

# Bases de Données Avancées, M1 Informatique et Miage

S. Cerrito

Année 2013-2014, Evry

## Plan du Cours

1. Introduction et rappels du modèle relationnel
2. Modèles à objet et objet-relationnel
3. Modèles de données NoSQL : XML et le Modèle dit “Données Semi-structurées”, puis plus si temps

*Supports de cours à l'adresse :*

`http://www.ibisc.univ-evry.fr/~serena/teach13-14.html`

# Table des matières

<b>1</b>	<b>Notions essentielles des BD relationnelles</b>	<b>4</b>
<b>2</b>	<b>Fondements des Langages de Requête (qqe soit le modèle)</b>	<b>13</b>
2.1	Les opérateurs de l’algèbre relationnelle . . . . .	15
<b>3</b>	<b>Limites du modèle relationnel</b>	<b>16</b>
<b>4</b>	<b>Modèle à Objets</b>	<b>19</b>
4.1	Sous-classes et Héritage . . . . .	42
4.2	Un langage de Requête pour les BD Objet : OQL. Quelques Notions . . .	49
<b>5</b>	<b>Bases Relationnelles-Objet</b>	<b>61</b>

# INTRODUCTION

## Historique

- Avant 1970 : BD=fichiers d'enregistrements, “modèles” *réseaux* et *hiérarchique* ; pas de vraie indépendance logique/physique.
- En 1970 : modèle *relationnel* (Codd) : vraie indépendance logique/physique.
- Années 80 et 90 : nouveaux modèles :
  - modèle à objets et object-relationnel
  - modèle à base de règles (Datalog)
- Fin années 90 : données dites *semi-structurées* (XML).

Centre de ce cours : modèle à objets et object-relationnel, puis modèle semi-structuré.

# 1 Notions essentielles des BD relationnelles

## Mots clés :

- Univers  $U$ , Attributs  $A_1, \dots, A_n$
- Domaine  $Dom(A)$  d'un attribut  $A$
- Schéma d'une relation dont le nom est  $R$ .
- $n$ -uplet sur un ensemble  $E$  d'attributs
- Relation (ou “table”) sur un schéma de relation
- Schéma d'une BD
- Base de données  $B$  sur un schéma de base

Un *univers*  $U$  est un ensemble fini et non-vide de noms, dits *attributs*.

Le *domaine* d'un attribut  $A$  ( $Dom(A)$ ) est l'ensemble des valeurs possibles associé à  $A$ .

**Exemple :**

$U = \{NomFilm, Realisateur, Acteur, Producteur, NomCinema, Horaire\}$

$Dom(NomFilm) = Dom(Realisateur) = Dom(Acteur) = Dom(Producteur) =$   
 $Dom(NomCinema) =$  chaînes de caractères.

$Dom(Horaire) = \{h.m \mid h \in [0, \dots, 24], m \in [0, \dots, 59]\}$

Un *schéma d'une relation* dont le nom est  $R$  est un sous-ensemble non-vide de l'univers  $U$ .

**Suite de l'exemple :**

- Schéma de la relation  $Film = \{NomFilm, Realisateur, Acteur, Producteur\}$
- Schéma de la relation  $Projection = \{NomFilm, NomCinema, Horaire\}$

**Intuition :** Format de deux tables.

*Film* :

NomFilm	Realisateur	Acteur	Producteur
⋮	⋮	⋮	⋮

*Projection* :

NomFilm	NomCinema	Horaire
⋮	⋮	⋮

Soit  $E = \{A_1, \dots, A_n\}$  le schéma d'une relation. Un *n-uplet*  $n$  sur  $E$  est un ensemble  $\{A_1 : v_1, \dots, A_n : v_n\}$  où  $v_i \in \text{Dom}(A_i)$ .

Un *n-uplet* possible sur le schéma de *Projection* :

$\{NomFilm : \text{"Jugez – moi coupable"}, NomCinema : \text{"Gaumont Alesia"}, Horaire : 13.35\}$ ,

ce qui est la même chose que  $\{NomFilm : \text{"Jugez – moi coupable"}, Horaire : 13.35\}, NomCinema : \text{"Gaumont Alesia"}\}$ .

Toutefois, le plus souvent on note :

$\langle \text{"Jugez – moi coupable"}, \text{"Gaumont Alesia"}, 13.35 \rangle$ .

Pourquoi ?

Si  $E' \subset E$ , la *restriction* d'un *n-uplet*  $t$  à  $E$  se note  $t(E')$ .

La restriction de  $\langle \text{"Jugez – moi coupable"}, \text{"Gaumont Alesia"}, 13.35 \rangle$  à

$\{NomCinema, NomFilm\}$  est :

$\langle \text{"Jugez – moi coupable"}, \text{"Gaumont Alesia"} \rangle$ .

Une *relation* (table)  $r$  sur un schéma de relation  $S$  est un ensemble d' $n$ -uplets sur  $S$ . On dit aussi :  $S$  est le schéma de  $r$ .

**Exemple.**

*Film* :

NomFilm	Réalisateur	Acteur	Producteur
nf1	r1	a1	p1
nf1	r1	a2	p1
nf2	r2	a1	p2
nf3	r2	a1	p2

*Projection :*

NomFilm	NomCinema	Horaire
nf1	nc1	h1
nf1	nc2	h2
nf2	nc1	h3
nf3	nc2	h1

Un **schéma  $S$  d'une base** sur un univers  $U$  est un ensemble non-vidé d'expressions de la forme  $N(S)$  où  $S$  est un schéma de relation et  $N$  un nom de relation.

