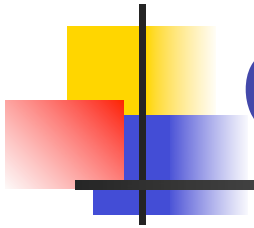
# Le modèle ODMG

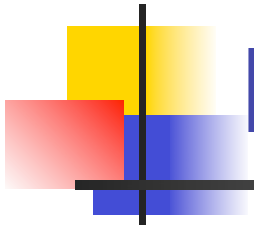
ODL, OQL, OML

Khalid Nafil
k.nafil@um5s.net.ma



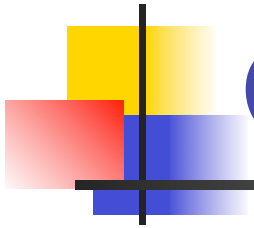
# CONTEXTE

- Object Database Management Group
  - Fondé en septembre 91 par 5 constructeurs:
    - O2 Technology
    - Objectivity
    - Object Design
    - Ontos
    - Versant
  - Version 2.0 96 avec: 10 auteurs
    - POET Soft, Barry & Ass., American Man. Syst., Windward Sol., Lucent
- Propose un "standard" pour les SGBDO



# Le standard de l'ODMG

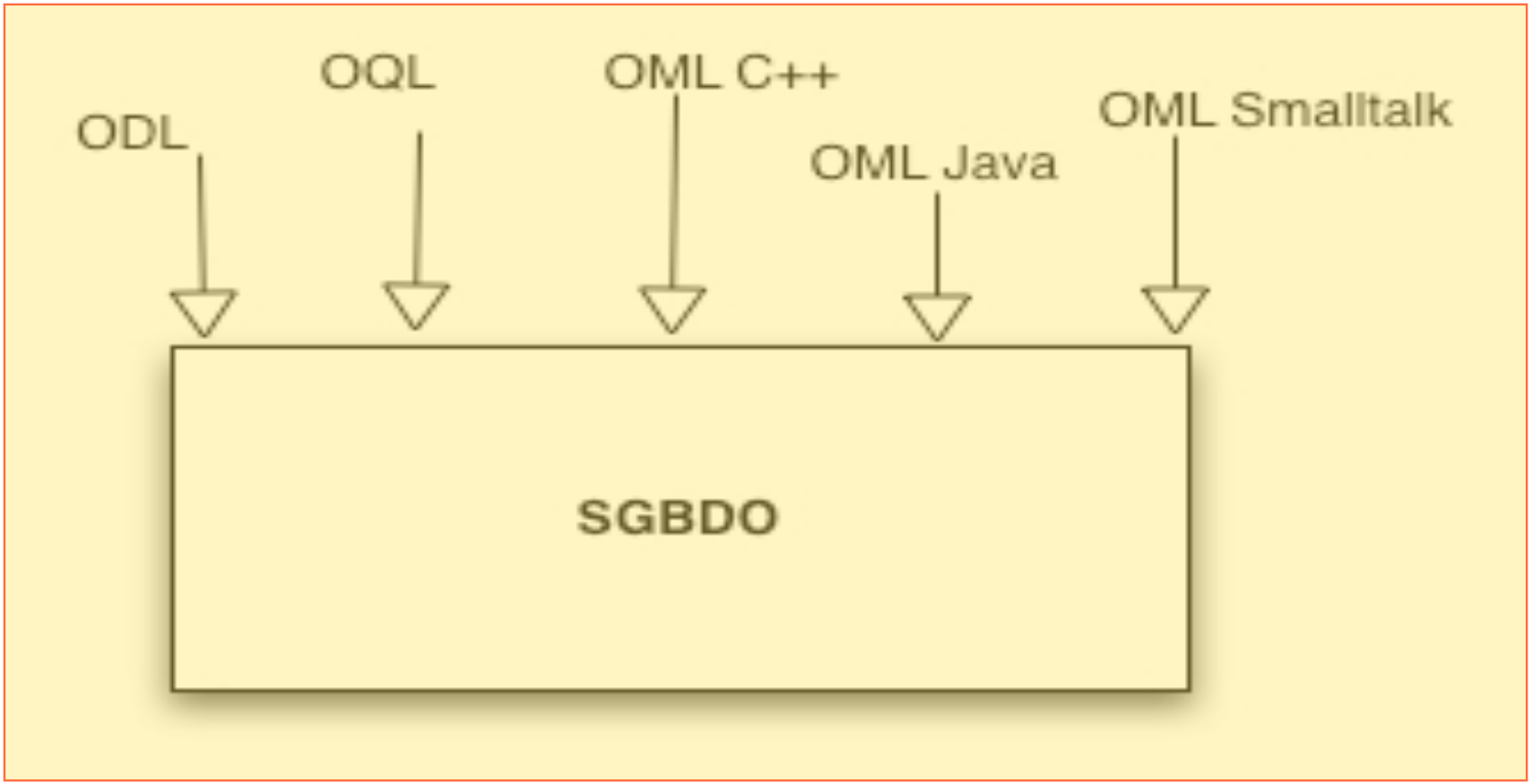
- L'ODMG (Object Database Management Group) vise à réaliser pour les BDOO l'équivalent de la norme SQL
- L'ODMG définit :
  - Le langage ODL
  - Le langage OQL
  - Le langage OML



