Master SRI
(Systèmes de recherche d'informations)

Bases de données orientées objets

http://www.larbiguezouli.com/

ou

http://fac-sciences.univ-batna.dz/cs/enseignants/guezouli_larbi_site/index.html

Présenté par Dr Larbi GUEZOULI

Sommaire

Chapitre I
Introduction

1. Différents type d'SGBDs
   a) Bases de données hiérarchiques
   b) Bases de données réseaux
   c) Bases de données relationnelles
2. Limites des SGBD
   a) Avantages
   b) Inconvénients
3. Nouveaux besoins
4. Caractéristiques des BDOO

Chapitre I
Introduction

Dans les **systèmes de recherche d'informations** on manipule des données **complexes** comme: texte, image, graphique, audio, vidéo…

Une telle quantité d'informations complexe ne peut être **stocké**, d'une manière pratique, que dans des **bases de données** qui manipulent des données **complexes**.

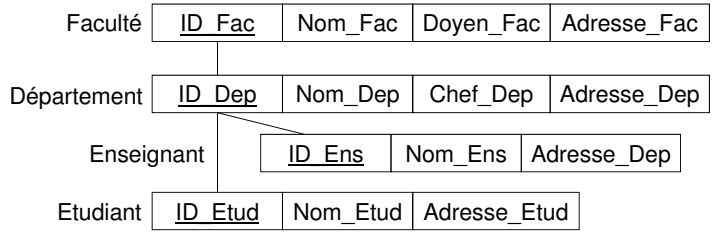
Pour le besoin de notre formation, nous avons choisi de voir les **bases de données orientées objet** en détail.

Chapitre I
Introduction

1. Différents type d'SGBDs:

a) Bases de données hiérarchiques

- Dans une base de données **hiérarchique**, les nœuds de la hiérarchie sont des **enregistrements** connectés entre eux par des liens



Chapitre I
Introduction

b) Bases de données réseaux

- Le modèle réseau est une **extension** du modèle hiérarchique. Il utilise des **pointeurs** additionnels afin d'augmenter la flexibilité du modèle hiérarchique

Chapitre I
Introduction

c) Bases de données relationnelles

- Les SGBDR sont les plus vendus et les plus utilisés
- Un modèle relationnel est constitué de tables (appelées entités), de colonnes définissant ces tables (appelées attributs) et de relations entre ces diverses tables

1	A
2	B
2	C
2	D

table 1

2	E
2	C
3	F
4	G

table 2

1	A
2	B
2	C
2	D
2	E
3	F
4	G

table1 ∪ table2

1	A
2	B
2	D

table1 - table2

1
2

Projection (colonne1)(table1)

2	B
2	C
2	D

Sélection (colonne1=2)(table1)

Chapitre I
Introduction

2. Limites des SGBD

a) Avantages:

- Le modèle de données est très **simple** (tables, relations…) et donc facile à comprendre pour les utilisateurs
- Le modèle repose sur une base **formellement définie**, ce qui a permis de définir des langages de manipulation de données (LMD) standardisés tels que (SQL, QUEL, ...)

b) Inconvénients:

- Le modèle de données simple **ne représente pas facilement** les objets du monde réel. Ces objets sont éclatés en plusieurs relations, ce qui multiplie les jointures dans les requêtes des utilisateurs
- **L'incompatibilité** des LMD relationnels et des langages de programmation
- Le développement d'applications est **très lent** et les applications sont difficilement **maintenables**

3. Nouveau besoins:

Les nouvelles applications ont besoin de gérer des **objets complexes** comme le texte, les graphiques, les cartes, les images, …

Pour cela, les **bases de données objets** sont nées, ils regroupes:

- les **bases de données**
- et les **langages de programmation orientés objets** (Simula, Eiffel, Smalltalk, C++, JAVA). Ces langages permettent d'accroitre la productivité des développeurs, ils se basent sur:
  - Les **objets** (valeurs, méthodes)
  - **L'héritage**, permet de donner à une classe fille les même propriétés de la classe mère sans avoir à les redéfinir.

