Conception et Programmation objet Avancées

Contrat pédagogique et révisions objet et UML

Sylvie Coste

IUT de Lens

2021–2022

Plan

Objectifs du module et compétences visées (PPN 2013)

Objectifs

- Produire une conception détaillée en appliquant des patrons de conception, la mettre en œuvre en utilisant des bonnes pratiques de programmation orientée objet

Compétences visées

- *Analyse d'une solution informatique*
- *Conception technique d'une solution informatique*
- *Réalisation d'une solution informatique*

À l'issue du cours, vous devriez être capable de . . .

- gérer un projet en tirant profit d'un *environnement de travail intégré (IDE)*
- concevoir une solution en utilisant les *patrons de conception (design patterns)*
- réaliser cette solution en respectant les *bonnes pratiques de programmation objet*
- ...

Thèmes abordés

- Approfondissement de la modélisation objet pour la conception et la programmation
- Compréhension et mise en œuvre de patrons de conception, éléments d'architecture logicielle
- Notions avancées de programmation orientée objets (responsabilité unique, principe ouvert-fermé, notions de dépendance et de couplage)
- (Sensibilisation aux tests d'intégration)

Déroulement/Activités

Cours Magistral

- 12h (1 par quinzaine)
- réflexion individuelle/de groupe, mini-test, cours magistral

TD/TP

- 30h (1 créneau hebdomadaire puis 3 par quinzaine)
- exercice individuel, mini-évaluation

Communication

La plateforme pédagogique Moodle

- http://moodle.univ-artois.fr
- M3105 : Conception et programmation objet avancées
- Forum (Nouvelles) et messagerie

Mattermost

- Équipe CPA-DUT2-INFO

Bureau

- 10E

Absences/retards

Retards

Les retards gênent l'enseignant et l'ensemble du groupe : arrivez donc à l'heure !

Absences

Vous devez prévenir de vos absences prévues vos enseignants et rattraper auprès de vos camarades avant la séance suivante

Toute absence non justifiée à une évaluation entraîne la note 0

Les absences justifiées donnent lieu à un rattrapage en fin de semestre

Évaluations

Évaluations formatives

- Au fur et à mesure du CM
- Brèves
- Le cours doit être su au fur et à mesure (pas de document)

Évaluations (si en présentiel)

- 3 DS de même coefficient répartis sur le semestre
- Durée d'1h
- Documents autorisés : *à définir à chaque DS*

Outils

Basique

- Langage de programmation : Java
- IDE : IntelliJ IDEA
- La notation UML

Si on a le temps

- Les tests : JUnit
- Gestionnaire de version : Git

Bibliographie (disponible à la bibliothèque de l'IUT)

Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates.
Design patterns : tête la première.
O'Reilly, 2005.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns (Catalogue de modèles de conception réutilisables).
Vuibert, 1999.

Craig Larman.
UML2 et les design patterns (Analyse et conception orientées objet et développement itératif).
Pearson Education, 2009.

Alan Shalloway and James R. Trott.
Design Patterns par la pratique.
Eyrolles, 2002.

Plan

Génie logiciel

Définition

L'ensemble des méthodes, des techniques et outils concourant à la production de logiciel, au-delà de la seule activité de programmation

- comment créer un logiciel
- avec de bonnes pratiques de conception, d'implémentation et de maintenance
- permet d'obtenir une amélioration de la qualité du logiciel

Qualités attendues d'un code

- *Capacité fonctionnelle* : est-ce que l'application correspond aux besoins exprimés ?
  - *Conformité, pertinence ...*
  - *Intégrité/sécurité*
- *Fiabilité (robustesse)* : est-ce que le logiciel maintient son niveau de services dans tous les cas prévus d'utilisation ?
- *Facilité d'utilisation* : ergonomie
- *Rendement et efficacité* : utilisation optimale des ressources
- Maintenabilité : est-ce que les évolutions du logiciel demandent peu d'effort ?
  - Extensibilité : facilité d'ajout de nouvelles fonctionnalités
  - Réutilisabilité (tout ou partie)
  - Testabilité : facilité de préparation des procédures de test
- *Portabilité* : transfert sous différents environnement

