Module MLBDA
Master Informatique
Spécialité DAC
Cours 2 – SGBD Objet

SGBD Objet

- Le modèle objet
- Le standard ODMG
- Le langage d'interrogation OQL

Modèle Objet

- Concepts :
  - Valeur et objet
  - Identité d’objet
  - Classe
  - Héritage

Valeur

- Valeur atomique (valeur de type simple : caractère, entier, booléen, …)
- Valeur complexe : les tuples, les ensembles, les listes, sont des valeurs complexes
  - Ex : [Max, 25, Paris] , {Jean, Marc, Paul}

Objet

- Un objet a un identificateur et une valeur.
- Un objet atomique a une valeur atomique
- Un objet complexe est construit en appliquant les constructeurs *tuple*, *set*, *list*, *array*, à des objets atomiques ou complexes.
- L'utilisation des constructeurs est indépendante du type des objets (orthogonalité du système de types)
- Ex : (i1, 25)

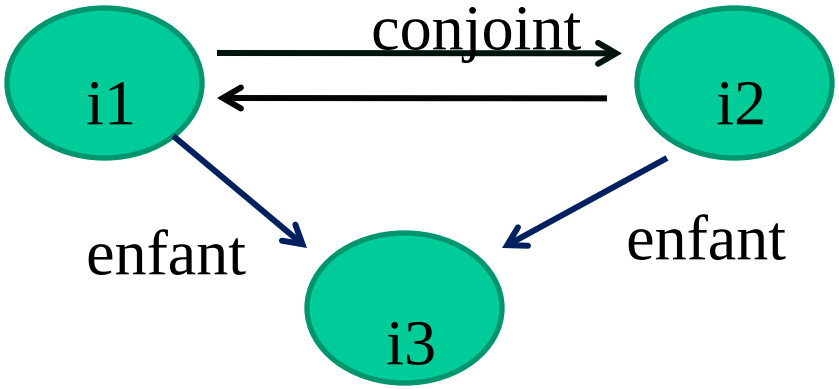
  (i2, [Max, i1, Paris, {Jean, Marc, Paul}])

Identité d’objet

- Chaque objet a une identité indépendante de sa valeur
- L’identificateur est géré par le système (correspond à une clef interne)
- Deux objets sont identiques s’ils ont le même identificateur, et sont égaux s’ils ont la même valeur.
- Les objets peuvent être représentés par un graphe de composition, qui peut comporter des cycles

Identité d'objet

- L'identité d'objet permet
  - le partage d'objets,
  - les cycles entre objets,
  - le maintien automatique des contraintes référentielles,
  - la mise à jour de la valeur sans changer l'identité,
  - plusieurs niveaux de comparaison (égalité profonde, égalité superficielle)
- Ex:



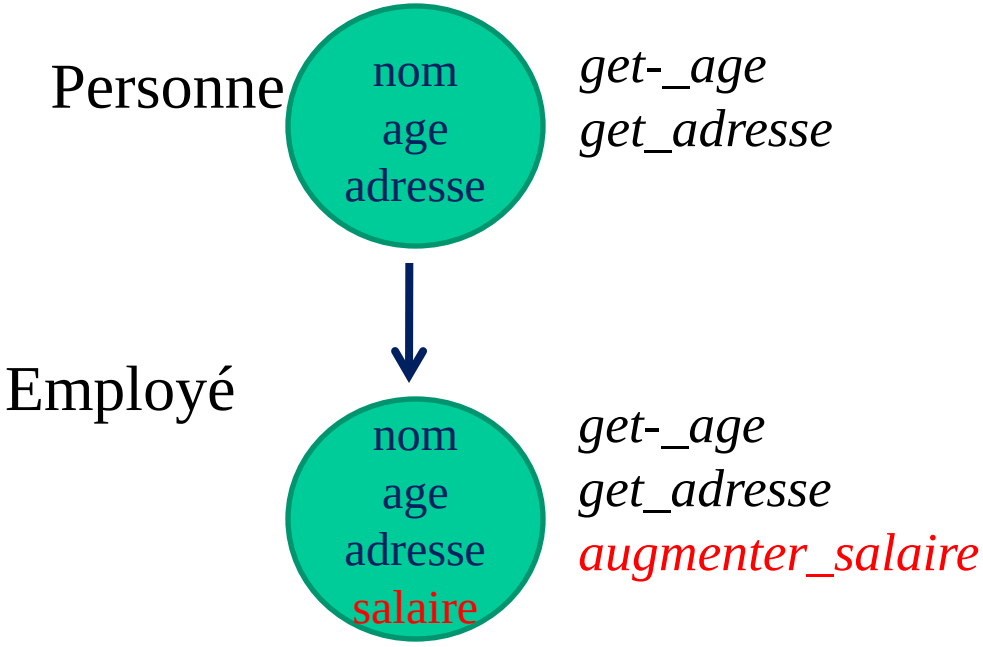
Classe

- Les objets partageant des caractéristiques communes (structure et comportement) sont regroupés dans des classes
- Les classes permettent une abstraction des informations et de leur représentation. Ce sont les concepts utilisés pour décrire le schéma de la base.
- Ex:

```
Class Personne
        [nom : string,
        age : integer,
        adresse : string,
        conjoint : Personne,
        enfants : {Personne} ]
    method get_age() : integer,
        get_adresse(): string
```

