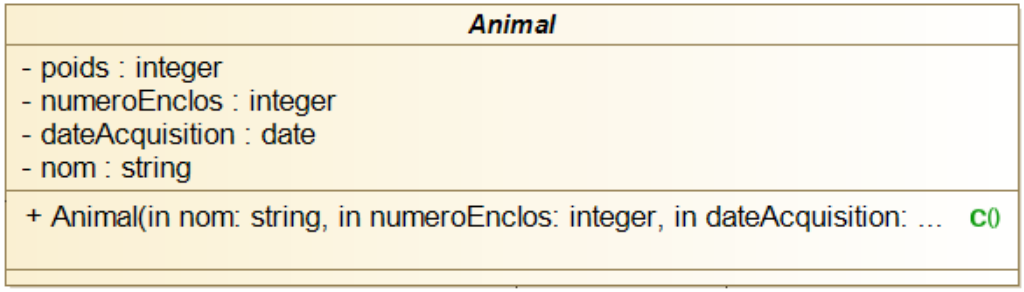
- Polymorphisme :
- Utilisation de plusieurs objets de classes différentes sans se soucier de son type
- Exemple :
- On manipule des instances de « Stagiaire » et « Personnel » de la même façon
- Code Java :

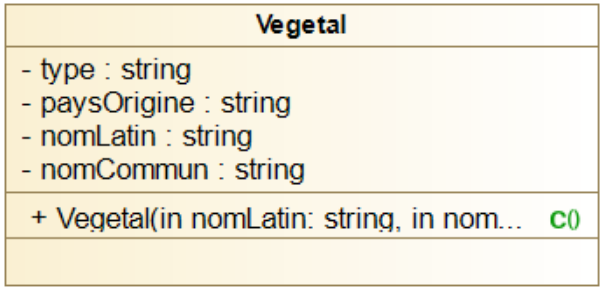
```
Personne stagiaire1 = new Stagiaire(150230, "Jebot", "Thomas", LocalDate.of(1990, 5, 12),
LocalDate.of(2022, 3, 28));
Personne personnel1 = new Personnel(220123, "Tobej", "Sophie", LocalDate.of(1985, 4, 20),
LocalDate.of(2022, 3, 21), "Formateur");


System.out.print("Prix repas pour " + stagiaire1.getIdAfpa() + " : " + stagiaire1.prixRepas());
System.out.print("Prix repas pour " + personnel1.getIdAfpa() + " : " + personnel1.prixRepas());
```

Même appel

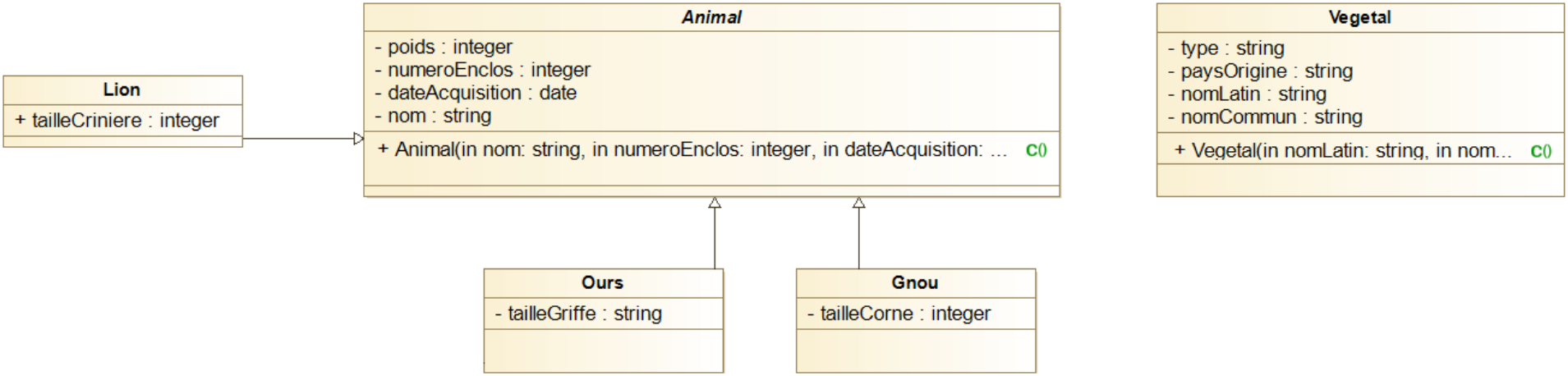
- Une interface est une classe abstraite sans attributs :
- Décrit des méthodes abstraites à **implémenter**
- Permet de donner des méthodes à une sous-classe