Processus de développement

Ne consiste pas simplement en l'écriture du code

- Recueil et analyse des besoins (conceptuelle)
- Modélisation (conception de l'architecture)
- Implémentation
- Tests
- Mise en production (utilisation)
- Maintenance et évolution

Plan

UML

Unified Modelling Language

- langage de modélisation qui permet de représenter et de communiquer les divers aspects d'un logiciel
- pas un langage de programmation ni une méthode !
- un ensemble de notations communes
- norme riche (permet de spécifier du début à la fin) et ouverte (en constante évolution depuis sa création)
- sur la base de la notion d'objet

$\Longrightarrow$ Reprenez vos cours de COO de semestre 2 !

Les (principaux) diagrammes

Besoin des utilisateurs

- diagramme des cas d'utilisation

Aspect statique - représentation des données

- diagramme de classes

Aspect dynamique des objets – cycle de vie

- diagramme état/transition

Diagramme d'activités - interaction entre les objets

- diagramme de séquence

L'approche orientée objet

Le système est vu comme un ensemble d'objets

- Chaque objet gère l'information sur son état
- L'architecture est liée aux objets du domaine (données et traitements associés)
- Les évolutions sont plus locales, donc plus faciles

But : permettre une conception modulaire

Application

Une application est un ensemble d'objets collaborant

- Le comportement d'une application (orientée objet) repose sur la communication entre les objets qui la composent
- Les objets communiquent en échangeant des messages
  - Constructeurs
  - Destructeurs (implicites en Java)
  - Accesseurs (« getters »)
  - Modificateurs («setters»)
  - ...

Les classes

Description abstraite d'un ensemble d'objets « de même nature»

- n'a pas d'état, ni d'identité (vue statique du système)
- c'est un modèle pour la création de nouveaux objets
- contrat garantissant les compétences minimales d'un objet

- des éléments concrets (un livre)
- des éléments abstraits (une commande)
- des composants d'une application
  (les éléments de l'interface graphique)
- des structures informatiques (des listes)
- des éléments comportementaux
  (des événements d'un agenda)

Classe vs Objet

Exemple

Marguerite est une instance de la classe *Vache*

Objet/Classe a deux aspects

- Interface : vue externe de l'objet
- Corps : implémentation des comportements et des attributs

- L'utilisateur/programmeur ne connaît que l'interface
- L'implémentation est masquée et non accessible

Diagramme de classes

Vue statique

- C'est le plus important pour la modélisation
- Il décrit la structure interne du système
- Il permet de fournir une représentation abstraite des objets qui vont interagir

- Le diagramme de cas d'utilisation montre le système du point de vue des acteurs

Visibilité - Encapsulation

Encapsulation

L'encapsulation est un principe de conception consistant à protéger le cœur d'un système des accès venant de l'extérieur

- La visibilité définit le degré de protection
  - + : visibilité publique i.e. toutes les classes y ont accès (cas le plus fréquent pour les méthodes)
  - - : visibilité privée i.e. inaccessible à tout objet hors de la classe (attributs et opérations internes à la classe)
  - # : visibilité protégée i.e. les sous-classes y ont accès (cas le plus fréquent pour les attributs)
  - ~ : visibilité de paquetage
- Pas de visibilité par défaut en UML
- Visibilité publique : la classe s'engage à toujours fournir le service

Représentation d'une classe (diagramme de classe)

Classe documentée

Livre
auteur titre cote emprunte
estDisponible() estRangeACoteDe() titreContient() ecritPar()

Classe détaillée

Livre
- auteur : String - titre : String - cote : String - emprunte : boolean
+ estDisponible() : boolean + estRangeACoteDe(unLivre : Livre) : boolean + titreContient(chaine : String) : boolean + ecritPar() : String