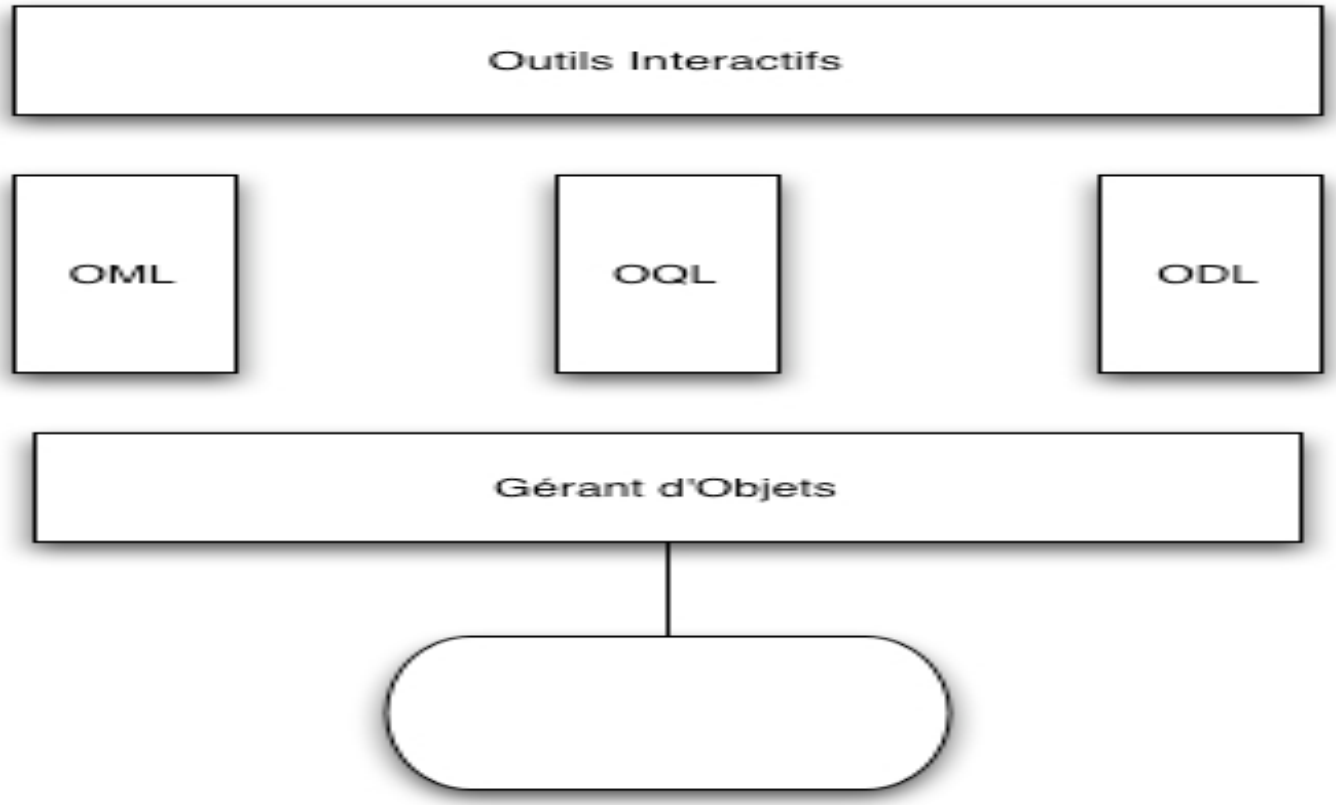
# Contenu de la proposition

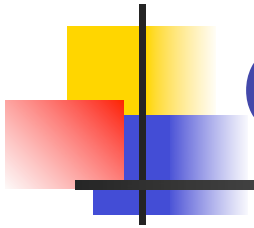
- Interfaces externes d'un SGBDO utilisés pour réaliser des applications
- Un langage de définition des interfaces des objets persistants : ODL
- Le langage OQL
- Intégration proposée avec C++, Smalltalk et Java
- Le langage OML

# Interfaces d'accès à un SGBD



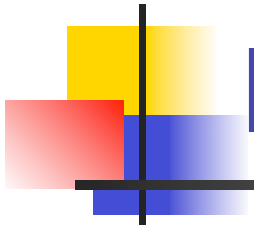
# Architecture type d'un SGBDO conforme àl'ODMG





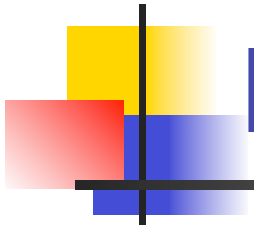
# Gérant d'objets

- Permet de gérer :
  - La persistance des objets
  - L'attribution des identifiants
  - Les méthodes d'accès
  - Les aspects transactionnels



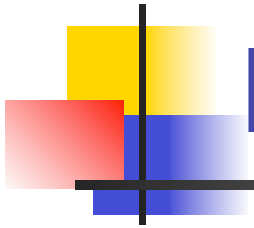
# Préprocesseur ODL

- Langage de définition de schéma des bases de données objet proposé par l'ODMG
- Permet de :
  - Compiler les définitions d'objets
  - Générer les données de la métabase



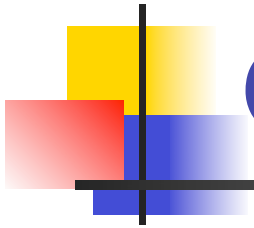
# Le composant OML

- Spécifique à chaque langage de programmation
- Permet de manipuler les objets conformes aux définitions depuis un langage de programmation (C++, Java, Smalltalk)



