Dossier 2
Introduction aux bases de données objet-relationnelles

La norme SQL3 a considérablement étendu les possibilités du langage (enrichissement des jointures, expressions de tables `WITH`, triggers, gestion des erreurs, requêtes récursives, etc.). De plus, cette version donne à l'utilisateur les outils pour implémenter des bases de données objet-relationnelles.

Les bases de données objet-relationnelles combinent un moteur relationnel et un système objet. Ainsi, comme l'a indiqué Stonebraker, il est maintenant possible de manipuler des **données complexes** (ce qui n'est pas possible avec des bases de données relationnelles qui ne gèrent que les types primitifs) et de conserver, en même temps, la possibilité de réaliser des **requêtes d'interrogation sophistiquées** (ce qui n'est pas possible avec des bases de données objet natives). Cette alternative doit aussi permettre aux entreprises de continuer à utiliser leur base de données traditionnelle tout en leur donnant la possibilité de rajouter une couche objet.

Les bases de données objet-relationnelles reposent sur les grands principes relationnels que nous avons vus les années précédentes auxquels s'ajoutent cinq extensions : les types abstraits de données (TAD), l'identité des objets (OID : Object IDentifier qui permet d'accéder aux objets par l'intermédiaire de références), les collections d'objets, les méthodes et l'héritage.

1 Les Types Abstraits de Données (TAD)

Dans les bases de données objet-relationnelles, les types abstraits de données se rapprochent du concept de classe qui est utilisé en programmation orientée objet. En effet, ces TAD permettent à l'utilisateur de définir ses propres types de données (`CREATE TYPE`) qui vont permettre ensuite d'instancier des objets ayant la même structure et le même comportement. De plus, comme nous le verrons dans les paragraphes qui suivent, les TAD permettent aussi de mettre en œuvre différents mécanismes tels que l'encapsulation, l'héritage et le polymorphisme. Les TAD peuvent être soit simples (composés uniquement d'attributs de type primitif) soit composés (leur création fait intervenir d'autres TAD).

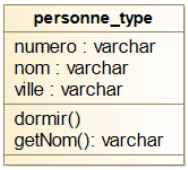
1.1 Les TAD simples

Dans un TAD simple, tous les attributs du type sont primitifs (`VARCHAR`, `DATE`, `NUMBER`, etc.). Lors de la création d'un TAD, l'encapsulation se traduit par la déclaration de méthodes (fonctions ou procédures) qui pourront être appelées sur des instances du TAD.
Attention : un type abstrait de données **ne possède pas de contraintes ni de clé primaire** !

ex : on décide de créer un Type Abstrait de Données que l'on nomme personne_type. Une personne possède un numéro, un nom et une ville. Une personne doit pouvoir dormir et retourner son nom.

Création d'un type abstrait de données

```
CREATE OR REPLACE TYPE personne_type AS OBJECT
(numero VARCHAR(5), nom VARCHAR(20), ville VARCHAR(20),
MEMBER PROCEDURE dormir,
MEMBER FUNCTION getNom RETURN VARCHAR) ;
```



La commande SQL*Plus `DESC` (ou bien `DESCRIBE`) permet d'extraire la structure du premier niveau d'un type.
Par exemple, `DESC personne_type`

Un TAD peut servir à créer différentes tables objet ou bien d'autres TAD.

Dans notre exemple, où l'on souhaite gérer la bibliothèque de l'IUT, on veut utiliser ce même type pour créer deux tables objet différentes. La table Etudiant et la table Auteur

Etudiant

numéro	nom	ville
E1	Zetofrais	Montpellier
E2	Bricot	Montpellier
E3	Zeblouze	Nîmes

Auteur

numéro	nom	ville
A1	Roques	Montpellier
A2	Palleja	Toulouse
A3	Crampes	Toulouse

C'est au moment de la création d'une table objet qui utilise un type abstrait, qu'il est possible de déclarer les différentes contraintes et en particulier la contrainte de clé primaire. Contrairement aux tables relationnelles qui contiennent des tuples, les tables objet contiennent des objets (des instances du TAD).

Création de tables objet utilisant un TAD

```
CREATE TABLE Etudiant OF personne_type
(CONSTRAINT pk_Etudiant PRIMARY KEY (numero),
 CONSTRAINT nn_nom_Etudiant CHECK (nom IS NOT NULL)) ;
CREATE TABLE Auteur OF personne_type
(CONSTRAINT pk_Auteur PRIMARY KEY (numero)) ;
```

Une fois les tables créées, les mises à jour sont réalisées comme dans les tables relationnelles des bases de données relationnelles classiques. Il est possible de modifier un type ainsi que tous les objets qui en dépendent (autres types ou tables) grâce à l'instruction `ALTER TYPE`.

Insertion de données

```
INSERT INTO Etudiant VALUES ('E1', 'Zetofrais','Montpellier') ;
```

On peut utiliser le constructeur de type mais, dans le cas présent, cela n'est pas obligatoire :

```
INSERT INTO Etudiant VALUES (personne_type('E2', 'Bricot','Montpellier'));
```

Mise à jour de données

```
UPDATE Etudiant e SET e.ville = 'Nîmes' WHERE e.numero = 'E1' ;
```

Suppression de données

```
DELETE FROM Etudiant e WHERE e.nom = 'Zetofrais' ;
```

Modification d'un type (ainsi que des types et des tables qui en dépendent)

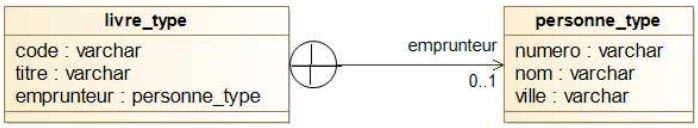
```
ALTER TYPE personne_type ADD ATTRIBUTE (dateNaissance DATE) CASCADE;
```

1.2 Les TAD complexes utilisant d'autres TAD imbriqués

Nous avons vu dans le paragraphe précédent, qu'un TAD peut être utilisé pour créer des tables. Il peut aussi être utilisé pour créer un autre TAD. Et il est alors possible d'imbriquer un type dans un autre.