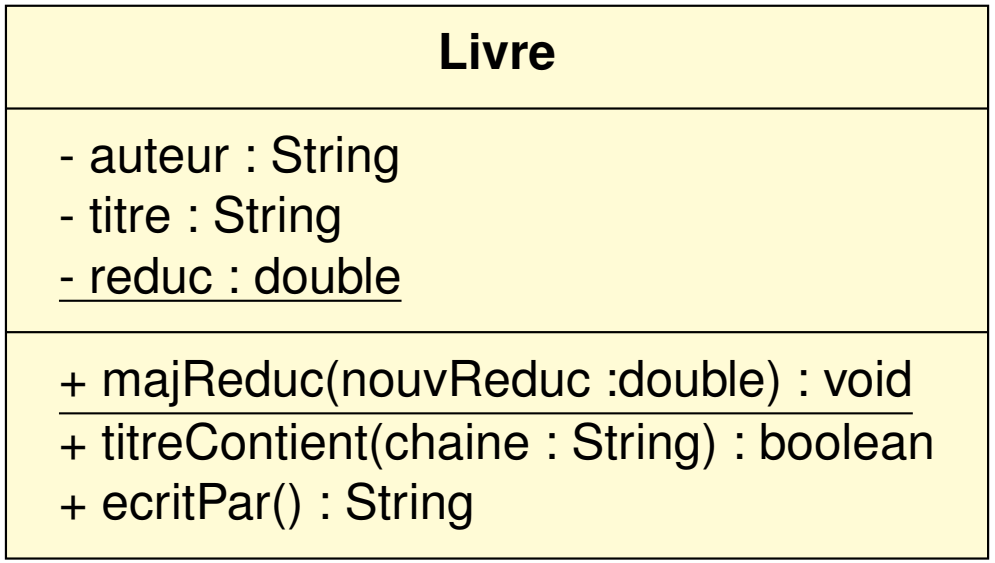
Classe non documentée

mediatheque : :Livre

Attributs et méthodes de classe

Notation

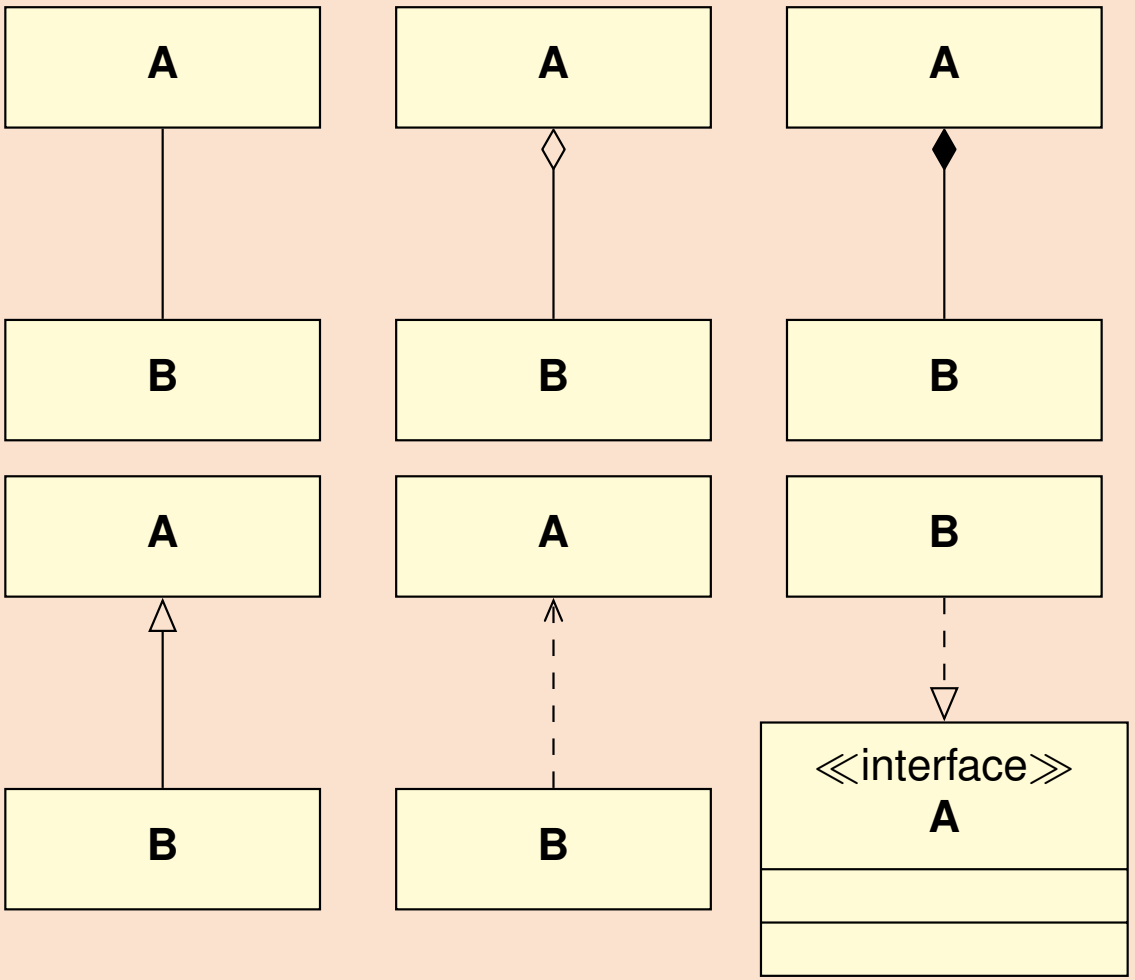
- Souligné en UML
- *static* en Java



- Un attribut de classe est partagé par toutes les instances
- Il garde une valeur unique
- Une méthode de classe porte uniquement sur des attributs de classe

Relations entre classes

Que représente chaque flèche ?

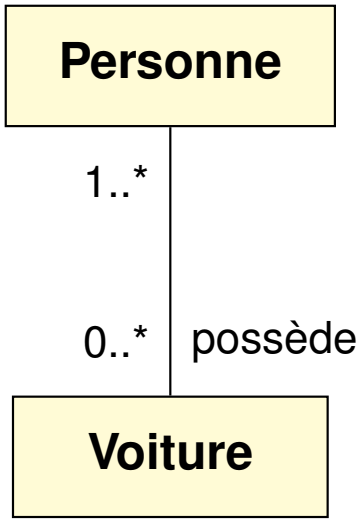


Relations entre classes : association

- Relation structurelle (sémantique) entre deux (ou plusieurs) classes
- Abstraction des différents liens qui peuvent exister
- Réflexive (en général)

Peut être complétée

- Nom
- Rôle
- Multiplicité

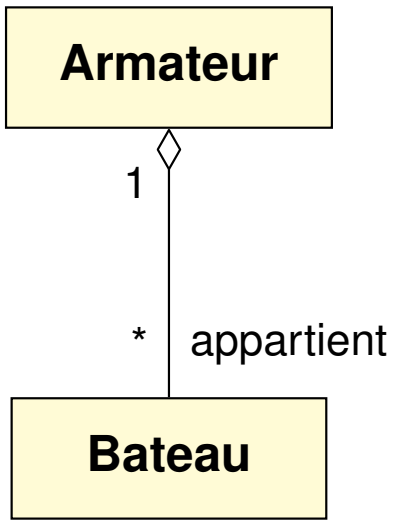


Relations entre classes : agrégation

- Forme particulière d'association entre deux classes
- L'agrégation est une association non symétrique
- Une agrégation peut exprimer qu'une classe fait partie d'une autre