**Exemple**(on omet les  $\{\}$ ).

$U =$

$\{NomFilm, Realisateur, Acteur, Producteur, NomCinema, Horaire, Spectateur\}$

$S =$

$\{$   
 $Film(NomFilm, Realisateur, Acteur, Producteur),$   
 $Projection(NomFilm, NomCinema, Horaire), Aime(Spectateur, NomFilm)$   
 $\}$

**Schéma de la base = Format des données de la base.**

Quel est le format de la base de l'exemple ?

- Une *base de données* (relationnelle)  $B$  sur un schéma de base  $\mathcal{S}$  (avec univers  $U$ ) est un ensemble de relations finies  $r_1, \dots, r_n$  où chaque  $r_i$  est associée à un nom de relation  $N_i$  et est telle que si  $N_i(S) \in \mathcal{S}$ , alors  $r_i$  a  $S$  comme schéma.
- On peut aussi imposer des *contraintes* sur les données. Par exemple : les *dépendances fonctionnelles*, qui fixent, entre autres, les *clés* des relations (cours SGBD L3).
- Ces contraintes, dites d'*intégrité*, font aussi partie de la spécification du format des données de la base.

## Exemple d'une base.

<i>Film</i>			
NomFilm	Réalisateur	Acteur	Producteur
nf1	r1	a1	p1
nf1	r1	a2	p1
nf2	r2	a1	p2
nf3	r2	a1	p2

<i>Projection</i>		
NomFilm	NomCinema	Horaire
nf1	nc1	h1
nf1	nc2	h2
nf2	nc1	h3
nf3	nc2	h1

<i>Aime</i>	
NomFilm	Spectateur
nf1	s1
nf1	s2
nf2	s1
nf3	s3

## 2 Fondements des Langages de Requête (qqe soit le modèle)

- Informellement : *Requête sur une base* = question que l'on pose à la base.
- *Langage de requête* = langage permettant d'écrire des requêtes
- Importance d'un langage de requête formel et rigoureux :
  1. Conception de langages commerciaux
  2. Evaluation de la puissance d'expression de chaque langage commercial
  3. Possibilité de déterminer ce qu'un langage commercial ne pourra pas exprimer
  4. Notion d'équivalence entre deux expressions de requête  $\Rightarrow$  Optimisation "logique" de l'évaluation d'une requête

Deux langages formels pour le modèle relationnel : *algèbre relationnelle* et *calcul relationnel* (cours SGBD L3).

## 2.1 Les opérateurs de l'algèbre relationnelle

- Opérateurs ensemblistes : union ( $\cup$ ), intersection ( $\cap$ ), différence ( $\setminus$ ), produit cartésien ( $\times$ )
- projection sur un ensemble d'attributs  $E$  ( $\pi_E$ ), sélection d'un ensemble de  $n$ -uplets selon une condition  $C$  ( $\sigma_C$ ), jointure "naturelle" ( $\bowtie$ ), division ( $\div$ ), renommage ( $\rho$ ).

### 3 Limites du modèle relationnel

1. On ne peut pas imbriquer les informations
2. La structure du schéma est très rigide
3. On ne peut pas exprimer la clôture transitive d'une relation (par ex. vol(départ, arrivée) par rapport à vol\_direct(départ, arrivée))

Commençons par (1).

En relationnel (“première forme normale”) :

*OPERAS* :

<i>Auteur</i>	<i>Titre</i>	<i>Langue</i>
Mozart	La Flûte Enchantée	Allemand
Mozart	Don Juan	Italien
Mozart	Les noces de Figaro	Italien
Bizet	Carmen	Français
Bizet	Djamileh	Français

Redondance.

Si on imbrique :

*OPERAS :*

<i>Auteur</i>	<i>Opéra</i>	
Mozart	<i>Titre</i>	<i>Langue</i>
	La Flûte Enchantée	Allemand
	Don Juan	Italien
	Les noces de Figaro	Italien
Bizet	<i>Titre</i>	<i>Langue</i>
	Carmen	Français
	Djamileh	Français

On est sorti de la norme “Première Forme Normale”

## 4 Modèle à Objets

- Extension de concepts de langages comme C<sup>++</sup> or Java au cas des BD, où la *persistance* des données est primordiale.
- Concepts clés :
  - types,
  - classes et objets,
  - identité des objets,
  - héritage.

Ici, on choisi *ODL* (Object Definition Language) comme langage de spécification de la structure d'une BD à objets → Ecriture du **schéma**.

En **relationnel** : en SQL, `create table` permet de spécifier une table.

En **objet** : une déclaration ODL permet de spécifier une classe.

## Types

**Types atomiques** : integer, float, char, string, boolean et les *énumérations*.

Syntaxe d'un type énuméré : `enum NomType e11, ..., e1N.`

Par ex. : `enum CouldrapeauFr bleu, blanc, rouge`

Une classe aussi est un type atomique (voir après ce que c'est une classe, ici).

## Constructeurs de Types :

- `Set<Type>`. *Ex.* : `Set<integer>`. *NB* : ensembles finis.
- `Bag<Type>`. Si  $T$  est un type, `Bag<T>` est un type  $T'$  dont les valeurs sont des *multi-ensembles finis* d'éléments de type  $T$ . *Ex.* : `{1, 2, 1}` est de type `Bag<integer>`.
- `List<Type>`. *Ex.* : `List<char>` (les chaînes de caractères), `List<integer>`. *NB* : listes finies.
- `Array<Type, entier>` : type des tableaux de  $n$  (entier) éléments de type `Type`. *Ex.* : `Array<char, 10>`.
- `Dictionary<Type1, Type2>` : un type dont les valeurs sont des ensembles finis de couples  $\langle \text{clé}, \text{val} \rangle$ , où `clé` est de type `Type1` et `val` est de type `Type2`.
- `Struct Nom { Type1 Nom1, ..., TypeN NomN }`.  
*Ex.* : `Struct Adresse { string rue, string ville}`. (Type record !)