# Le composant OQL

- Langage d'interrogation de bases de données objets proposé par l'ODMG, basé sur des requêtes Select proches de celles de SQL
- Comporte un analyseur et un optimiseur du langage OQL capables de générer des plans d'exécution exécutables par le noyau



# Outils interactifs

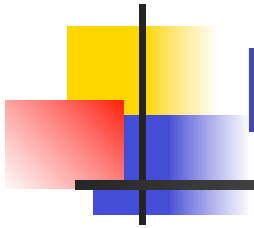
- Éditeur de classes
- Manipulateur d'objets
- Bibliothèques graphiques
- Débogueur, éditeur

# LE MODELE de l'ODMG

- Extension du modèle de l'OMG
  - l'OMG a proposé un modèle standard pour les objets permettant de définir les interfaces clients
  - le modèle est supporté par le langage IDL (déf. interface)
  - les BD objets nécessitent des adaptations/ extensions
    - instances de classes
    - collections
    - associations
    - persistance
    - transactions
- ODL se veut l'adaptation d'IDL aux BD

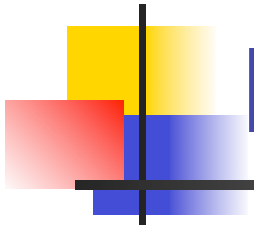
# Interface

- Spécification du comportement observable par les utilisateurs pour un type d'objets
- Voir exemple **calculateur**



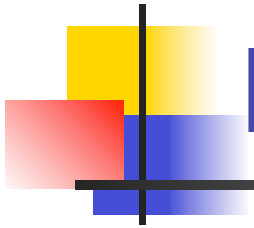
# Définition de classe

- Spécification du comportement et d'un état observables par les utilisateurs pour un type d'objets
- Une classe implémente ainsi une ou plusieurs interfaces
- Pour mémoriser les états abstraits de ses instances, une classe possède aussi une extension de classe



# Extension de classe

- Collection caractérisée par un nom contenant les objets crées dans la classe
- Class ordinateur (EXTENT ordinateurs key id) : calculateur {
- ….
- }

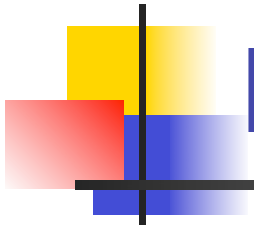


# Les classes : caractéristiques

- **Nom** : spécifié par le mot **class**
- **Nom collection** : spécifié par **extent**
- **Clés** : spécifiées par le mot **key**
- **Attributs** : spécifiés par **attribute**
- **Relations** : spécifiées par **relationship**
- **Opérations** : spécifiées en donnant la signature de l'opération

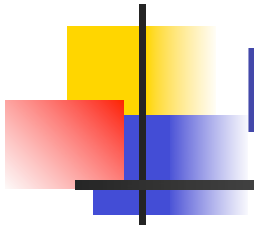
# Littéral

- Spécification d'un type de valeur correspondant à un état abstrait, sans comportement
- Correspondent aux types de base et aux structures



# Littéral : caractéristiques

- Un littéral n'a pas d'identificateur oid
- Ne peut être stocké directement de manière persistante
- Pour devenir persistant, il doit faire partie d'un objet
- On dénote trois types de littéraux

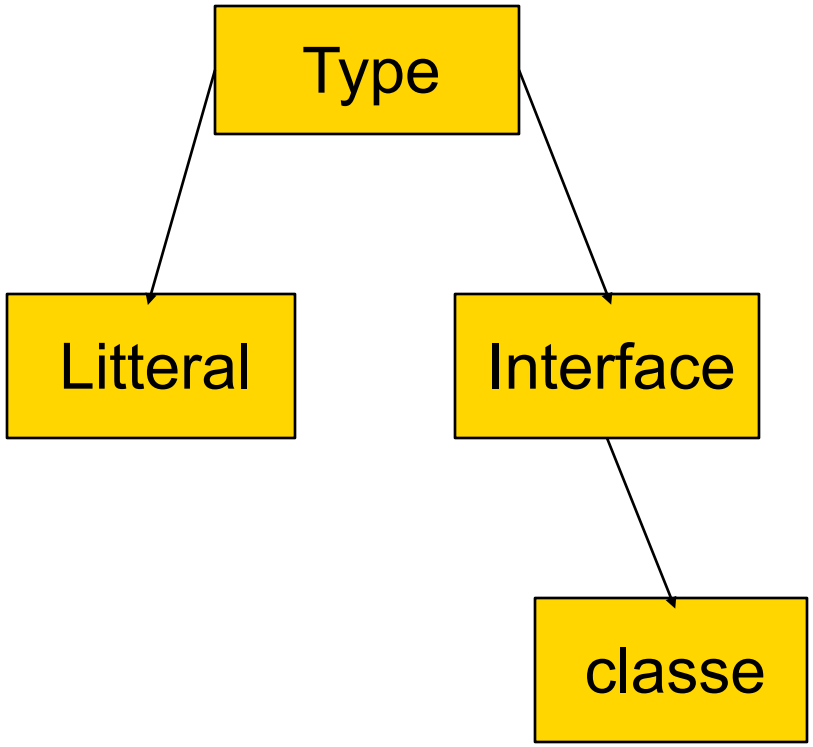


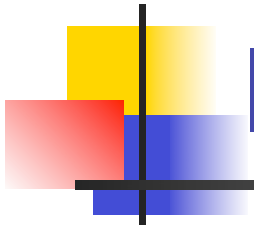
# Littéral : type

- **Atomic** : long, short, float, double, char, string, boolean, enum
- **Composé** : date, interval, time
- **Collection** : set, bag, list, array, dictionary (minuscules)

