

M1 Informatique -Sorbonne Université

MU4IN801 - MLBDA

Modèles et Langages Bases de Données Avancées

Bernd Amann

**COURS 2 – MODÈLES OBJET ET RELATIONNEL-
OBJET**

Evolution de SQL

Algèbre relationnelle (Codd, 1970)

SEQUEL (Chamberlin, Boyce, 1974)

SQL-86 (ANSI), SQL-89 (contraintes), SQL-92

SQL-99 : récursion, triggers, **types structurés, objets** ⇒ **objet-relationnel**

SQL:2003 : XML, fenêtrage

SQL:2006 : plus de XML (import/export, **XQuery**)

SQL:2008, SQL:2011, SQL:2016, ...

[Codd, E.F.](#) (June 1970). "A Relational Model of Data for Large Shared Data Banks".
[Communications of the ACM](#). **13** (6): 377–387.

Andrew Eisenberg and Jim Melton and Krishna G. Kulkarni and Jan-Eike Michels and Fred Zemke, SQL: 2003 has been published, SIGMOD Record, 33 (1) 2004, pages 119-126

Données complexes (schéma E/A)

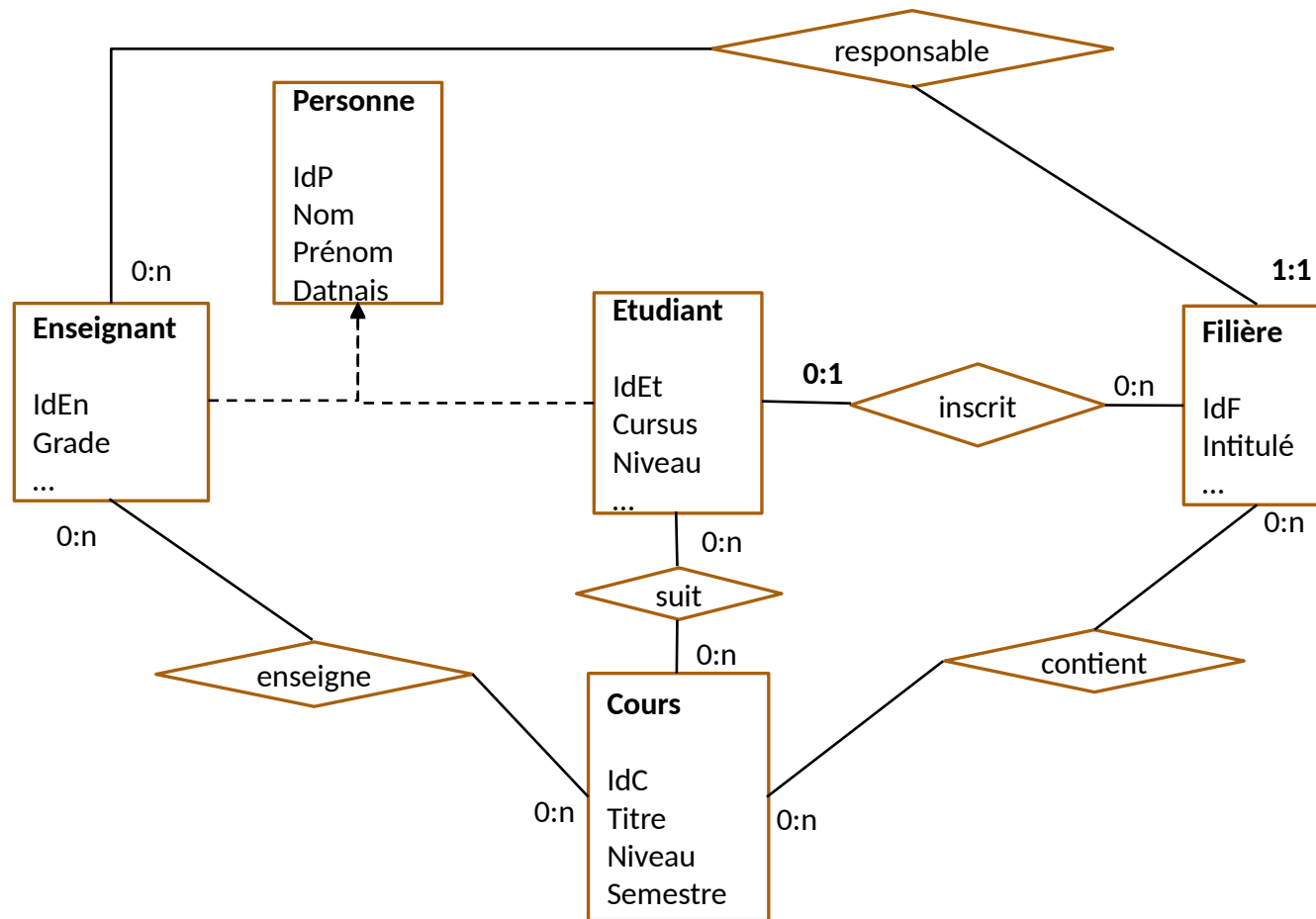


Schéma relationnel (SQL-89)

PERSONNE(IdP, Nom, Prenom, Datnais)

ENSEIGNANT (IdP*, Grade)

ETUDIANT (IdP*, Coursus, Niveau, IdF*)

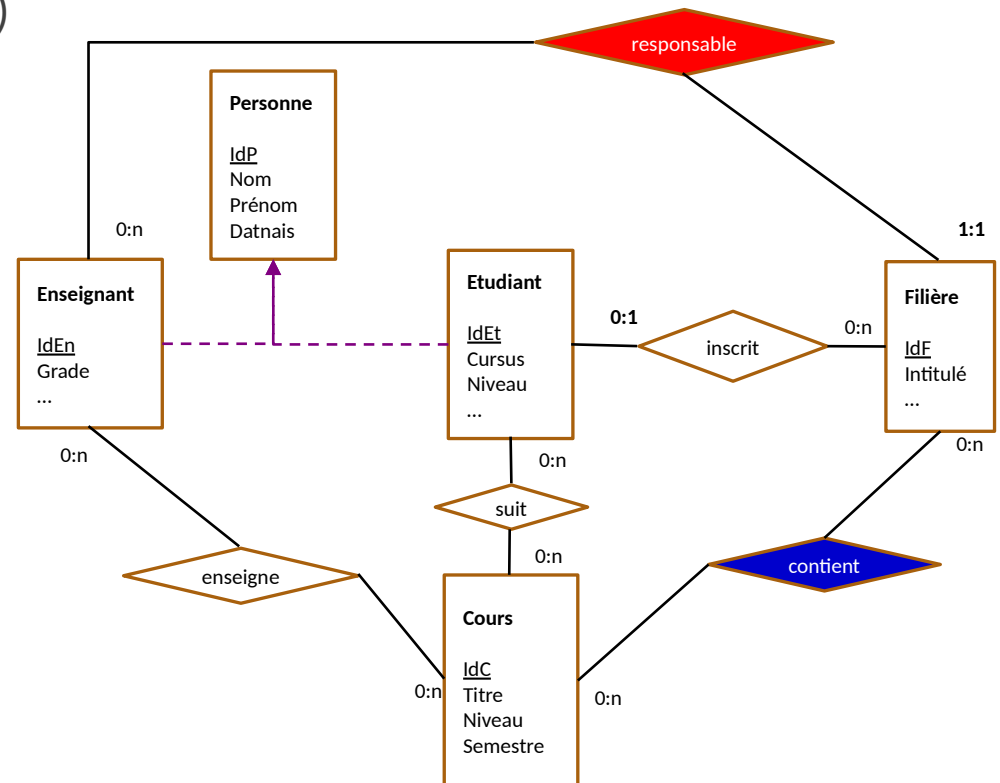
COURS (IdC, Titre, Niveau, Semestre)

