Conception et programmation objet avancées
Introduction

Petru Valicov
petru.valicov@univ-amu.fr

2017-2018

Objectifs

- Approfondir les notions de programmation objet (bonnes pratiques, patrons de conception)
- Apprendre à mener un projet de la phase de conception, jusqu'au produit final
- Utilisation d'une approche méthodologique
- Apprendre à programmer "proprement"
- Mise en œuvre des acquis de la programmation objet (`Java`)
- Sensibilisation aux tests

Planning du cours - 2h par séance

1. Généralités + rappels `UML` et notions d'objet (1 cours)
2. `Java` + les bonnes pratiques de la POO ($\approx$ 2 cours)
3. Patrons de conception ($\approx$ 3 cours)

TD+TP

- 3 séances de 4h de TD en salle machine.
- 3 séances de 4h de TP en salle machine.
- 16h de travail personnel (sans enseignant) dédiées au **projet**

Trois enseignants interviennent dans les séances :

sophie.nabitz@univ-avignon.fr - TD/TP
sebastien.nedjar@univ-amu.fr - TD/TP
petru.valicov@univ-amu.fr - CM/TD/TP

En TD et TP c'est **VOUS** qui travaillez $\Longrightarrow$ Pas de correction à copier/coller $\Longrightarrow$ analyse/amélioration de **VOTRE** solution

Retour de l'an dernier :

" Pas assez de temps pour le projet !" ☹

" Les TD et TPs sont longs !" ☹

" Pas de correction de TD/TP. Et si moi, j'aime apprendre avec une correction ? !" ☹

" Trop peu de temps pour le projet !" ☹

" Trop de travail !" ☹

" Encore plus de CPOA !" ☺

Morale : Il faut bosser dès le début et tout ira bien.

Contrôle des connaissances

Contrôle Continu (Projet)

- par équipe de 4 (les étudiants doivent être du **même groupe**)
- les notes seront individuelles
- but : **travailler en équipe**, respecter les principes de la CPOO
- période de réalisation : mi-octobre à mi-novembre
- **soutenances le 23.11.2017**

Examen (17.11.2017)

- Durée de 3 heures
- questions de cours + exercices
- tous les documents issus du cours autorisés
- l'utilisation intensive des documents est très déconseillée.

N'apprenez pas par cœur – ça va vous pénaliser !

Les outils

- La notation `UML`
- Langage de programmation : `Java` version ≥1.8 (certains exemples en `C++` en cours)
- IDE : `IntelliJ IDEA` ou autre (`Eclipse`, `NetBeans`, etc.)
- Les tests : `JUnit`
- Versioning : `Git` et `GitHub`
- GUI : `JavaFX`

...et pour commencer, au moins quelques principes de programmation :

KISS - **K**eep **I**t **S**imple and **S**tupid

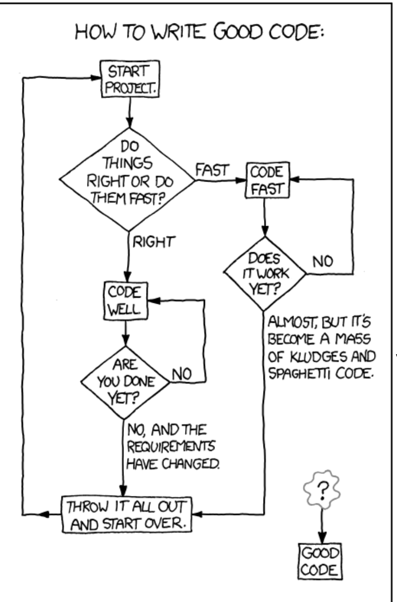
YAGNI - **Y**ou **A**in't **G**onna' **N**eed **I**t