Possibilité d'imbriquer les constructeurs de types (la déf. des types est récursive !)

Par ex. :

```
Struct famille {  
Set(string) enfants,  
Struct père { string nom, string prenom} LePere,  
Struct mère { string nom, string prenom } LaMère  
}
```

## Classes et Objets

- Une **classe** est un *type abstrait* : on définit les propriétés d'un objet et ce que l'objet peut faire (*méthodes*).
- Un **objet**  $o$  est une instance d'une classe  $C$  et a un et un seul identificateur (*OID*).
- Déclaration d'une classe en ODL :  
`class Nom = { liste de propriétés et méthodes }`

- La sorte la + simple de propriété : un **attribut**.
- Déclaration d'un attribut : on indique son type et son nom.

– *Exemple* :

```
class Film {  
  attribute string titre ;  
  attribute integer année ;  
  attribute integer longueur ;  
  attribute enum couleurs { couleur, noir&blanc } SorteFilm ;  
};
```

Attribut SorteFilm : de type énumération.

couleurs est le nom de ce type énuméré, tandis que le nom de l'attribut est SorteFilm.

- Un objet de cette classe : (“Autant en emporte le vent”, 1939, 231, couleur)

Un autre exemple de classe :

```
class Star {  
attribute string nom ;  
attribute Struct Adr { string rue, string ville } adresse  
};
```

Ici l'attribut qui se nomme adresse est de type "structure" (record) ; ce type s'appelle "Adr" et le type des ses 2 champs est string.

## Autre sorte de propriétés : Associations entre objets

- Mot clé dans la déclaration d'une classe : `relationship`
- Définition + riche de la classe `Film` :

```
class Film {  
    attribute string titre ;  
    attribute integer année ;  
    attribute integer longueur ;  
    attribute enum couleurs { couleur, noir&blanc } SorteFilm ;  
    relationship Set<Star> acteurs ;  
};
```

Possibilité d'indiquer les acteurs d'un film donné, qui est un objet de la classe `Film`.

Une `relationship` : comme une association en EA.

Le type d'une `relationship` (ensemble "d'arrivée") : une classe, ou bien construit à partir d'une classe avec **un** constructeur de **collection** : `Set` ou `Bag` ou `List`.

Ici : `Set<Star>`.

Donc :

- Type d'un **attribut** : construit à partir des types atomiques avec autant de constructeurs qu'on veut, peu importe lesquels.
- Type (ensemble “d'arrivée”) d'une **relationship** : une classe, ou construit à partir d'une classe avec un seul constructeur de collection.

## Associations INVERSEES entre objets : on navigue entre les classes.

- Mot clé : inverse
- Définition de classes encore + riches :

```
class Film {
  attribute string titre ;
  attribute integer année ;
  attribute integer longueur ;
  attribute enum couleurs { couleur, noir&blanc } SorteFilm;
  relationship Set<Star> acteurs
  inverse Star : :JoueDans };
```

```
class Star {
  attribute string nom ;
  attribute Struct Adr = { string rue, string ville } adresse ;
  relationship Set<Film> JoueDans
  inverse Film : :acteurs;
};
```

La notation  $C :: R$  signifie :  
la relationship  $R$  est une relation de la classe  $C$ .

Toute relationship ODL est binaire.

## Cardinalités des associations entre les classes

Relationship R *many-many* : le type de R et celui de son inverse sont un Set (ou un autre type collection) d'éléments d'une classe C. C'est le cas d'acteurs (et JoueDans)

pour l'exemple précédent :

le type de la relationship acteurs est Set<Star>

le type de JoueDans est Set<Film>.

## Cardinalités des associations entre les classes, suite

Autre exemple de relation many-many :

```
class Bière {  
  ... ;  
  relationship Set<Buveur> Fans  
  inverse Buveur : :Aime; }  
  
class Buveur {  
  ... ;  
  relationship Set<Bière> Aime  
  inverse Bière : :Fans;  
}
```

Une bière peut avoir plusieurs fans, et un buveur peut aimer plusieurs bières.

**N.B.** Fans a comme type (= type du cible) un Set d'objets de Buveur et Aime a comme type un Set d'objets de Bière.

On dit où sont définies les relations inverses à l'aide de la notation : :.

## Cardinalités des associations entre les classes, suite

Relationship R *many-one* : d'un côté Set (ou un autre type collection), de l'autre la classe.

On enrichie l'exemple précédent des bières et des buveurs en y ajoutant :  
une bière peut avoir plusieurs *super-fans*, mais un buveur a une *seule*  
*bière-favorie*, dont il est un *super-fan* :



## Cardinalités des associations entre les classes, suite

```
class Bière {  
    ... ;  
    relationship Set<Buveur> Fans  
    inverse Buveur : :Aime;  
    relationship Set<Buveur> super-fans  
    inverse Buveur : :bière-favorie;  
}
```

```
class Buveur {  
    ... ;  
    relationship Set<Bière> Aime  
    inverse Bière : :Fans;  
    relationship Bière bière-favorie  
    inverse Bière : :super-fans;  
}
```

## Cardinalités des associations entre les classes, suite

Relationship R *one-one* : le type de la relation est une classe, des 2 cotés. Exemple : une fille a un seul époux et un garçon a une seule épouse.

```
class Fille {  
  ... ;  
  relationship Garçon Epoux  
  inverse Garçon : :Epouse; }
```

```
class Garçon {  
  ... ;  
  relationship Fille Epouse  
  inverse Fille : :Epoux;  
}
```

**N.B** : l'expression Garçon joue 2 rôles différentes, dans la déf. de la classe Fille.

**Question** : est-il vrai ou faux que la définition donnée permet le célibat ?

Une classe C peut être telle que l'inverse d'une de ses relationships est défini dans C elle même.

```
class Personne {  
    ... ;  
    relationship Personne Epoux  
    inverse Epouse;  
    relationship Personne Epouse  
    inverse Epoux; }
```

**N.B** : Puisque on écrit : `inverse Epouse`, sans le `:` pour spécifier où `Epouse` est définie, c'est implicite que `Epouse` est définie dans la même classe `Personne` ; idem pour `Epoux`.