Exemple

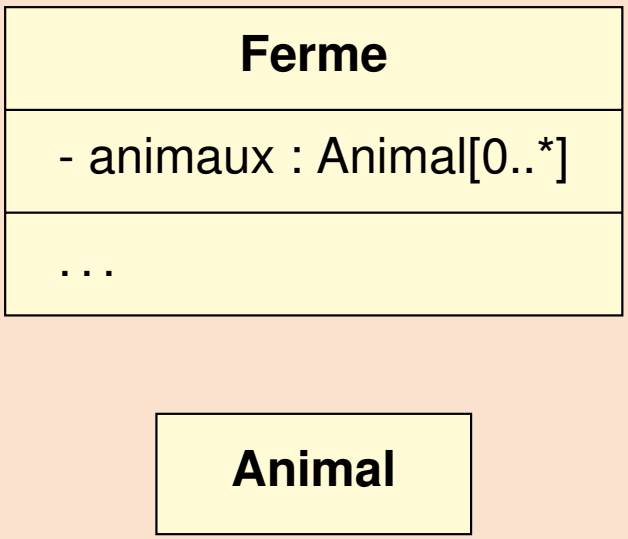
- Un bateau peut appartenir à un armateur
- L'armateur peut posséder plusieurs bateaux



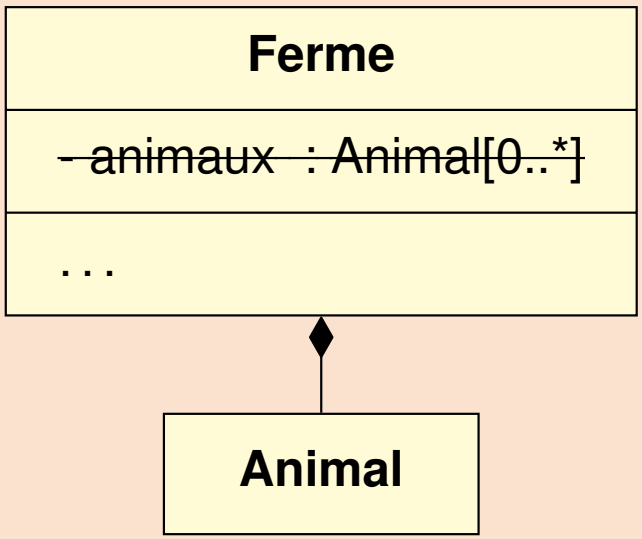
Relations entre classes : composition

- La composition est une agrégation forte
- Une classe est composée d'objets d'une (ou plusieurs) autres classes
- Les objets des attributs *font partie* des objets de la classe

Rep. avec les attributs

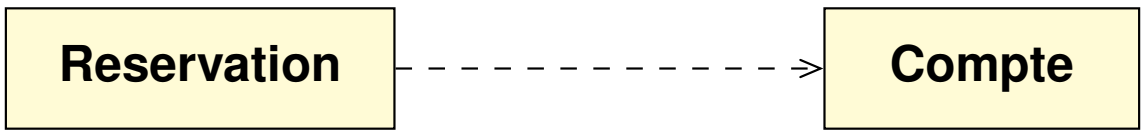


Rep. avec des relations



Dépendance

- Exprime qu'une classe en utilise une autre
- Toute modification de la classe utilisée peut se répercuter sur la classe utilisatrice



Généralisation/spécialisation

Généralisation

- Factorisation dans une classe (appelée super-classe) de propriétés communes à plusieurs classes
- Notion d'héritage en Java
- Relation non transitive et non-réflexive
- Généralisation multiple (pas possible en Java)