DRY - **D**ont **R**epeat **Y**ourself

Un peu de biblio

- E. Gamma, R. Helm, R. Johnson et J. Vlissides. **Design Patterns. Elements of Reusable Object Oriented Software**. *Addison Wesley* : 1995.
- E. Freeman, E. Robson, B. Bates, K. Sierra. **Head First - Design Patterns** (régulièrement mis à jour) *O'Reilly* : 2014. dispo dans la base SAFARI du SCD avec le compte AMU : http://proquestcombo.safaribooksonline.com/
- R.C. Martin. **Clean Code - A Handbook of Agile Software Craftmanship** *Prentice Hall* : 2008.
- A. Shvets. **Design Patterns Explained Simply.** *SourceMaking* : 2015 https://sourcemaking.com/design_patterns

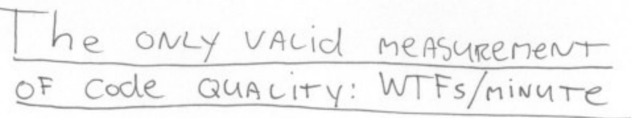
Bien coder

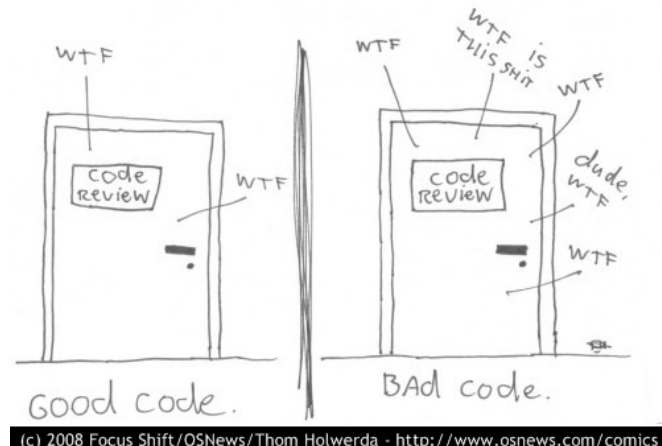


La *rapidité* prime sur la *qualité* ?

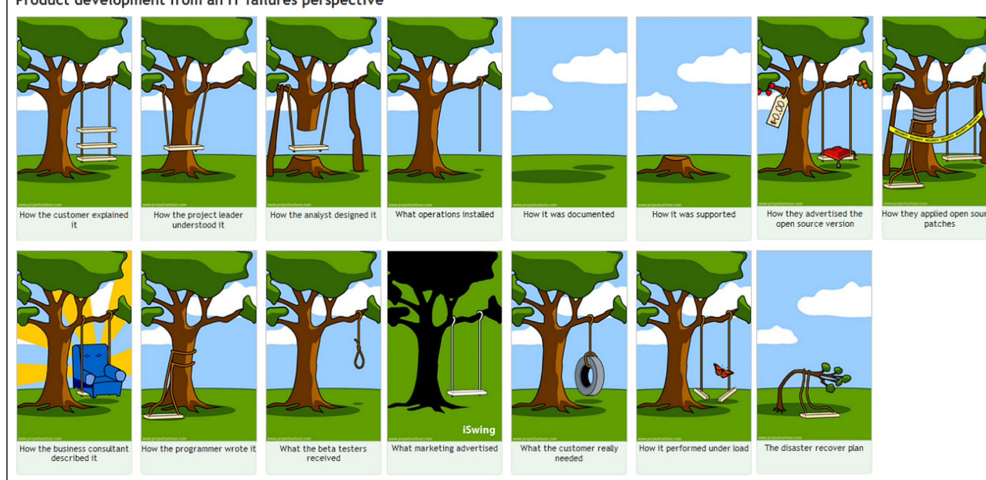
Parmi vous il y en a qui y croient vraiment !

Bien coder





Savoir ce qu'on fait



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.

L'art et la manière de créer un logiciel. Dépasse le cadre purement technique.

- développe les bonnes pratiques de conception, d'implémentation et de maintenance d'un logiciel ;
- permet d'obtenir une amélioration de la qualité du logiciel.

Les qualités d'un logiciel/code bien écrit

Beaucoup !