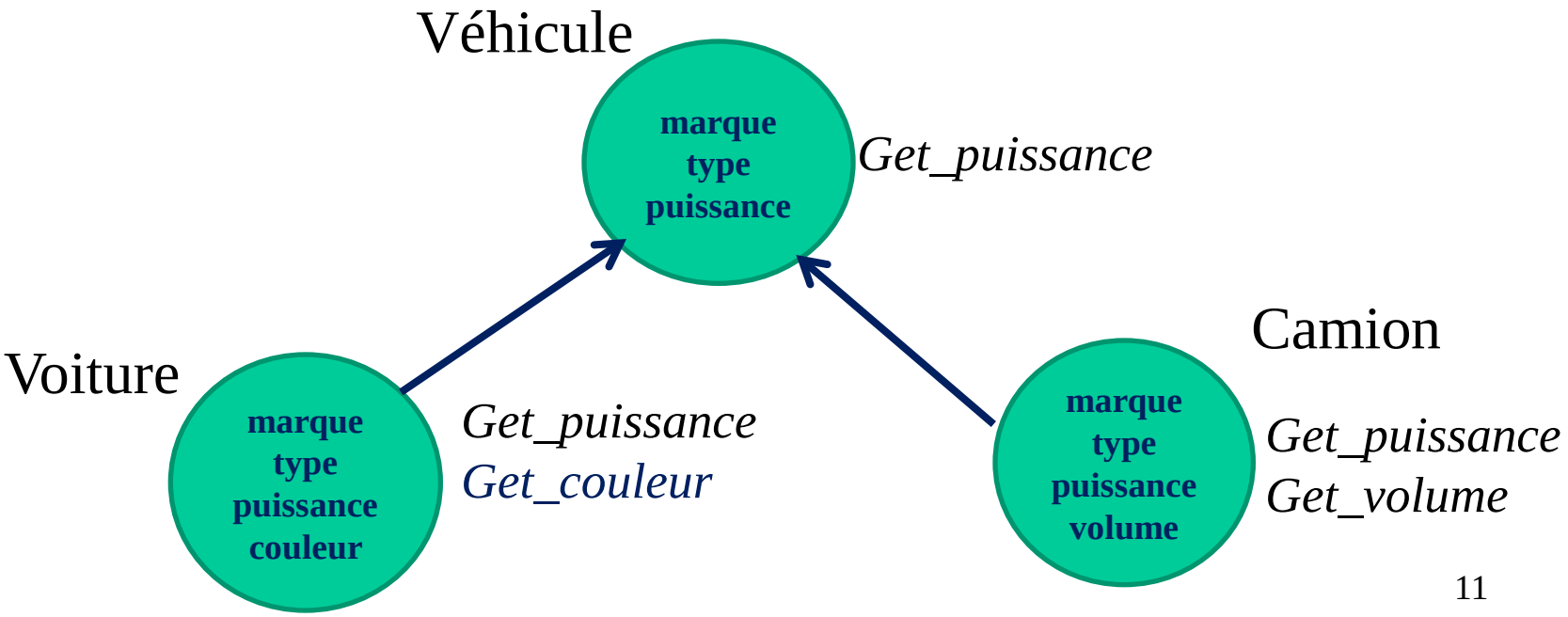
Héritage

- Factoriser des classes ayant des propriétés communes (structure et/ou méthodes). Une sous-classe hérite des propriétés de sa super-classe.
- Spécialisation : affiner une classe en une sous-classe
  - Spécialiser la classe *Personne* en la classe *Employé*



Héritage

- Généralisation : création d'une super-classe regroupant les caractéristiques communes à plusieurs classes.
  - Généraliser les classes *Voiture* et *Camion* en une classe *Véhicule*



L'apport des modèles objets

- Identité d'objets
  - introduction de pointeurs invariants
  - possibilité de chaînage
- Encapsulation des données
  - possibilité d'isoler les données par des opérations
  - facilite l'évolution des structures de données
- Héritage d'opérations et de structures
  - facilite la réutilisation des types de données
  - permet l'adaptation à son application
- Possibilité d'opérations abstraites (polymorphisme)
  - simplifie la vie du développeur

Définitions

BDO = base de données dont les éléments sont des objets (avec identité et opérations)

schéma BDO = graphe de classes des objets de la base

SGBDO =

SGBD : persistance, langage de requêtes, gestion du disque, partage de données, fiabilité des données, sécurité

+

- langages objet (C++, Smalltalk, Java, etc.)
- objets structurés et non structurés
- objets volumineux (multimédias)
- objets complexes (avec associations inter-objets)
- composants et appel d'opérations
- héritage et surcharge d'opération

Persistance

Un objet persistant est un objet stocké dans la base, dont la durée de vie est supérieure au programme qui le crée.

Une donnée est rendue persistante par une commande explicite (LP) ou par un mode automatique (BD):

- attachement à une racine de persistance
- attachement à un type de données

Les noms du schéma servent de points d'entrée à la base de données (les relations en relationnel, les noms de classe, ou des objets nommés en BDOO)

ODMG
(Object Database Management Group)