FILIERE (IdF, Intitule, **IdR***)

ENSEIGNE (IdEns*, IdC*)

SUIT (IdEt*, IdC*)

CONTENU (IdF*, IdC*)



Modèles Objet (rappel)

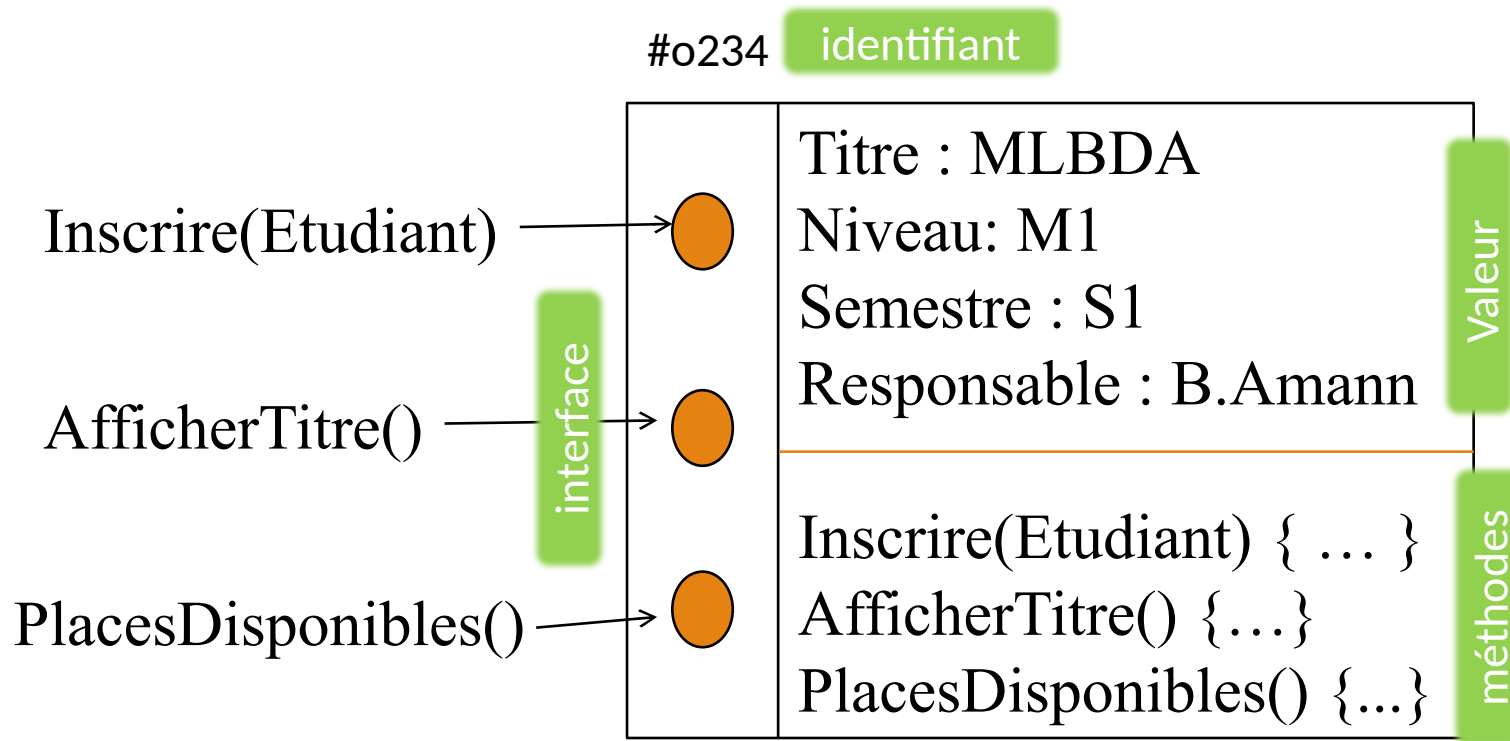
Principes

- Une entité est représentée par un **objet identifié**
- Les objets sont reliés entre eux par des **liens de composition** et des **liens d'héritage**
- Les **opérations** sont associées à l'objet

Concepts proche du modèle conceptuel E/A:

- Valeur et objet
- Identité d'objet
- Classe
- Héritage

Objet : exemple



Valeurs et types

Valeurs atomique :

- type simple : caractère, entier, booléen, ...

Valeurs complexes :

- Valeurs construits à partir de *valeurs* et d'*identifiants d'objets* (atomiques ou complexes)

Constructeurs de types / valeurs complexes :

- **Tuple** (n-uplet) : liste d'attributs de types diverses identifié par position ou nom

Collections d'éléments *d'un même type* :

- Ensemble / Multi-ensemble : *non-ordonnée* sans / avec doublons
- Listes : ordonnée de taille variable
- Array : ordonné de taille fixe

Exemple :

- ['Durand', 'Paul', '25-10-98', 'Informatique', {'MLBDA', 'LRC'}]

Objets

Un **objet** est un couple (identifiant, *valeur*) :

- **l'identifiant** est indépendante de la valeur et géré par le système
- **objet atomique** : objet avec une *valeur atomique*.
- **objet complexe** : objet avec une *valeur complexe*.