Par exemple, on décide de créer un TAD que l'on nomme livre_type. Un livre possède un code, un titre et, dans le cas où il serait emprunté, un étudiant emprunteur (bien sûr si le livre n'est pas emprunté cet attribut a pour valeur `NULL`).

Création d'un type complexe




```
CREATE TYPE livre_type AS OBJECT (code VARCHAR(5), titre VARCHAR(20),
                                  emprunteur personne_type) ;
```

Maintenant il suffit de créer une table *'Livre'* de type `livre_type` puis d'y insérer des objets.

Livre

code	titre	emprunteur		
		numéro	nom	ville
L1	UML en Action	E1	Zetofrais	Montpellier
L2	Merise pour les nuls	E2	Bricot	Montpellier
L3	Java et UML	E3	Zeblouze	Nîmes

Création d'une table objet

```
CREATE TABLE Livre OF livre_type
(CONSTRAINT pk_Livre PRIMARY KEY (code),
 CONSTRAINT un_titre_Livre UNIQUE (titre)) ;
```

Insertion

Ici, pour insérer des objets dans la table *'Livre'* il faut faire appel au constructeur de `personne_type` :

```
INSERT INTO Livre VALUES ('L1','UML en Action',
                          personne_type ('E1', 'Zetofrais', 'Montpellier')) ;
```

Modification

On veut modifier la ville de l'emprunteur du livre L1

```
UPDATE Livre l SET l.emprunteur.ville = 'Nîmes' WHERE l.code = 'L1' ;
```

Suppression

```
DELETE FROM Livre l WHERE l.emprunteur.nom = 'Zetofrais' ;
```

Requête d'accès aux données

```
SELECT l.titre FROM Livre l WHERE l.emprunteur.nom = 'Zetofrais' ;
```

2 Les références (ou pointeurs)

Contrairement aux tuples qui se trouvent dans les tables relationnelles, les objets des tables objet n'ont pas nécessairement besoin d'une clé primaire car ils peuvent être identifiés par un **OID** - Object IDentifier - unique qui représente l'adresse logique de l'objet (dans un environnement distribué ou répliqué, le SGBD est programmé pour garantir l'unicité des OID). Grâce à cela, les liens entre les différentes tables peuvent être réalisés par des **références** (`REF`) pointant sur l'OID d'un objet. Dans une base de données objet-relationnelle, les références peuvent soit se substituer entièrement aux clés étrangères, soit les compléter.

Sous Oracle, l'utilisation des références (plutôt que des clés étrangères) permet de simplifier sensiblement l'écriture des requêtes d'accès aux données. En effet la navigation entre les différentes tables est alors réalisée par l'intermédiaire des pointeurs ce qui rend les **jointures implicites**. Ces pointeurs permettent également de gérer les contraintes d'intégrité référentielles grâce à la clause `REFERENCES`. Mais, pour rendre la gestion de cette contrainte plus facile, il est fortement conseillé de n'utiliser dans le schéma que des tables objet. Toutefois, ces références ont l'inconvénient de ne pas pouvoir jouer le rôle de clé primaire. En effet, il n'est pas possible de déclarer une contrainte `UNIQUE` sur une référence (mais il est tout de même possible d'implanter une contrainte `NOT NULL`).

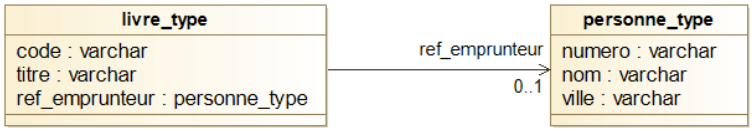
Quoi qu'il en soit, les références vont permettre, tout comme les clés étrangères, de gérer les associations de type un-plusieurs ainsi que plusieurs-plusieurs.

2.1 Association 'un – plusieurs'

L'utilisation de références permet d'éliminer toute redondance d'information. Les associations 'un – plusieurs' peuvent être traduites par une référence placée dans la table fille.

exemple : On souhaite à nouveau gérer la bibliothèque de l'IUT. Un livre peut être emprunté par au plus **un seul** étudiant. Par contre un étudiant peut emprunter **plusieurs** livres (pour simplifier, on va dire qu'il n'y a pas de quota maximum).

Création de types

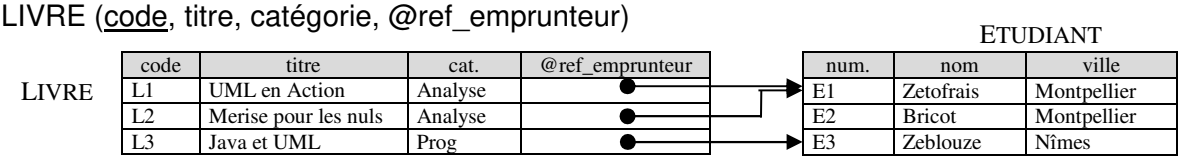


Pour indiquer qu'un attribut est une référence on utilise le mot clé **`REF`** suivi du type de l'objet qu'il pointe (attention si on oublie le mot clé `REF` on crée un type imbriqué !)

```
CREATE TYPE personne_type AS OBJECT (numero VARCHAR(5), nom VARCHAR(20),
                                                        ville VARCHAR(20)) ;
/
CREATE TYPE livre_type AS OBJECT (code VARCHAR(5), titre VARCHAR(20),
                  categorie VARCHAR(20), ref_emprunteur REF personne_type) ;
```