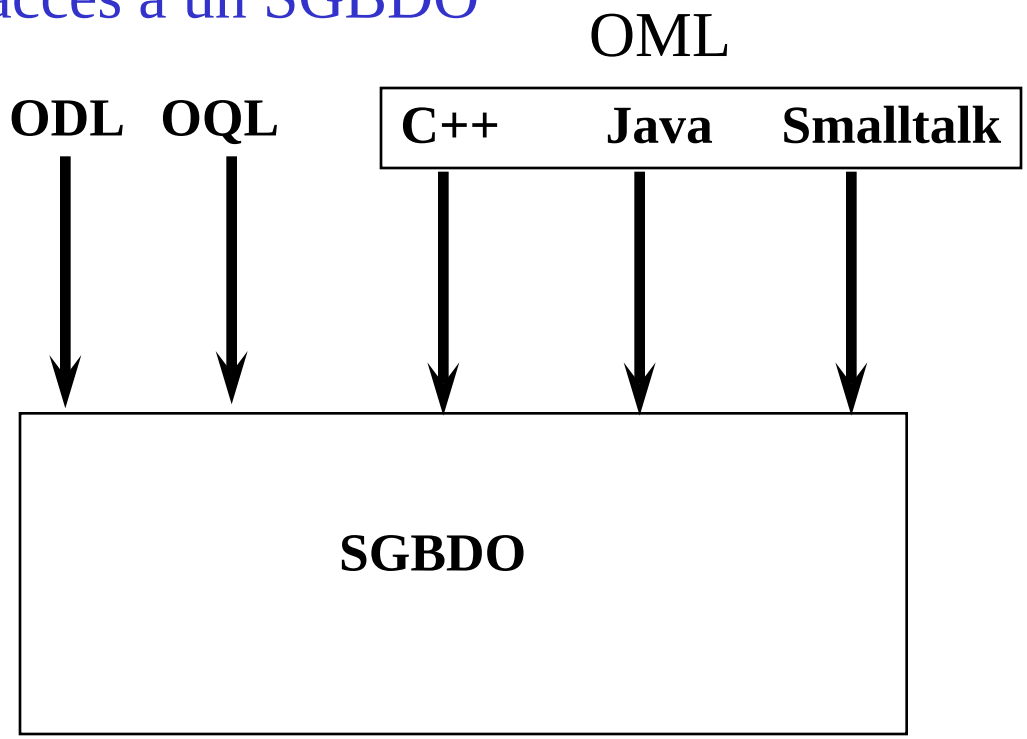
Proposé en 1996, par l'OMG.

Définit les standards pour les BDO

- modèle BDO, extension du modèle OMG
- langage de définition d'objets ODL
- OQL langage de requêtes fonctionnel
- interfaces avec LPO (C++, Smalltalk, Java, etc.)

Contenu de la proposition

- Adaptation du modèle objet de l'OMG
- Interfaces d'accès à un SGBDO

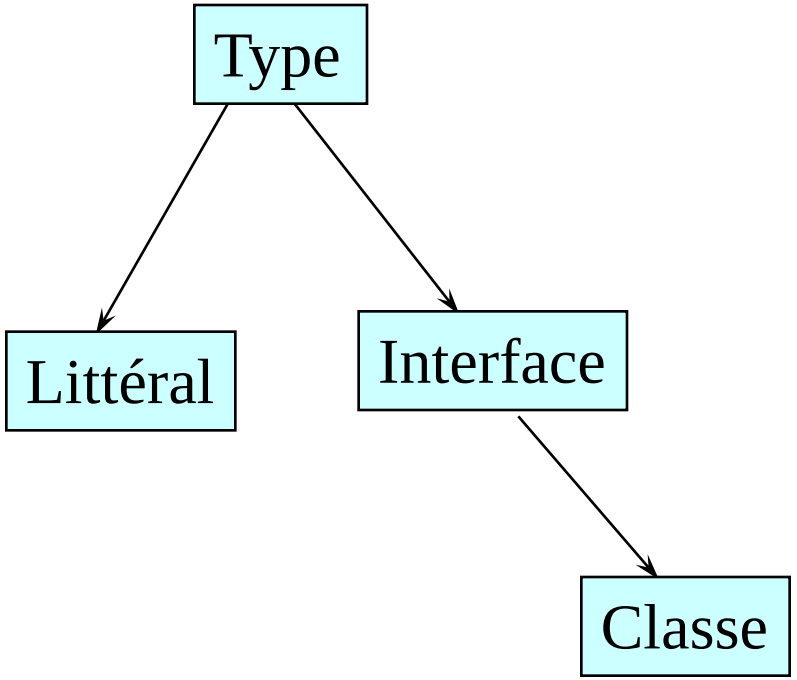


Le Modèle

- Extension du modèle de l'OMG
  - l'OMG a proposé un modèle standard pour les objets
  - le modèle est supporté par le langage IDL (def. interface)
  - les BD objets nécessitent des adaptations/extensions
    - instances de classes
    - collections
    - associations
    - persistance
    - transactions
- Un candidat pour un profil BD de l'OMG

Type, Interface et classe

- Un modèle objet-valeur
- Valeurs = littéraux
  - entier, réel, string, ...
  - structure<>, enum<>
- Les objets
  - implémentent des interfaces (comportement)
  - peuplent des extensions de classes (comportement + état)
- Deux types d'héritage
  - comportement (interface)
  - d'état (classe)



Les Objets : Instances

- Identifiés par des OID :
  - gérés par le SGBD OO pour distinguer les objets
  - permettent de retrouver les objets, restent invariants
  - Peuvent être nommés par les utilisateurs
- Persistants ou temporaires :
  - les objets persistants sont les objets BD, les autres restent en mémoire
- Peuvent être simples ou composés :
  - atomiques, collections, structurés

Propriétés communes

- un ensemble d'opérations héritées pour:
  - création, verrouillage, comparaison, copie, suppression
- création des objets par des "usines"

```
interface ObjectFactory { Object new(); };
```

- héritage d'un type racine :

```
interface Object {
void lock(in Lock_Type mode) raises
  (LockNotGranted);
boolean try_lock(in Lock_Type mode);
boolean same_as(in Object anObject);
Object copy(); void delete() ; };
```