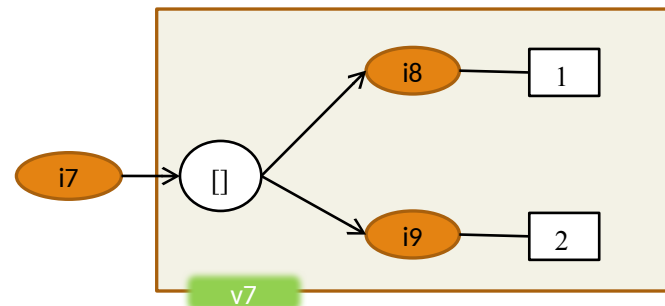
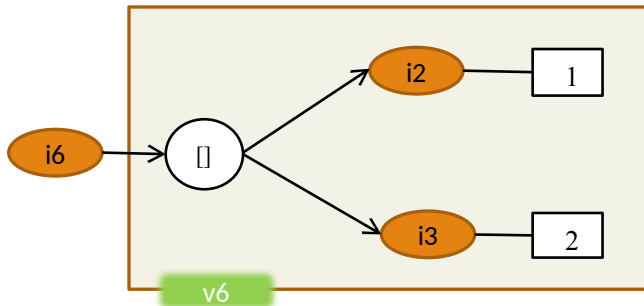
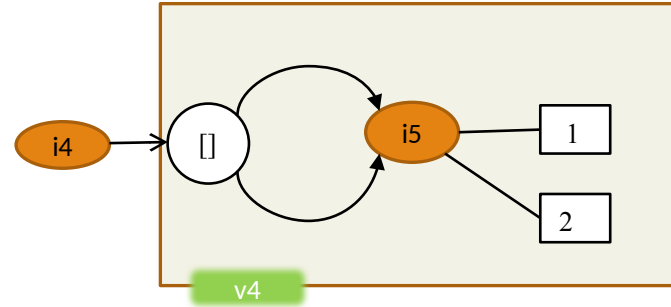
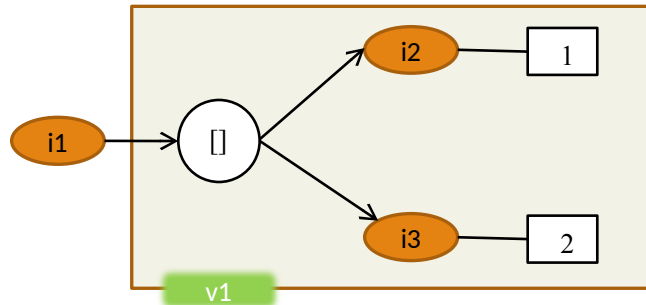
Exemples

- (**#i1**, ['MLBDA', 'M1', 'S1', i3])
- (**#i2**, ['Durand', 'Paul', '25-10-98', 'Informatique', {#i1, #i4}])

Deux objets (#i1,v1) et (#i2,v2) sont

- **identiques** s'ils ont le même identifiant : #i1=#i2
- **égaux** s'ils contiennent les mêmes valeurs et objets.
- **égaux en profondeur** (deep equal) s'ils contiennent les mêmes valeurs et des objets qui sont égaux en profondeur.