# Type, Interface et classe

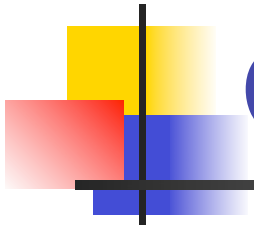
- Un modèle objet-valeur
- Valeurs = littéraux
  - entier, réel, string, ...
  - strcture<>, enum<>
- Les objets
  - implémentent des interfaces (comportement)
  - peuplent des extensions de classes (comportement + état)
- Deux type d'héritage
  - comportement  (interface vers interface)
  - d'état (classe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





# Interface : exemple

```
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();
}
```



# Classe : exemple

Class ordinateur (extent ordinateurs key id) : calculateur {

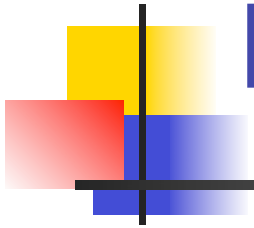
Attribute short id ;

Attribute float accumulateur ;

Void start() ;

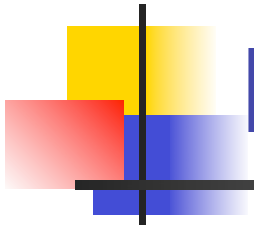
Void stop() ;

}



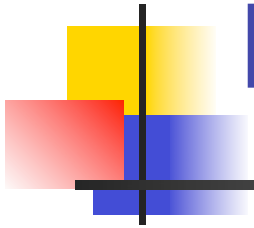
# Les Objets = Instances

- Identifiés par des OIDs :
  - gérés par le SGBD OO pour distinguer les objets
  - permet de retrouver les objets, reste invariant
  - Peuvent être nommés par les utilisateurs
- Persistants ou transiants :
  - les objets persistants sont les objets BD, les autres restant en mémoire (transiants)
- Peuvent être simples ou composés :
  - atomiques, collections, structurés



# Les Objets : caractéristiques

- **Identificateur** : valeur interne au Sgbd
- **Nom** : nom persistant. Typiquement, on associe un nom seulement aux collections d'objets
- **Durée de vie** : persistant ou transcient
- **Structure** : atomique ou collection d'objets. Un objet atomique est défini par les clauses interface et class



# Propriétés communes

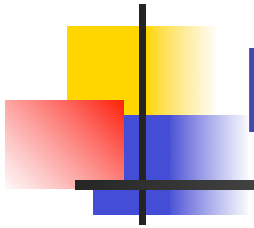
- un ensemble d'opération héritées pour:
  - création, verrouillage, comparaison, copie, suppression
- création des objets par des "usines"
  - interface ObjectFactory { Object new(); };
- héritage d'un type racine :
  - interface Object {
  - void lock(in Lock_Type mode) raises (LockNotGranted);
  - 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 à une interface spécifique
- 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 }



# Exemple de collections

- Class Personne

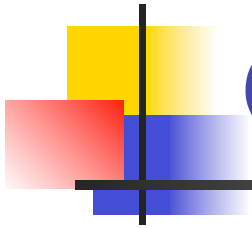
  {attribute string name;

  attribute set <string> emails;

  attribute list <T-address> addresses;

  attribute bag <string> computers;