4. Caractéristiques des BDOO:

- Un **modèle de données** qui permet de représenter des structures de données **complexes**
- Les **données** et les **traitements** ne sont plus séparés. Les méthodes font partie de la déclaration des objets
- **L'héritage**
- Les objets ont des **identités différentes** pour pouvoir les distinguer

Chapitre II

Systèmes de gestion de bases de données orientées objets

I. Caractéristiques supportées par les SGBDOO
II. Structure de données
III. La dynamique
IV. Concepts objet

I. Caractéristiques supportées par les SGBDOO

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

Ces concepts sont : l'identité, l'encapsulation, les classes et types, l'héritage, l'agrégation, le polymorphisme et l'extensibilité.

1. Identité

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

L'identifiant ne doit pas être géré par le programmeur mais par le système.

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

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

2. Encapsulation

L'encapsulation est un **mécanisme** provenant des langages orientés objets. Il permet de **cacher l'implantation** d'un objet (valeurs) et l'utilisateur ne connait rien sur les **constituantes internes** des données contenues dans cet **objet**.

Dans ce contexte, les **interfaces** (sélecteur de méthodes applicables à l'objet) constituent la seule partie **visible** par l'utilisateur de l'objet implanté.

2. Encapsulation (suite)

Exp:

3. Classes et types

Une **classe d'objets** permet de regrouper des objets possédant les mêmes caractéristiques.

Un **type** permet de décrire la structure de données.

4. Héritage

L'héritage peut être **simple** (une seule superclasse par sous-classe) ou **multiple** (plusieurs superclasses pour une sous-classe).

Les SGBDOO doivent **supporter** le concept d'héritage **simple**, mais l'héritage **multiple** reste **facultatif**.

5. Agrégation

L'agrégation est un **type** de relation entre les objets.

Elle permet de décrire un objet par les objets qui le composent.

6. Polymorphisme

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

Lors d'une **application** d'une **méthode** à un objet, la méthode à exécuter dépend de la classe de l'objet et ne peut donc être **connue** qu'à **l'exécution** (pas à la compilation).

6. Polymorphisme (suite)

Exp. en java

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


7. Extensibilité

Un SGBDOO doit **supporter** la possibilité d'ajouter de **nouveaux types** afin de prendre en compte de nouveaux domaines d'application.

II. Structure de données

Les **objets** sont décrits par des **attributs**, et sont regroupés en **classes**.

Chaque objet a une **identité** qui le distingue de tous les autres. On l'appelle **oid** de l'objet (en anglais "**o**bject **id**entity").

À ce jours, chaque SGBDO possède son **propre modèle de données** OO. Il n'y a **pas** encore de **norme**. Nous présentons ici les points essentiels sur lesquels il y a consensus:

1. Structure complexe

La **structure** d'une classe est définie en utilisant des constructeurs comme: **TUPLE, SET**.

Le constructeur « **TUPLE** » crée un type tuple composé d'une suite d'attributs:

$$\text{TUPLE } (att_1:dom_1, \ldots att_n:dom_n)$$

où dom_i peut être un **domaine prédéfini** (STRING, REAL, INT, DATE…) comme il peut être défini par un **constructeur** (TUPLE, SET) ou bien peut être un nom d'une **classe**.

1. Structure complexe (suite)

Le constructeur d'ensemble « **SET** » crée un type ensemble composé d'un ensemble de valeurs:

SET domaine

où domaine à la même définition précédente.

Certains SGBDO proposent des **constructeurs de collection** autres que l'ensemble, tels que la **liste**, le **multi-ensemble**, le **tableau**, ...

