Cours

Systèmes de Gestion des Bases Données Avancées

Master 1, ISI.
2023-2024

Dilekh Tahar

tahar.dilekh@univ-batna2.dz

Département d'informatique. Faculté des mathématiques et informatique. Université Batna 2

BASES DE DONNEES ORIENTEES OBJETS

Plan de cours

- Evolution des applications et des SGBD
- Structure de données (complexe)
- Identité
- Lien de composition
- Hiérarchie de généralisation / spécialisation
- Méthodes et encapsulation
- Population et persistance
- ODL : Object Definition Language
- OQL : Object Query Language

I. Evolution des applications et des SGBD

Nouvelles applications

- conception assistée par ordinateur
- production assistée par ordinateur
- génie logiciel
- systèmes d'informations géographiques
- systèmes multimédia
- recherche et intégration de données de la toile
- …

Nouveaux besoins

- objets structurés, volumineux
- nouveaux types de données
- transactions longues
- …

I. Evolution des applications et des SGBD

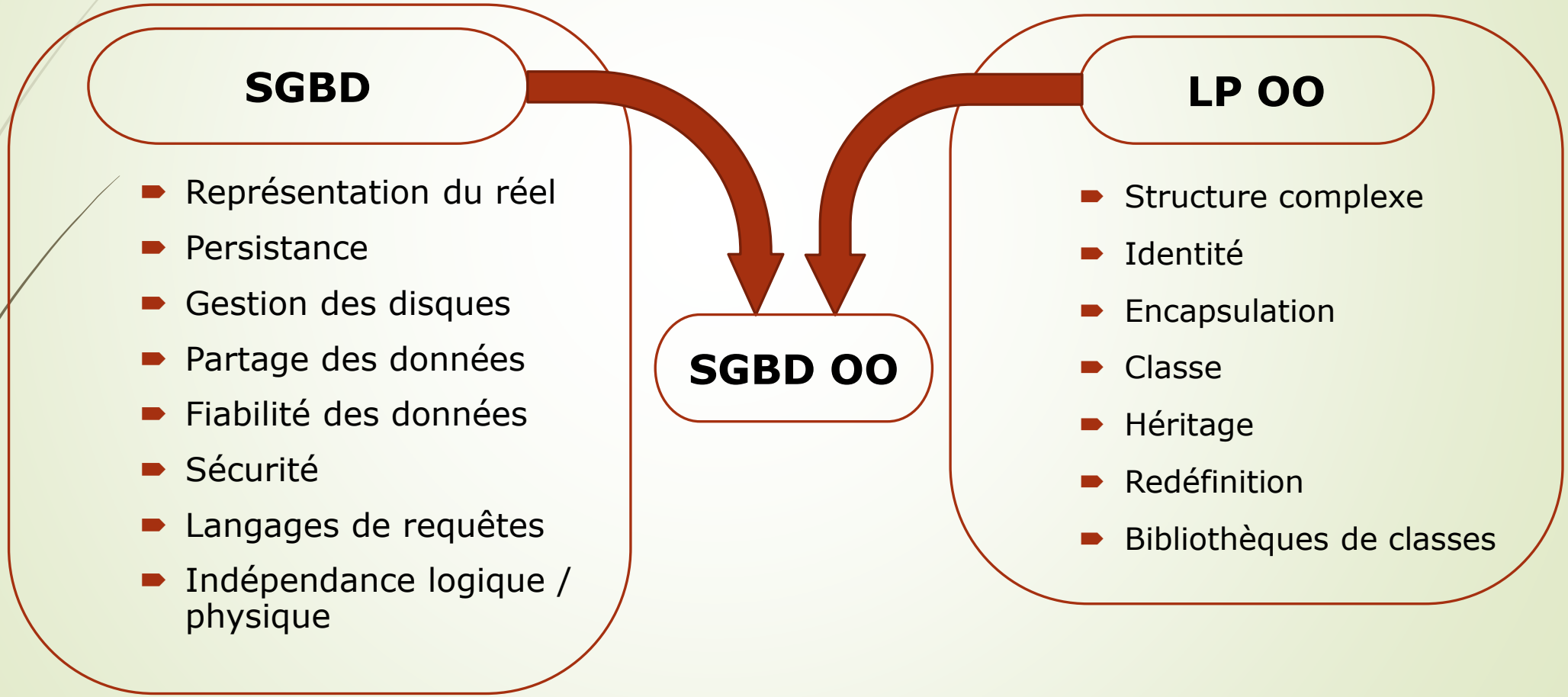
Approche Orienté Objet

- Ensemble de méthodologies et d'outils pour concevoir et réaliser des logiciels structurés et réutilisables, par composition d'éléments indépendants.
- Objectif : productivité des programmeurs
- Concepts essentiels
  - Langages de programmation OO

    C++, Java, Python …
  - L'encapsulation
    - interface visible : opérations (méthodes)
    - implémentation cachée : structure et code
  - Héritage

I. Evolution des applications et des SGBD

SGBD OO = LPOO + BD



I. Evolution des applications et des SGBD

Avantages des SGBDs par rapport aux LPs OO

- persistance des données
- indépendance modèles logique et physique
- LMD déclaratif
- optimisation par le SGBD
- intégrité des données : l'exactitude, l'exhaustivité, la cohérence, la sûreté
- confidentialité, fiabilité, concurrence, gestion de transactions, …

I. Evolution des applications et des SGBD

Avantages des SGBDs OO par rapport aux SGBDs R

- permet la manipulation d'objets à structure complexe
- interface compatible avec les LP-OO
- nouveaux types de données (image, son…)
- nouvelles transactions
- …

I. Evolution des applications et des SGBD

Diversité des modèles

- Norme ODMG (1993): est une spécification d'interfaces pour SGBDO, mais de nombreux SGBDO ne la suivent pas.
- SQL3 : est une norme pour les SGBD relationnels-objets, (définie par ISO)
  - SQL3 : est une extension du SQL relationnel aux fonctionnalités orientées objet et à d'autres fonctionnalités

I. Evolution des applications et des SGBD

Caractéristiques des SGBDOO

Les SGBDOO possèdent certaines caractéristiques propres au modèle objet qui les distinguent des autres applications.

- Structure de données
- l'identité ;
- l'encapsulation ;
- Les types et les classes,
- l'héritage,
- l'agrégation,
- le polymorphisme et l'extensibilité.