Comparaison d'objets

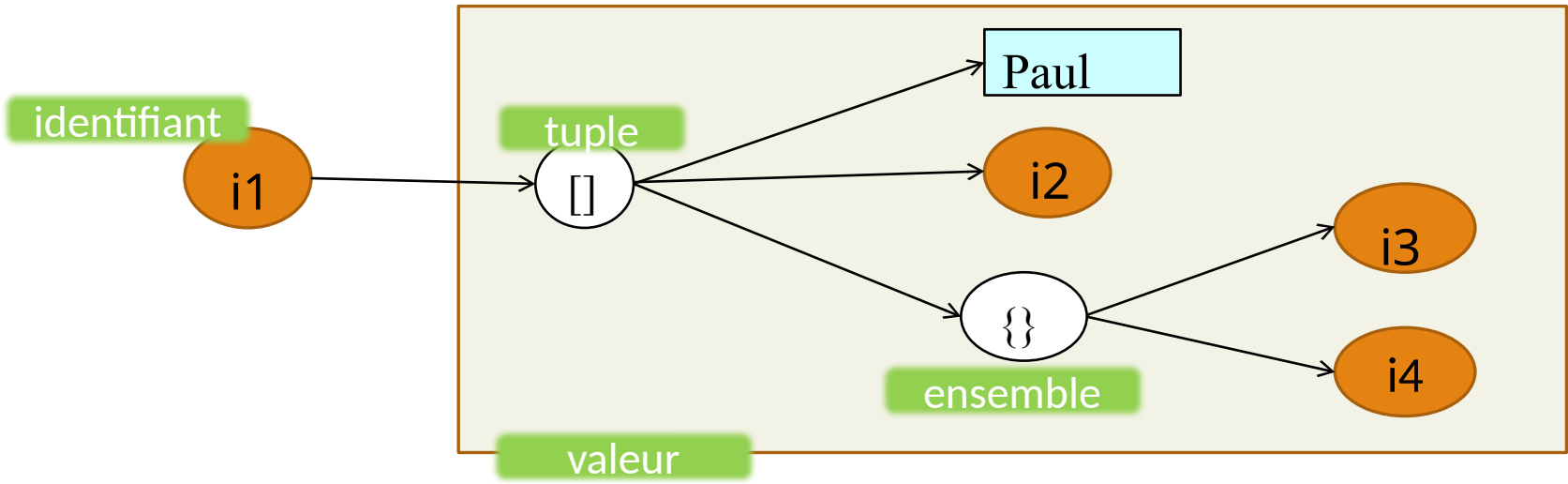


Identique : I
 Égal : E
 Égal en profondeur : P

$I \Rightarrow E \Rightarrow P$

objet	i1	i4	i6	i7
i1	I	-	E	P
i4	-	I	-	-
i6	E	-	I	P
i7	P		P	I

Graphe d'objets

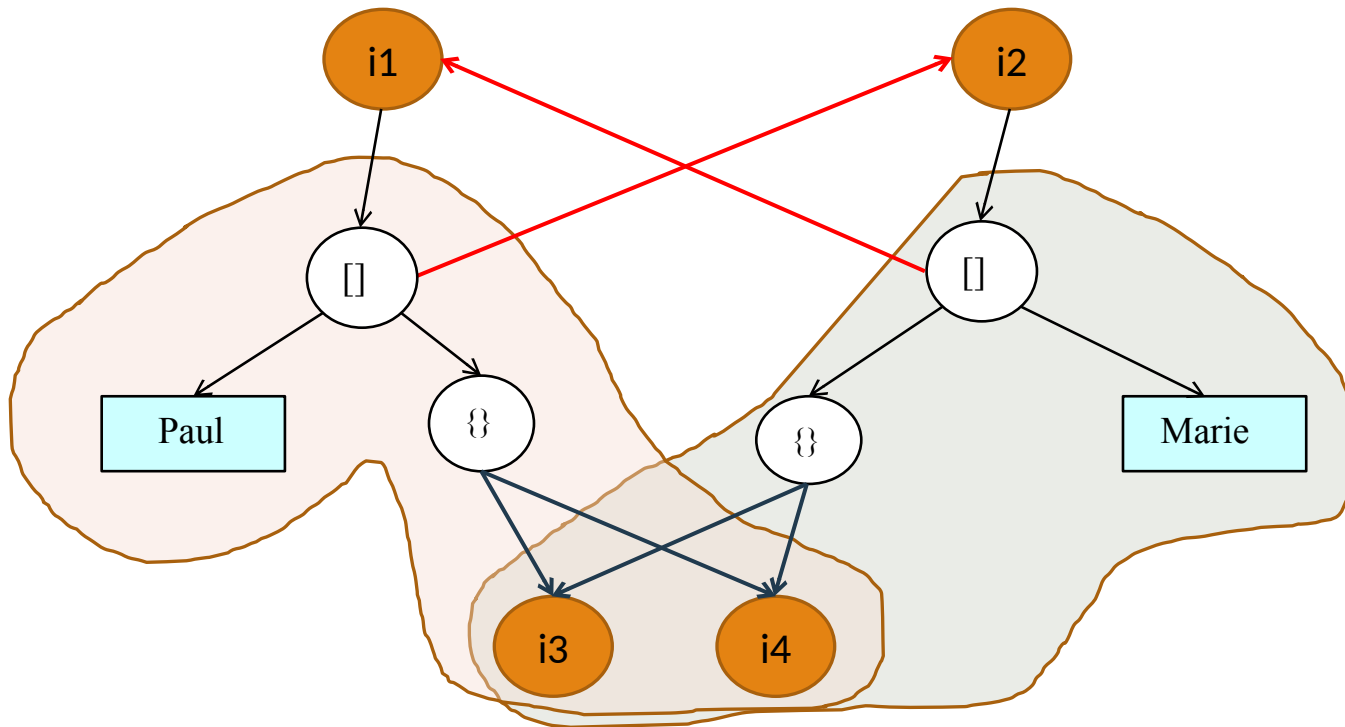


L'identité d'objet permet

- le **partage** d'objets,
- les **cycles** entre objets,
- le maintien automatique des **contraintes référentielles**

Cycle et partage d'objets

Les objets peuvent être représentés par un graphe de composition, qui permet de partager des objets et peut comporter des cycles



Classes

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.

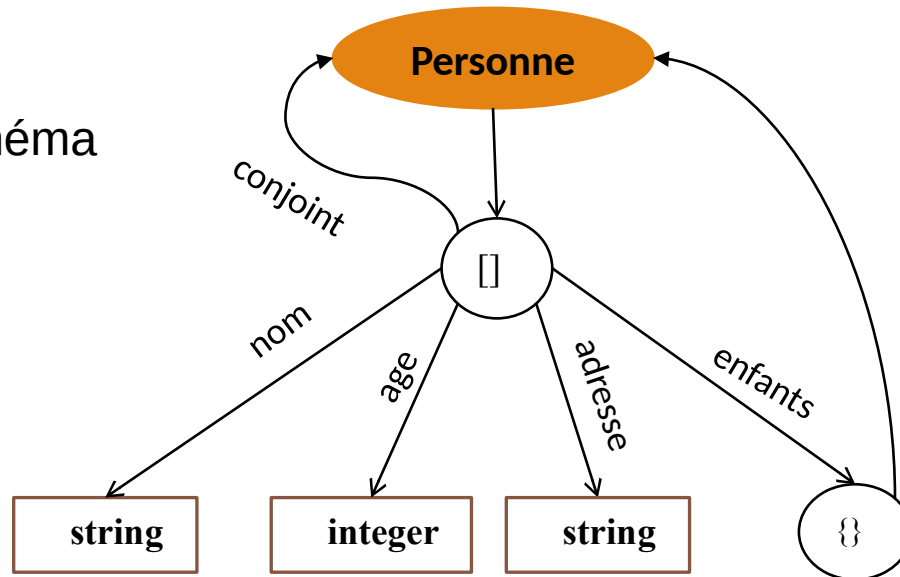
Exemple

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

Classes

```
class Personne [nom : string, age : integer, adresse : string,  
                conjoint : Personne, enfants : {Personne} ] ...
```