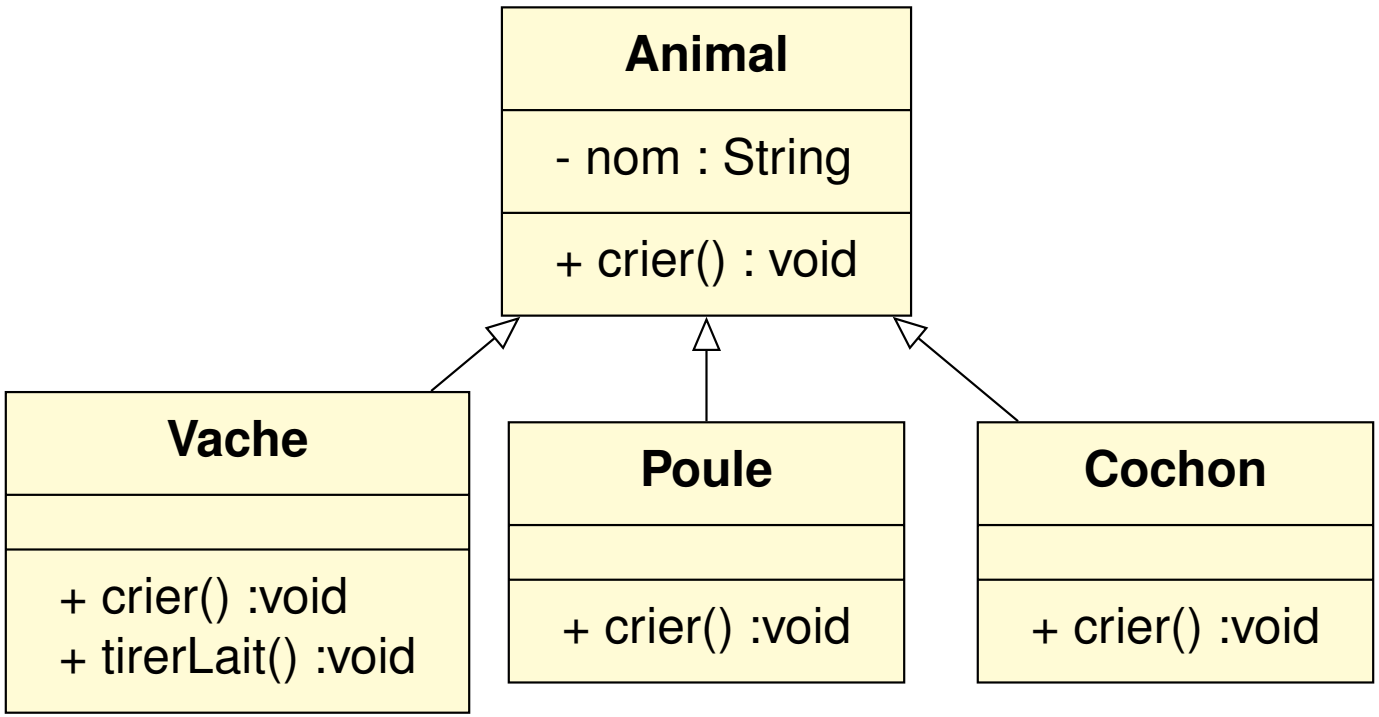
Spécialisation

- Inverse de la généralisation
- Plusieurs sous-classes spécialisées

Héritage

Exemple

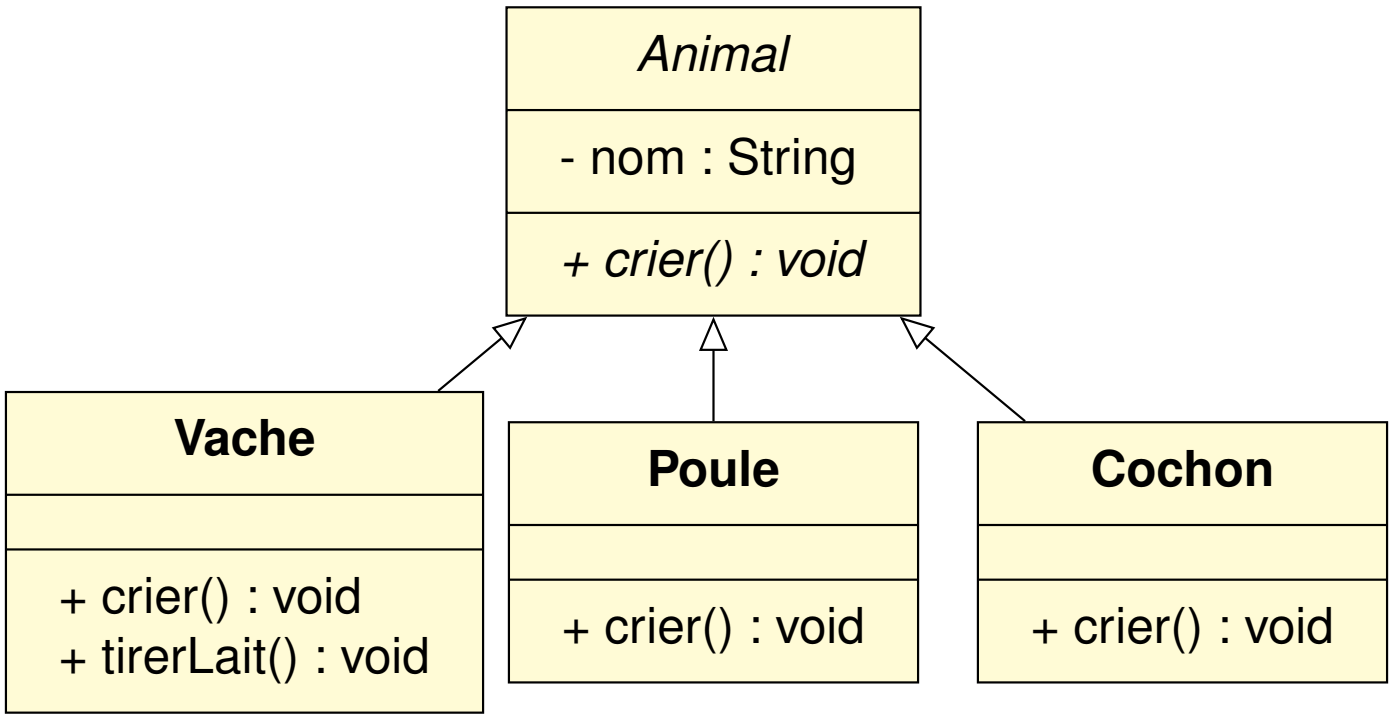
- Animal généralise Vache, Poule, Cochon



Classes abstraites

- Classe qui ne peut pas être instanciée
- Sert uniquement de super-classe à d'autre classes
- Peut contenir des méthodes abstraites pour forcer toutes les sous-classes à implémenter ces méthodes

- Exemple précédent : *Animal* peut-être une classe abstraite