## **Associations liant + que 2 classes**

En ODL on peut spécifier seulement des relations binaires. Comment faire si l'on veut modéliser une relation  $R$  entre  $n$  classes, où  $n > 2$ ? On introduit une nouvelle classe  $C$  qui joue le rôle de  $R$  et on définit  $n$  relations binaires entre  $C$  et chaque  $C_i$ .

**Exemple** On veut modéliser une relation *contrats* qui lie une star, un film et un studio.

On crée une nouvelle classe `Contrat` :

```
class Contrat {  
  attribute integer salary;  
  relationship Film leFilm  
  inverse ...;  
  relationship Studio leStudio  
  inverse ...;  
  relationship Star laStar  
  inverse ...;  
};
```

Ici, `leFilm` est le nom de la relation qui lie un objet de la classe `Contrat` au seul objet de la classe `Film` auquel **ce** contrat fait référence et `Film` est le “type d’arrivée” de cette relation (un contrat est lié à un objet de la classe `Film`).

Pour pouvoir remplir la partie ... qui suit `inverse` après la déclaration de la relation `leFilm` de la classe `Contrat`, en y écrivant, par ex. :

```
Film : :ContratsPour
```

il faut modifier la définition de la classe `Film`, en y ajoutant :

```
relationship Set<Contrat> ContratsPour inverse Contrat : :leFilm;
```

## Méthodes

- *Méthode* = code exécutable qui peut être appliqué à un objet.
- En ODL, on n’écrit pas le code d’une méthode : on indique juste son nom et sa *signature* : les types des entrées/sorties. *ODL est un langage de spécification !*
- Comme dans les langages de programmation orientés objet, l’objet au quel la méthode s’applique est un argument “caché” d’une méthode *m*.
- Une méthode *m* peut soulever des *exceptions*. Mot clé : `raises`.
- Les paramètres d’une méthode *m* sont spécifiés par :
  - (a) `in` (entrée)
  - (b) `out` (sortie).

Une méthode peut aussi (c) renvoyer une valeur.

Différence entre (b) et (c) : selon (b) la méthode *m* se comporte comme une procédure d’un langage impératif (C, par ex.), selon (c) *m* est comme une fonction. Si (b), alors le type du résultat est `void` (vide), car on n’a pas calculé une fonction, mais effectué une action.

## *Exemple*

Définition encore + riche de la classe `Film`, avec 3 méthodes :

```
class Film {
attribute string titre ;
attribute integer année ;
attribute integer longueur ;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Stars> acteurs
inverse Star : :JoueDans;
float longueurHeures() raises(ErreurHoraire) ;
void NomsActeurs(out Set<String>) ;
void AutresFilms(in Star, out Set<Film>)
raises(PasValable) ;;
};
```

Lire :

```
float longueurHeures( ) raises(ErreurHoraire)
```

ainsi : longueurHeures est une **fonction**, dont le type d'arrivée est float, et peut soulever des exceptions.

NomActeurs et AutresFilms sont des **procédures**.

AutresFilms prend en entrée un paramètre de type Star. Etant donné un objet *f* de la classe Film **et une star *s* qui joue dans *f***, cette procédure affecte à une variable de type Set<Film> l'ensemble des autres fils dans les quels *s* joue.

## 4.1 Sous-classes et Héritage

Avec ODL on peut déclarer qu'une classe  $C$  est une *sous-classe* d'une autre classe  $D$ .

Mot-clé : extends

Par ex. :

```
class Cartoon extends Film {  
relationship Set<Star> voix ;  
}
```

Ici, les objets de Cartoon ont, par rapport à Film, la relation nouvelle voix avec l'ensemble de stars qui donnent leur voix aux personnages.

De plus, une sous-classe  $C$  de  $D$  hérite toutes les propriétés de  $D$ . Dans l'exemple : tout objet de Cartoon a les attributs titre, année, longueur, SorteFilm et les relations Acteurs, etc.

## Héritage Multiple

Supposons que nous avons défini aussi une classe `Policier`, qui est une autre sous-classe de `Film`. Le film *Roger Rabbit* appartient au même temps à `Cartoon` et `Policier`.

Comment faire en ODL ? Déclarer une nouvelle classe : `CartoonPolicier` extends va être suivi par plusieurs noms de classes, séparées par “:”

```
class CartoonPolicier extends Policier : Cartoon ;
```

## Héritage multiple : problème des conflits de noms

*Exemple.* Une sous-classe de `Film` qui s'appelle `FilmAmour` a un attribut `fin`, de type énuméré `{ happy, triste }`. Une autre sous-classe de `Film` qui s'appelle `FilmTribunal` a aussi un attribut `fin`, de type énuméré `{ coupable, innocent }`.

Ambiguïté pour la classe `FilmTribunalEtAmour`, sous-classe des 2.

Le standard ODL ne dicte pas quoi faire.

Possibilités :

- Préciser la classe dont il faut hériter la signification de l'attribut `fin`
- Renommer un attribut. Par ex., attribut `fin` de `FilmTribunal`  $\rightsquigarrow$  `verdict`

Un objet d'une classe  $C$  qui est une sous-classe de la classe  $D$  hérite aussi les méthodes de  $D$ .

C'est exactement comme dans les langages de programmation orientés objet.