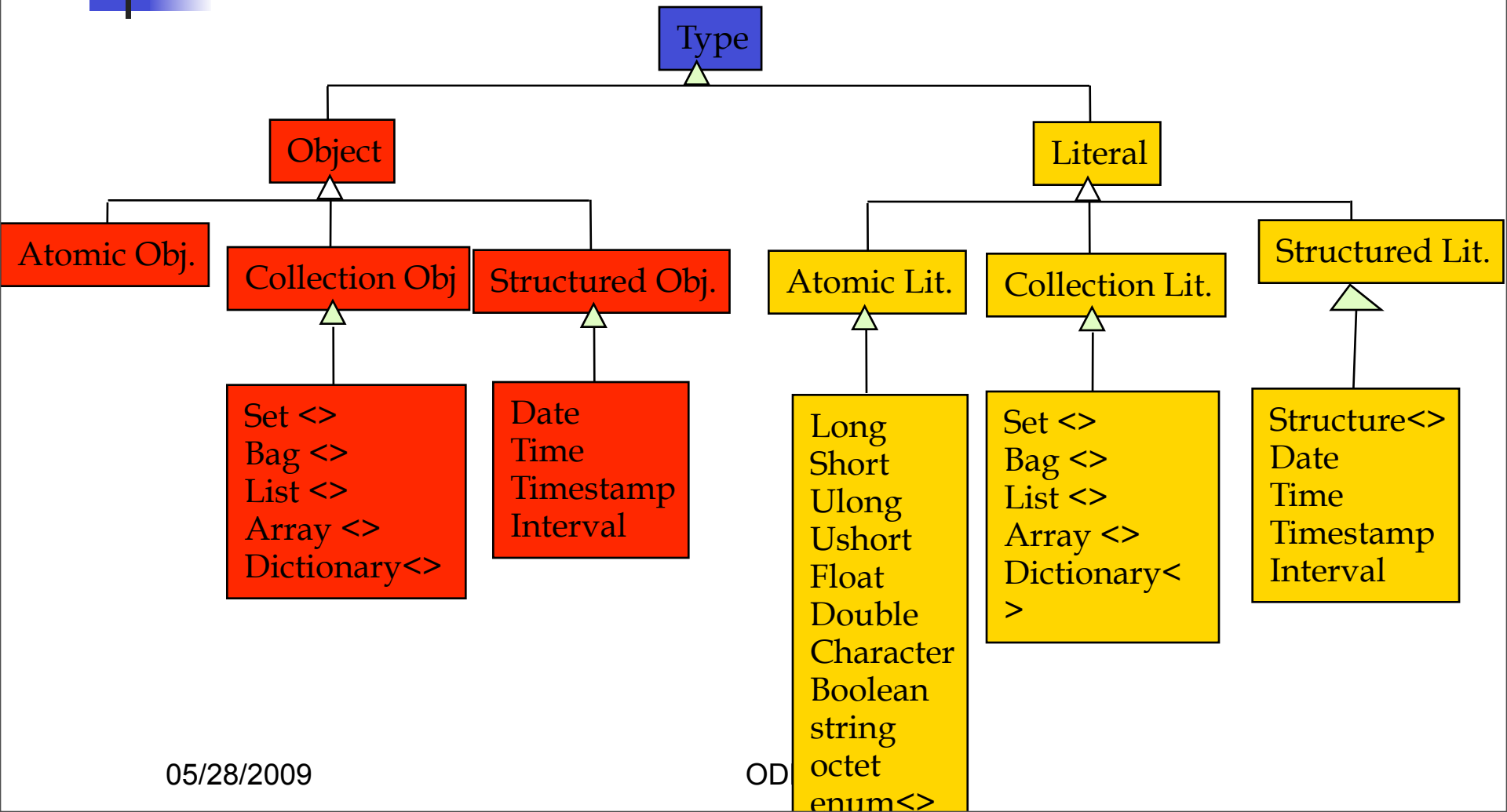
  }



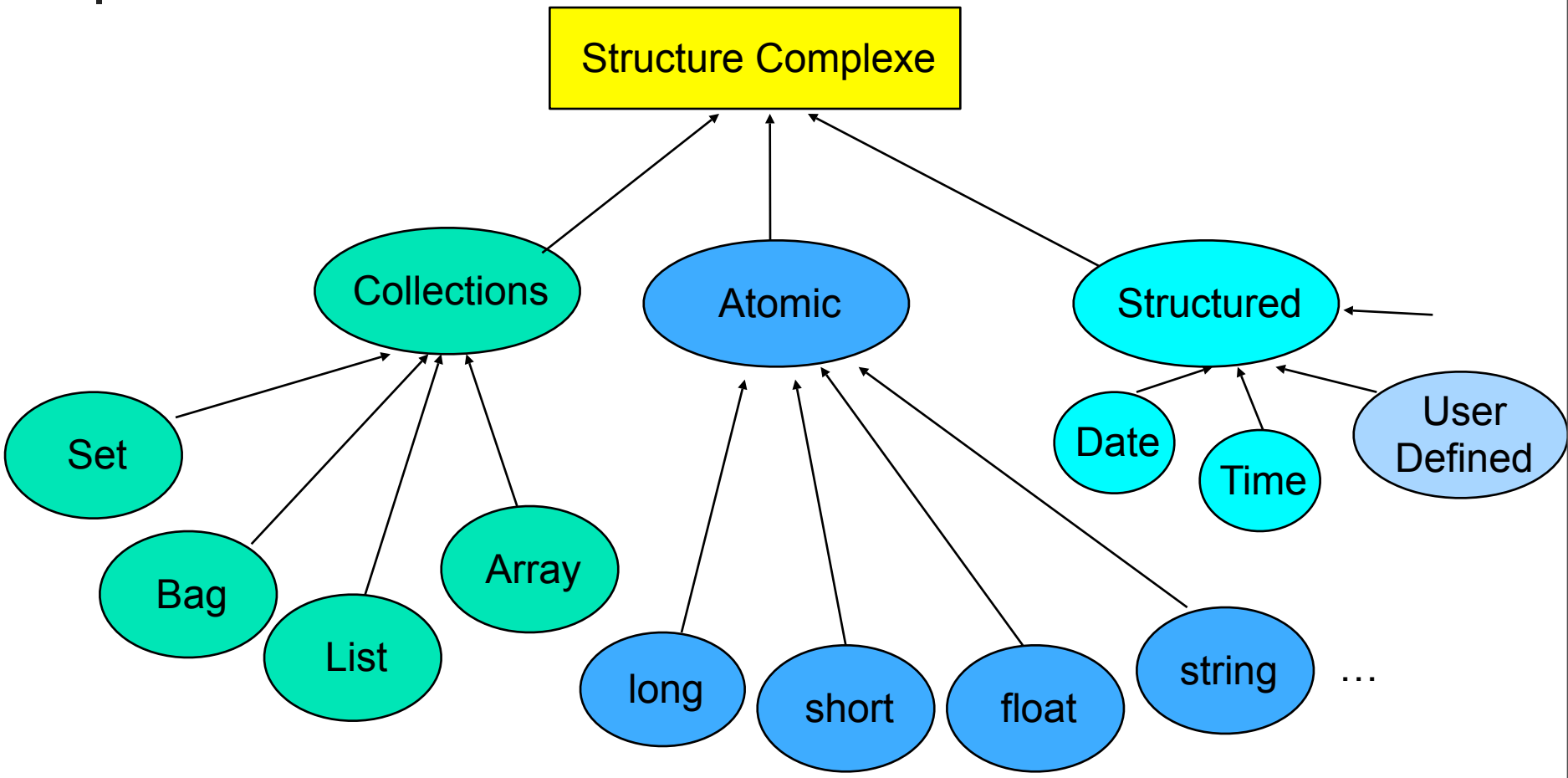
# Objets structurés

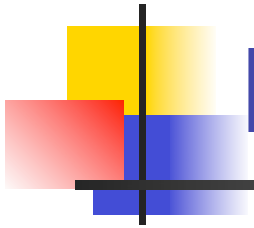
- Des objets structurés permettent de gérer le temps :
  - DATE (mois, jour, an)
  - INTERVAL (jour, heure, minute, seconde)
  - TIME (heures avec zones de temps en « ms  »)
  - TIMESTAMP (encapsule date et temps)
- Il sont la version objet des litéraux correspondants
  - fournissent en plus des opérations :
    - ajout d'intervalles