Les objets collections

- Support de collections homogènes :
  - **Set<t>, Bag<t>, List <t>, Array<t>, Dictionary<t,v>**
  - héritent d'une interface commune collection

```
Interface Collection : Object {
unsigned long cardinality();
boolean is_empty(), is_ordered(),
  allows_duplicates(), contains_element(in any
  element);
void insert_element(in any element);
void remove_element(in any element)
  raises(ElementNotFound);
Iterator create_iterator() ;
Bidirectionalterator
  create_bidirectional_iterator() ; };
```

Objets collections

- Un itérateur permet d'itérer sur les éléments :

```
Interface Iterator { void reset() ; any
  get_element() raises(NoMoreElements);
void next_position raises(NoMoreElements);
replace_element (in any element)
  raises(InvalidCollectionType) ; ...};
```

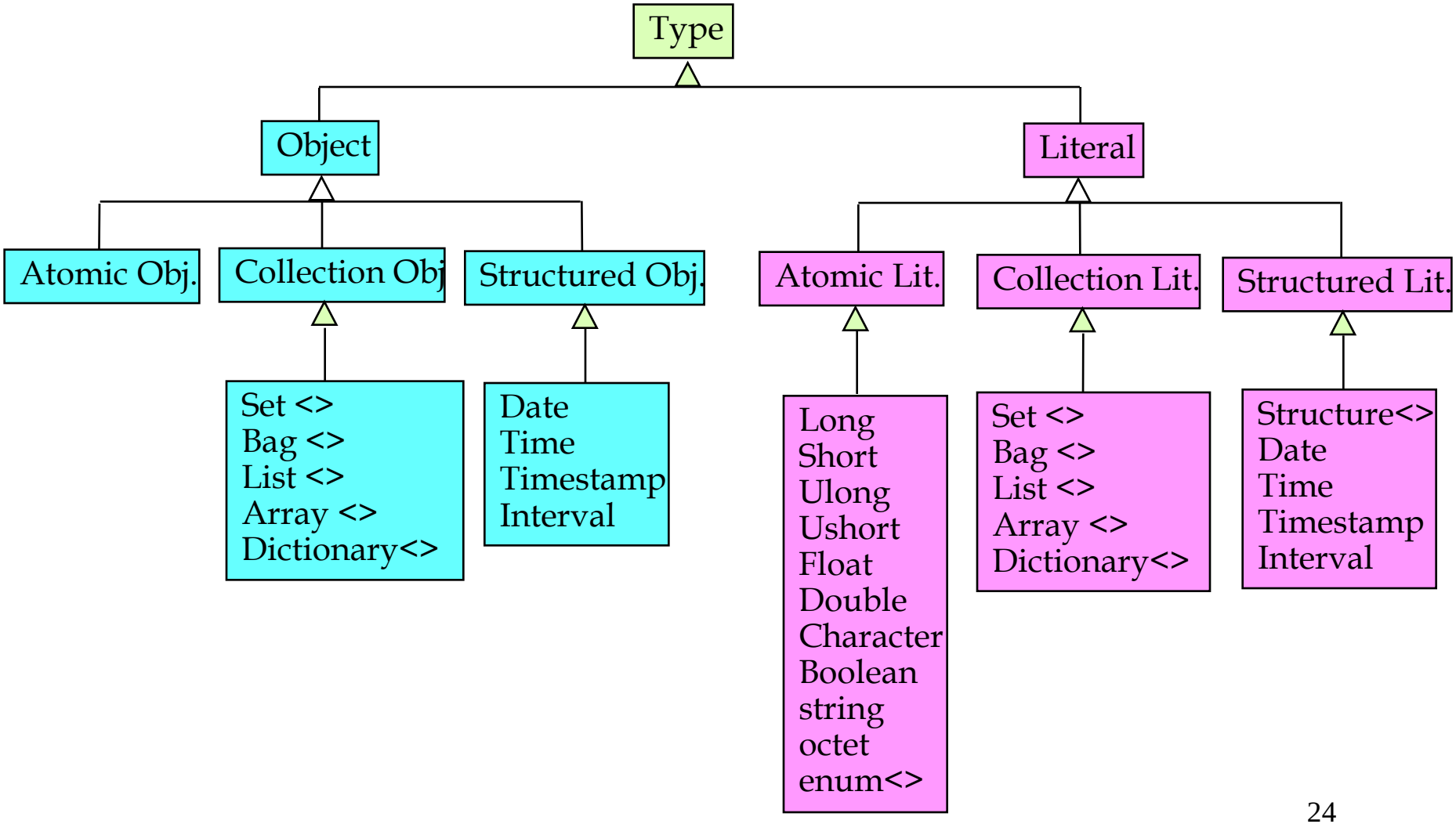
- Chaque collection a une interface spécifique

Exemple

- Exemple : Dictionnaire
  - une collection de doublets <clé-valeur>

```
Interface Dictionary : Collection
exception keyNotFound(any key);
void bind(in any key, in any value);
  //insertion
void unbind (in any key)raise(KeyNotFound);
  //suppression
void lookup (in any key)raise(KeyNotFound);
  //recherche
boolean contains_key(in any key) ;
  // test d'appartenance }
```