On peut ensuite utiliser ces types pour créer des tables d'objets. Par convention, dans le schéma objet-relationnel, on fera précéder les références du caractère @

ETUDIANT (numero, nom, ville)
LIVRE (code, titre, catégorie, @ref_emprunteur)



Création de tables objet

La contrainte d'intégrité référentielle peut être assurée sur les références de tables objet grâce à la clause `REFERENCES`. Dans ce cas, la cohérence du fils vers son père (lors d'insertion et de modification dans la table fille) est totalement assurée. La cohérence du père vers le fils (lors de suppression dans la table père) est également assurée ; mais les messages d'erreur retournés par Oracle, lors de suppressions non autorisées, peuvent être surprenants.

```
CREATE TABLE Etudiant OF personne_type
(CONSTRAINT pk_Etudiant PRIMARY KEY (numero)) ;
CREATE TABLE Livre OF livre_type
(CONSTRAINT pk_Livre PRIMARY KEY(code),
 CONSTRAINT rf_emprunteur_Livre ref_emprunteur REFERENCES Etudiant);
```

Fonctions permettant de manipuler des pointeurs

Oracle propose trois fonctions particulières pour manipuler des références.

- La fonction `REF()` : qui s'applique à un objet et qui retourne l'adresse de l'objet. Cette fonction est très utile lorsque on fait des insertions ou des mises à jours dans les tables afin de connaitre l'adresse des objets qui doivent être ciblés par des pointeurs.
  Par exemple : `SELECT REF(e) FROM Etudiant e WHERE e.numero = 'E1'`
- La fonction `DEREF()` : il s'agit de la fonction inverse de `REF()`. Cette fonction s'applique à un pointeur et retourne le contenu de l'objet pointé.
  Par exemple : `SELECT DEREF(l.ref_emprunteur) FROM Livre l WHERE l.code = 'L1'`
- La fonction `VALUE()` : cette fonction s'applique à un objet nommé par un alias, et retourne le contenu de l'objet pointé par l'alias.
  Par exemple : `SELECT VALUE(e) FROM Etudiant e WHERE e.numero = 'E1'`

Insertion

```
INSERT INTO Etudiant VALUES ('E1',' Dupont',' Montpellier') ;
INSERT INTO Livre VALUES ('L1',' UML en Action', 'Analyse', NULL) ;
INSERT INTO Livre VALUES ('L2', 'Merise pour les nuls', 'Analyse',
                    (SELECT REF(e) FROM Etudiant e WHERE e.numero = 'E1')) ;
```

Modification d'un pointeur

```
UPDATE Livre l
SET l.ref_emprunteur = (SELECT REF(e) FROM Etudiant e WHERE e.numero='E1')
WHERE l.code = 'L1' ;
```

Suppression d'un lien d'un pointeur (pas de l'objet)

Il s'agit en fait de la modification d'un attribut d'un objet. Cela est réalisé avec un `UPDATE`.

```
UPDATE Livre l
SET l.ref_emprunteur = NULL
WHERE l.code = 'L1' ;
```

Requêtes d'interrogation

Comme nous l'avons indiqué, les jointures sont implicites. Pour cela, l'utilisation des **alias** est obligatoire (pour ne pas avoir d'ambiguïté avec une expression de type `[nomSchema.nomTable]`). On peut remarquer que toutes les requêtes doivent partir de la table fille (ici la table `livre l` qu'on retrouve dans le `FROM`) puisqu'il n'est pas possible de naviguer à contre-courant d'un pointeur.

exemple : Nom des personnes qui possèdent un livre d'analyse

```
SELECT l.ref_emprunteur.nom
FROM Livre l
WHERE l.categorie = 'Analyse'
AND l.ref_emprunteur IS NOT NULL ;
```

exemple : Titre des livres empruntés par la personne E1

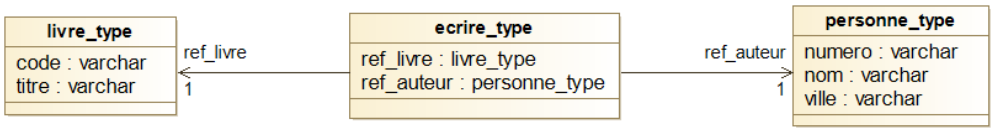
```
SELECT l.titre
FROM Livre l
WHERE l.ref_emprunteur.numero = 'E1' ;
```

2.2 Association 'plusieurs – plusieurs'

On peut également gérer les associations 'plusieurs – plusieurs' avec des références (même si dans ce cas-là, il est plus naturel d'utiliser des collections). Il suffit de créer une table intermédiaire comme on le fait dans les bases de données relationnelles.

exemple : Dans notre bibliothèque, un livre peut être écrit par **plusieurs** auteurs. Bien entendu un auteur peut écrire **plusieurs** livres.

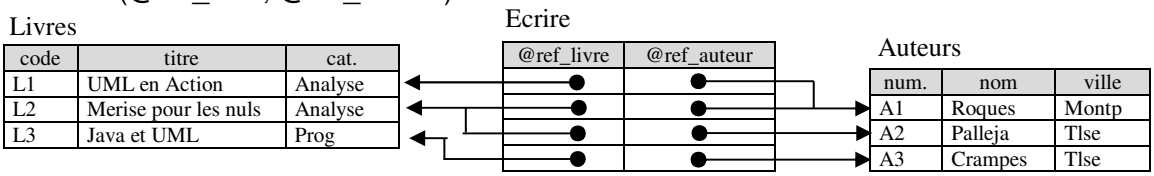
Création des types




```
CREATE TYPE personne_type AS OBJECT (numero VARCHAR(5),nom VARCHAR(20),
                                                    ville VARCHAR(20)) ;
CREATE TYPE livre_type AS OBJECT (code VARCHAR(5), titre VARCHAR(20),
                                                  categorie VARCHAR(20)) ;
CREATE TYPE ecrire_type AS OBJECT (ref_auteur REF personne_type,
                                               ref_livre REF livre_type) ;
```