- Validité - correspond aux spécifications définies par le cahier des charges
- Fiabilité (robustesse) - gestion des conditions "anormales"
- Facilité d'utilisation (ergonomie)
- Extensibilité
- Réutilisabilité (en tout ou en partie)
- Compatibilité
- Efficacité - utilisation optimale des ressources matérielles
- Portabilité - transfert sous différents environnement matériels et logiciels
- Intégrité - aptitude du logiciel à protéger son code et ses données contre des accès non autorisés
- Vérifiabilité - facilité de préparation des procédures de test

Tests

La phase essentielle consistant à vérifier et valider un système informatique ainsi que ses composantes.

Les outils de tests peuvent être manuels ou automatisés.

Les tests peuvent être classifiés selon leur niveau ou leurs caractéristiques

" *Tester un programme peut démontrer la présence de bugs, jamais leur absence.*"

Edsger W. Dijkstra

Tests

- Tests unitaires
  - vérifient qu'un composant du système respecte ses spécifications
  - la composante est testée individuellement
- Tests d'intégration des modules
  - vérifient que l'ensemble des composants d'un module fonctionnent ensemble
  - détecte principalement les problèmes d'interface entre composants
- Tests d'intégration du logiciel
  - vérifient que tous les modules du logiciel fonctionnent ensemble
- Tests du système
  - le bon fonctionnement du logiciel dans son environnement d'exécution (compatibilités avec les logiciels existants)
  - tests de performance (+tests de profil)
- Tests de validation - importants pour le client !
  - tests du logiciel dans les conditions définies par l'utilisateur
- Tests en boîte noire - vérifient que les sorties sont bien celles prévues par l'utilisateur

Concepts objet et `UML` - Rappels

UML - Unified Modeling Language

Définition

UML est un **langage de modélisation** orienté objet standard qui permet de représenter (de manière graphique) et de communiquer les divers aspects d'un système informatique.

- Apparu au milieu des années '90 (G. Booch, I. Jacobson et J. Rumbaugh). La version actuelle - UML 2.2
- **langage de modélisation** $\neq$ **langage de programmation**
- C'est juste un ensemble de notations ayant comme base la notion d'*objet*

Reprenez vos notes de cours de l'an dernier !

Les diagrammes UML

Besoin des utilisateurs

- **diagramme des cas d'utilisation**

Aspect statique (vue structurelle) – représentation des données

- *diagramme objet*
- **diagramme de classes**

Aspect dynamique des objets – cycle de vie

- **diagramme État/Transition**
- *diagramme d'activités*

Interaction entre les objets (vue fonctionnelle)

- **diagramme de séquence**
- *diagramme de collaboration*

L'approche orientée objet

- Le système est vu comme un ensemble d'objets.
- Chaque objet gère l'information concernant son propre état.
- L'architecture est liée aux objets du domaine (données+traitements).
- Les évolutions sont plus locales, donc plus faciles.
- But ultime : permettre une conception modulaire.

Les objets - pourquoi ?

Ce sont des abstractions

- mettent en avant les caractéristiques essentielles
- définissent une représentation simplifiée
- dissimulent les détails
- regroupent un ensemble de données et de compétences

Impact sur la façon de penser :

héritage ou **composition** ou **interface** ?

Les objets

Définition

Objet = identité + état + comportement

- Chaque objet possède une **identité** unique qui lui est propre et qui le caractérise
  - exemple : deux étudiants homonymes n'ont pas la même identité
  - dans une base de données : clef primaire
  - en `C++` et `Java` : référence unique (mot clef `this`)
- L'**état** : valeur de tous les attributs d'un objet à un instant T
  - évolue dans le temps
  - à un instant T il est la conséquence des comportements passés
  - par exemple, l'état d'un étudiant : nom, prénom, date de naissance, adresse, diplômes obtenus, diplôme en cours, . . .
- Le **comportement** regroupe toutes les compétences d'un objet

  $\Longrightarrow$ les *méthodes* de la classe

Application

- Une application est un ensemble d'objets collaborant entre eux.
- 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)
  - Iterateurs
  - ...

Les classes

Une classe est une 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 (ex. livre, client)
- des éléments abstraits (ex. commande de marchandise)
- des composants d'une application (ex. les éléments de l'interface graphique)
- des structures informatiques (ex. tables, liste de contacts)
- des éléments comportementaux (ex. des tâches, paiement)

Classe vs Objet

Une **instance** de la classe HUMAIN est une personne donnée (par ex. Philippe - objet de type HUMAIN).