Schéma

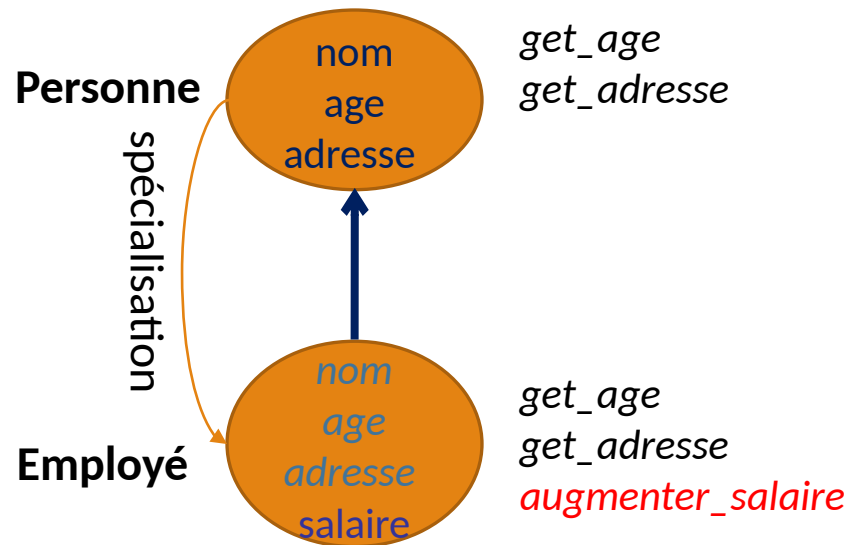


Héritage et spécialisation

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

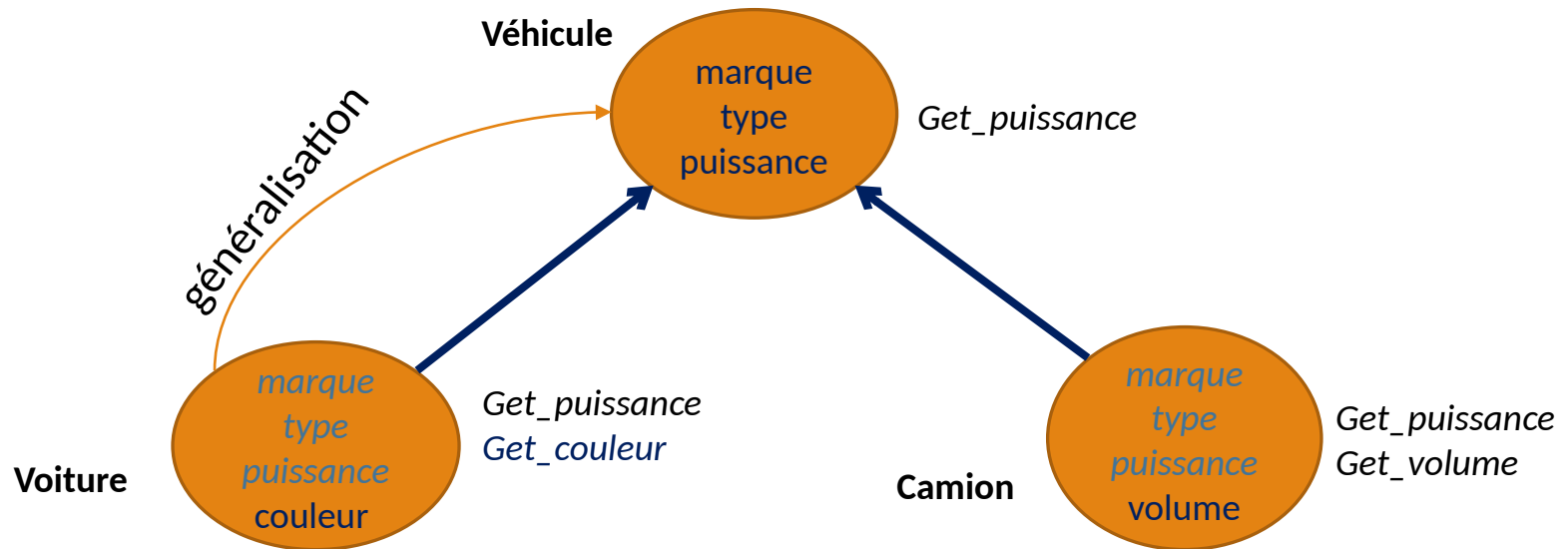
Exemple : spécialiser la classe Personne en la sous-classe Employé



Héritage et généralisation

Généralisation : création d'une **super-classe** regroupant les caractéristiques communes à plusieurs classes.

Exemple : Généraliser les classes Voiture et Camion en une classe Véhicule



Encapsulation

Un objet est constitué

- de **données**
- d'opérations applicables à ces données (**méthodes**)

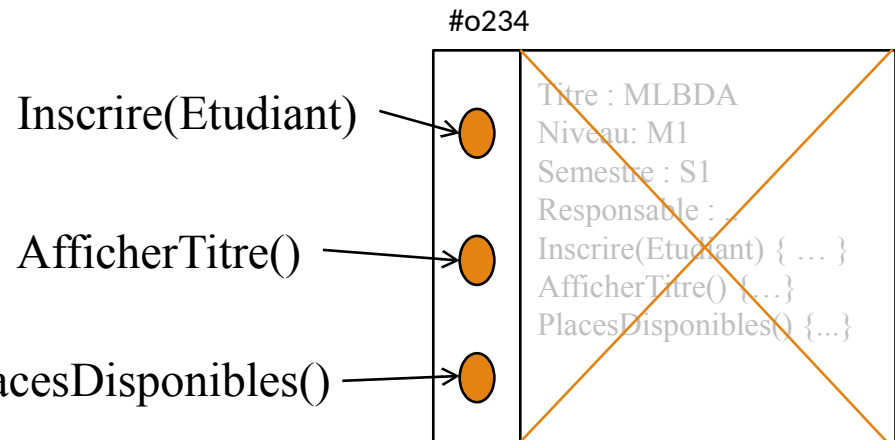
Interface

- description des opérations applicables à l'objet

Implémentation

- structure des données et code des opérations)

Le développeur ne voit que l'interface !



L'apport des modèles objets

Identité d'objets

- introduction de pointeurs invariants
- possibilité de chaînage

Héritage d'opérations et de structures

- facilite la réutilisation des types de données
- permet l'adaptation à son application

Encapsulation des données

- possibilité d'isoler les données par des opérations
- facilite l'évolution des structures de données

Possibilité d'opérations abstraites (polymorphisme)

- simplifie la vie du développeur

Définitions : Base de Données Objet (BDO)

BD Objet :

- Les données sont des objets (avec identité et méthodes)
- Le schéma est un graphe de classes

SGBD Objet :

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

+ « Objet » :

- langages objet (C++, 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

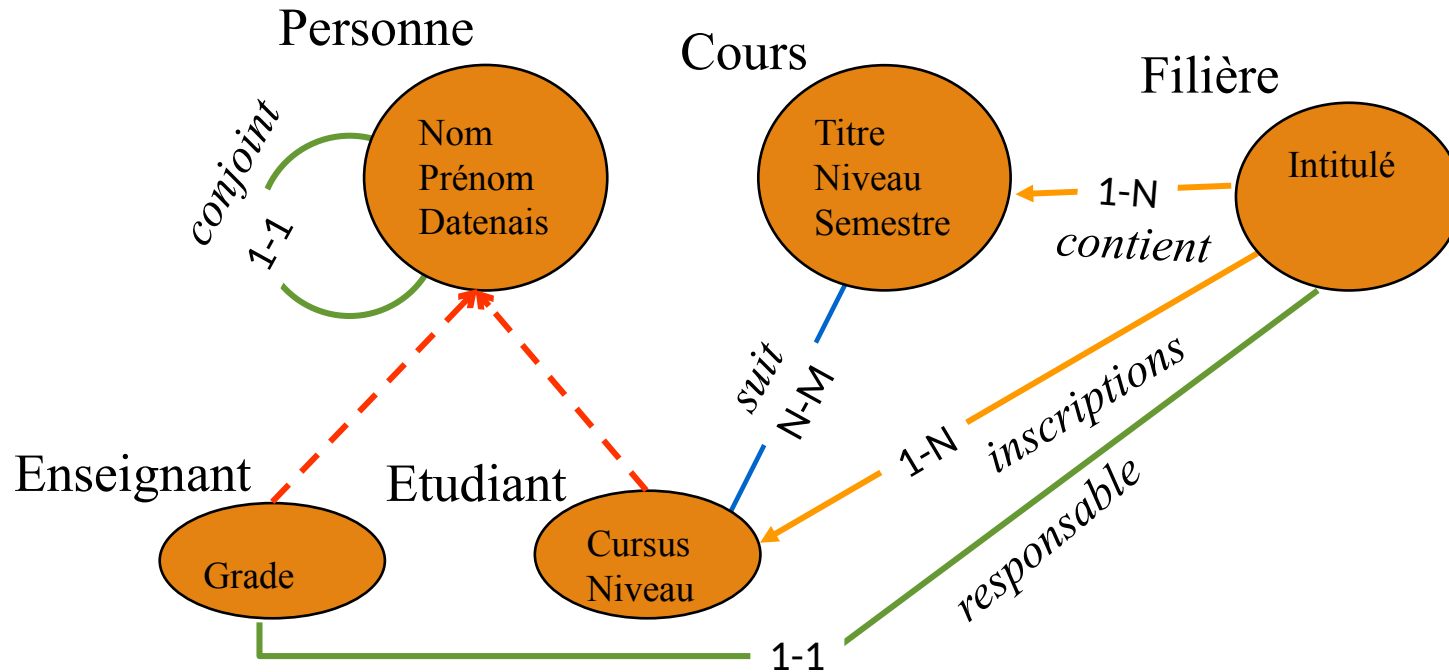
Persistance d'objets

Un **objet persistant** est un objet stocké dans la base, dont **la durée de vie** est indépendante du programme qui le crée.