On peut ensuite utiliser les types précédents pour obtenir ce schéma objet-relationnel :

LIVRE (code, Titre, Catégorie)
AUTEUR (numéro, Nom, Ville)
ECRIRE (@ref_livre, @ref_auteur)



Création des tables

```
CREATE TABLE Auteur OF personne_type
(CONSTRAINT pk_Auteur PRIMARY KEY (numero)) ;
CREATE TABLE Livre OF livre_type
(CONSTRAINT pk_Livre PRIMARY KEY (code)) ;
```

Toutefois, les références ne peuvent pas être clé primaire. Néanmoins, il est toujours possible d'indiquer une contrainte `NOT NULL` sur les références de la table intermédiaire et de programmer 'à la main' un trigger qui garantit l'unicité des objets référencés.

```
CREATE TABLE Ecrire OF ecrire_type
(CONSTRAINT nn_refAuteur_Ecrire CHECK (ref_auteur IS NOT NULL),
 CONSTRAINT ref_auteur_Ecrire ref_auteur REFERENCES Auteur,
 CONSTRAINT nn_refLivre_Ecrire CHECK (ref_livre IS NOT NULL),
 CONSTRAINT ref_Livre_Ecrire ref_livre REFERENCES Livre) ;

CREATE OR REPLACE TRIGGER Trig_Insert_Ecrire
BEFORE INSERT ON Ecrire FOR EACH ROW
DECLARE
   nb NUMBER;
BEGIN
   SELECT COUNT(*) INTO nb
   FROM Ecrire e
   WHERE e.ref_auteur = :NEW.ref_auteur AND e.ref_livre = :NEW.ref_livre ;
   IF (nb <> 0) THEN RAISE_APPLICATION_ERROR (-20001, 'doublon');
   END IF;
END;
```

Insertion d'objets

```
INSERT INTO Auteur VALUES ('A2', 'Palleja', 'Toulouse') ;
INSERT INTO Livre VALUES ('L2', 'Merise pour les nuls', 'Analyse') ;
INSERT INTO Ecrire VALUES ((SELECT REF(a) FROM Auteur a WHERE a.numero = 'A2'),
                           (SELECT REF(l) FROM Livre l WHERE l.code = 'L2')) ;
```

Requêtes d'interrogation

exemple : Numéro et nom des auteurs qui ont écrit un livre d'Analyse

```
SELECT DISTINCT e.ref_auteur.numero, e.ref_auteur.nom
FROM Ecrire e
WHERE e.ref_livre.categorie = 'Analyse' ;
```

3 Les Collections

Sous Oracle, on peut gérer les collections soit avec les tables imbriquées (nested tables) soit avec des tableaux prédimensionnés (traduction de *variable size arrays* – `VARRY`) que nous avons déjà vus en 2ème année de BUT sous PostgreSQL. Contrairement aux `VARRY`, les nested tables ont l'avantage d'être des collections non ordonnées et non limitées en taille. De plus, il est possible de réaliser des requêtes SQL sur les nested tables. Pour ces raisons, nous ne verrons ici que l'utilisation des collections à travers les nested tables.

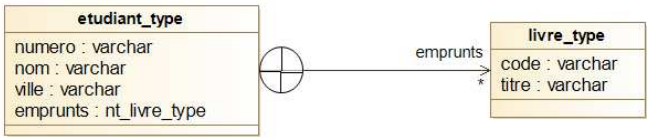
Les nested tables permettent de placer à l'intérieur d'un objet, une table imbriquée. Ainsi, dans les relations qui possèdent une nested table (et une collection en général), les attributs de la table imbriquée ne sont pas en dépendance fonctionnelle avec la clé primaire de la table externe. De ce fait, ces tables ne sont pas en première forme normale. On dit parfois qu'elles sont en NF² (Not First Normal Form).

Les nested tables peuvent être des collections d'objets ou bien des collections de références qui pointent sur des objets.

3.1 Collections d'objets imbriqués (sans pointeur)

exemple : soit des étudiants qui possèdent un numéro, un nom et un ville. Un étudiant peut emprunter **plusieurs** livres (qui ont un code et un titre). Un livre peut être emprunté que par **un** seul étudiant.

Création des types



Afin d'imbriquer une table de `livre_type` à l'intérieur du TAD `etudiant_type`, il faut créer un type intermédiaire `nt_livre_type` qui permet de stocker une table de `livre_type`. Cela peut se faire grâce à l'instruction `AS TABLE OF`. Il suffit ensuite d'indiquer que l'attribut `emprunts` est de type `nt_livre_type`.

```
CREATE TYPE livre_type AS OBJECT (code VARCHAR(5), titre VARCHAR(20)) ;
CREATE TYPE nt_livre_type AS TABLE OF livre_type ;
CREATE TYPE etudiant_type AS OBJECT (numero VARCHAR(5), nom VARCHAR(20),
                                     ville VARCHAR(20), emprunts nt_livre_type) ;
```

Ensuite, pour stocker les données, il suffit de créer qu'une seule table Etudiant. Tous les livres seront imbriqués à l'intérieur de cette table.
Sur le schéma objet-relationnel, les collections sont reconnaissables par des {}.