Ensemble des définitions ODL décrivant les propriétés des classes et leur relations hiérarchiques

=

Définition d'un **schéma** de base de données **à objets**

## **Extension d'une classe**

Une définition d'une classe  $C$  en ODL spécifie le format (schéma) de la classe. Pour faire référence à l'ensemble des instances de la classe (l'“extension” de  $C$ ) et pouvoir interroger la base : mot clé `extent`.

```
class Film (extent Films)
attribute string titre ;
:
```

*N.B.* : Expression `Film`  $\neq$  Expression `Films`.

(On aurait pu utiliser `TOTOS`, à la place de `Films`, certes, mais pas `Film`).

**N.B.** Le mots clé `extent` est différent du mot clé `extends` :-)

Même si chaque objet a une identité unique on PEUT vouloir traiter plusieurs objets comme “non-distinguables” par rapport aux propriétés observables, même si chacun garde sa propre identité.

C’est alors sensé de déclarer une *clé* : mot clé = key

```
class Film
  (extent Films key (titre, année))
{
  attribute string titre ;
  :
}
```

## 4.2 Un langage de Requête pour les BD Objet : OQL. Quelques Notions

- ODMG : *Object Data Management Group* (auteur aussi de ODL).

Voir :

[http://en.wikipedia.org/wiki/Object\\_Data\\_Management\\_Group](http://en.wikipedia.org/wiki/Object_Data_Management_Group)

- *Object Query Language* : la norme proposée par le groupe ODMG pour les langages de requête à objet, et implémentée par les SGBD à objets participant à ce groupe.
- Notation à la SQL. Mais pas de compatibilité avec le SQL relationnel.
- Utilisé comme extension d'un langage de programmation hôte, comme  $C^{++}$  ou Java.

## *Format Général d'une requête OQL*

```
SELECT liste d'expressions  
FROM liste d'une ou plusieurs déclarations de variables.  
WHERE condition C de selection
```

Une variable est déclarée en indiquant :

1. Une expression C dont la valeur est une collection d'objets.
2. Le nom de la variable, x, par exemple.

Analogie entre la déclaration d'une variable et l'indication d'une relation, après le FROM, dans une requête du SQL relationnel.

### **Syntaxes alternatives pour déclarer une variable :**

a la place de C x on peut aussi écrire :

```
x in C
```

```
x : C
```

## Notation “.”

Soit  $o$  un objet d’une classe  $C$ .

- Si  $p$  est un attribut,  $o.p$  est la valeur de  $p$  pour  $o$ .
- Si  $p$  est une relationship,  $o.p$  est l’objet ou la collection d’objets reliés à  $o$  par  $p$ .
- Si  $p$  est une méthode (éventuellement avec paramètres  $a_1, \dots, a_k$ ),  $o.p(\dots)$  est le résultat de l’application de  $p$  à  $a$ .

*N.B* : Puisque la déclaration de la méthode `NomsActeurs` de la classe `Film` est :  
`void NomsActeurs(out Set<String> ) ;`  
cette méthode est une procédure. Donc si `MonFilm` est un objet de la classe `Film`,  
l’expression `MonFilm.NomsActeurs(mesStars)` ne renvoie pas de valeur,  
mais, comme effet de bord, fait si que la valeur de la variable `output mesStars` de la  
méthode, qui est de type `Set<String>`, soit un ensemble de noms d’acteurs.

Attention : si  $o.p$  est une collection, ça n’a pas de sens de lui appliquer la notation pointée  $o.p.q$  ! Voir les exemples suivants.

## Exemple d'une BD à objet pur illustrer OQL (spécifiée en ODL)

```
class Film
(extent Films key (titre, année))
{
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Star> acteurs inverse Star : :JoueDans;
relationship Studio appartient-à inverse Studio : :possède;
float longueurHeures() raises(pasLongueurTrouvée);
void NomsActeurs(out Set<String>);
void autresFilms (in Star, out Set<Film>) raises(PasActeur);
};

class Star
(extent Stars key nom)
{
attribute string nom;
attribute Struct Adr
{string rue, string ville } adresse;
relationship Set<Film> JoueDans inverse Film : :acteurs;
};

class Studio
(extent Studios key nom)
{
attribute string nom;
attribute string adresse;
relationship Set<Film> :possède inverse Film : :appartient-à;
};
```

## Exemples de Requêtes OQL

```
SELECT m.année  
FROM Films m  
WHERE m.titre = ``Autant en emporte le vent``
```

Ici, `Files m` déclare que la variable `m` doit recevoir une valeur qui appartient à l'*extension* `Files` de la classe dont le nom est `Film`.

Une écriture alternative de cette déclaration de variable :

```
m in Files.
```

Dans cette requête, écrire `m.titre` a un sens, car la variable `m` reçoit un objet, qui est un élément de la classe des films.

## Exemples de Requêtes OQL, suite

```
SELECT DISTINCT s.nom  
FROM Films m, m.acteurs s  
WHERE m.appartient-à.nom = ``Disney``
```

Comme en SQL, le mot clé DISTINCT élimine les répétitions dans le multi-ensemble (bag) résultat.

## Exemples de Requêtes OQL, suite

Liste des titres des films du studio Disney, ordonnés par leur longueur ; si deux films ont la même longueur, trier selon l'ordre alphabétique des titres.

```
SELECT m.titre  
FROM Films m  
WHERE m.appartient-à.nom = ``Disney``  
ORDER BY m.longueur, m.titre
```

## Exemples de Requêtes OQL, suite

Les noms des acteurs du film 'Casablanca'

```
SELECT s.nom  
FROM Films m, m.acteurs s  
WHERE m.titre = ``Casablanca``
```

**Attention** : `m.acteurs` est une collection, et la valeur de la variable `s` sera un élément de cette collection.

## Exemples de Requêtes OQL, suite

Attention, l'écriture suivante de la requête qui calcule les noms des acteurs du film Casablanca n'aurait pas été correcte :

```
SELECT m.acteurs.nom  
FROM Films m  
WHERE m.titre = ``Casablanca``
```

En fait, `m.acteurs` renvoie une **collection**, donc la notation pointée `m.acteurs.nom` n'est pas correcte ! Comparer avec l'écriture précédente, qui était correcte.