Hiérarchie de Types

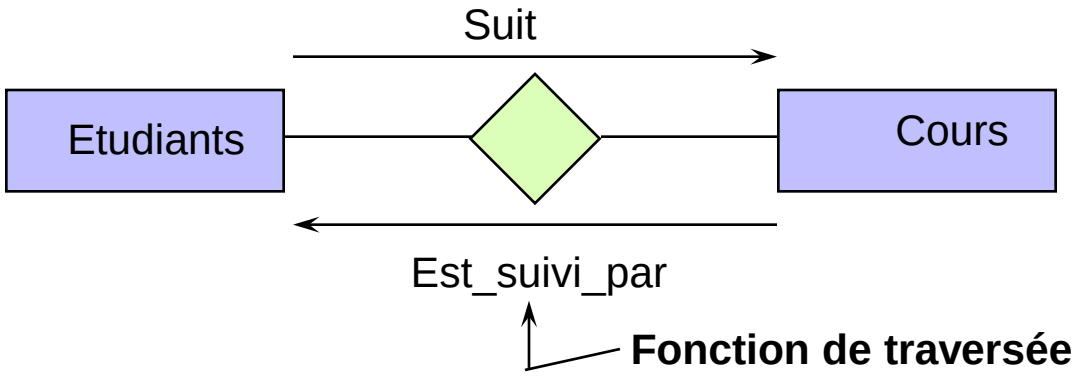


Les Attributs

- Une propriété permettant de mémoriser un littéral ou un objet
- Peut être vue comme deux fonctions :
  - Set_value
  - Get_value
- Propriétés:
  - type de ses valeurs légales
  - son nom

Ex :
```
Interface Personne {
        attribute string nom;
        attribute date datnais;
         }
```

Les Associations (Relationships)



- Associations binaires, bi-directionnelles de cardinalité (1:1), (1:N), (N:M).
- Opérations:
  - **`Add_member, Remove_member`**
  - **`Traverse, Create_iterator_for`**

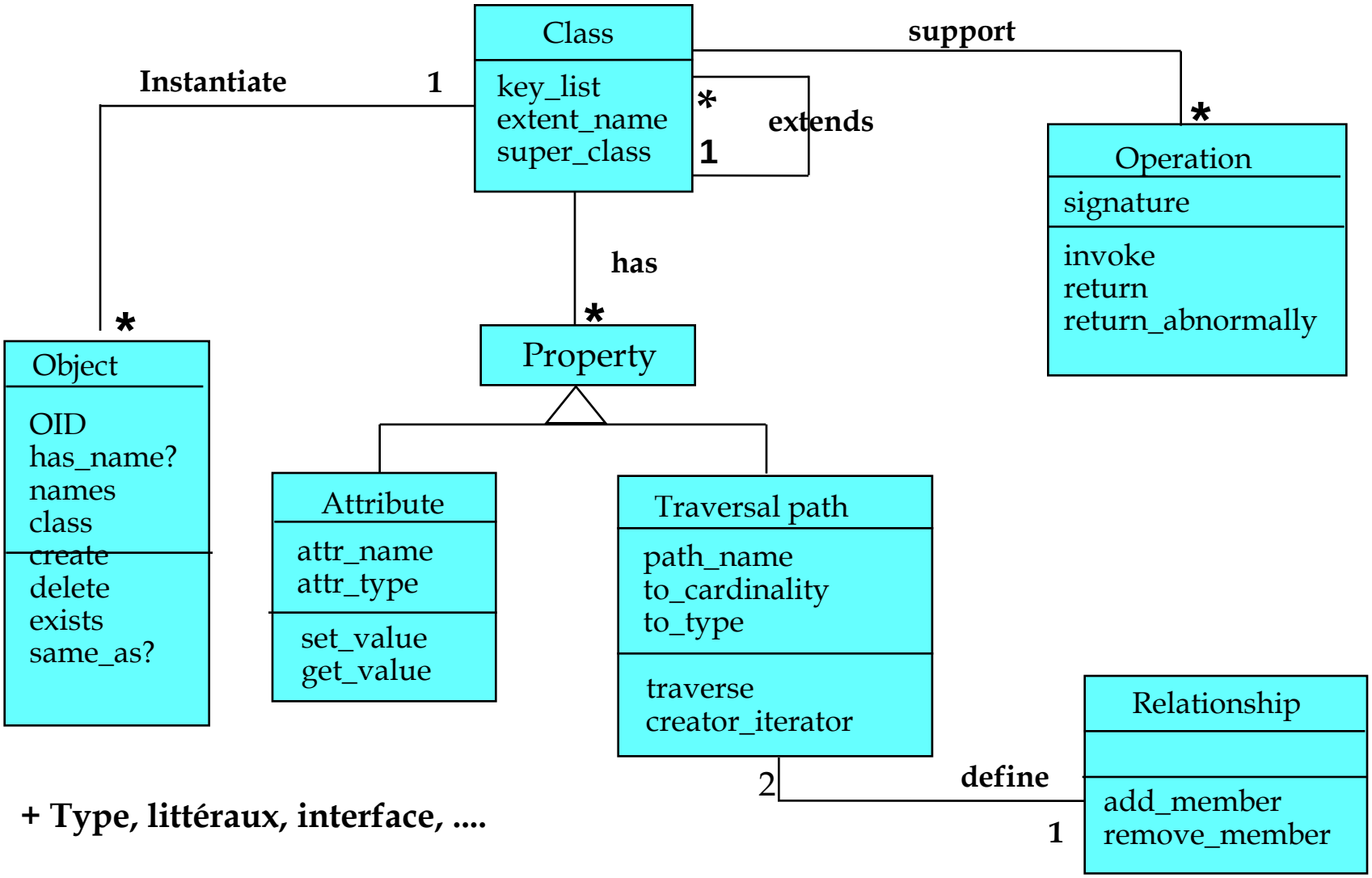
Exemple

- Exemple :
  - **Interface étudiants { ...**
    Relationship list<cours> suivre inverse cours::est_suivi_par;
    };
  - **Interface cours { ...**
    Relationship set<étudiants> est_suivi_par inverse étudiants: :suivre;
    };
- Le SGBDO est responsable du maintien de l'intégrité
  - Référence dans les deux sens si inverse déclaré
  - Ce n'est pas le cas pour un attribut valué par un objet
  - Ex : **attribut list<cours> suivre**
  - pas de chemin inverse, pas d'intégrité référentielle