Exp.

```
CLASS Etudiant {
      num : INT,
      nom : STRING,
      prénoms : SET STRING,
      date-nais : DATE,
      adresse : TUPLE (n° : INT,
         rue : STRING,
         ville : STRING,
         pays : STRING)
         cours-suivis : SET TUPLE ( nom-cours : STRING,
                                    note : REAL)
}
```


2. Liens de composition

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

2. Liens de composition (suite)

Exp.

```
CLASS Etudiant {
    num : INT,
    nom : STRING,
    prénoms : SET STRING,
    cours-suivis : SET TUPLE (cours : Cours,
    note : REAL) }
CLASS Cours {
    nomC : STRING, .... }
```

La classe **Cours** est une classe composante de la classe **Etudiant**.

Il existe des dérivées de ces liens: dépendance, partage et contrainte inverse.

Une classe **composante** peut être, ou non, **dépendante** de sa classe **composite**, c-à-d, un objet ***x*** composant est **dépendant** d'un objet père ***y*** si la destruction de ***y*** entraîne automatiquement celle de ***x***.

Un objet **composant** peut être, ou non, **partagé** entre plusieurs objets de **la même classe composite**. Par exemple: l'objet ***x*** est, en même temps, **composant** de l'objet ***a*** et de l'objet ***b***.

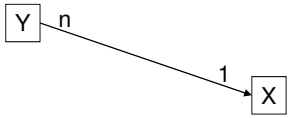
Quelques SGBDO acceptent la **contrainte inverse**: l'objet composite dépend de ses objets composants.

Des **cardinalités** peuvent être associées au **lien de composition**, comme par exemple:

La contrainte « ***X*** est une classe composante non partagée de la classe ***Y*** », c-à-d un objet de ***X*** est lié à un seul objet de la classe ***Y*** :

cardmax(< *lien de composition* X-Y >, X) = 1

cardmax(< *lien de composition* X-Y >, Y) = n



Le lien de référence est orienté de la classe composite vers la classe composante, et il est utile de décrire ce lien de composition et son inverse pour avoir un accès facile dans les deux sens.

Exp.

```
CLASS Cours {
  nomC : STRING,
  etudiants : SET Etudiant
}
```

L'attribut `etudiants` de `Cours` est un **attribut référence inverse** de l'attribut `cours-suivis` de la classe `Etudiant` définie dans l'exemple précédent.

Certains SGBDO permettent de déclarer les attributs inverses par la clause « INVERSEOF ».

Exp.

```
CLASS Cours {
  nomC : STRING,
  etudiants: SET Etudiant INVERSEOF cours-suivis
  cours
}
```

3. Identité d'objet

L'identifiant doit être **permanent**, **fixe** (ne change pas pendant la vie de l'objet) et **unique** dans la **base** et dans le **temps** (deux objets différents dans une base n'auront jamais le même identifiant, même s'ils n'existent pas en même temps).

Ces **caractéristiques** de l'identifiant permettent à plusieurs objets de **partager** le même objet composant.

3. Identité d'objet (suite)

On peut dire que **deux objets** sont **égaux** dans trois cas:

- **L'égalité d'identité**

obj1 == obj2

Signifie : « le **même oid** »

- **L'égalité de surface**

obj1 =1 obj2

signifie: « les objets obj1 et obj2 ont la **même valeur au premier niveau** » C'est-à-dire tous leurs attributs ont la même valeur, que ce soient des attributs-valeur ou des attributs-référence.

3. Identité d'objet (suite)

- **L'égalité de profondeur**

obj1 =* obj2

signifie: «les objets obj1 et obj2 ont la **même valeur en profondeur**» c-à-d:

- les attributs-référence des objets obj1 et obj2 constituent des graphes semblables, et
- les attributs-valeur correspondants, à tous les niveaux, sont égaux.

On a les propriétés suivantes:

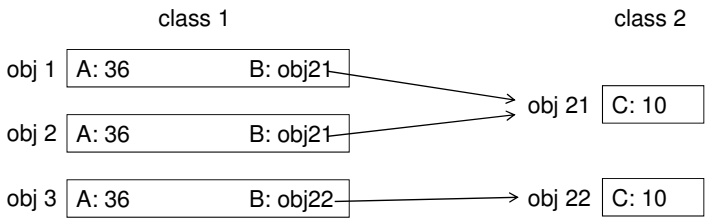
obj1 == obj2 ⇒ obj1 =1 obj2

obj1 =1 obj2 ⇒ obj1 =* obj2

3. Identité d'objet (suite)

Exp.