II. Structure de données

Concepts principaux

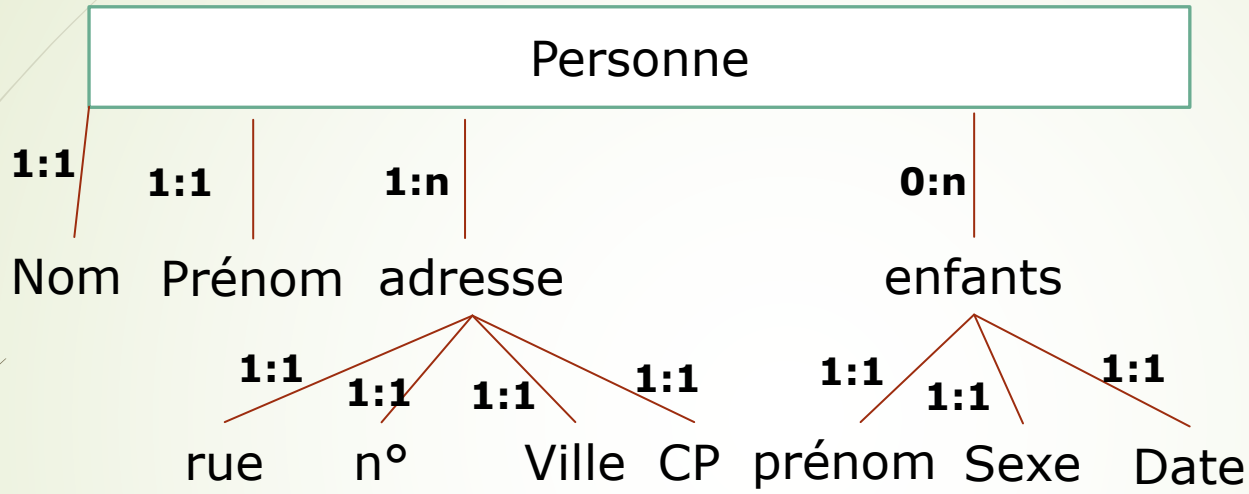
- **objet** –> objet, classe d'objets
- **propriété** –> attribut
  méthode
- **lien** –> lien de composition

Représentation multiple –> hiérarchie de généralisation/
spécialisation, héritage

II. Structure de données (complexe)

- **Objectif** : représentation directe des objets du monde réel
- **Exemple** : Personne (monde réel) un objet représenté par :
  - nom,
  - prénoms,
  - adresse (rue, n°, ville, codeP)
  - et enfants (prénoms, sexe, dateNais).

II. Structure de données (complexe)



- En **relationnel** : 2 relations, N tuples
  - Personne (<u>n°</u>, nom, prénom, adresse_rue, adresse_n°, adresse_ville, adresse_codeP)
  - Personne_enfant (<u>n°P, n°enfant</u>, prénom, sexe, dateNais)

II. Structure de données (complexe)

- **Mais En OO** : un seul objet
- CLASS **Personne** // class personne en ODMG (ODL)

```
{   ATTRIBUTE STRING  nom ;
    ATTRIBUTE STRING prénom ;
    ATTRIBUTE adresse : STRUCT
        {   STRING rue,
            INT n°,
            STRING ville,
            INT CP}
    ATTRIBUTE enfants : LIST STRUCT
        {   STRING prénom,
            ENUM {"M", "F"} sexe,
            DATE date}
}
```

II. Structure de données (complexe)

définir des classes Constructeurs de structure complexe :

- attribut complexe : STRUCT
  - attribut multivalué => constructeur de collection
    - ensemble : SET
    - liste : LIST
    - multi-ensemble : BAG
    - tableau à une dimension : ARRAY
  - Impact sur le SGBD :
    - LMD : comment accéder aux valeurs ?
      - notation pointée
      - variables sur les attributs multivalués
    - stockage d'objets complexes, gros, de taille variable

II. Structure de données (complexe)

Types définis par l'application

- définir des types de données adaptés à l'application
  - type T-Adresse
  - types Point, Ligne, Polygone
  - types Image, Son …
- Comme les classes d'objets, les types de données définis par l'application ont :
  - une structure complexe
  - des opérations (méthodes)

RI : 2 types d'attributs :

- attribut valeur (domaine = STRING, INT… ou complexe)
- attribut référence (domaine = une classe d'objets)

II. Structure de données (complexe)

Types définis par l'application

Types de données - Exemple

```
TYPEDEF T-Adresse STRUCT
{   STRING rue ,
    STRING n°,
    STRING ville,
    INT CP }
```

```
CLASS Personne {
    ATTRIBUTE nom : STRING ,
    ATTRIBUTE prénom : STRING ,
    ATTRIBUTE adresse : T-Adresse ,
    ATTRIBUTE enfants : LIST STRUCT enfant
{       STRING prénoms,
        sexe ENUM {'M', 'F'} ,
        DATE date } }
```

III. Identité en orienté objet

L'identifiant (OID - Object Identifier ou PID Persistent Identifier) permet de distinguer chaque objet de la base.

Objectif : Identifier les objets indépendamment de leur valeur et de leur adresse.

- L'identifiant (**oid** ou la référence de l'objet) est **géré par le système**.

- Un **objet** est un couple (oid, valeur) tel que:

(oid1, [nom: N, prénom: P, Enfants: {E1, E2, E3}])

III. Identité en orienté objet

- Cet identité doit être
  - **permanent** : existe toute la durée de vie de l'objet
  - **fixe** : ne change pas durant la vie de l'objet
  - **unique** dans la base et dans le temps

- Intérêt de l'identité d'objet
  - les **caractéristiques** de l'identifiant permettent à plusieurs objets de **partager** le même objet composant.
  - Moyen efficace pour référencer un objet

III. Identité en orienté objet

Teste d'égalité (Trois test d'égalité !)

L'égalité de deux objets, o1 et o2, peut se mesurer à :

1. **l'égalité d'objets,** notée : **o1 == o2**, et qui **signifie** : le même objet (le même **oid**)

2. **l'égalité de valeur (l'égalité de surface),** notée: **o1 = o2**, et qui **signifie** : les attributs des objets o1 et o2 ont la **même valeur**, que ce soient des **attributs-valeur** ou des **attributs-référence**.

III. Identité en orienté objet

Teste d'égalité (Trois test d'égalité !)

L'égalité de deux objets, o1 et o2, peut se mesurer à :

3. **l'égalité de valeur après fermeture (l'égalité de profondeur),** notée : **o1 = f o2 ,** et qui **signifie** : les objets o1 et o2 ont la même "valeur après fermeture, **c'est-à-dire**:
   1. Les attributs-référence des objets o1 et o2 constituent des graphes équivalents, et
   2. les attributs-valeur correspondants, à tous les niveaux, sont égaux.

les propriétés suivantes sont vérifiées :

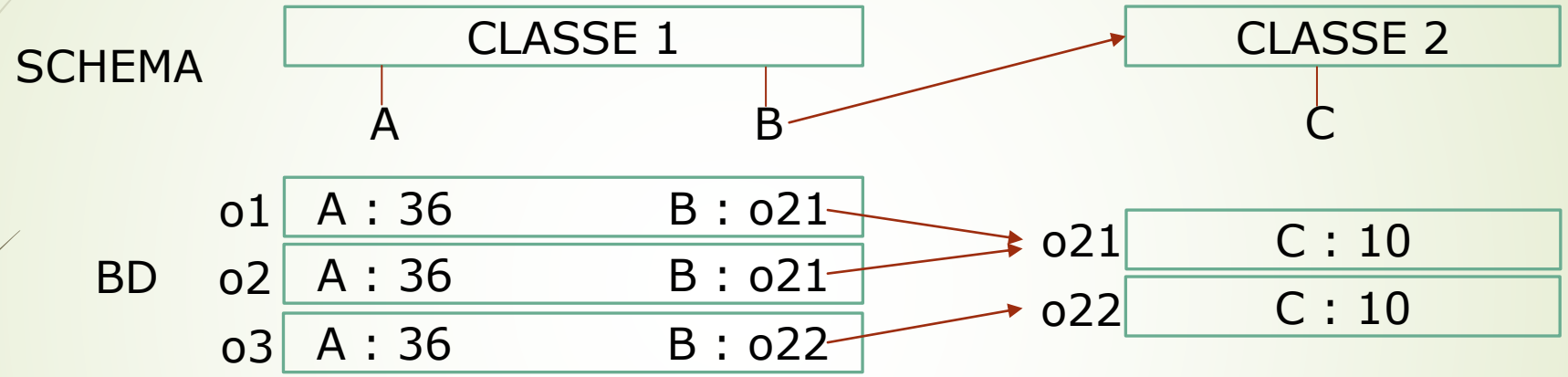
obj1 == obj2 obj1 = obj2

obj1 = obj2 obj1 = f obj2

- **RI** : 2 types d'attributs :
  - attribut valeur (domaine = STRING, INT… ou complexe)
  - attribut référence (domaine = une classe d'objets)