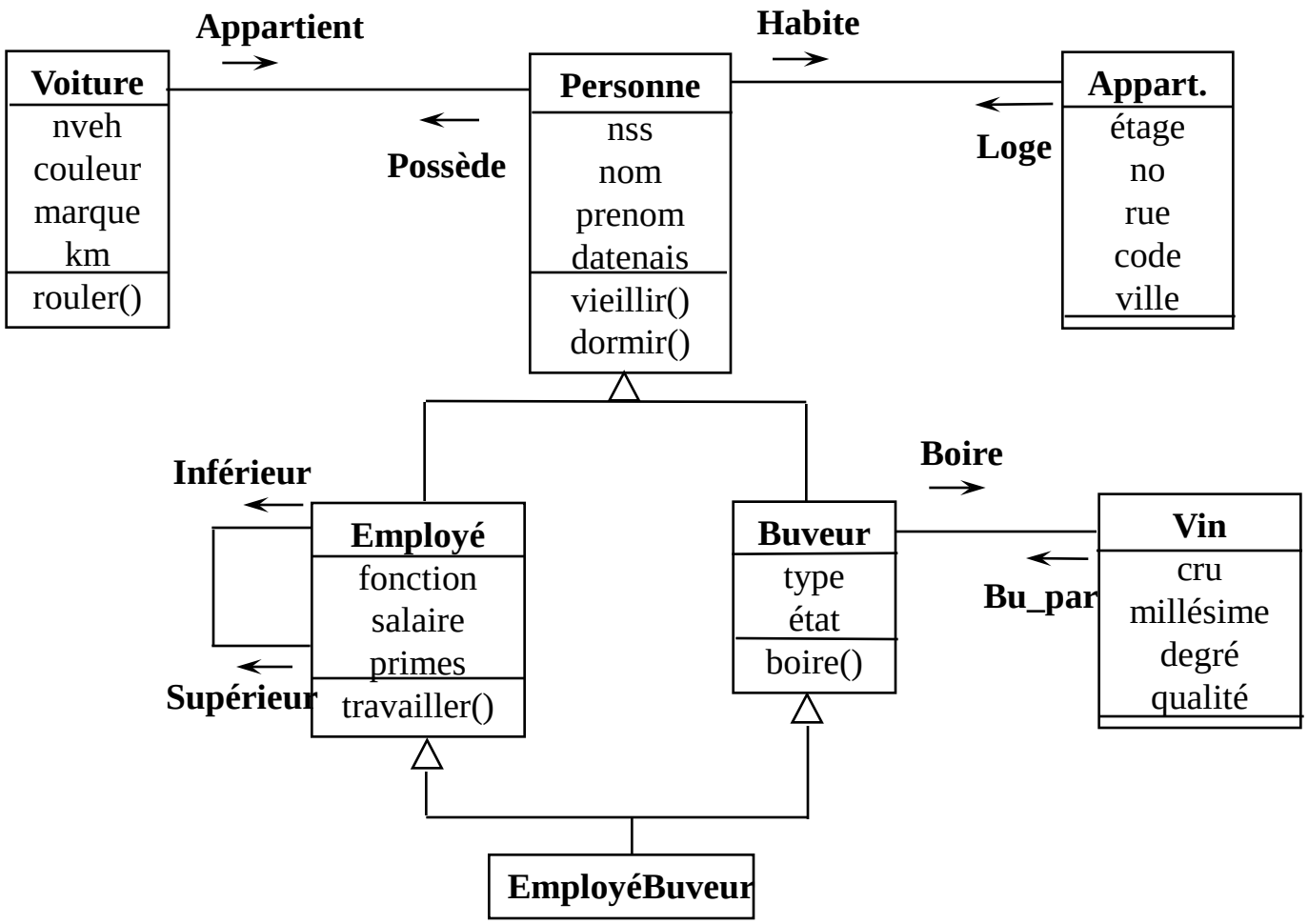
Les Opérations

- Représentent le comportement du type
- Propriétés:
  - nom de l'opération
  - nom et type des arguments (in)
  - nom et type des paramètres retournés (out)
  - nom des conditions d'erreurs (raises)

Méta-modèle du modèle ODMG



Exemple de base



Exemple de définition

```
Interface Personne {  // interface abstraite pour
  implémentation dans classe
  attribute string nss ;
  attribute string nom ;
  attribute string prenom ;
  attribute date datenaissance;
  relationship Appart habite inverse Appart:: loge;
  // relationship
  relationship Voiture Possede inverse
  Voiture::Appartient;
  short vieillir();
  void dormir();
  short age();
};
```

Exemple (suite)

```
class Employé : Personne(extent Employés key
  nss)
//classe avec extension de personne + {
  attribute enum fonct{ingénieur, secrétaire,
  analyste, programmeur} fonction;
  attribute float salaire ;
  attribute list<float> primes ; //attribut
  multi-valué
  relationship Employé inferieur inverse
  supérieur;
  relationship Employé supérieur inverse
  inférieur;
  void travailler();
};
```

Le langage OQL

- Permettre un accès facile à une base objet
  - via un langage interactif autonome
  - par intégration dans un des langages de l'ODMG (C++, Smalltalk, Java)
- Offrir un accès non procédural
  - permettre des optimisations automatiques (ordonnancement, index,…)
  - garder une syntaxe proche de SQL
- Rester conforme au modèle de l'ODMG
  - application d'opérateurs aux collections extensions ou imbriquées
  - permettre de créer des résultats littéraux, objets, collections, …
- Permettre des mises à jour limitées via les méthodes