ETUDIANT (numéro, nom, ville, emprunts{code, titre})

Etudiant

numéro	nom	ville	emprunts {code, titre}	
			code	titre
E1	Zetofrais	Montpellier	L1	UML en Action
			L2	Merise pour les nuls
E2	Bricot	Montpellier		
E3	Zeblouze	Nîmes	L3	SQL pour les Ninjas

Création de la table

La clause `STORE AS` permet de nommer la structure interne qui stocke les enregistrements de cette table imbriquée. Cette espace ne sera pas accessible lorsqu'on fera des requêtes, mais il peut être utile pour gérer les contraintes sur la table imbriquée.

```
CREATE TABLE Etudiant OF etudiant_type
(CONSTRAINT pk_Etudiant PRIMARY KEY (numero))
NESTED TABLE emprunts STORE AS tabLliv_nt;
```

Insertions dans la table

Lors de l'insertion de lignes, il faut utiliser le constructeur de `nt_livre_type` afin de créer une nested table et d'initialiser l'attribut `emprunts`. Si cela n'est pas fait, la valeur de l'attribut `emprunts` pointe sur `NULL` et il ne sera pas possible par la suite de rajouter des livres dans `emprunts`.
Afin de manipuler une table imbriquée dans une requête, on peut utiliser la fonction `TABLE()` qui permet grâce à une requête SQL de sélectionner certaines tables imbriquées de la collection. On peut aussi utiliser la fonction `THE()` qui est équivalent pour les insertions.

```
INSERT INTO Etudiant VALUES ('E1', 'Zetofrais', 'Montpellier',
                             nt_livre_type (livre_type('L1', 'UML en Action'),
                                            livre_type('L2', 'Merise pour les nuls')));

INSERT INTO Etudiant VALUES ('E2', 'Bricot', 'Montpellier', nt_livre_type());

INSERT INTO Etudiant VALUES ('E3', 'Zeblouze', 'Nimes', nt_livre_type());

INSERT INTO THE (SELECT e.emprunts FROM Etudiant e WHERE e.numero = 'E3')
            VALUES ('L3', 'SQL pour les Ninjas');

INSERT INTO TABLE (SELECT e.emprunts FROM Etudiant e WHERE e.numero = 'E2')
            VALUES ('L4', 'Java et UML');
```

Requête d'interrogation

Pour faire une requête d'interrogation sur une table imbriquée, il est possible d'utiliser les fonctions `TABLE()` ou `THE()`. Mais la fonction `THE()` est plus limitée car elle ne peut retourner qu'une seule table de la collection. Si la requête SQL retourne plusieurs tables imbriquées cela provoque une erreur. La fonction `TABLE()` peut également être associée à une autre table dans le `FROM`. Une jointure implicite est alors réalisée par le SGBD entre la table imbriquée et la table externe qui contient la table imbriquée (cela n'est pas possible avec un `THE()`).

Titre des livres empruntés par l'étudiant E1 (Mélanie Zétofrais)

```
SELECT emp.titre
FROM THE(SELECT e.emprunts FROM Etudiant e WHERE e.numero = 'E1') emp;

SELECT emp.titre
FROM TABLE(SELECT e.emprunts FROM Etudiant e WHERE e.numero = 'E1') emp;

SELECT emp.titre
FROM Etudiant e,
     TABLE(e.emprunts) emp
WHERE e.numero = 'E1';
```

Nom de l'emprunteur du livre SQL pour les Ninjas

```
SELECT e.nom
FROM Etudiant e,
     TABLE(e.emprunts) emp
WHERE emp.titre = 'SQL pour les Ninjas';
```

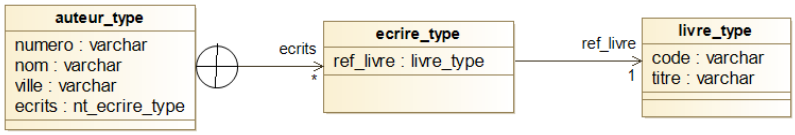
3.2 Collections de pointeurs

Les collections avec des pointeurs permettent de gérer les associations plusieurs-plusieurs sans avoir de la redondance. Elles peuvent aussi être utilisées pour gérer des associations un-plusieurs en plaçant la collection du côté de la table mère (par exemple, dans l'exemple du §2.1 on pourrait avoir un étudiant qui a une collection de livres).

3.2.1 Collection pour gérer une association plusieurs-plusieurs

exemple : soit des auteurs qui ont un numéro, un nom et une ville. Un auteur a écrit **plusieurs** livres qui sont décrits par un code et un titre. Et un livre peut être écrit par **plusieurs** auteurs.

Création des types



On peut soit mettre une collection de livres dans auteur, soit une collection d'auteurs dans livre (soit les deux). Si on choisit d'avoir une collection de livres dans les auteurs, il faut commencer par créer un type `ecrire_type` qui contient des pointeurs sur des `livre_type`. Ensuite, comme précédemment, afin d'imbriquer une table de `ecrire_type` à l'intérieur du TAD `auteur_type`, il faut créer un type intermédiaire `nt_ecrire_type` qui permet de stocker une table de `ecrire_type`. Il suffit ensuite d'indiquer que l'attribut `ecrits` est de type `nt_ecriture_type`.

```
CREATE TYPE livre_type AS OBJECT (code VARCHAR(5),titre VARCHAR(20)) ;
CREATE TYPE ecrire_type AS OBJECT (ref_livre REF livre_type) ;
CREATE TYPE nt_ecrire_type AS TABLE OF ecrire_type ;
CREATE TYPE auteur_type AS OBJECT (numero VARCHAR(5), nom VARCHAR(20),
                                   ville VARCHAR(20), ecrits nt_ecrire_type) ;
```

Dans le schéma objet-relationnel on se retrouve donc avec deux tables.