Soient les deux classes suivantes:



3. Identité d'objet (suite)

Exp. (suite)

Nous avons alors:

(obj1.B == obj2.B) vrai mais (obj1 == obj2) faux

(obj1 =1 obj2) vrai mais (obj1 =1 obj3) faux

(obj1 =* obj3) vrai

(obj21 =1 obj22) vrai mais (obj21 == obj22) faux

(obj1 =1 obj2 $\Rightarrow$ obj1 =* obj2) vrai

4. Graphe de généralisation des classes

La notion d'héritage permet de définir deux types de classes: classe **CS** (**Classe Spécifique**) d'une autre classe **CG** (**Classe Générique**), notée:

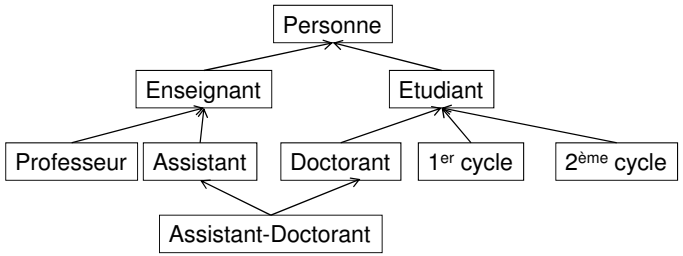
CS Is-a CG

CS représente un **sous-ensemble** de **CG**. Du point de vue conceptuel, l'ensemble des objets de CS est inclut dans celui de CG;

4. Graphe de généralisation des classes (suite)

Exp.

Soit le graphe de généralisation des classes décrivant les personnes dans une université.



Exp. (suite)

Déclaration des classes:

```
Class Personne {
   nom : String [30] ;
   prénoms : Set String [30] ;
   adr : String [100] ;
   METHODS afficher ;
}
```

Exp. (suite)

```
Class Etudiant
Is-a Personne {
       n°E : Int ;
       dateN : Date ;
       études : Set Tuple : (année : Int ,
       diplôme : String [30] ) ;
       cours-obtenus : Set Tuple ( cours : Cours ,
       année : Int ,
       notes : Set Real ) ;
       cours-suivis : Set Cours ;
   METHODS afficher ; /* surcharge */
       nouveauétud (nom : String, prénoms : String, n°E : Int, adr :
       String, dateN : Date ) ;
       inscrire (cours : Cours ) ;
       aobtenu (cours : Cours, notes : Set Int ) ;
}
```


Exp. (suite)

```
Class Enseignant
Is-a Personne {
       tél : Int ;
       statut : String [30] ;
       coursdonnés : Set Cours ;
   Methods
       afficher ; /* surcharge */
       nouvelens (nom : String, prénoms : String, statut : String ) ;
       assure (cours : Cours ) ;
       nassureplus (cours : Cours ) ;
}
```


Exp. (suite)

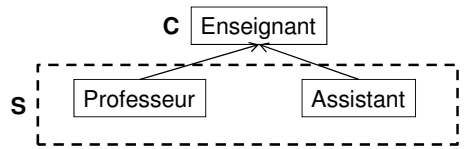
Dans cette base de données, tout **professeur** est à la fois un **Professeur**, un **Enseignant** et une **Personne.**

Le graphe de généralisation de classes ne doit pas contenir de **cycles**.