## Exemples de Requêtes OQL, suite

On veut l'ensemble des couples de stars qui vivent à la même adresse :

```
SELECT DISTINCT Struct(star1 :s1, star2 : s2)
FROM Stars s1, Stars s2
WHERE s1.adresse = s2.adresse AND s1.nom < s2.Nom
```

Pour chaque couple qui passe le test on produit un record, avec 2 champs, nommés `star1` et `star2`. Le type de chaque champ est la classe `Star`.

Le type du résultat final de cette requête (qui est un ensemble de couples d'acteurs) est :

```
Set<Struct{star1 : Star, star2 : Star}>
```

## Exemples de Requêtes OQL, suite

### *Utilisation de “sous-requêtes”*

Une autre façon de chercher les noms des acteurs des films du studio Disney (la requête du premier exemple) :

```
SELECT DISTINCT s.nom
FROM (SELECT m
FROM Films m
WHERE m.appartient-à.nom = ''Disney'') f,
f.acteurs s
```

La variable *m* renvoie la collection des films du studio Disney, et cette collection est le résultat de la sous-requête :

```
SELECT m
FROM Films m
WHERE m.appartient-à.nom = ''Disney''
```

Donc la variable *f* va être affecté, chaque fois, à un objet de cette collection (un film de Disney) et la variable *s* prend valeur dans la collections des acteurs du film de Disney *f*.

## Exemples de Requêtes OQL, suite

### Sous-classes

Si on a déclaré `Cartoon` comme sous-classe de `Film`, et `TheCartoons` comme son ensemble de valeurs (avec le mot-clé `extends` de ODL), alors la requête

```
SELECT m
FROM Films m
WHERE m.appartient-à.nom = ``Disney``
ORDER BY m.longueur, m.titre
```

crée un ensemble d'objets contenant les films du studio Disney, peu importe s'ils sont de la classe `Cartoon` ou pas.

Comparer à :

```
SELECT m
FROM TheCartoons m
WHERE m.appartient-à.nom = ``Disney``
ORDER BY m.longueur, m.titre
```

# 5 Bases Relationnelles-Objet

## Le relationnel-objet sur Oracle

Un système de gestion de bases de données *relationnel-objet* est, essentiellement, un SGBD relationnel, mais qui **simule** la définition de classes, donc **l'imbrication de structures typique du modèle objet**, et l'héritage.

Historiquement, la recherche sur le relationnel-objet a commencé au début des années 90. Beaucoup de ces idées ont été incorporées dans SQL.

IBM's DB2, Oracle, et Microsoft SQL Server les supportent (avec des niveaux de réussite variables).

Au département : Oracle (version 11), et on va va étudier certaines de ses caractéristiques objet. L'adresse permettant de récupérer la doc de Oracle 11 est en ligne.

## Definition de Types : UDT(*User Defined Types*)

En plus des types usuels (VarChar, Number, Date etc), l'utilisateur peut travailler avec ses propres types.

Oracle permet de définir des types **OBJECT**. Syntaxe :

```
CREATE TYPE t AS OBJECT (  
    liste d'attributes et methodes  
);  
/
```

Ceci déclare le nouveau type, qui s'appelle t, dont chaque élément est un objet structuré, comportant des attributs et des méthodes. Tout ce qui est de type OBJECT **possède un oid et peut être référencé** (voir après).

## Definition de Types, suite.

Syntaxe alternative, permettant, éventuellement, de redéfinir un type :

```
CREATE OR REPLACE TYPE t AS OBJECT (  
    liste d'attributes et methodes  
);  
/
```

## Definition de Types, suite

**Exemple.** Définition d'un type pour représenter un point par ses coordonnées :

```
CREATE TYPE PointType AS OBJECT (  
    x NUMBER,  
    y NUMBER  
);  
/
```

**NB :**

Analogie avec la définition d'un type structuré ayant deux champs, x et y dont les types sont atomiques, en ODL. **Imbrication !**

## Definition de Types, suite

Un type OBJECT peut être utilisé pour déclarer d'autres types OBJECT. Par exemple, on peut définir un type pour les lignes géométriques (finies, c.à.d. les segments), où on indique 2 points de la ligne :

```
CREATE TYPE LineType AS OBJECT (  
    end1 PointType,  
    end2 PointType  
);  
/
```

**NB :**

Analogie avec la définition d'un type structuré ayant deux champs, ayant eux mêmes un type structuré (PointType), en ODL.

## Tables

On peut alors créer une table relationnelle *qui n'est pas en première forme normale*.

Creation d'une table de lignes, où, de plus, chaque ligne est nommée par la valeur de l'attribut `lineID` :

```
CREATE TABLE Lines (  
    lineID INT,  
    line    LineType  
);
```

Analogie avec la définition d'une classe en ODL, ayant deux attributs, `lineID` et `line`, où le second a un type structuré.

## Elimination de Types

Pour éliminer un type comme `LineType` :

```
DROP TYPE Linetype;
```

(“to drop” en anglais : laisser tomber, jeter.)

**N.B.** On doit d’abord effacer toutes les tables et les autres types qui utilisent ce type (dans l’exemple : la table `Lines`).

**Construction d'objets (valeurs).** Constructeurs pre-définis pour créer des valeurs, dont les noms sont les noms des types.

**Suite de l'exemple.** Pour créer une valeur de type `PointType` : le mot `PointType` (nom du type) suivi par les valeurs en `()`.