Objet/Classe a deux aspects :

- Interface : vue externe de l'objet
- Corps : implémentation des comportements (opérations) et des attributs.

Encapsulation :

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

Diagramme de classes (modèle statique)

Le diagramme les plus important pour la modélisation orientée objet.

Décrit la **structure interne** du système.

Représente les classes (avec les attributs et méthodes) et les relations entre elles.

Permet de fournir une représentation abstraite des objets du système qui vont interagir ensemble pour **réaliser** les cas d'utilisation.

Diagramme de classes

Plusieurs niveaux de détails

Produit

Produit
-num_produit -photo_produit -description_produit -prix -marge -/fraisDePort +estDisponible

⟵ Détails d'analyse

Détails de conception ⟶

Produit
-num_produit: int -photo_produit -description_produit: String -prix: float -marge: int -/fraisDePort: float +estDisponible: boolean

Produit
-num_produit: int -photo_produit -description_produit: String -prix: float -marge: int -/fraisDePort: float +estDisponible: boolean
+majPrix() +majMarge()

Visibilité - Encapsulation

L'**encapsulation** est un principe de conception consistant à protéger le coeur d'un système des accès intempestifs venant de l'extérieur.

- La visibilité définit le degré de protection

  ≪ + ≫ : visibilité **publique** *i.e.* toutes les classe y ont accès (cas le plus fréquent pour les méthodes)

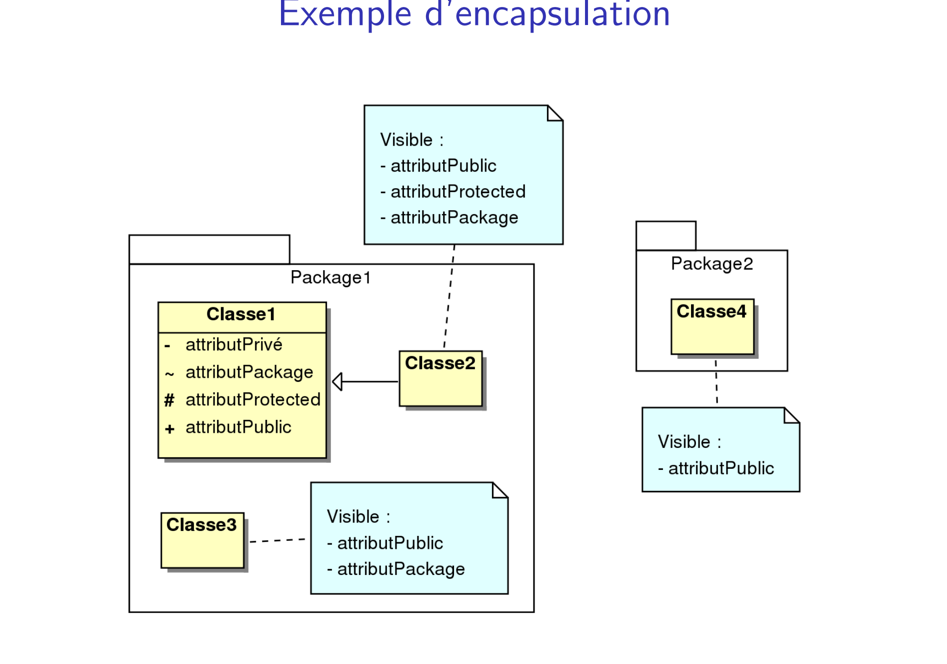
  ≪ - ≫ : visibilité **privée** *i.e.* inaccessible à tout objet hors de la classe (attributs et opérations internes à la classe)

  ≪ # ≫ : 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 ce service ("contrat")

Exemple d'encapsulation



Attributs et opérations de classe

- Souligné (`static` en `Java`)
- Attribut de classe : partagé par **toutes** les instances de la classe (garde une valeur unique)
- Opération de classe : sur des attributs de classe uniquement

Produit

-num_produit: int
-photo_produit
-description_produit: String
-prix: float
-marge: int
-/fraisDePort: float
+estDisponible: boolean