Un objet est rendu **persistant**

- par une commande explicite (write) ou
- par un mode automatique :
 - attachement à une **racine de persistance** : *ensemble nommé de valeurs d'un type*
 - attachement à un **type de données** : *table relationnel, classe*

Exemple



- 1-N : une filière contient zéro ou plusieurs cours et chaque cours appartient à une filière au maximum
- N-M : un étudiant suit zéro ou plusieurs cours et chaque cours est suivi par zéro ou plusieurs étudiants
- 1-1 : une filière a un enseignant responsable au maximum et un enseignant est responsable d'une filière au maximum

L'objet-relationnel

MODÈLE RELATIONNEL

- tables
- attributs/domaines
- clés / clés étrangères

+

MODÈLE OBJET

- collections
- identifiant d'objet
- héritage
- méthodes
- types utilisateurs
- polymorphisme

Une « classe » est définie par un *type abstrait de données* avec des méthodes, et des tables

La ***persistance*** des objets est assuré par des **tables**

L'objet-relationnel : SQL3 (SQL:99)

Extension du modèle relationnel

- attributs complexes / multivalués : tuple, liste, tableau, ensemble, ...
- héritage sur relations et types
- domaine type abstrait de données (structure cachée + méthodes)
- identité d'objets

Extension de SQL

- définition des types complexes avec héritage
- appels de méthodes en résultat et qualification
- imbrication des appels de méthodes
- surcharge d'opérateurs

Objet « Paul »

[Nsécu : 1234

Nom : Paul

Adresse : Paris

**Enfants : { [Prénom : Léa, Age : 7],
[Prénom : Max, Age : 5] }**

Voitures : { [Marque : Ferrari

Type : F355

Photo :



],

[Marque : Citroën

Type : 2CV

Photo :



] }

]

Objet « Paul »

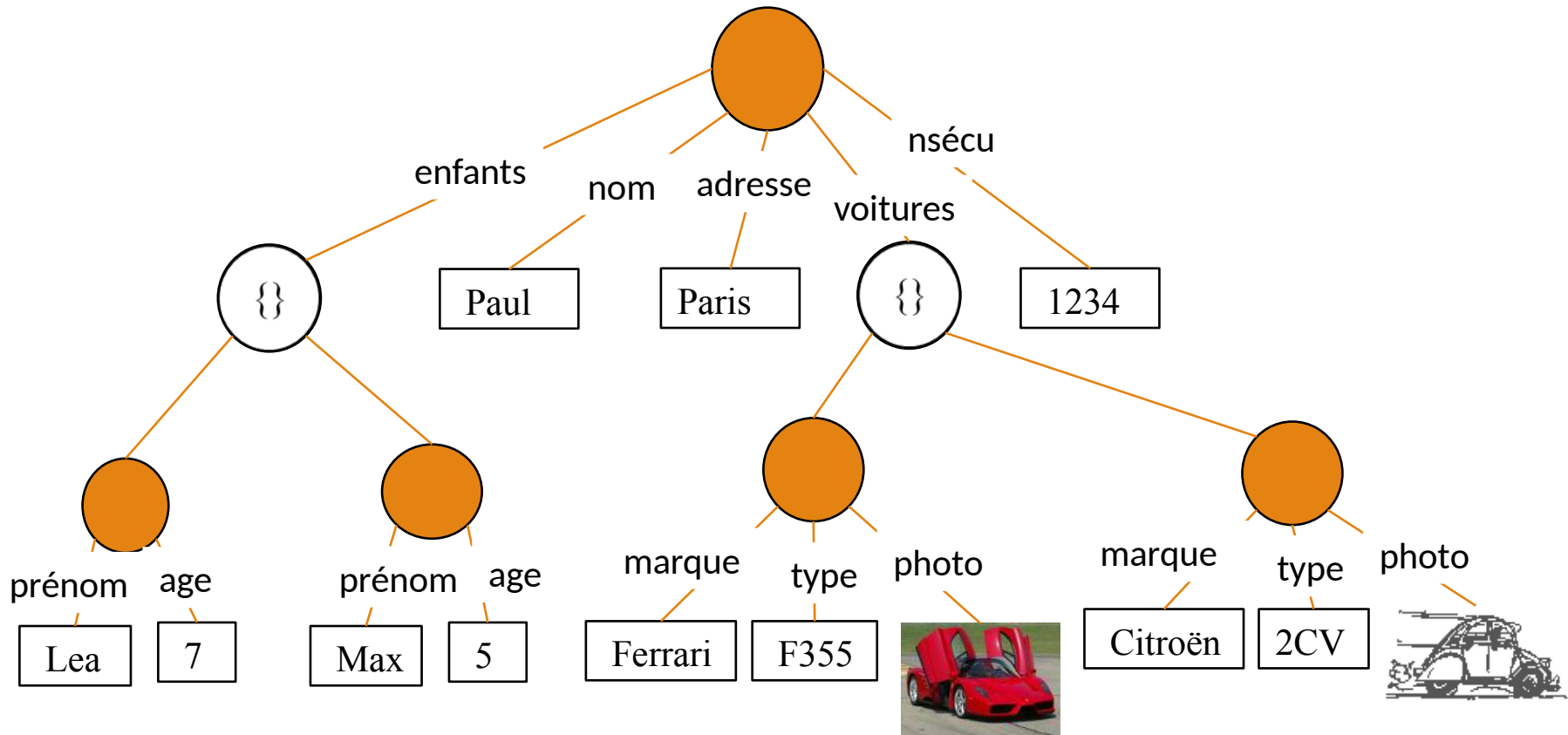




Table d'objet

NSécu	Nom	Adresse	Enfants		Voitures		
1234	Paul	Paris	Prénom	Age	Ferrari	F355	
			Léa	7			
			Max	5	Citroën	2C V	

Valeur complexe

Les concepts « objet » de SQL3 (SQL:99)

Types de données objet

- Définition de types abstraits
- Identité d'objet

Objets complexes

- Constructeurs de types (tuples, set, list, ...)
- Utilisation de référence (OID)

Héritage

- Définition de sous-types
- Définition de sous-tables
- Surcharge de méthodes

Le modèle SQL3 (SQL:99)