Construction d'une ligne de la table `Lines`, la 27, qui part du point (0,0) et arrive au point (3,4) :

```
INSERT INTO Lines
VALUES(27, LineType(
                PointType(0.0, 0.0),
                PointType(3.0, 4.0)
                )
);
```

**N.B.** On utilise `INSERT` comme pour l'insertion dans une table standard. On a créé 2 valeurs de type `PointType` et 1 valeur de type `LineType`.

**Déclarations de méthodes.** Une déclaration de type peut inclure la déclaration de méthodes, à l'aide de `MEMBER FUNCTION` ou `MEMBER PROCEDURE`. Comme en ODL, il faut spécifier les types des entrées et sortie.

**Suite de l'exemple.** On ajoute une fonction `length` au type `LineType`, qui produit la longueur d'une ligne et la multiplie par un facteur (`scale`).

```
CREATE TYPE LineType AS OBJECT (  
    end1 PointType,  
    end2 PointType,  
    MEMBER FUNCTION length(scale IN NUMBER) RETURN NUMBER,  
    PRAGMA RESTRICT_REFERENCES(length, WNDS)  
);  
/
```

La dernière instruction (`PRAGMA . . .`) dit que la méthode ne change pas la BD. (`WNDS = Write No Database State`). C'est une clause nécessaire, si l'on veut utiliser `length` dans une requête.

## Définition du code d'une méthode

Le code de la méthode est donné à part, dans une instruction `CREATE TYPE BODY`. Ici, on ne répète pas les types des entrées (et des sorties des procédures). La variable spéciale `SELF` dans la déf. de la méthode indique la valeur courante.

## Suite de l'exemple

```
CREATE TYPE BODY LineType AS
    MEMBER FUNCTION length(scale NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN scale *
            SQRT((SELF.end1.x-SELF.end2.x)*(SELF.end1.x-SELF.end2.x)
                (SELF.end1.y-SELF.end2.y)*(SELF.end1.y-SELF.end2.y)
            );
    END;
END;
/
```

## Quelques exemples de requête relationnelle-objet

Calcul du double de la longueur des chaque ligne :

```
SELECT  ll.line.length(2.0)
        FROM Lines ll;
```

La requête utilise la méthode `length`.

**Attention** : Il faut utiliser une variable (ici, `ll`) pour accéder à un champ avec la notation “.”. Par exemple, l’écriture suivante ne marchera pas :

```
SELECT  line.length(2.0)
        FROM Lines;
```

C’est comme pour OQL : après le `FROM` on déclare une variable comme recevant une valeur dans une collection fixée (ici : la table des lignes).

Quelques exemples de requête relationnelle-objet, suite

Coordonnées de la première extrémité de chaque ligne :

```
SELECT ll.line.end1.x, ll.line.end1.y  
       FROM Lines ll;
```

Seconde extrémité de chaque ligne, comme valeur de type PointType :

```
SELECT ll.line.end2  
       FROM Lines ll;
```

Une réponse possible : `PointType(3.0,4.0)`.

## Syntaxe Create TABLE ... OF...

Les deux écritures suivantes créent une table “dont le schéma est LineType” :

```
CREATE TABLE Lines1 (  
    end1 PointType  
    end2 PointType  
);
```

```
CREATE TABLE Lines1 OF LineType;
```

**Mais, attention, seulement cette seconde écriture permet d'utiliser la méthode length.**

Pourquoi ?

## Types Références

Si  $t$  est un type OBJECT, REF( $t$ ) (ou REF  $t$ ) est les type des références à des valeurs de type  $t$ , par le biais de leurs OID.

Exemple :

définition d'une table Lines2 dont chaque ligne est donnée par référence à 2 points :

```
CREATE TABLE Lines2 (  
    end1 REF PointType,  
    end2 REF PointType  
);
```

## Exemple, suite : définition du contenu d'une table avec REF.

Supposons qu'on a déjà une table `Points` dont les valeurs sont de type `PointType`. On remplit la table `Lines2` avec des valeurs qui sont toutes les lignes dont les extrémités (gauche et droite) sont des éléments de la table `Points` :

```
INSERT INTO Lines2
    SELECT REF(pp), REF(qq)
    FROM Points pp, Points qq
    WHERE pp.x < qq.x;
```

**Attention** : On peut référencer seulement un objet qui existe déjà dans une table. Par exemple, ceci ne marchera pas :

```
INSERT INTO Lines2
    VALUES(REF(PointType(1.3,2.5)), REF(PointType(3.4,4.7)));
```

Les objets `PointType(1,2)` et `PointType(1,2)` ne vivent dans aucune table !

Pour [suivre une référence](#) avec le ., on fait comme si la valeur d'un attribut de type référence était littéralement la valeur du type référencé.

**Exemple** : Calcul des coordonnées x des extrémité de chaque ligne élément de Lines2 :

```
SELECT l1.end1.x, l1.end2.x  
FROM Lines2 l1;
```

## Encore sur l'usage de REF

Un autre exemple de déclaration de type et de table à l'aide des références :

```
CREATE TYPE T-Personne AS OBJECT
  (NSS NUMBER
   Nom VARCHAR(18)
   Prenom VARCHAR(18)
   conjoint REF T-Personne) /
```

L'attribut conjoint pointera à un oid de type T-Personne.

Penser à une *relationship* conjoint d'une classe de personnes en ODL !

```
CREATE TABLE LesPersonnes OF T-Personne
```

On a créé une table d'objets de type T-personne.

## Encore sur l'usage de REF, suite de l'exemple

La clause REF(variable-objet) donne en résultat l'oid d'un objet.