LIVRE (code, titre)
AUTEUR (numero, nom, ville, écrits{@ref_livre})

Auteur

numero	nom	ville	ecrits{ref_livre}
			@ref_livre
A1	Roques	Montp	
A2	Palleja	Tlse	
A3	Crampes	Tlse	

Livre

code	titre
L1	UML en Action
L2	Merise pour les nuls
L3	SQL pour les Ninjas

Création de la table

```
CREATE TABLE Livre OF livre_type
(CONSTRAINT pk_Livre PRIMARY KEY (code));
CREATE TABLE Auteur OF auteur_type
(CONSTRAINT pk_Auteur PRIMARY KEY (numero))
NESTED TABLE ecrits STORE AS tablivre_nt;
```

Insertions dans la table

```
INSERT INTO Livre VALUES ('L2', 'Merise pour les nuls');
INSERT INTO Auteur VALUES ('A2','Palleja', 'Toulouse', nt_ecrire_type());
INSERT INTO THE(SELECT a.ecrits FROM Auteur a WHERE a.numero = 'A2')
                          SELECT REF(l) FROM Livre l WHERE l.code = 'L2';
INSERT INTO TABLE(SELECT a.ecrits FROM Auteur a WHERE a.numero = 'A2')
                          SELECT REF(l) FROM Livre l WHERE l.code = 'L3';
```

Requête d'interrogation

Titre des livres écrits par l'auteur A2

```
SELECT ecr.ref_livre.titre
FROM THE(SELECT a.ecrits FROM Auteur a WHERE a.numero = 'A2') ecr;
```

```
SELECT ecr.ref_livre.titre
FROM Auteur a,
     TABLE(a.ecrits) ecr
WHERE a.numero = 'A2';
```

Nom de l'auteur qui a écrit le livre SQL pour les Ninjas

```
SELECT a.nom
FROM Auteur a,
     TABLE(a.ecrits) ecr
WHERE ecr.ref_livre.titre = 'SQL pour les Ninjas';
```

Le nombre de livres écrits par l'auteur Palleja

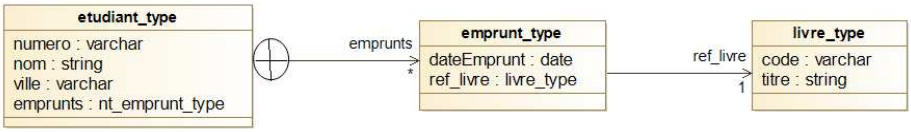
```
SELECT cardinality(a.ecrits)
FROM Auteur a,
WHERE a.nom = 'Palleja';
```

3.2.2 Collection pour gérer une association un-plusieurs

exemple : Un livre peut être emprunté par au plus **un seul** étudiant. Par contre un étudiant peut emprunter **plusieurs** livres. On souhaite connaitre la date de l'emprunt.

On pourrait gérer ce cas sans collection. En mettant dans le type `livre_type` une référence vers l'`etudiant_type` qui a emprunté le livre (voir § 2.1). Ou alors on peut utiliser une collection pour connaitre les livres empruntés par un étudiant.

Création des types



En plus d'avoir le livre emprunté, on veut aussi la date de l'emprunt. Le type `emprunt_type` est donc composé de deux attributs (cela n'est pas spécifique aux associations un-plusieurs ; on pourrait avoir la même chose avec une association plusieurs-plusieurs qui est porteuse).

```
CREATE TYPE livre_type AS OBJECT (code VARCHAR(5),titre VARCHAR(20)) ;
CREATE TYPE emprunt_type AS OBJECT (dateEmprunt DATE, ref_livre REF livre_type) ;
CREATE TYPE nt_emprunt_type AS TABLE OF emprunt_type ;
CREATE TYPE etudiant_type AS OBJECT (numero VARCHAR(5), nom VARCHAR(20),
                          ville VARCHAR(20), emprunts nt_emprunt_type) ;
```

LIVRE (<u>code</u>, titre)
ETUDIANT (<u>numero</u>, nom, ville, emprunts{dateEmplrunt, @ref_livre})

Etudiant

numero	nom	ville	emprunts{dateE, ref_livre}	
			dateE	@ref_livre
E1	Zétofrais	Mtp	15/01	●
			20/01	●
E2	Zeblouse	Mtp	16/01	●
E3	Bricot	Sète	18/01	●

Livre

code	titre
L1	UML en Action
L2	Merise pour les nuls
L3	SQL pour les Ninjas

Création de la table

```
CREATE TABLE Livre OF livre_type
(CONSTRAINT pk_Livre PRIMARY KEY (code));
CREATE TABLE Etudiant OF etudiant_type
(CONSTRAINT pk_Etudiant PRIMARY KEY (numero))
NESTED TABLE emprunts STORE AS tabEmprunts_nt;
```

Un trigger pourrait empêcher qu'un livre soit emprunté par plusieurs étudiants.

Insertions dans la table

```
INSERT INTO Livre VALUES ('L2', 'Merise pour les nuls');
INSERT INTO Etudiant VALUES ('E1','Zeblouse', 'Mtp', nt_emprunt_type());
INSERT INTO TABLE(SELECT e.emprunts FROM Etudiant e WHERE e.numero = 'E1')
              SELECT '15/01/2023', REF(l) FROM Livre l WHERE l.code = 'L2';
```

Requête d'interrogation

Nom des étudiants qui ont emprunté un livre le 15/01

```
SELECT e.nom
FROM Etudiant e,
     TABLE(e.emprunts) emp
WHERE emp.dateEmprunt = '15/01/2023';
```

4 Les méthodes

Dans le modèle objet-relationnel il est possible de définir des méthodes (procédures ou fonctions) dans un TAD. Comme nous l'avons vu dans le paragraphe 1.1, ces méthodes doivent être, dans un premier temps, déclarées lors de la création du TAD (via l'instruction `DECLARE TYPE`). Elles doivent ensuite être implémentées à l'aide de l'instruction `CREATE TYPE BODY`. Ces méthodes peuvent être des méthodes d'instance `MEMBER` (applicables sur un objet précis de la table) ou bien des méthodes de classe `STATIC` (applicables sur la table).