III. Identité en orienté objet

Teste d'égalité (Exemple)



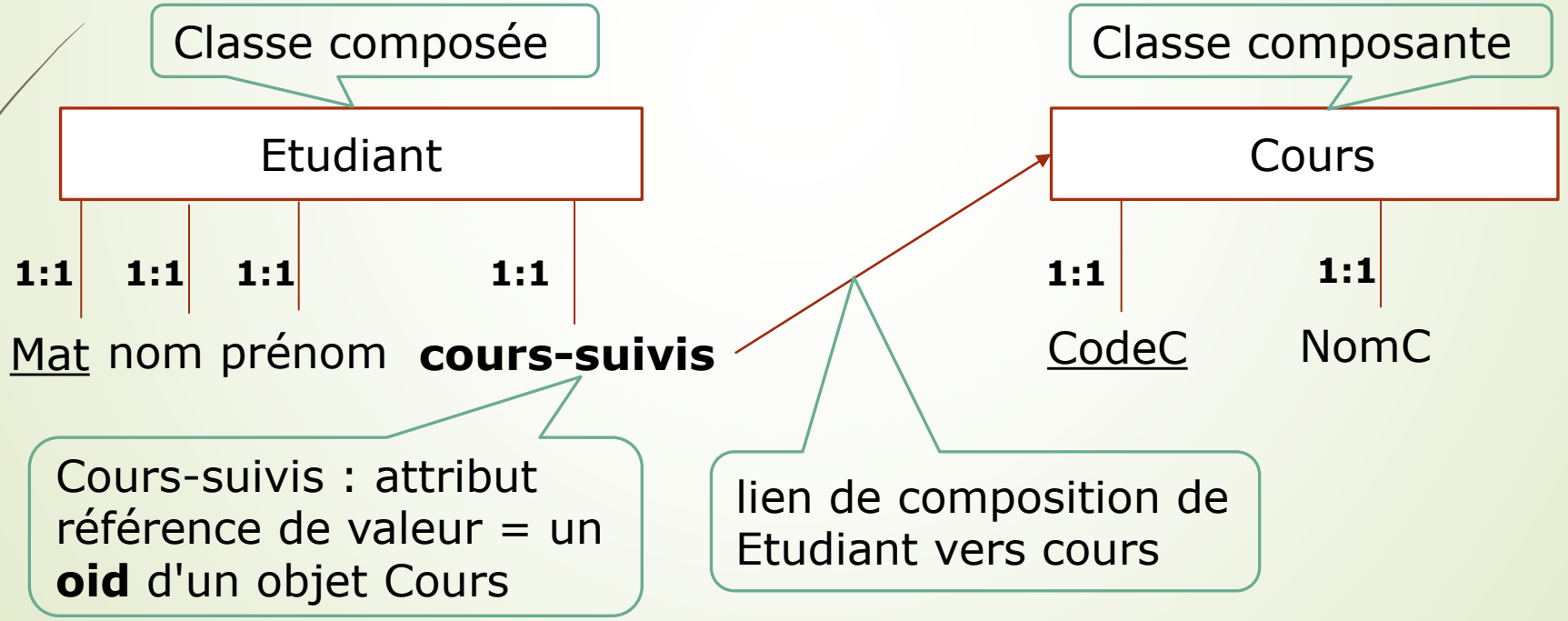
Nous avons alors:

- (o1.B == o2.B) vrai et (o1 = o2) vrai , mais (o1 == o2) faux
- (o1 = o3) faux mais (o1 =f o3) vrai
- (o21 = o22) vrai mais (o21 == o22) faux

IV.Liens de composition

Objectif : représenter les liens de composition qui existent entre objets du monde réel.

Un **lien de composition** relie une classe C1 à une classe C2 si les objets de C1 (appelés "**objets composés**") sont composés d'objets C2 (appelés "**objets composants**").



IV.Liens de composition

```
CLASS Etudiant {
    INT Mat;
    STRING nom;
    LIST STRING prénoms;
    cours-suivis : SET STRUCT {
          cours : Cours ;
          note : FLOAT }
    }
```

```
CLASS Cours {
     STRING CodeC ,
     NomC : STRING ,
     … }
```

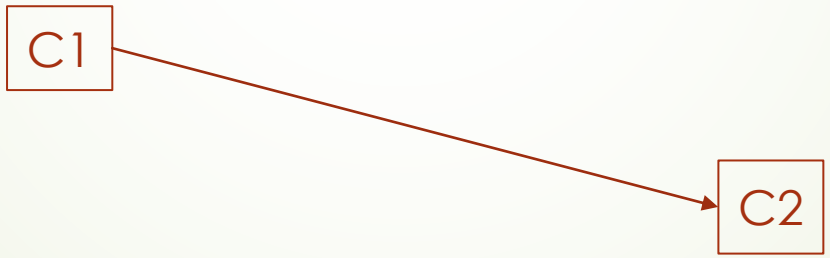
- Les **liens de composition** sont implantés par des **attributs-référence** qui pointent sur les **objets composants**. Leur valeur est l'**oid** de l'objet référencé.
- On peut avoir des **objets partagés** qui sont composants de plusieurs objets composés.

IV.Liens de composition : contraintes d'intégrité

- Objet composant peut être : **partagé / non partagé** entre plusieurs objets de la même classe composée.
  - Une classe d'objets composants C2 partagés d'une classe C1, si **deux** (ou plus) objets de C1 peuvent partager le même objet de C2.
- Objet composant peut être : **dépendant / non dépendant** de son (d'un de ses) objet de la classe composée.
  - l'existence de l'objet composant dépend de celle l'objet composé (c.-à-d. destruction composite => destruction composant)
- Certains SGBDO assurent la contrainte inverse : l'objet **composé dépend** de ses **composants**.
- Les contraintes de dépendance définissent l'ordre selon lequel les objets doivent être créés (et détruits)

IV.Liens de composition : contraintes d'intégrité

- Parfois des cardinalités peuvent être associées au lien de composition.
- Par exemple, la contrainte "C2 est une classe composante non partagée de la classe C1"
  - s'exprimer de la façon suivante : "la cardinalité maximale du lien de composition **C1-C2** du côté de **C2** est **1**".