Des **contraintes** d'intégrité de **couverture** et de **disjonction** peuvent être associées aux classes spécifiques d'une classe générique:

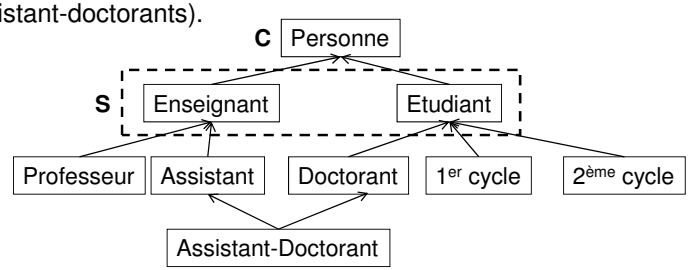
- Un sous-ensemble **S** de classes spécifiques immédiates d'une classe générique **C** est **disjoint** si toute occurrence de **C** peut être occurrence **d'au plus** une des classes de **S**.

Dans l'exemple précédent, un assistant ne peut pas être aussi un professeur, on a donc une contrainte de **disjonction** entre ces deux sous-classes.



- Un sous-ensemble **S** de classes spécifiques immédiates d'une classe générique **C** est dit **couvrant** si toute occurrence de **C** est aussi occurrence **d'au moins** une des classes de **S**.

Dans l'exemple précédent, il y a une contrainte de couverture: **Enseignant et Etudiant constituent une couverture de Personne**. Mais il n'y a **pas de disjonction** entre Enseignant et Etudiant, car il peut y avoir une personne qui est à la fois Enseignant et Etudiant (cas d'un assistant-doctorants).



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

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

III.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.

Les méthodes

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.

Les méthodes (suite)

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.

Les méthodes (suite)

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.

Les méthodes (suite)

Exp.

```
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
```


IV.Concepts objet et langage OQL

Le groupe **ODMG** (Object Database Management Systems) propose la **standardisation** du langage de requêtes **OQL** (Object Query Language). OQL est un langage de type **déclaratif**.

1. Requêtes, points d'entrée et résultats

En OQL, on peut donner à tout objet un ou plusieurs noms permanents, qui pourront servir de **point d'entrée** dans la base. Pour cela, il suffit de déclarer un nom grâce à l'instruction "Name". Par exemple: **Name directeur : Personne**

1. Requêtes, points d'entrée et résultats (suite)

Il suffit maintenant d'affecter un objet à ce nom.

Exp.

directeur = Personne (nom:"toto", prénoms: SET("p1","p2"), adresse: "adr")

En OQL, le **résultat** d'une requête peut être de **différents types**.

1. Requêtes, points d'entrée et résultats (suite)

Exp.

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. Requêtes, points d'entrée et résultats (suite)

Exp. (suite)

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. Requêtes, points d'entrée et résultats (suite)

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.

Exp. **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 la requête SELECT:

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

Tel que:

<définition du résultat> : c'est une expression qui

- désigne un objet / une collection d'objets
  - `SELECT p FROM p IN Enseignants`
  - `SELECT p.cours_assurés FROM p IN Enseignants`
- désigne une valeur / collection de valeurs
  - `SELECT p.nom FROM p IN Enseignants`
  - `SELECT p.prénoms FROM p IN Enseignants`
- construit une valeur complexe

```
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))
```


collection$_i$: peut être

- collection quelconque de la base de données
  - population :

```
SELECT p FROM p IN Enseignants WHERE …
```

  - autre collection :

```
… FROM p IN Enseignants, c IN p.cours_assurés
```

- requête : ex. 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)
```


condition: peut être l'une des conditions suivantes:

- –condition élémentaire
- –(condition)
- –condition AND condition
- –condition OR condition

•condition élémentaire: peut être

expression1 opérateur_comparaison expression2

Exemples : (e est un Etudiant)

```
e.nom = 'N'
e.age() < 20
COUNT(e.prénoms) > 2
```

ou avec un quantificateur **existentiel** ∃ **ou universel** ∀. (Voir section 4)

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.

Exp. 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 ;
```


b. Quantificateur universel

L'expression:

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

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

Exp. La requête suivante retrouve les enseignants qui ne donnent que des cours de 3ème cycle (ainsi que ceux qui ne donnent aucun cours).