Afin d'illustrer cela nous allons reprendre l'exemple de la bibliothèque avec le schéma objet-relationnel et la déclaration des types qui suivent :

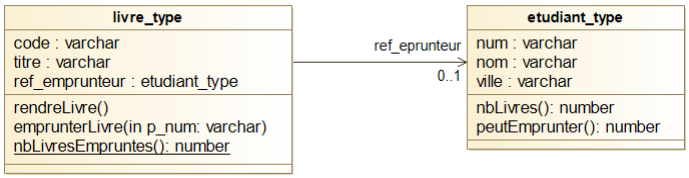


Schéma objet-relationnel

ETUDIANT (num, nom, ville)
LIVRE (code, titre, catégorie, @ref_emprunteur)

Déclaration des types et des méthodes

```
CREATE OR REPLACE TYPE etudiant_type AS OBJECT
(num VARCHAR(5), nom VARCHAR(20), ville VARCHAR(20),
 MEMBER FUNCTION nbLivres RETURN NUMBER,
                    -- La fonction nbrLivres retourne le nombre d'emprunts de l'étudiant
 MEMBER FUNCTION peutEmprunter RETURN NUMBER);
              -- La fonction peutEmprunter retourne 1 si l'étudiant peut emprunter ; 0 sinon

CREATE OR REPLACE TYPE livre_type AS OBJECT
(code VARCHAR(5), titre VARCHAR(20), ref_emprunteur REF etudiant_type,
 MEMBER PROCEDURE rendreLivre,
       -- La procédure rendreLivre libère le livre (il est alors attribué à aucun adhérent)
 MEMBER PROCEDURE emprunterLivre (p_num IN VARCHAR),
       -- La procédure emprunteLivre attribue le livre à l'adhérent p_num passé en paramètre
 STATIC FUNCTION nbLivresEmpruntes RETURN NUMBER );
   -- méthode de classe qui retourne le nombre de livres de la bibliothèque qui sont empruntés
```

Création des tables objet

```
CREATE TABLE Etudiant OF etudiant_type
(CONSTRAINT pk_Etudiant PRIMARY KEY (num)) ;
CREATE TABLE Livre OF livre_type
(CONSTRAINT pk_Livre PRIMARY KEY (code)) ;
```

Implémentation du corps des méthodes

- Les fonctions *nbLivres()* et *peutEmprunter()* du type etudiant_type :

La fonction `nbrLivres` retourne un entier (`RETURN NUMBER`) qui correspond au nombre d'emprunts de l'étudiant sur qui on appelle la méthode. Ce nombre est stocké dans la variable locale `v_nb`. On réalise une requête qui calcule le nombre de livres empruntés par l'objet étudiant qui appelle la fonction (on connaît son identité grâce à `SELF`). On stocke le résultat dans `v_nb` (`INTO v_nb`). Puis on retourne ce résultat.

La fonction `peutEmprunter` retourne 1 si l'étudiant peut emprunter un livre ; 0 sinon. Un étudiant peut emprunter un livre s'il possède moins de trois livres. On appelle donc la méthode `nbLivres` sur `SELF`. Si le résultat de cet appel est supérieur ou égal à trois la méthode retourne 0 ; 1 sinon

```
CREATE OR REPLACE TYPE BODY etudiant_type AS
   MEMBER FUNCTION nbLivres RETURN NUMBER IS
   v_nb NUMBER ;
   BEGIN
      SELECT COUNT(*) INTO v_nb
      FROM Livre l
      WHERE l.ref_emprunteur.num = SELF.num;
      RETURN v_nb ;
   END nbLivres ;

   MEMBER FUNCTION peutEmprunter RETURN NUMBER IS
   BEGIN
      IF SELF.nbLivres() >= 3
      THEN RETURN 0 ;
      ELSE RETURN 1 ;
      END IF;
   END peutEmprunter ;
END ;
```

L'appel d'une méthode fonction peut se faire directement sur n'importe quel objet étudiant :

```
SELECT e.nbLivres()
FROM Etudiant e
WHERE e.num = E1 ;
```

- Procédures *rendreLivre()* et *emprunterLivre(…)* du type livre_type :

La procédure `rendreLivre` permet de prendre en compte le fait que le livre sur qui est appelé la méthode n'est plus emprunté. C'est-à-dire que la référence `ref_emprunteur` de ce livre passe à `NULL`. Il suffit d'indiquer que, dans la table livre, la nouvelle valeur de `ref_emprunteur` passe à `NULL` pour le livre dont le code est égal au code de l'objet qui appelle la méthode `(SELF)`.

La procédure `emprunterLivre` permet à l'étudiant passé en paramètre d'emprunter le livre sur lequel la méthode est appelée. Cela n'est possible que si l'étudiant peut emprunter des livres (c'est-à-dire que la méthode `peutEmprunter` appliquée à cet étudiant retourne 1). Dans le cas contraire, cette méthode ne fait rien.

```
CREATE OR REPLACE TYPE BODY livre_type AS
   MEMBER PROCEDURE rendreLivre IS
   BEGIN
      UPDATE Livre l
      SET l.ref_emprunteur = NULL
      WHERE l.code = SELF.code;
   END rendreLivre ;
   MEMBER PROCEDURE emprunterLivre (p_num IN VARCHAR) IS
   v_possible NUMBER;
   BEGIN
      SELECT e.peutEmprunter() INTO v_possible
      FROM Etudiant e
      WHERE e.num = p_num;
      IF (v_possible = 1) THEN
         UPDATE Livre l
         SET l.ref_emprunteur = (SELECT REF(e) FROM Etudiant e
                                 WHERE e.num = p_num)
         WHERE l.code = SELF.code;
      END IF;
   END emprunterLivre;
   …
END ;
```