Extension objet du relationnel, inspirée de C++

- **types** : types abstrait avec fonctions (méthodes)
- **objets** : instances de type référencée par un identifiant d'objets (OID)
- **collections** : constructeur de type ensemblistes
 - table, tuple, set, list, bag, array
- **fonctions** :
 - associée à une base, une table ou un type
 - écrite en SQL, SQL3 PSM (Persistent Stored Module) ou langage externe (C, C++, Java, etc.)
- sous-typage et héritage
- association : références vers des objet simples ou ensemblistes

Types atomiques

Types atomiques prédéfinis (SQL:89)

- Varchar2(<longueur>)
- Number(<longueur>)
- Date

Création de types atomiques (SQL:99):

```
create type nsecu as varchar2(15);  
create type taille as number(3) ;  
create type datenais as Date;
```

Impossible dans Oracle

Types objet (SQL3/SQL:99 Oracle)

Types « objet »:

```
create type <nom-type> as object (  
    (<nom-attribut> [ref] <type>, )+  
    (<declaration-methodes>)*  
);  
/
```

```
drop type <nom-type> [force];
```

Exemple :

```
create type Personne as object (  
    nom Varchar2(10),  
    nss Varchar2(20),  
    datenais Date) ;  
/
```

Dans Oracle il faut terminer
la commande **create type** avec '/'
dans une ligne séparée
(pas toujours présent dans la suite)

Types ensemblistes (SQL:99 Oracle)

Types ensemblistes (array):

- table : ensemble avec doublons
- varray : collection ordonnée (liste) avec doublons et taille max

```
create type <nom-type> as  
           (table | varray(<longueur>)) of <type>;  
/
```

Exemple:

```
create type Retraites as table of Personne;  
/  
create type Centenaires as varray (50) of Personne ;  
/
```

Attributs de type objet

LesEtudiants

NOM	SUIT		
Max	MLBDA	Doucet	20
Lucie	MLBDA	Doucet	20
Marie	MLBDA	Doucet	20
Paul	LRC	Ganascia	30

```
create type Etudiant as object (  
  nom varchar2(15),  
  suit Cours);  
/
```

LesCours

TITRE	RESP	HEURES
MLBDA	Doucet	20
LRC	Ganascia	30

```
create type Cours as object (  
  titre varchar2(15),  
  resp varchar2(20),  
  heures number(2) );  
/
```

suit est un attribut de type **objet** et contient la **valeur** d'un objet de type Cours

Références (type ref)

Le constructeur de type ref définit une référence vers objet identifié par son OID (partage d'objets).

LesEtudiants

NOM	SUIT
Max	
Lucie	
Marie	
Paul	

LesCours

TITRE	RESP	HEURES
MLBDA	Doucet	20
LRC	Ganascia	20

```
create type Cours as object (  
    titre varchar2(15),  
    resp varchar2(20),  
    heures number(2)  
);  
/
```

```
create type Etudiant as object (  
    nom varchar2(15),  
    suit ref Cours  
);  
/
```

L'attribut **suit** contient un **identifiant** (référence) d'un objet de type **Cours**.

Utilisation des références : Association 1-1

On utilise le type ref pour faire référence à un objet. Les associations se modélisent à l'aide d'attributs en utilisant le type REF.

Association 1-1 :

```
create type Personne as object  
  (nom Varchar2(10),  
   conjoint ref Personne);  
/
```

Une personne a un conjoint, qui est une personne qui l'a aussi comme conjoint (cycle).

« Remplace » les clés étrangères du modèle SQL-89

Association 1-N : table de références

Le type de l'attribut doit être une collection (table ou varray)

```
create type Personne;  
/  
create type Ens_enfant as table of ref Personne;  
/  
create type Personne as object (  
    nom varchar(10),  
    adresse varchar2(30), datenais date,  
    enfants Ens_enfant  
);  
/
```

Association N-M : deux tables de références

Un cours est suivi par plusieurs étudiants et un étudiant suit plusieurs cours

```
create type Cours; /* type incomplet (cycle) */  
/  
create type Ens_cours as table of ref Cours;  
/  
create type Etudiant as object (  
    nom varchar2(15),  
    suit Ens_cours);  
/  
create type Ens_etud as table of ref Etudiant;  
/  
create type Cours as object (  
    titre varchar2(15),  
    resp ref Personne,  
    suiviPar Ens_etud);
```

Sous-typage et héritage : not final et under

- **not final** : le type objet Personne peut être spécialisé (**final** par défaut)
- **not instantiable** : type sans instance

```
create type Personne as object
    (nom varchar2(10), adresse varchar2(30), datenais date)
    not final not instantiable;
/
create type Salarie under Personne
    (affectation varchar2(10), repos varchar2 (15) );
/
create type Etudiant under Personne
    (universite varchar2(20), no_etudiant number(10)) ;
```

Méthodes

Fonctions ou procédures associées à un type d'objet.

Modélisent le comportement d'un objet

Ecrites en PL/SQL ou JAVA, et sont stockées dans la base.

Déclaration méthode

```
member function <nom-fonction>  
    [(<nom-para> in <type>, ...)]  
    return    <type-resultat>  
member procedure <nom-proc>  
    [(<nom-para> in <type>, ...)]
```

Implémentation / définition

```
create type body <type-objet> as  
    <nom-méthode> is  
    <declaration var-locales>  
    begin  
        <corps de la methode>  
    end;  
    /
```

Méthodes : implémentation (body)

Le corps de la méthode est défini dans le type de l'objet « receveur ».

```
create type body <type-objet> as  
    <declaration-methode> is  
    <declaration var-locales>  
begin  
    <corps de la methode>  
end;
```

```
create type Personne as object (  
    nom Varchar2(10),  
    nss Varchar2(10),  
    datenais Date,  
    member function get_age  
        return Number) ;
```

```
create type body Personne as  
    member function get_age  
        return Number is  
begin  
    return sysdate - datenais;  
end get_age;  
end;
```

Méthodes « abstraites » et spécialisation

not instantiable : méthode abstraite (sans implémentation dans le type indiqué)

overriding : spécialisation d'une méthode

```
create type Personne as object (  
    nom varchar2(10),  
    adresse varchar2(30), datenais date,  
    not instantiable member function statut return varchar2)  
not final not instantiable;
```

```
create type Etudiant under Personne (  
    université varchar2(20),  
    no_etudiant number(10),  
    overriding member function statut return varchar2) ;
```

On n'indique pas la longueur

Stockage des données

Les objets sont stockés dans des **tables** (racines de persistance)

- comme n-uplet,

```
create table <nom-table> of <nom-type>;
```

On termine pas avec '/'

- ou comme attribut d'un n-uplet.

```
create table <nom-table> (  
    (<nom-attribut> [ref] <nom type>, )+  
);
```


Exemples de racines de persistance

Objet comme n-uplet dans une table :

```
create table LesPersonnes of Personne;
```

LesPersonnes est une table d'objets de type *personne* (racine de persistance).