+majPrix()
+majMarge()

```java
import java.awt.Image;

public class Produit {
    private int numProduit;
    private Image photoProduit;
    private String descriptionProduit;
    private float prix;
    private static int marge;
    private float fraisDePort;
    private boolean estDisponible;

    public void majPrix(float prix) {
        if (prix > 0)
            this.prix = prix;
    }

    public static void majMarge(int marge){
        if (marge > 0)
            Produit.marge = marge;
    }
}
```

Attributs dérivés

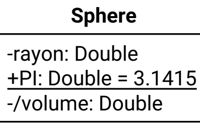
- peuvent être calculés à partir d'autres attributs et de formules de calcul
- **Exemples** :

`dateLocation` $\Rightarrow$ `dateRendue`

Livre

-nom: String
-dateLocation: Date
-/dateLimiteRendu: Date
-/nonRendu: boolean

π, `rayon` $\Rightarrow$ `Volume`



- peu utilisés

Les classes *in a nutshell*

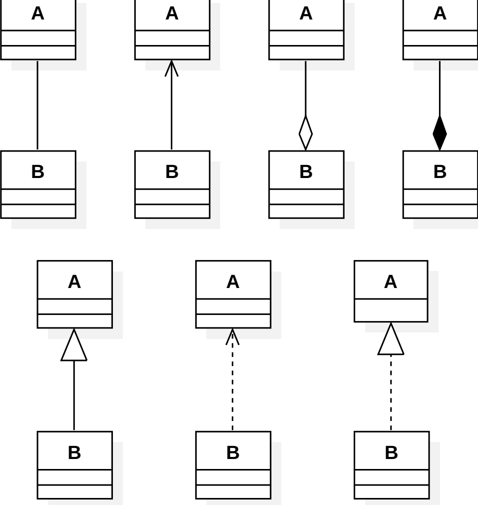
Encapsulation

- Rassembler les attributs et les méthodes en masquant les détails d'implémentation
- Garantir l'intégrité des données de l'objet
- Peut être réalisée en particulier par des 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.

Relations entre classes



Révisez (et respectez) bien les notations ! ! !

Relations entre classes - associations

Association

- Relation entre plusieurs classes.
- Représente une abstraction des différents liens qui peuvent exister entre les objets.

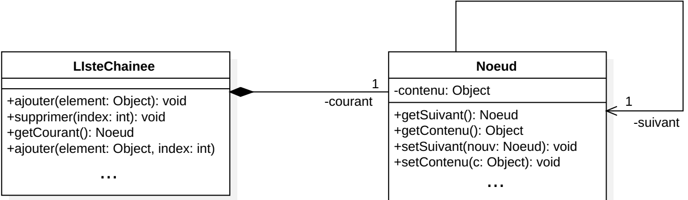
Agrégation

- Forme particulière d'association entre deux classes.
- Est une association non symétrique.
- Représente une inclusion (au sens large du terme).

Composition

- La composition est une agrégation forte (agrégation par valeur).
- Exprime une contenance structurelle entre des instances.
- Dépendance des cycles de vie : la destruction de l'agrégat implique la destruction de l'agrégé.

Relations entre classes - associations




```java
public class ListeChainee {
    private Noeud courant;

    /*constructeur - souvent implicite dans le diagramme*/
    public ListeChainee() { ... }

    public void ajouter(Object element) { ... }

    public void supprimer(int index) { ... }

    public Noeud getCourant() { ... }

    public void ajouter(Object element, int index) { ... }

    /* d'autres attributs et méthodes */
}
```

```java
public class Noeud {
    private Noeud suivant;
    private Object contenu;

    /* constructeur */
    public Noeud(...) { ... }

    public Noeud getSuivant() { ... }

    public Object getContenu() { ... }

    public void setSuivant(Noeud nouv) { ... }

    public void setContenu(Object c) { ... }

    /* d'autres attributs et méthodes */
}
```

Relations entre classes - classe-association

Personne
-estMarie: boolean
-estSansEmploi: boolean
-dateDeNaissance: Date
-/age: int
-prenom: String
-nom: String
+getNom()
+afficherEmployeurs()