La requête suivant insère un objet (Philippe Tochat) et *initialise son lien de référence* conjoint (sur Annie Muller) :

```
INSERT INTO LesPersonnes(NSS,Nom,Prenom,Conjoint) VALUES
(17924457911, 'Rochat', 'Philippe',
SELECT REF(p) FROM LesPersonnes p
WHERE p.Nom='Muller' AND p.Prenom='Annie')
```

**N.B** : Annie Muller doit exister déjà, pour pouvoir la référencer. Comment éviter la circularité ? A voir en TD.

## Encore sur les références, suite de l'exemple

La clause `DEREF(oid)` permet de récupérer la valeur de l'objet identifié par l'oid.

La requête :

```
SELECT p.Nom, p.conjoint FROM LesPersonnes p;
```

donnera les noms des personnes et l'oid de leur conjoint. En revanche, la requête :

```
SELECT p.Nom, Deref(p.conjoint) FROM LesPersonnes p;
```

donnera, pour chaque personne `p`, son nom, et la “valeur” de son conjoint (qui est un objet), donc : son NSS, son nom, son prénom, et l'oid de son conjoint (c.à.d. l'oid de `p`, si le conjoint du conjoint de `p` est `p`).

## Relations imbriquées

On a vu comment définir une table où la valeur d'un attribut est structurée (exemple : table `Line`). On peut aussi créer une table où la **valeur d'un attribut** est une **relation elle-même**.

Penser à l'exemple avec les musiciens et leur opéras du début du cours ! On **simule des tables imbriquées** !

## Relations imbriquées, déclaration du schéma

Définition du schéma d'une table dont les éléments sont des polygones, tout polygone étant une table (de points) lui-même.

D'abord, la création du type `PolygonType` :

```
CREATE TYPE PolygonType AS TABLE OF PointType;  
/
```

Le type que l'on vient de créer est tel que chaque valeur de ce type est elle-même une table relationnelle dont les  $n$ -uplets sont tous de type `PointType`.

## Relations imbriquées, déclaration du schéma, suite.

Création de la table des polygones :

```
CREATE TABLE Polygons (  
    name    VARCHAR2(20),  
    points  PolygonType)  
    NESTED TABLE points STORE AS PointsTable;
```

Chaque ligne de la table créée est un polygone, qui a un nom (“triangle1”, triangle2, “rectangle1”, “hexagone1”, etc) et est constitué par un ensemble (table) de points.

Lors de l’emploi du type `PolygonType` comme domaine d’un attribut d’une autre table (ici : comme domaine de `points`, pour la table `Polygons`), Oracle crée physiquement une table annexe dans laquelle il stockera les valeurs de cet attribut.

Donc, lors de la création de la table `Polygons` on doit définir le nom de cette table annexe ; c’est le rôle de l’instruction `NESTED TABLE points STORE AS PointsTable` qui donne un nom à cette table annexe.

## Relations imbriquées, remplissage des tables

La valeur de chaque relation de niveau inférieur est représentée par une liste de valeurs du type approprié (PolygonType dans notre exemple).

Exemple d'insertion d'un polygone qui est un carré :

```
INSERT INTO Polygons VALUES(  
    'square', PolygonType(PointType(0.0, 0.0), PointType(0.0, 1.0),  
                          PointType(1.0, 0.0), PointType(1.0, 1.0))  
);
```

## Relations imbriquées, exemples de requêtes

### Les points du carré :

```
SELECT points
FROM Polygons
WHERE name = 'square' ;
```

### Les points du carré qui sont sur la diagonale (x=y) :

```
SELECT ss
FROM THE(SELECT points
        FROM Polygons
        WHERE name = 'square'
        ) ss
WHERE ss.x = ss.y ;
```

(sous-requête, relation du niveau inférieur dans le FROM avec le mot-clé THE et utilisation de la variable `ss` pour la nommer.) Le THE transforme une relation imbriquée en une relation ordinaire, à laquelle on peut appliquer un FROM.

## Hiérarchie de types OBJECT

En Oracle, à partir de la version 9, on peut créer des hiérarchies de types OBJECT.

Pour déclarer un type  $t$  pour lequel on va déclarer des sous-type, il faut ajouter la clause `NOT FINAL` lorsque on crée  $t$ .

Pour déclarer un sous-type  $t'$  d'un type non final  $t$ , il faut utiliser le mot-clé `UNDER`.

**Pas d'héritage multiple !** Un type  $t'$  peut avoir un seul sur-type  $t$ .

## Exemple de hiérarchie de types OBJECT

```
CREATE TYPE T-Personne2 AS OBJET
(NSS NUMBER
 Nom VARCHAR(18)
 Prenom VARCHAR(18)
 Adresse VARCHAR(200))
NOT FINAL/
```

```
CREATE TYPE T-ETUDIANT UNDER T-Personne2
(Université VARCHAR(18)
 Département VARCHAR(18))
/
```

## Exemple de hiérarchie de types OBJECT, suite

Quand le type  $t'$  est un sous-type de  $t$ , une valeur de type  $t'$  est aussi une valeur de de typê  $t$ .

```
CREATE Table LesPersonnes OF T-Personne2;
```

```
INSERT INTO LesPersonnes VALUES(T-ETUDIANT  
(22222222 'Muller', 'Annie',  
'Rue du Marché, 1, Paris', 'Evry', 'Informatique'))
```

```
CREATE TABLE LivresEmpruntés(livre VARCHAR(20),  
emprunteur T-Personne2);
```

```
INSERT INTO LivresEmpruntés VALUES('Les Bases de Données',  
(T-ETUDIANT  
22222222 'Muller', 'Annie', 'Rue du Marché, 1,  
Paris', 'Evry', 'Informatique'))
```

## Hiérarchie de types OBJECT, fin

La clause

```
<valeur> IS OF <nom-sous-type>
```

teste si la valeur appartient au sous-type.