    - extraction de mois, jour, an
    - ...

# Hiérarchie de Types



# Structures complexes

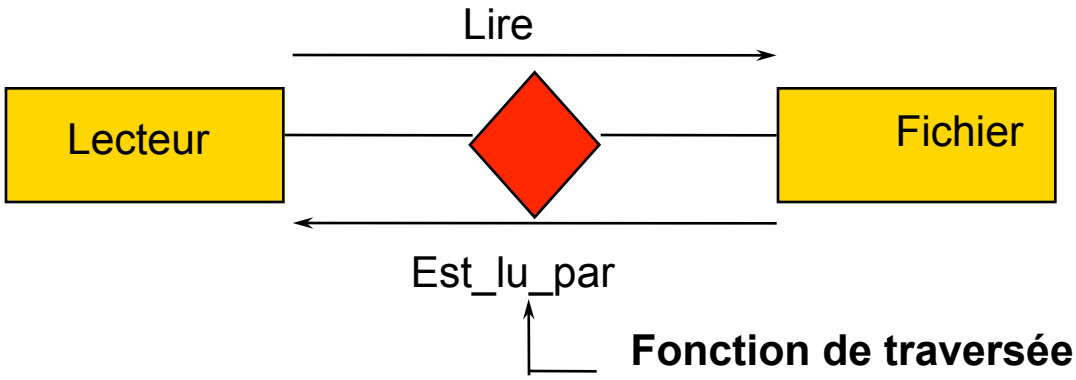




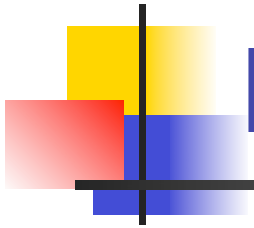
# Les Attributs

- Modélisent les états abstraits des objets
- Une propriété permettant de mémoriser un litéral ou un objet
- Peut être vu comme deux fonctions :
  - Set_value
  - Get_value
- Propriétés:
  - son nom
  - type de ses valeurs légales
- Un attribut n'est pas forcément  implémenté.