-gerant 1 — -compagniesGerees *
* -employes — -employeurs *

Entreprise
-nom: String
-capitalisation: float
+afficherEmployés()
+getNom()

Emploi
-intitulePoste: String
-dateDebut: Date
-salaire: float
+getPersonne()
+getEntreprise()
+getIntitulePoste()

```java
public class Entreprise {
  /* Les attributs "triviaux" */
   private Personne gerant;
   private ArrayList<Emploi> employes;

   public String getNom() {
     return nom;
   }

public void afficherEmployes(){
  for(Emploi e : employes)
     System.out.println(e.getPersonne().getNom());
   }
}
```

```java
public class Personne {
   /* Les attributs "triviaux" */

    private ArrayList<Entreprise> compagniesGerees;
    private ArrayList<Emploi> employeurs;

    public String getNom(){
      return nom;
    }

    public void afficherEmployeurs(){
       for(Emploi e : employeurs)
          System.out.println(e.getEntreprise().getNom());
    }
}
```

```java
public class Emploi {
   /* Les attributs "triviaux" */
    private Personne p;
    private Entreprise e;

    public Entreprise getEntreprise(){
         return e;
    }
    public Personne getPersonne(){
         return p;
    }
    public String getIntitulePoste(){
         return intitulePoste;
     }
}
```

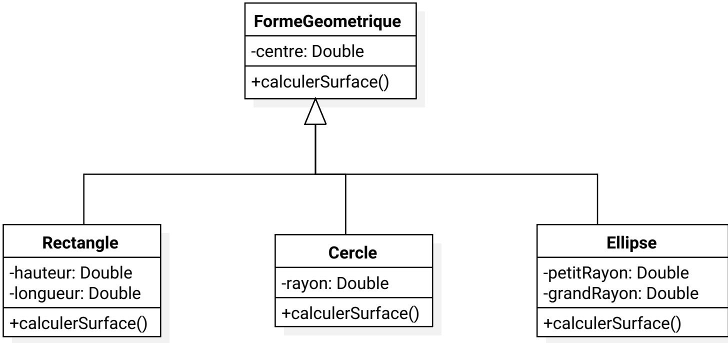
Relations entre classes - Généralisation

Spécialisation et généralisation : Le mécanisme qui permet de définir de nouvelles classes à partir de classes déjà existantes.

- **Généralisation** : factorisation dans une classe (appelée super-classe) de propriétés de plusieurs classes
- **Spécialisation** : inverse de la généralisation, consiste à créer à partir d'une classe plusieurs classes spécialisées.

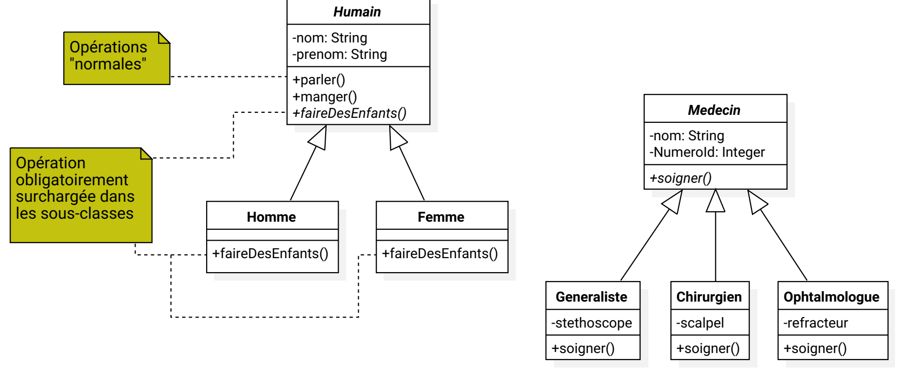
Relations entre classes - Généralisation

- indique qu'une classe est un cas plus générale d'une autre
- se traduit par la notion d'**héritage** en `Java`
- transitive et non-réflexive
- généralisations multiples (pas en `Java`)