- Exemple :
- On cherche à modéliser les animaux et les plantes d'un zoo

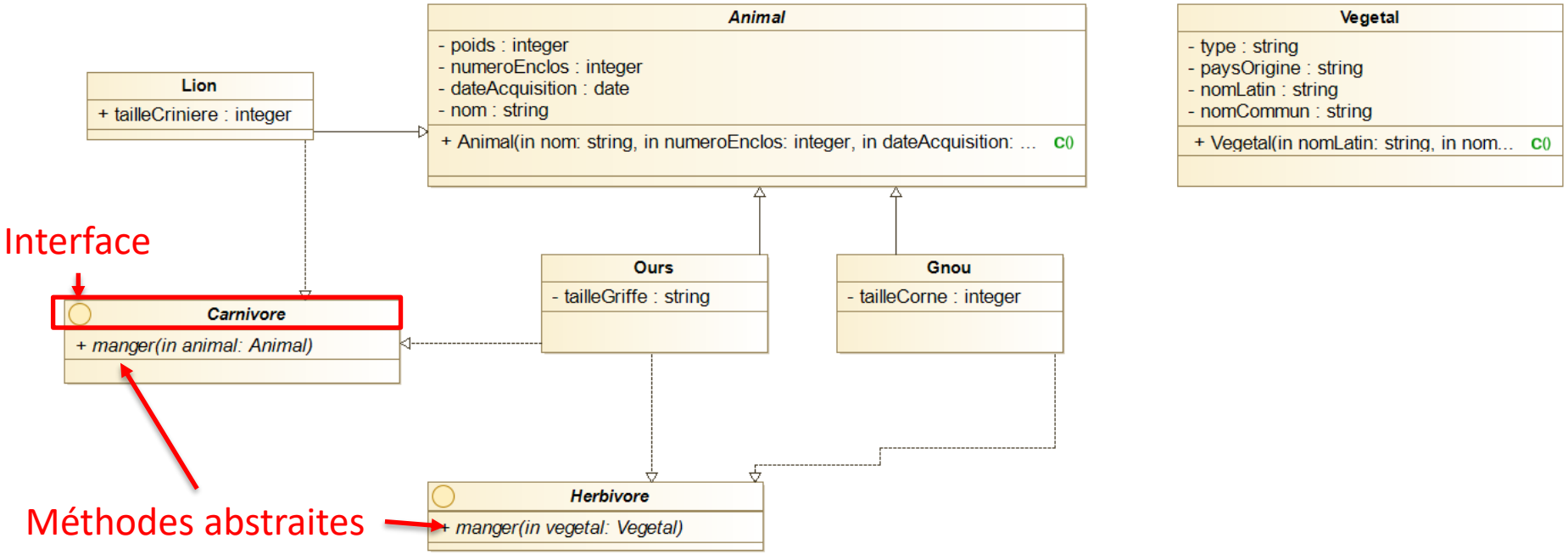




- Plusieurs animaux sont à représenter **: Lion, Ours, Gnou**
- Diagramme UML (non complet) :



- On cherche à ajouter une méthode manger qui dépendra du régime de l'animal (herbivore, carnivore, insectivore…)
- Possibilité d'utiliser une interface :



```java
abstract public class Animal {
	// Code de la classe
}

public interface Carnivore {
	public abstract void manger(Animal animal);
}

public interface Herbivore {
	public abstract void manger(Vegetal vegetal);
}

public class Gnou extends Animal implements Herbivore {
	@Override
	public void manger(Vegetal vegetal) {
		// Délicieux.
	}
}
```

```java
public class Ours extends Animal implements Carnivore, Herbivore {

	@Override
	public void manger(Vegetal vegetal) {
		// Tout manger !
	}

	@Override
	public void manger(Animal animal) {
		// Tout manger !
	}
}
```

Agence nationale pour la formation professionnelle des adultes
Établissement public à caractère industriel et commercial
Tour Cityscope
3 rue Franklin, 93100 Montreuil
824 228 142 RCS BOBIGNY