IV.Liens de composition : contraintes d'intégrité

Lien inverse

- Le lien de référence est orienté de la classe composée vers la classe composante.
  - Cela implique que, dans les requêtes, il est beaucoup plus facile et rapide d'aller de l'objet composé aux objets composants que l'**inverse**.
- Pour avoir un accès aisé dans les deux sens dans la base de données, il est utile de décrire le lien de composition et son **inverse**.
- **Exemple** :

```
CLASS Cours {
    CodeC : STRING ,
    NomC : STRING ,
    Etudiants set Etudiant }
```

Etudiants est un **attribut référence inverse** de l'attribut cours-suivis de la classe Etudiant

IV.Liens de composition : contraintes d'intégrité

Lien inverse

- Certains systèmes permettent de déclarer l'existence des attributs inverses par une clause "**INVERSE**". L'exemple s'écrit alors:

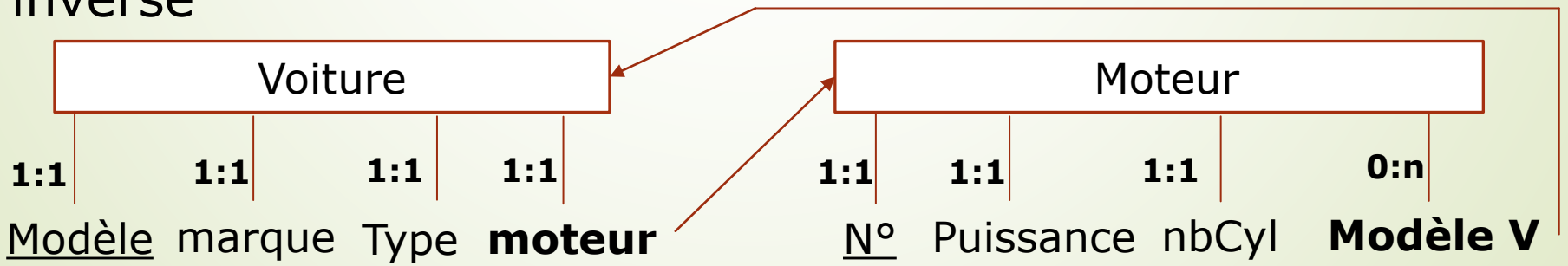
- **Exemple** :

```
CLASS Cours {
    CodeC : STRING ,
    NomC : STRING ,
    Etudiants set Etudiant INVERSE Etudiant.cours-suivis.cours}
```

IV.Liens de composition : contraintes d'intégrité

Contraintes de composition

- objet composant : partagé / non partagé
- objet composant : dépendant / non dépendant
  - destruction composite => destruction composant
- cardinalités :
  - minimale, maximale
  - inverses (=> partagé / dépendant)
- lien inverse

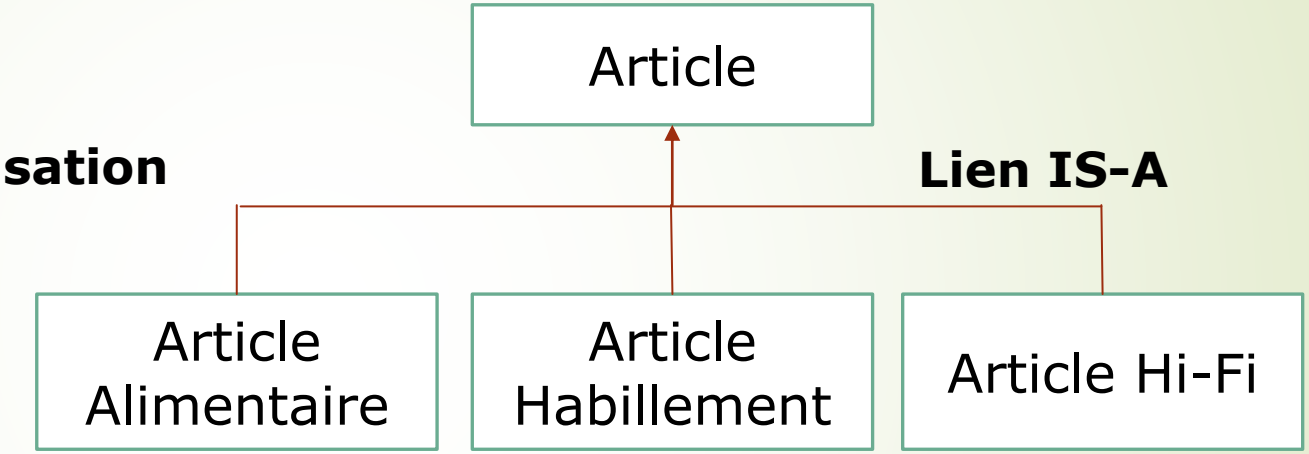


V. Hiérarchie de Généralisation/Spécialisation

Classe Générique

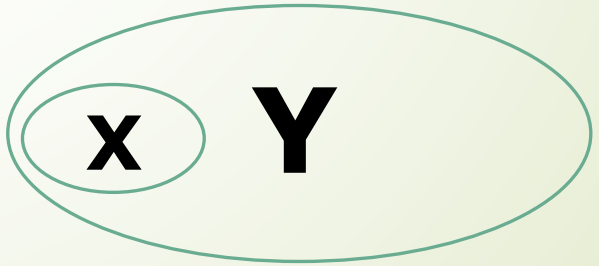
Spécialisation | **Généralisation**

Classe Spécifiques



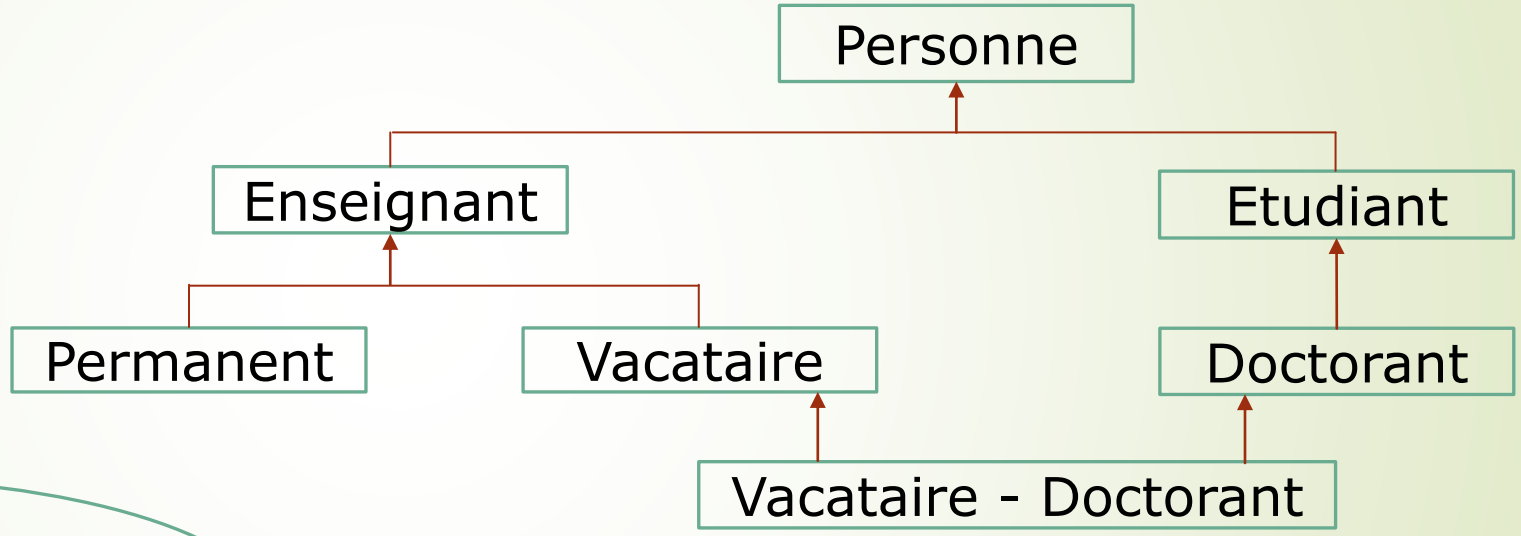
- X Est un Y
- X sous-type de Y
- Y sur-type de X