Caractéristiques

- Langage de requêtes fonctionnel
  - composition d'expressions
  - manipulation des types des langages de programmation objet
  - appel d'opérations avec mises à jour

- Fonctionnalités
  - select avec expressions de chemins
  - opérateurs ensemblistes : **`union, intersect, except`**
  - quantificateurs : **`for all, exists`**
  - agrégats : **`count, sum, min, max, avg`**
  - autres : **`order by, group by, define`**

Concepts nouveaux

- Expression de chemin mono-valuée
  - Séquence d'attributs ou associations mono-valués de la forme X1.X2…Xn telle que chaque Xi à l'exception du dernier contient une référence à un objet ou un littéral unique sur lequel le suivant s'applique.
  - Utilisable en place d'un attribut SQL
- Collection dépendante
  - Collection obtenue à partir d'un objet, parce qu'elle est imbriquée dans l'objet ou pointée par l'objet.
  - Utilisable dans le FROM

Forme des Requêtes

- Forme générale d'une requête

  Expressions fonctionnelles mixées avec un bloc select étendu

  Select [<type résultat>] (<expression> [, <expression>] ...)

  From x in <collection> [, y in <collection>]...

  Where <formule>

- Type résultat
  - automatiquement inféré par le SGBD
  - toute collection est possible (bag par défaut)
  - il est possible de créer des objets en résultat
- Syntaxe très libre, fort contrôle de type

Schéma

```
Class Personne {
   String nom;
   Date datenais;
   Relationship set <ref <Personne>> parents inverse
   enfants;
   Relationship list <ref<Personne>> enfants inverse
   parents;
   Relationship ref <Appartement> habite inverse est-
   habite-par;
   Int age();}
Class Employe : Personne { float salaire; }
Class Appartement {
   int numero;
   Relationship ref <Batiment> batiment inverse
   appartements;
   Relationship ref <Personne> est-habite-par inverse
   habite; }
Class Batiment {
   Adresse adresse;
   Relationship list <ref>Appartement>> appartements
   inverse batiment;
   }
```

Requêtes simples

Racines de persistance :

extensions des classes

```
set <Personne> lespersonnes;
set <Employe> lesemployes;
set <Appartement> lesappartements;
```

nommer un objet

martin est le nom d'un objet désignant l'employé qui s'appelle Martin.

Requêtes simples :

`5 + 2`	renvoie l'entier 7
`martin`	renvoie l'objet Employé de nom martin
`martin.enfants`	renvoie la liste des enfants de martin
`martin.age`	renvoie le résultat de la méthode age appliquée à martin

Sélection-Projection

Select … From… Where…

extraire d'une collection des éléments vérifiant une condition

```
Select e
From e in lesemployes
Where e.salaire > 2000

Select distinct e.salaire
From e in lesemployes
Where e.age < 30
```

Expression de chemin

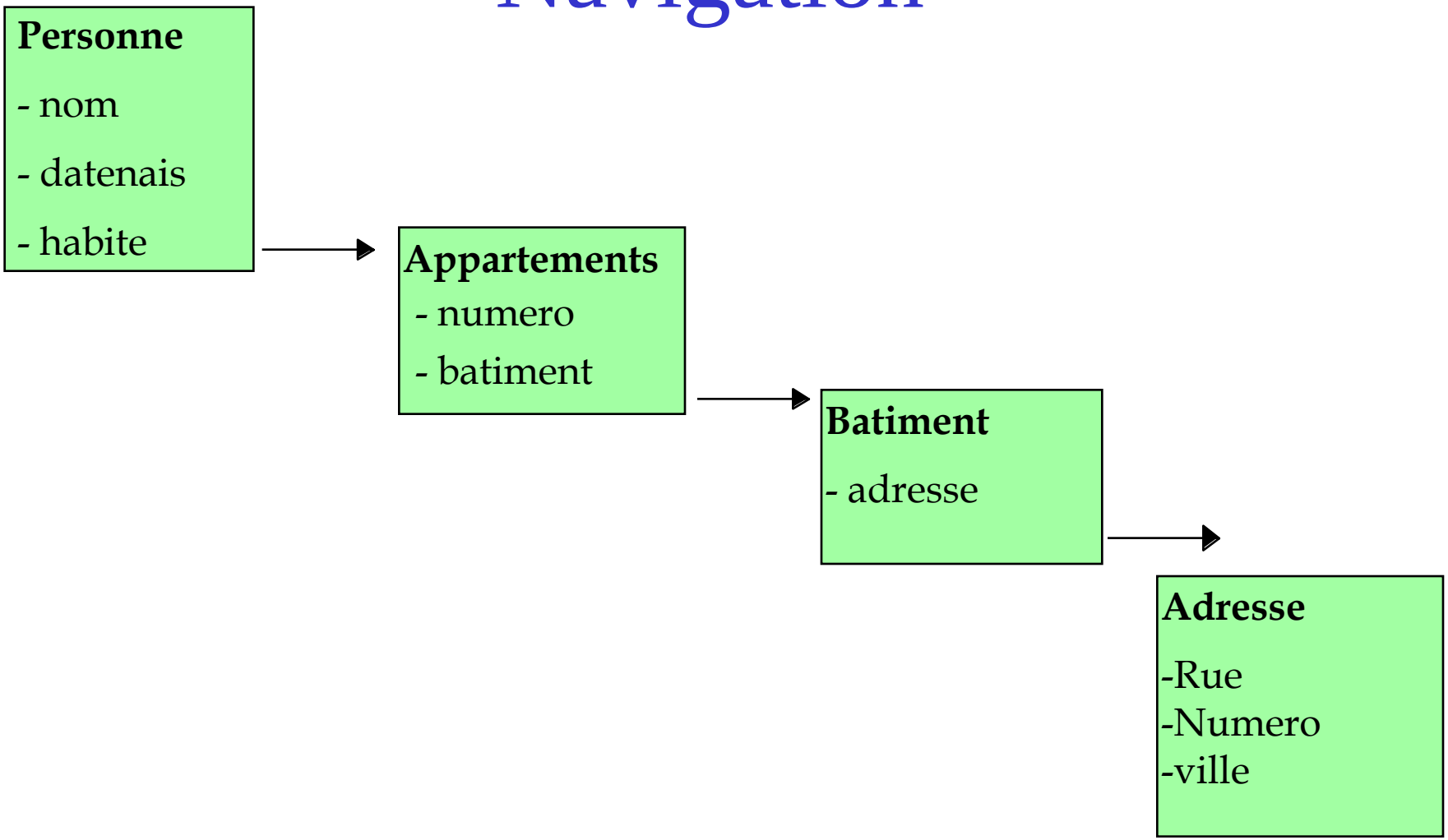
Un chemin permet de naviguer à travers les objets :