Classe abstraite

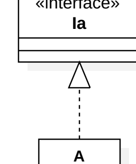
- Classe qui ne peut pas être instanciée
- Sert uniquement de super classe à d'autres classes
- Exemple : les hommes et les femmes vs *humains*
- Peut contenir des méthodes abstraites (pour garantir que toutes les classes filles aient ces méthodes)



Interfaces

- Classe ≈ nom + données + opérations (privées, publiques etc.)
- Souvent on s'intéresse uniquement aux *services* rendus par la classe *i.e.* à son **interface**

- Une interface est définie comme une classe, avec les mêmes compartiments.
- On dit que `A` est une **réalisation** de l'interface `Ia`
- Équivalente à une classe abstraite pure en `C++`



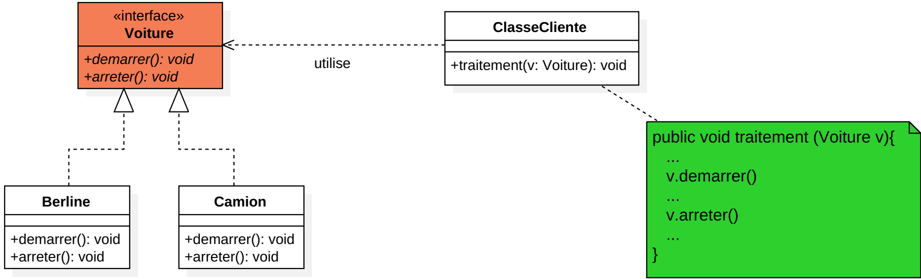
Classe abstraite vs Interface

Une interface c'est...

- *une sorte de* classe abstraite :
  - pas d'attribut (sauf les attributs `static final`)
  - pas de méthodes concrètes
  - en gros, rien n'est concret
- un ensemble de services (méthodes) rendus par les classes qui l'implémentent

Classe cliente 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.

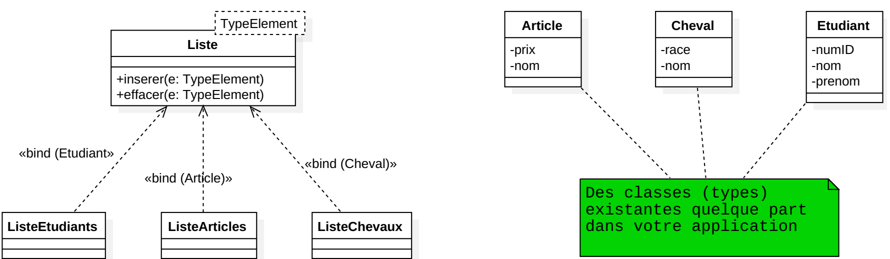


- Toute classe implémentant l'interface pourra être utilisée - principe de substitution

Classe paramétrée

Généricité

En Orienté Objet, correspond à la possibilité de paramétrer une Classe, une fonction, ou une méthode d'une classe par un paramètre générique formel représentant un type arbitraire.



Une classe paramétrée est essentiellement utilisée pour représenter une "structure de données"

Diagrammes de classes - Exemple

Etudiant
-numeroEt : Integer
Professeur
-numen: Integer
Adhérent
#nom : String
#prenom : String
#adresse : String
#email : String
#dateAdhesion: Date
+emprunterLivre(): void
+retournerLivre(): void
1..*
Bibliothèque
+ajouterAdherent(): void
+enleverAdherent(): void
+ajouterDocument(): void
1..*
*
+consulte
*
Document
-titre: String
-numero: Integer
1
Emprunt
-dateEmprunt: Date
-dateRetour : Date
+prolongerDateRetour(): void
0..4
Volume
-auteur : String
-année: Integer
-disponible: Boolean
Journal
-dateParution : Date
Livre
-genreLitteraire: String
Dictionnaire
-sujet: String
Manuel
-niveauEtude: String
-matiere: String

Les objets sont vivants !

- les diagrammes de classes représentent le modèle statique
- le cycle de vie des objets n'y apparaît pas
- un bon diagramme de classes permet d'anticiper les problèmes dus aux aléas des aspects dynamiques...
- ... mais le programmeur à encore pas mal de travail !