Inclusion de populations :
tout X est un Y

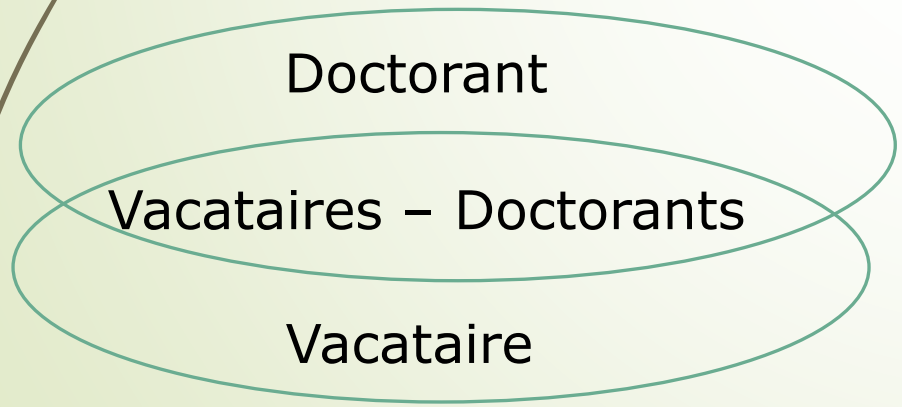


V. Hiérarchie de Généralisation/Spécialisation

L'héritage multiple



Populations



L'**héritage multiple** est possible: c'est le cas de la classe Assistant-Doctorant.

V. Hiérarchie de Généralisation/Spécialisation

Polymorphisme : appelé aussi **surcharge**

le polymorphisme permet **d'associer** un **code spécifique** à une **méthode héritée** qui garde le même nom.

Dans ce cas, un **conflit** peut survenir lors de l'héritage des attributs et des méthodes: deux attributs ou méthodes de deux sur-classes peuvent avoir le même nom.

Exemple en Java

```
public class A {
    void f() {
        code 1;
    }
};
public class B extends public A {
    void f() {
        code 2;
    }
};
```

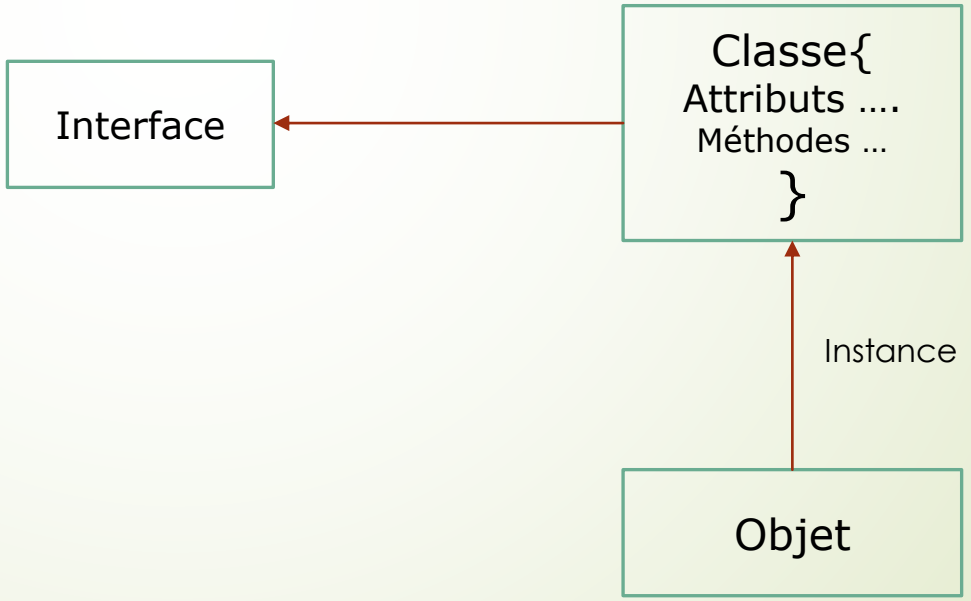
Par exemple, pour **Assistant-Doctorant**, quelle est la méthode « afficher » qu'il faut utiliser? Celle de Enseignant ou celle de Etudiant?

Pour **résoudre** ce **conflit**, les SGBDO peuvent:

1. soit ils refusent la définition d'un tel schéma;
2. soit ils demandent au concepteur de choisir la sur-classe «**dominante**» dont la sous-classe héritera;
3. soit ils appliquent une **règle** (par exemple la première classe citée dans la clause **Is-a**).

VI.Méthodes et encapsulation (la dynamique)

- C'est la **dynamique** des valeurs du **contenu caché** d'un objet. Les valeurs des attributs d'un objet peuvent être changés par l'utilisateur en passant par une **interface** contenant les méthodes qui manipules ces attributs.
- C'est le principe de **l'encapsulation** où la **structure** des objets et le **code** des méthodes sont **cachés** à l'utilisateur, et seules les **méthodes** définies dans **l'interface** sont **visibles** par l'utilisateur.



VI.Méthodes et encapsulation (la dynamique)

La déclaration d'une classe d'objets est séparée en 2 parties:

- **L'interface** qui est **visible** aux utilisateurs : définition de la **signature de chaque méthode** (**nom** de la méthode, liste des **paramètres** d'appel avec leur type, **type** du **résultat** s'il en existe);

- **L'implantation**, qui est **invisible** aux utilisateurs: description des **types** des attributs, et du **corps** de chaque méthode.

VI.Méthodes et encapsulation (la dynamique)

Différents **degrés d'encapsulation** sont définis:

- **Encapsulation stricte**: tout accès aux données d'un objet d'une autre classe se fait par appel à une méthode de la classe de l'objet;
- **Encapsulation des écritures**: tout accès en écriture se fait par appel à une méthode, les accès en lecture peuvent être faits directement;
- **Encapsulation partielle**: les données publiques sont accessibles directement, et les données privées sont accessibles uniquement par appel aux méthodes.