Objets comme attribut d'un n-uplet :

```
create table LesFamilles (  
    nom Varchar2(10),  
    pere Personne,  
    mere Personne);
```

Les attributs **pere** et **mere** de la relation LesFamilles sont des objets de type *Personne*.

Sous-typage et héritage

```
// types
create type Personne as object
    (nom varchar2(10), adresse varchar2(30), datenais date)
    not final;
/
create type Salarie under Personne
    (affectation varchar2(10), repos varchar2 (15));
/
create type Etudiant under Personne
    (universite varchar2(20), no_etudiant number(10)) ;
/
// tables
create table LesPersonnes of Personne
    (primary key (nom)) ;

create table LesSalaries of Salarie under LesPersonnes ;
```

« under » n'est pas
implémenté dans Oracle

Stockage des collections

Attribut de type **varray** : collection de taille « fixe »

- On définit un type T1 comme **varray** sur T2
- L'attribut est de type T1

```
create type <T1> as varray (<N>) of <T2>  
/  
create table <nom-table> of <nom-type>  
  <nom-attribut> <T1>;
```

Attribut de type **table** : collection de taille variable

- On définit l'attribut comme une table imbriquée (**nested table**) avec un nom (**store as**)

```
create table <nom-table> of <nom-type>  
  nested table <nom-attribut>  
    store as <nom-table-imbriquee>;
```

Tables imbriquées

```
create type Toeuvre as object (  
    titre varchar2(30),  
    otype varchar2(20),  
    odate Date);  
/  
create type ens_oeuvres as table of Toeuvre;  
/
```

```
create table Artiste (  
    nom varchar2(30),  
    Datenais Date,  
    œuvres ens_oeuvres)  
nested table œuvres store as Tab1;
```

Nom	Datenais	oeuvres		
Picasso	1881	oeuvre	otype	oDate
		Guernica	Huile	1937
		Le rêve	Huile	1932
		Tête de femme	Sculpture	1931
Van Gogh	1853	oeuvre	otype	oDate
		Iris	Huile	1889
		tournesols	Huile	1888

Stockage des tables imbriquées

```
create type Toeuvre as object (  
    titre varchar2(30),  
    otype varchar2(20),  
    odate Date);  
  
/  
create type ens_oeuvres as table of Toeuvre;  
/  
create table Artiste (  
    nom varchar2(30),  
    Datenais Date,  
    œuvres ens_oeuvres)  
    nested table œuvres store as Tab1;
```

Artiste

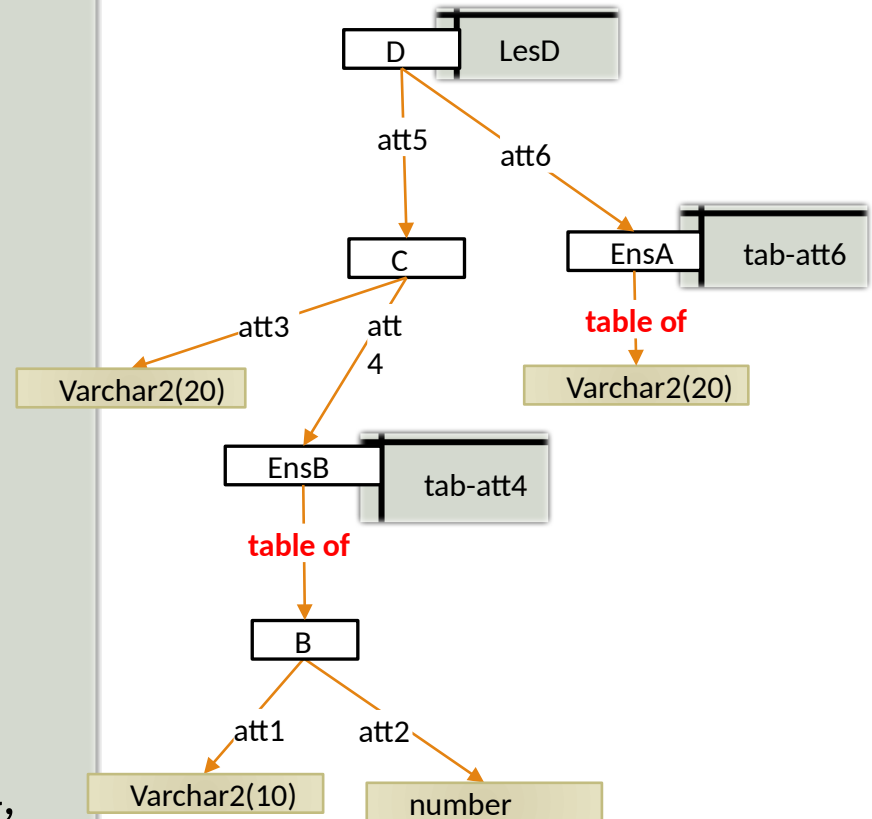
Nom	Datenais	oeuvres
Picasso	1881	index 1
Van Gogh	1853	index 2

Tab1

Tindex	oeuvre	otype	oDate
index1	Guernica	Huile	1937
index1	Le rêve	Huile	1932
index2	Iris	Huile	1889
index1	Tête de femme	Sculpture	1931
index2	tournesols	Huile	1888

Exemple nested table imbriquée

```
create type EnsA as table of varchar2(20);  
/  
create type B as object (  
    att1 varchar2(10),  
    att2 number);  
/  
create type EnsB as table of B;  
/  
create type C as object (  
    att3 varchar2(20),  
    att4 EnsB);  
/  
create type D as object (  
    att5 C,  
    att6 EnsA );  
/  
create table LesD of D  
    nested table att5.att4 store as tab-att4,  
    nested table att6 store as tab-att6;
```



Compilation et validation (info pour les TME)

Exemple

```
create type A;  
/  
create type Ens_A as table of ref A;  
/  
create type A as object  
  (nom varchar(10),  
   adresse varchar2(30));  
/
```

Ens_A est valide

```
drop type A force;
```

Ens_A n'est plus valide

```
create or replace type A as object  
  (nom varchar(10),  
   adresse varchar2(30),  
   datenais date);
```

Ens_A n'est pas valide

```
alter type Ens_A compile;
```

Ens_A est valide à nouveau

Compilation et validation (info pour les TME)

Afficher tous les objets invalides

```
select object_name, object_type, status  
from user_objects  
where status = 'INVALID';
```

Afficher les erreurs

```
show errors ;
```

Afficher les erreurs d'un type donné

```
show errors type <nom_type> ;
```

Recompiler tous les objets

```
exec dbms_utility.compile_schema(schema => 'AMANN', compile_all => false);
```