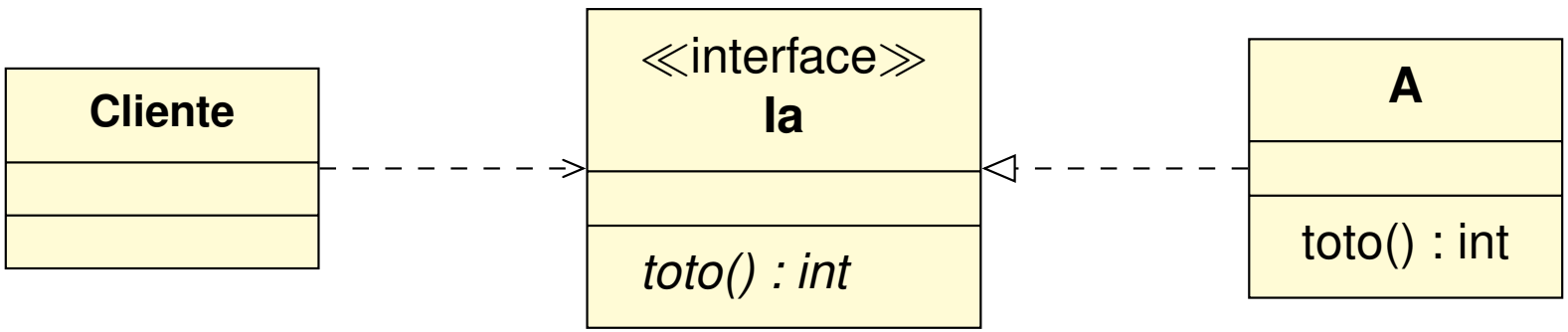
Classes abstraites en Java

```
public abstract class Animal {
  private String nom;

  public Animal(String leNom){
    nom=leNom;
  }

  abstract void crier();

  String getNom(){
    return nom;
  }
}
```

```
class Vache extends Animal{

  public Vache(String sonNom){
    super(sonNom);
  }

  void crier(){
    System.out.println("Meuh !");
  }

  void tirerLeLait(){
    System.out.println("Tire");
  }
}
```