VI.Méthodes et encapsulation (la dynamique)

- La **surcharge** des méthodes dans les sous-classes permet de donner naissance au principe de **liaison dynamique**.
- Les SGBDO utilisent le mécanisme de **liaison dynamique** pour le choix du corps des méthodes dans le cas de redéfinition de méthodes dans une sous-classe.
- Exemple

```
A Personne;
B Enseignant;
A.afficher(); //Liaison dynamique vers la méthode
                 afficher de la classe Personne
A = B;
A.afficher(); // Liaison dynamique vers la méthode
                 afficher de la classe Enseignant
```

VI.Méthodes et encapsulation (la dynamique)

Exemple d'encapsulation CLASS Personnel

Interface visible

- INT salaire()
- VOID newService(servoid : Service)
- VOID afficher()

Signatures des méthodes

Implémentation invisible

- ATTRIBUTE STRING AVS ;
- ATTRIBUTE STRING nom ;
- ATTRIBUTE STRING adresse ;
- ATTRIBUTE INT sal_mensuel ;
- RELATIONSHIP service : Service INVERSE Service.personnes

structure des données

VI.Méthodes et encapsulation (la dynamique)

Exemple d'encapsulation : CLASS Personnel

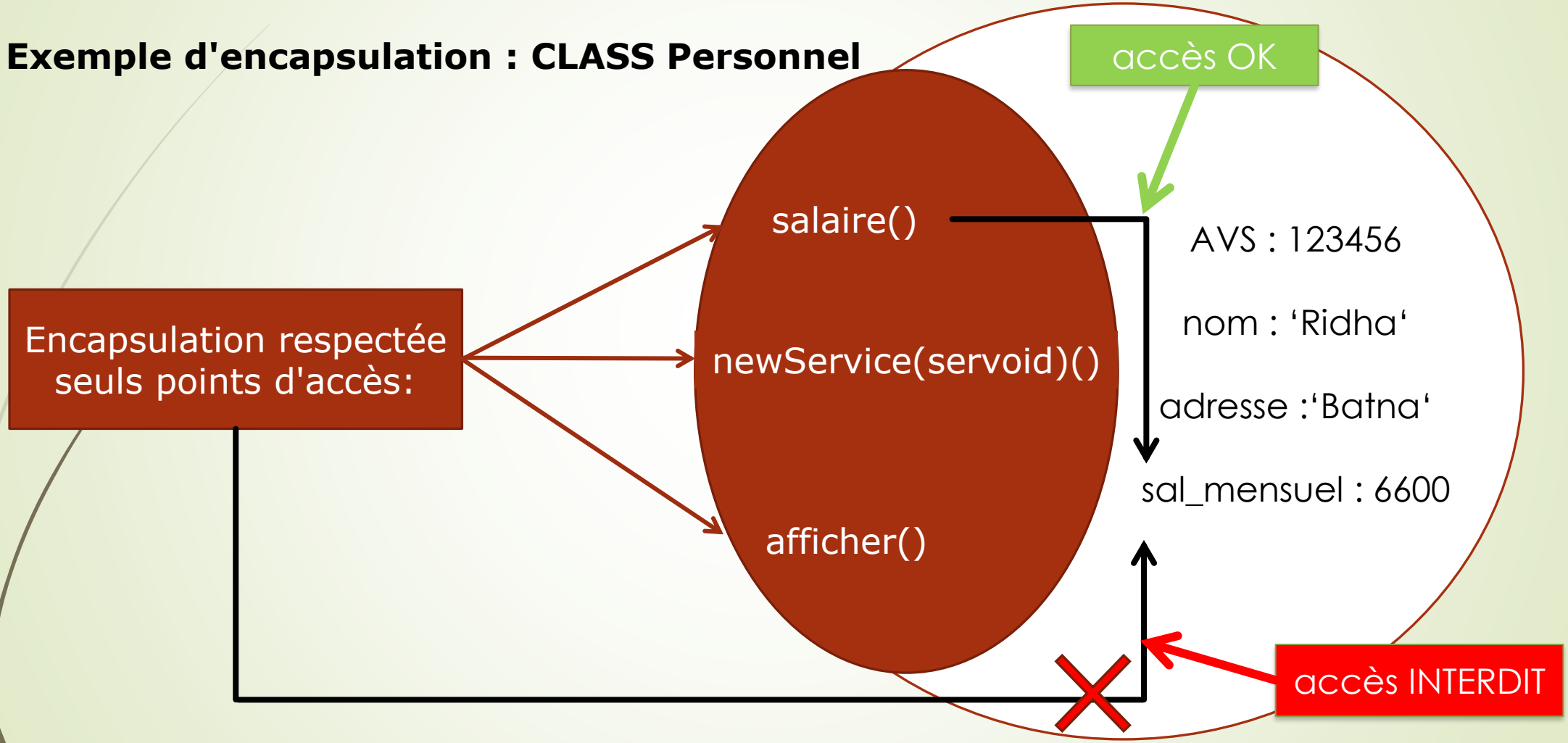
- Implémentation invisible (suite) : code des méthodes

```
salaire ()
    { return sal_ mensuel }
newService (servoid: Service)
    { self.service := servoid }
afficher ()
    {   PRINT('AVS:', self.AVS) ;
        PRINT('nom:', self.nom) ;
        PRINT('adresse:', self.adresse) ;
        PRINT('salaire mensuel:', self.sal_mensuel) ;
    }
```

Encapsulation : seul l'objet lui-même (c-à-d les instructions de ses méthodes) peut accéder à ses attributs

VI.Méthodes et encapsulation (la dynamique)

Exemple d'encapsulation : CLASS Personnel



ODL : **O**bject **D**efinition **L**anguage

Introduction

Définition d'interface (Interface definiton)

- Spécification du **comportement** observable par les utilisateurs (ou d'une partie de celui-là) pour **un type d'objets.**

```
INTERFACE CALCULATEUR {
        CLEAR();
        FLOAT ADD(IN FLOAT OPERAND);
        FLOAT SUBSTRACT (IN FLOAT OPERAND);
        FLOAT DIVIDE(IN FLOAT DIVISOR);
        FLOAT MULTIPLY (IN FLOAT MULTIPLIER);
        FLOAT TOTAL();
}
```

Introduction

interface = spécification d ’un type

- super (Héritages simple et multiple)
- extent, clés candidates
- Attributs

  attribute <type> <nomattr>;

- associations et associations inverses

  relationship <type> <nomasso> inverse <nom-d-interface>::<nomasso>;

- Méthodes

  <type-retourné> <nommeth> (<type-paramêtre> : <type>, ...) raise (<type-d-exception>);

:<type-paramêtre> : in, out, inout

Introduction

Définition de classe (Class definiton)

- Spécification du **comportement** et d'un **état** observables par les utilisateurs pour un **type d'objets**.

Définition d'Extension de classe (Class extent)

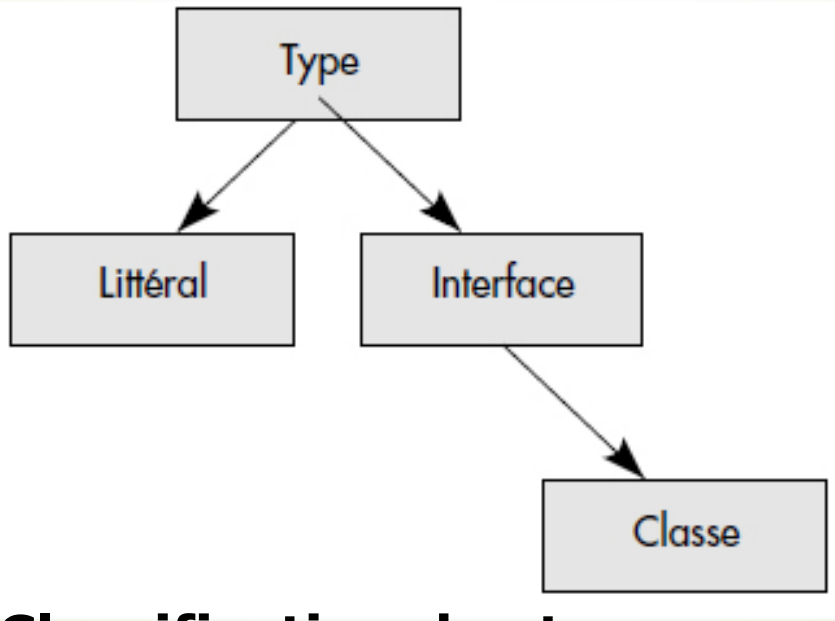
- Collection caractérisée par un nom contenant les objets créés dans la classe

```
CLASS ORDINATEUR (EXTENT ORDINATEURS KEY ID) : CALCULATEUR {
        ATTRIBUTE SHORT ID ;
        ATTRIBUTE FLOAT ACCUMULATEUR;
        VOID START();
        VOID STOP();
}
```

Introduction

Définition de littéral (Literal définition)

- Spécification d'un type de valeur correspondant à un état abstrait, sans comportement. Exemple : entier, réel, chaîne de caractères, mais aussi aux structures.