# Les Associations



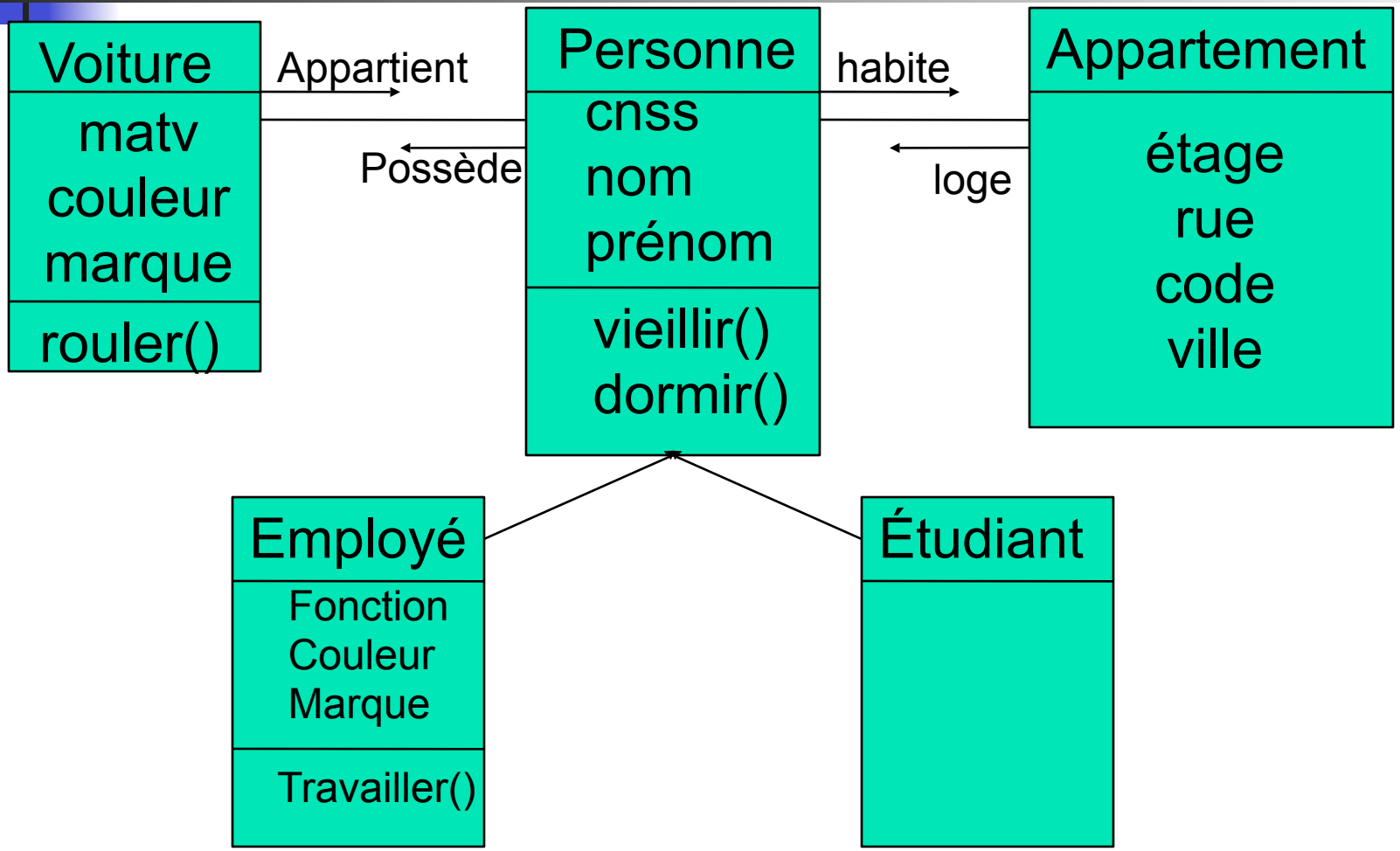

- Associations binaires, bi-directionnelles de cardinalité (1:1), (1:N), (N:M).
- Opérations:
  - Add_member, Remove_member
  - Traverse, Create_iterator_for

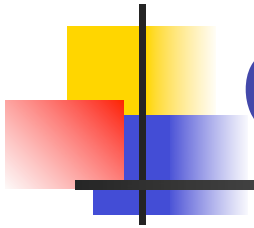


# Les Opérations

- 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)

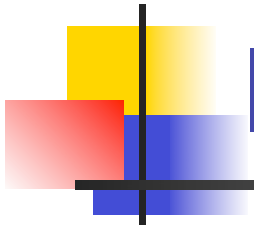
# Exemple de schéma ODL





# Classe voiture

```
class voiture(extent voitures key matv)
   { attribute string matv;
....
relationship personne appartient inverse
   personne::possede;
short rouler (in short distance) ;};
```

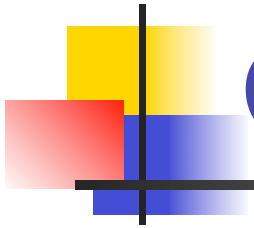


# Interface personne

```
interface personne { attribute string cnss;
    ….;
relationship appartement habite inverse
    appartement::loge;
relationship voiture possede inverse
    voiture::appartient;
short vieillir();
….; } ;
```

# Classe employe

```
class employe:personne(extent employes
    key cnss){ attribute enum fonct
    { ingenieur, scretaire, analyste,
    programmeur} fonction;
attribute float salaire;
attribute list<float> primes;
void travailler(); };
```

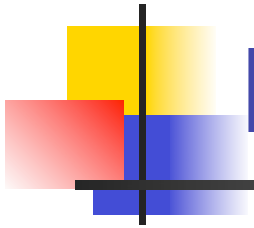


# Class appartement

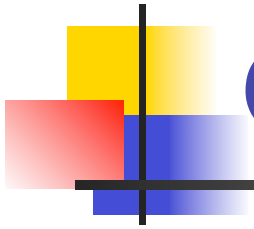
class appartement (extent apparts) {

attribute struct adresse (short etage,
unsigned short numero, string rue,
unsigned short code, string ville);

relationship set <personne> loge inverse
personne::habite; } ;

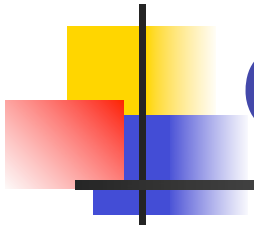


# Le langage OQL



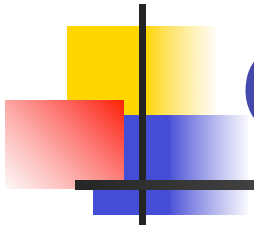
# OQL

- Défini à partir d'une première proposition issue du système O2 développé à l'INRIA



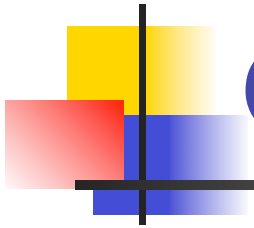
# Objectifs d'OQL

- Offrir un accès non procédural pour permettre des optimisations automatiques
- Garder une syntaxe proche de SQL
- Rester conforme au modèle de l'ODMG, en permettant l'interrogation de ttes les collections d'objets, le parcours d'associations, l'invocation d'opérations,…
- Permettre de créer des résultats littéraux, objets, collections,…
- Supporter des m-à-j via opérations sur ces objets



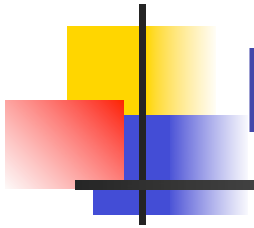
# OQL : principes

- Le langage est construit de manière fonctionnelle
- Une question est une expression fonctionnelle qui s'applique sur un littéral, un objet ou une collection d'objets
- Le langage est fortement typé