Ex. `p.habite.batiment.adresse.rue` (association 1-1)

Pour traverser des collections, on utilise une variable dans une expression select-from-where :

Ex.
```
select e.nom from p in lespersonnes, e in p.enfants
```

Navigation


Personne
- nom
- datenais
- habite
Appartements
- numero
- batiment
Batiment
- adresse
Adresse
-Rue
-Numero
-ville

Jointure

Relie plusieurs collections :

```
Select e.habite.batiment.adresse
From p in lespersonnes
       e in p.enfants
```

/* *adresse des enfants de chaque personne* */

```
Select p
from p in lespersonnes
        b in (select distinct a.batiment
              from a in lesappartements)
Where p.nom = b.adresse.rue
```

/**Personnes qui habitent dans une rue qui porte leur nom**/

Construction des résultats

La structure du résultat est implicite.

Filtrer un ensemble (une liste) renvoie un ensemble (une liste).

On peut aussi construire des résultats à l'aide des constructeurs **`struct, set, list, bag, array`**

```
Struct (nom : "Martin", salaire : 2000)
Select struct (personne : p.nom,
               adresse : p.habite.batiment.adresse)
From p in lespersonnes
```

Résultats imbriqués

Imbrication des **select** au niveau **select**

```
Select struct (
      moi : p.nom,
      monadresse : p.habite.batiment.adresse,
      mesenfants : (select struct (
                  nom : e.nom,
                  adresse :
  e.habite.batiment.adresse)
                  from e in p.enfants)
            )
      From p in lespersonnes
```

Appel de méthodes

En OQL, on peut appeler des méthodes avec ou sans paramètres, dans la clause select ou dans la clause where.

```
Select max(select e.age
                from e in p.enfants )
From p in lespersonnes
Where p.nom = “Paul “
```

Si la méthode a des paramètres, ils sont donnés entre parenthèses.

Création d'objets

Le résultat des requêtes est une valeur. On peut créer des objets en utilisant le nom de la classe. Certains attributs peuvent avoir une valeur par défaut.

```
Employe (nom :“Martin “ , salaire:2000)
Employe (select struct (nom:enf.nom,
  salaire:1500)
          from p in lespersonnes
                    enf in p.enfants
          where p.nom = “Martin “ )
```

Opérateurs

Agrégats : **count, min, max, sum, avg**

ex: `count(lesemployes)`

Opérateurs ensemblistes : **union, intersect, except**

Define : permet de nommer le résultat d'une requête

ex : `define <nom> as <requete>`

Like : comparaison de chaînes

ex : `like "Ma"` renvoie tous les noms commençant par Ma

Quantificateurs

Quantificateur universel : **for all**

```
For all x in collection : predicat(x)
```

Ex : `for all e in lesemployes : e.salaire > 2000`

Renvoie true si tous les employes ont un salaire supérieur à 2000, false sinon.

Quantificateur existentiel : **exists**

```
Exists x in collection : predicat(x)
```

Ex :

```
exists e in lesemployes : e.salaire > 5000 and
   e.age < 22
```

Opérateurs

Group by : regroupe les objets d'une collection selon les valeurs de certains attributs

```
Select e.age, avg(select e.salaire from partition)
from e in lesemployes
Group by e.age;
```

Order by : permet de trier

```
Select p from p in lespersonnes order by p.age, p.nom
```

Opérateurs de conversion

Element : extrait l'unique élément d'un ensemble

```
element (select e from e in lesemployes where e.nom =
   "Martin ")
```

Listtoset : transforme une liste en ensemble

```
Listtoset (list(1,2,3,2))
```

renvoie l'ensemble contenant 1, 2 et 3

Flatten : aplatit une collection imbriquée

```
Flatten (list (set 1,2,3), set(3,4,5,6), set(7)))
```

renvoie l'ensemble contenant 1,2,3,4,5,6,7

Applications des SGBD objet

- Domaines d'applications
  - Internet/Intranet
  - technique
    - gestion de réseau, CAD, CASE, CAM, etc.
  - systèmes bancaires et financiers
  - systèmes d'information géographiques
  - systèmes d'information médicaux
- Besoins
  - objets structurés et non structurés
  - associations complexes
  - sources de données hétérogènes