Classification des types

Définition d'une classe

Syntaxe

Héritage d'interface (:) ou
de classe (Extends) ?

Class nom-classe1 (**Extent** nom-population, **Key** < clés>) {

Attribute < def-attribut> ;

Relationship nom-relation1 : [**List** / **Set** / **Bag** / **Array**] nom-classe2 [**Inverse** nom-classe2.nom-relation2] ;

<domaine-résultat> nom-méthode ([nom-paramètre : <domaine-paramètre> , ...]) ;

} ;

Définition d'un domaine

Syntaxe

Typedef nom-domaine

[**List** / **Set** / **Bag** / **Array**] <domaine-s/c>) ;

avec:

<domaine-s/c> :: <domaine-simple> / <domaine-complexe>

<domaine-simple> :: **String** / **Int** / **Float** / **Date**/ **Boolean** / **Enum** (…,valeur)

<domaine-complexe> :: **Struct** nom-domaine {<def-attribut>; …}

La structure complexe

Syntaxe

STRUCT (type_attr$_1$ nom_attr$_1$, … type_attr$_i$ nom_attr$_i$) ;

où " type_attr$_i$ " est le domaine de l'attribut de nom " nom_attr$_i$ "

Il est soit :

- un domaine prédéfini (STRING, FLOAT, INT, DATE...), ou
- un type défini par l'utilisateur, ou
- le nom d'une classe

Exemple sur la définition d'une classe

```
CLASS Etudiant (extent lesEtudiants  key num)      // déclaration d'une classe d'objets
{
attribute INT num ;                                 // attribut monovalué de type prédéfini
attribute STRING nom ;                              // attribut monovalué de type prédéfini
attribute LIST <STRING>  prénoms ;                  // attribut multivalué de type liste
attribute DATE date-nais ;                          // attribut monovalué de type prédéfini
attribute adresse : STRUCT                          // attribut monovalué complexe
        (INT num ,
        STRING rue ,
        STRING ville ,
        STRING pays  ) ;
attribute cours-suivis : SET STRUCT                 // attribut complexe et multivalué
        (STRING nom-cours ,
        FLOAT note ) ;
}
```

Exemple 2

```
TYPEDEF T_Adresse  STRUCT          // déclaration d'un nouveau type utilisateur complexe
        (INT num ,
        STRING rue ,
        STRING ville ,
        STRING pays  ) ;
CLASS Etudiant (extent lesEtudiants  key num)   // voir utilisation de EXTENT en section 2.1.1
{
attribute INT num ;
attribute STRING nom ;
attribute LIST <STRING>  prénoms ;
attribute DATE date-nais ;
attribute T_Adresse  adresse ;                  // attribut complexe de type défini par l'utilisateur
attribute cours-suivis : SET STRUCT
        (STRING nom-cours ,
        FLOAT note ) ;
}
```

Liens de composition

Exemple

```
CLASS Etudiant{                          // classe composée
attribute INT num ;
attribute STRING nom ;
attribute LIST <STRING> prénoms;
attribute cours-suivis : SET STRUCT
        (Cours cours ;                   // attribut référence de 2è niveau
         FLOAT note }
} ;
CLASS Cours{                             // classe composante
attribute STRING nomC ;
 ....
} ;
```

Version non autorisée en ODMG !!!

Liens de composition

Exemple

```
CLASS Etudiant{
attribute INT num ;
attribute STRING nom ;
attribute LIST <STRING> prénoms;
attribute SET <Cours-suivis> cours-suivis;
} ;
Class Cours-suivis {                    // classe intermédiaire
attribute att : Struct
        (Cours cours ;                  // attribut référence de 1er niveau
         FLOAT note  )
} ;
CLASS Cours{                            // classe composante
attribute STRING nomC ;
 ....
} ;
```

Version autorisée en ODMG !!!

Définition d'une association & lien Inverse

Exemple

```
class Enseignant (extent  lesEnseignants)
{
   attribute short Numbureau;
   relationship set<Etudiant> tuteur_de inverse Etudiant :: tutoré_par;
   ……
};
class Etudiant  (extent lesEtudiants  key numInscription)
{
   attribute string numInscription;
   relationship Enseignant tutoré_par inverse Enseignant :: tuteur_de;
   ……
};
```

Graphe G/s (l'héritage multiple))

Exemple

```
Interface Personne {
   attribute string nom;
   attribute string addresse;
   attribute date dateNais;
};
class Enseignant : Personne (extent  lesEnseignants) {
   attribute short numBureau;
   attribute statut : enum ("prof", "assistant")

   ....
};
class Etudiant  : Personne  (extent lesEtudiants) {
   attribute string numInscription;
   .....
};
```

OQL : Object Query Language

Introduction

- Langage de requêtes standard pour BD OO défini par ODMG
- Langage de requêtes uniquement
  - insertions, suppressions et mises à jour faites par des méthodes
- Syntaxe similaire à celle de SQL MAIS non compatible avec SQL

Objectifs :

- manipuler les éléments et les collections
- utiliser pleinement les concepts OO
  - structure complexe
  - hiérarchie de généralisation / spécialisation
  - méthodes

1. Points d'entrée, Requêtes et Résultats

En OQL, on peut donner à tout objet un ou plusieurs noms permanents, qui pourront server de point d'entrée dans la base.

Pour cela, il suffit de déclarer un nom grâce à l'instruction « **Name** »

Name directeur : **Personne** ; *variable permanente*

directeur = Personne (nom:'n', prénoms: LIST('p1', 'p2'), adresse: 'adr'); *Apres l'affectation : création d'un objet permanent nommé.*

1. Points d'entrée, Requêtes et Résultats

Requêtes

Exemples de requêtes :

Phil	objet (non imprimable)
Phil.nom	valeur (imprimable)
Phil.age()	appel de méthode de résultat valeur
LesEtudiants	collection d'objets

Select distinct e.nom

From e in Etudiants

where e.age() < 20;

Cette requête crée un **ensemble** de valeurs de type **SET**(STRING), contenant les noms des étudiants de moins de 20 ans, sans double.

1. Points d'entrée, Requêtes et Résultats

Select distinct struct (nom: c.nomC , prereq: (**Select** p.nomC
From p in c.a_prerequis))
From c in Cours ;

Cette requête crée un ensemble de valeurs de type:

SET(STRUCT(nom:STRING,
prereq:SET(STRING)),

Contenant pour chaque cours, son nom et l'ensemble des noms de ses cours pré-requis.