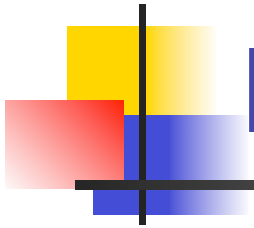
# OQL : Concepts nouveaux

- 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 litéral unique sur lequel le suivant s'applique.
  - Utilisable en place d'un attribut SQL
- Collection dépendante
  - Collection obtenue à partir d'un objet, soit parcequ'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
  - 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 résultats
- Syntaxe très libre, fort contrôle de type



# Exemple de requêtes

- Calcul d'une expression
  - ((string) 10*5/2) || "toto"
  - ===> string
- Accéder un attribut d'un objet nommé
  - mavoiture.couleur
  - ===> litteral string

# Parcours d'association monovaluées

- avec sélection de structure (défaut)
  - Select struct (name : e.nom, cite : e.habite.adresse.ville)

    from e in Employes

    where e.grade = 'ingenieur'
  - ===> littéral bag <struct(Name,Cite)>

# Parcours d'association multivaluées

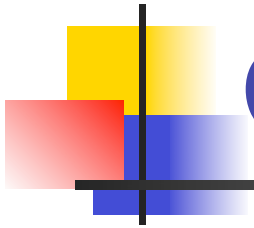
- Utilisation de collections dépendantes
  - Select e.nom, e.prenom
    from e in employes, k in e.adresse
    where k.ville = "kenitra"
  - ==> litteral
    bag<struct<nom:string,prenom:string>

# Résultat imbriqué

- Imbrication des select au niveau select
  - select distinct struct (nom : e.nom, inf_mieux_payes :
  - select i
  - from i in e.inferieur
  - where i.salaire > e.salaire))
  - from e in employes
  - ===> litteral de type set <struct (nom: string,
  - inf_mieux_payes : bag<employes>)>
- et aussi au niveau du FROM (requête collection)

# Invocation de méthodes

- En résultat ou dans le critère
  - select distinct e.nom,
  - e.habite.adresse.ville, e.age()
  - from e in employes
  - where e.salaire > 10000 and e.age() < 30
  - ===> litteral de type set <struct>

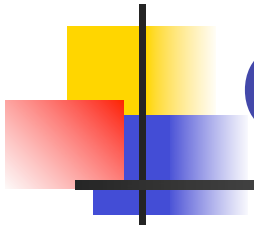


# Création d'objets

- Expressions de constructions
  - Si C est le nom d'une classe, p1, ..., pn des propriétés de la classe et e1, ..., en des expressions
  - alors C(p1 : e1, ..., pn : en) est une expression de construction
- Création d'objet
  - employe (cnss:460869, nom:"jamil", salaire: 10000)
  - employe (select struct(cnss:c.cnss, nom:c.nom, salaire: 4000)
  - from c in chauffeurs
  - where not exist e in employes : e.cnss=c.cnss )

# Définition d'objets via requêtes

- Définition de macros
  - Define <nom>  as <question>
  - permet de définir un objet de nom <nom> calculé par la question
- Exemple:
  - Define Ingenieurs as
  - Select e
  - From e in Employes
  - Where e.grade = "ingenieur"

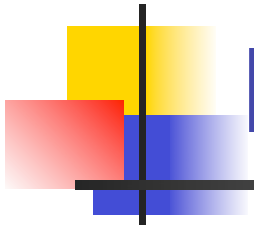


# Quantification

- Quantificateur universel
  - for all x in collection : predicat(x)
  - Exemple:
  - for all e in Employes : e.age < 18
- Quantificateur existentiel
  - exists x in collection: predicat(x)
  - Exemple:
  - exists v in Employés.posséde : v.marque = "Renault"

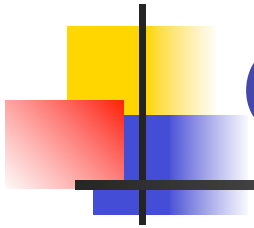
# Calcul d'agrégats

- Similaire à SQL
- avec possibilité de prédicats
  - select e
  - from e in employes
  - group by (bas : e.salaire < 7000,
  - moyen : e.salaire >= 7000 and e.salaire < 21000,
  - haut : e.salaire >= 21000)
  - ===>struct<bas: set(emp.),moyen:set(emp.),haut:set (emp.)>



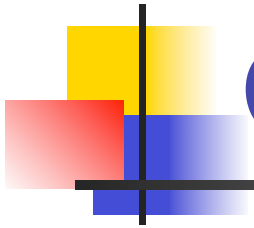
# Expressions de collections

- Conversion de collections
  - element (select v.marque
  - from v in voitures
  - where v.numero = "2A20275") ===> string



# Gestion des transactions

- Objet Transaction créé par Factory
  - begin() pour ouvrir une transaction ;
  - commit() pour valider les mises à jour de la transaction ;
  - abort() pour défaire les mises à jour de la transaction ;
  - Possibilités d'imbriquer des transactions
  - Contrôle de concurrence niveau objet (explicite ou défaut);



# Conclusion

- Tentative de création d'un standard pour SGBDO
  - vise à la portabilité des applications
  - langage de requêtes très complet (et complexe)
  - modèle abstrait et mapping C++, Java
- Extension de SQL pour collections imbriquées :
  - des différences avec SQL2 (sémantique, typage fort, …)
  - des déficiences (contrôle, vues,…)
  - des spécificités (nomination des objets, des requêtes, ..)
  - difficile à implémenter ...