Interfaces

- Souvent, on veut définir les services rendus par la classe i.e. son interface
- Une interface est définie comme une classe
- Une interface ne contient que les signatures des méthodes que doivent implémenter les classes qui réalisent l'interface



Interfaces en Java

```
public interface Ia {
  public int toto();
}

public class A implements Ia{
  public int toto(){
    return 0+0;
  }
}
```

```
public class Cliente{
  public static void main(String args[]){
    Ia calcul = new A();
    System.out.println(calcul.toto());
    }
}
```

Classes clientes d'une interface

- Quand une classe dépend d'une interface pour réaliser ses opérations, elle est dite classe cliente de l'interface
- On utilise une relation de dépendance entre la classe cliente et l'interface requise
- Toute classe implémentant l'interface pourra être utilisée : principe de substitution

Classes – bilan

Encapsulation

- Rassembler les attributs et les méthodes en masquant les détails d'implémentation
- Garantir l'intégrité des données de l'objet
- Constituants privés des objets

Polymorphisme

- Poly = plusieurs, morphisme = forme
- Propriété d'un élément de pouvoir se présenter sous plusieurs formes
- Capacité donnée à une même opération de s'effectuer différemment suivant le contexte de la classe où elle se trouve

Surclassement (upcasting)

Définition

- Une classe B qui hérite d'une classe A définit un sous-type du type défini par A
- Toute instance de B peut être vue comme une instance de A
- Une référence déclarée de type A peut-être instanciée par un objet de type B
- On peut affecter à une référence d'un type donné une valeur qui correspond à un objet dont le type effectif est une sous-classe directe ou indirecte du type de la référence

Surclassement/Upcasting

Exemple

- Une Vache est un Animal
- L'ensemble des vaches est inclus dans l'ensemble des animaux

```
Animal a;
a = new Vache();
// et même
a = new VacheVosgienne();
```

Surclassement/Upcasting

Compilation/exécution

- Lorsqu'un objet est "sur-classé", il est vu comme un objet du type de la référence utilisée pour le déclarer
- Ses fonctionnalités sont restreintes à celles de proposées par la classe du type de la référence

```
Animal a;
a = new Vache(); // upcasting
a.crie();
// méthode définie pour tous les animaux
// résolution dynamique
// méthode définie pour Vache choisie à l'exécution
a.trait();
// le compilateur refuse
// trait() non définie dans la classe Animal
```

Transtypage/Downcasting

Définition

- Permet de forcer un type
- Il faut que le type de l'objet soit «compatible»avec la classe forcée : même classe ou sous-classe
- Permet d'accéder à des comportements spécifiques à la sous-classe
- Sinon, une erreur se produit à l'exécution
- Vérifier le type via instanceof

Transtypage/Downcasting

Exemple

```
Animal a = new Poule();
Poule p = (Poule) a; p.pond();
```

Provoque une erreur à l'exécution

```
Animal a = new Poule();
Vache v = (Vache) a;
```

Vérification du type

```
if (a instanceof Vache) {
  Vache v = (Vache) a;
  v.trait();
}
```