1. Points d'entrée, Requêtes et Résultats

Une requête sur une sur-classe fournit tous les objets de la sur-classe qui satisfont la condition de la requête, qu'ils appartiennent ou non à une sous-classe.

Exemple :

```
Select p
From p in Personne
Where p.nom='N' and p.prénom='P' ;
```

Cette requête crée un ensemble d'objets SET(Personne) contenant les personnes de nom N et de prénom P qu'ils soient **enseignants** ou **étudiants**, ou ni l'un ni l'autre.

2. Opérateurs sur les collections

Le format générale de **SELECT**:

SELECT [**DISTINCT**] <définition du résultat>
FROM variable1 **IN** collection1, …
[**WHERE** <condition>];

<définition du résultat> peut désigner :

1. **un objet/une collection d'objets**
   Exemple :
   SELECT p
   FROM p IN Enseignants
   SELECT p.cours_assurés
   FROM p IN Enseignants

2. **une valeur/collection de valeurs**
   Exemple :
   SELECT p.nom
   FROM p IN Enseignants
   SELECT p.prénoms
   FROM p IN Enseignants

3. **construit une valeur complexe**

Exemple :
SELECT STRUCT (nom:e.nom,
cours:e.cours_suivis ,
cours2: (**SELECT** c
FROM e.cours_suivis
WHERE c.cycle=2))
FROM e **IN** Etudiants
==> en résultat nous aurons
STRUCT (nom : STRING, cours :
SET(Cours), cours2 : **SET**(Cours))

2. Opérateurs sur les collections

Le format générale de **SELECT**:

SELECT [**DISTINCT**] <définition du résultat>
FROM variable$_1$ **IN** collection$_1$, …
[**WHERE** <condition>];

collection$_1$ peut être :

1. **collection quelconque de la BD**
   1. **population** :
      SELECT p **FROM** p **IN** Enseignants
      WHERE …
   2. **autre collection :**
      … **FROM** p **IN** Enseignants, c **IN** p.cours_assurés

2. **Requête :**
   Exemple : donner les noms des cours de cycle 2 suivis par E avec le nom du prof du cours
   SELECT STRUCT(nomcours : c.nomC, nomprof : c.prof.nom)
   FROM c **IN** (**SELECT** x **FROM** x **IN** E.cours_suivis
   WHERE x.cycle=2)

2. Opérateurs sur les collections

Le format générale de **SELECT**:

SELECT [**DISTINCT**] <définition du résultat>
FROM variable$_1$ **IN** collection$_1$, …
[**WHERE** <condition>];

condition peut être :

1. **Condition élémentaire**
   1. **Expr1 Opérateur_Com Expr2**:
      e.nom = ’ Mohamed’
      e.age() < 20
      COUNT(e.prénoms) > 2
   2. **ou avec un quantificateur :** $\exists$ *ou* $\forall$
      (voir diapos n°65, 66)
2. **condition AND condition**
3. **condition OR condition**

2. Opérateurs sur les collections

a. Quantificateur existentiel

L'expression:

EXISTS x IN <collection> : <condition(x)>

donne en résultat un booléen qui est vrai ssi **un élément au moins** de la collection satisfait la condition.

- **Exemple** : La requête suivante retrouve les enseignants qui donnent au moins un cours de 3ème cycle.

```
SELECT e
FROM e IN Enseignants
WHERE EXISTS c IN e.cours-assurés : c.cycle=3 ;
```

2. Opérateurs sur les collections

b. Quantificateur universel

L'expression:

FOR ALL x IN <collection> : <condition(x)>

donne en résultat un booléen qui est vrai ssi **tous les élément** de la collection satisfont la condition.

- **Exemple** : La requête suivante retrouve les enseignants qui ne donnent que des cours de 3ème cycle.

```
SELECT e
FROM e IN Enseignants
WHERE FOR ALL c IN e.cours-assurés : c.cycle=3 ;
```

3. Liens de composition

- Pour passer d'un objet à un de ses objets composants il suffit de traverser le liens de composition.
- **Exemple** : La recherche des noms des enseignants assurant les cours suivis par l'étudiant numéro 10 peut s'écrire:

```
Select distinct p.nom
from e in Etudiants, c in Cours, p in Enseignants
where e.num=10 and c in e.cours_suivis and p = c.prof;
```

ou bien:

```
Select distinct c.prof.nom
from e in Etudiants, c in e.cours_suivis
where e.num=10;
```

4. Fonctions d'agrégation

COUNT(collection)

MIN(collection)

MAX(collection)

AVG(collection)

SUM(collection)

- **Exemples** :
  1. **COUNT**(Etudiants) => nombre d'étudiants
  2. **COUNT**(**SELECT** p **FROM** p **IN** Personnes **WHERE** p.nom='N') => nombre de personnes ayant le nom 'N' dans la base

4. Fonctions d'agrégation

- **Exemples** :

3. Pour chaque étudiant, donner son nom, le nombre total de ses diplômes, le nombre de diplômes obtenus en 2003, et la première année où il a obtenu un diplôme.

```
SELECT STRUCT(nom : e.nom,
                    nbdiplomes : COUNT(e.diplomes),
                    nbdiplomes03 : COUNT(SELECT d
                                         FROM d IN e.diplomes
                                         WHERE d.année=2003),
                    premièreannée : MIN(
                                    SELECT d.année FROM d IN e.diplomes)
                    )
FROM e IN Etudiants
```

5. Instruction GROUP BY

- Permet de partitionner une collection en sous-groupes ayant même valeur pour certains attributs. Ces groupes sont appelés d'un nom prédéfini « **partition** »
- Syntaxe:

  GROUP variable **IN** collection

  BY (nom1: expression1 , …) critères de partition

  [**WITH** (nom'1: expression'1 , …)]

- Exemple :

  GROUP c **IN** Cours

  BY (cycle : c.cycle)

  WITH (nbcours : **COUNT**(**partition**)) ;

donne en résultat un ensemble de structures: SET (STRUCT(cycle:INT, nbcours:INT)) qui définissent pour chaque cycle le nombre de cours de ce cycle.

6. Transformation d'une collection en élément

- Si une collection **ne contient qu'un seul élément**, la fonction « **Element** » rend en réponse cet élément.

- Cette fonction est utile quand on sait que le résultat d'une requête **E** est un seul élément.

Element(<collection>) => <élément>

- **Attention**, si la collection comporte **plusieurs éléments**, l'instruction génère une **exception** lors de l'exécution.

7. Opérateurs ensemblistes

collection1 **UNION** collection2 (l'union)

collection1 **EXCEPT** collection2 (la différence)

collection1 **INTERSECT** collection2 (l'intersection)

- Les éléments **doivent** être de **types compatibles**:
  - Soit de même type
  - Soit sur-type commun -> comparaison sur la partie commune
- **Exemple** :

A.cours-suivis

UNION

(**SELECT** c

FROM e **IN** Etudiants, c **IN** e.cours_suivis

WHERE e.nom = 'N' **AND** e.prénoms = **LIST**('M'))

Cours suivis par A ou (inclusif) cours suivis par l'étudiant de nom 'N' et de prénom 'M'.

Questions ?