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


Il existe donc d'autres types de requêtes que celles de type "Select from where", comme par exemple:

Cours

c'est une requête qui fournit en résultat **l'ensemble** des **objets Cours**.

De même, si par exemple "bd" est une variable de type "Cours" et contenant le cours "bases de données":

bd.prof.nom

c'est une requête qui fournit en résultat les noms des professeurs du cours de bases de données.

3. Parcours des liens de composition

Pour passer d'un objet à un de ses objets composants il suffit de traverser le liens de composition.

Exp.

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;
```

3. Parcours des liens de composition (suite)

Exp. (suite)

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)

Exp.

`COUNT(Etudiants)` => nombre d'étudiants

`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 (suite)

Exp. 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 , …)]
```

5. Instruction GROUP BY (suite)

Exp. L'expression:

```
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

$collection_1$ UNION $collection_2$ (l'union)

$collection_1$ EXCEPT $collection_2$ (la différence)

$collection_1$ INTERSECT $collection_2$ (l'intersection)

Les éléments **doivent** être de **types compatibles**:

- Soit de même type
- Soit sur-type commun ➔ comparaison sur la partie commune

7. Opérateurs ensemblistes (suite)

Exp.

```
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'.

8. DB4Objects

C'est un système de gestion de bases de données objet open source.

Il permet aux développeurs JAVA et .NET de manipuler les bases de données Objet.

Nous utilisant la version « **db4o 8.0** » sous JAVA pour manipuler ce type de bases de données.

8. DB4Objects (suite)

Les opérations d'accès à la bases de données se trouvent dans une bibliothèque qu'il faut importer:

```
import com.db4o.*;
```

Avec ce système, la base de données est enregistrée dans un fichier. Nous déclarons le nom du fichier:

```
public static String DBOFILENAME = "c:/TD1.dbo";
```

Pour ouvrir la base:

```
ObjectContainer db = Db4o.openFile(DBOFILENAME);
```


8. DB4Objects (suite)

Pour stocker un objet dans la base:

```
db.store(obj);
```

Pour lire tous les objets d'une classe de la base:

```
ObjectSet result = db.query("MYCLASS".class);
```

Pour parcourir la liste d'objets récupérés de la dernière requête:

```
MYCLASS obj = null;
for (int i=1; i<=result.size(); i++) {
    obj = (MYCLASS) result.next();
}
```


8. DB4Objects (suite)

Pour récupérer les objets d'une classe qui ont une caractéristiques communes:

```
db.queryByExample(obj);
```

Exp:

```
MYCLASS myobj = new MYCLASS();
myobj.atribut = value;
ObjectSet<MYCLASS> result=db.queryByExample(myobj);
MYCLASS obj = null;
for (int i=1; i<=result.size(); i++) {
  obj = result.next();
}
```

8. DB4Objects (suite)

Créer une "List":

```
List myList = new LinkedList();
```

Parcoure d'une "List":

```
Iterator i = myList.iterator();
while (i.hasNext()) {
  MyCLASS myObject = (MyCLASS)i.next();
};
```


8. DB4Objects (suite)

Ajout d'un objet dans une "List":

```
myList.add(myObj);
```

Suppression d'un d'une "List":

```
myList.remove(index);
```

Taille d'une "List":

```
int t = myList.size();
```

Toute les opérationsde la bibliothèque db4o doivent être dans un bloc try:

```
try { opérations db.*;
} finally {
   db.close();
}
```


Bibliographie

1. Alfred, C., « Maximizing Leverage from an Object Database », IBM Systems Journal, Vol 33, N° 2, 1994, pp. 280-299.
2. Benzaken, V., Doucet, A., Bases de données orientées objet: origines et principes, Armand Colin, 1993.
3. Delobel, C., Lécluse, C., Richard, P., « Bases de données: des systèmes relationnels aux systèmes à objets », InterEditions, 1991.
4. Khoshafian, S., Baker, A.B., « Multimedia and Imaging Databases », Morgan Kaufmann Publishers, 1996.