L'appel d'une méthode procédure peut se faire en deux temps dans un bloc PL/SQL. D'abord on recherche l'objet livre sur lequel on veut appliquer la méthode (grâce à la fonction `VALUE(alias)`). Ensuite on appelle la méthode sur cette variable.

```
DECLARE
      v_livre livre_type ;
BEGIN
      SELECT VALUE(l) INTO v_livre
      FROM Livre l
      WHERE l.code = 'L2' ;
      v_livre.rendreLivre() ;
END ;
```

- Méthode de classe (`STATIC`) *nbLivresEmpruntes()* du type livre_type :

La méthode de classe `nbLivresEmpruntes` retourne le nombre de livres de la table *Livre* qui sont actuellement empruntés.

```
CREATE OR REPLACE TYPE BODY livre_type AS
   …
   STATIC FUNCTION nbLivresEmpruntes RETURN NUMBER IS
   v_nb NUMBER ;
   BEGIN
      SELECT COUNT(*) INTO v_nb
      FROM Livre l
      WHERE l.ref_emprunteur IS NOT NULL;
      RETURN v_nb ;
   END nbLivresEmpruntes ;
END ;
```

L'appel à une méthode de classe se fait en préfixant la méthode du nom de son TAD

```
SELECT livre_type.nbLivresEmpruntes()
FROM DUAL ;
```

5 Héritage

Il est possible de spécialiser un type que si celui-ci a été déclaré avec la directive `NOT FINAL`. Attention, si rien n'est spécifié, un type est par défaut déclaré comme `FINAL`. La directive `NOT INSTANTIABLE` indique qu'un type ne possède pas de constructeur (notion proche de celle de classe abstraite). Par défaut, un type est déclaré `INSTANTIABLE`.
Enfin, il est possible de redéfinir une méthode grâce au mot clé `OVERRIDING`

exemple : prenons le cas de notre bibliothèque où nous voulons aussi gérer des emprunteurs qui sont des profs. Un prof peut être défini comme un étudiant sauf qu'il possède en plus un salaire. De plus, un Prof peut emprunter autant de livres qu'il le souhaite (il n'est pas limité à 3 emprunts).

Déclaration des types et des méthodes

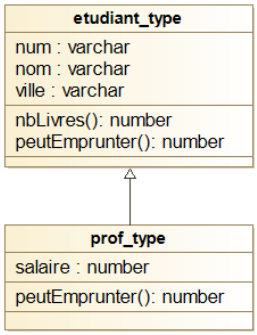
On crée un type `prof_type` qui hérite de `etudiant_type`. Dans `prof_type` on rajoute le salaire du prof et on redéfinit la méthode `peutEmprunter`. (il est à noter que cette conception peut paraître un peu douteuse car elle ne respecte pas le principe de Liskov – cf les principes SOLID).

```
CREATE OR REPLACE TYPE etudiant_type AS OBJECT
(num VARCHAR(5), nom VARCHAR(20), ville VARCHAR(20),
 MEMBER FUNCTION nbLivres RETURN NUMBER,
 MEMBER FUNCTION peutEmprunter RETURN NUMBER) NOT FINAL INSTANTIABLE ;

CREATE OR REPLACE TYPE prof_type UNDER etudiant_type
(salaire NUMBER,
 OVERRIDING MEMBER FUNCTION peutEmprunter RETURN NUMBER) FINAL INSTANTIABLE ;
```

Création des tables objet

```
CREATE TABLE Etudiant OF etudiant_type
(CONSTRAINT pk_Etudiant PRIMARY KEY (num)) ;
CREATE TABLE Prof OF prof_type
(CONSTRAINT pk_Prof PRIMARY KEY (num)) ;
```



Implémentation du corps des méthodes

```
CREATE OR REPLACE TYPE BODY prof_type AS
    OVERRIDING MEMBER FUNCTION peutEmprunter RETURN NUMBER IS
    BEGIN
      RETURN 1;
    END peutEmprunter ;
END ;
```

Il est à noter que si dans le code de la méthode on a besoin d'appeler une méthode d'une super-classe sur l'objet `SELF` (super.nomMethode() en Java), cela est possible en écrivant :

```
(SELF AS super_type).nomMethode()
```

Insertion d'objets

Il est ensuite possible d'insérer des enseignants dans la table *Prof*. On peut bien sûr aussi insérer à la fois des étudiants et des enseignants dans la table *Etudiant* (pour insérer des enseignants dans la table *Etudiant* il faut obligatoirement utiliser le constructeur `prof_type`).

```
INSERT INTO Etudiant VALUES ('E1', 'Titouplin', 'Montpellier');
INSERT INTO Prof VALUES ('P2', 'Palleja', 'Montpellier', 4800);
INSERT INTO Etudiant VALUES (etudiant_type('E3', 'Gator', 'Sête'));
INSERT INTO Etudiant VALUES (prof_type('E4', 'Deuf', 'Montpellier', 4200));
```

Requêtes d'interrogation dans la table Etudiant

On peut appeler la méthode `peutEmprunter()` sur tous les objets de la table *Etudiants* ; le polymorphisme est assuré. Pour afficher *tous* les attributs des différents objets de cette table, on peut utiliser la fonction **`VALUE(alias)`** qui renvoie un contenu sous la forme d'un type.

```
SELECT e.nom, e.peutEmprunter()
FROM Etudiant e;
```

```
SELECT VALUE(e)
FROM Etudiant e;
```

La fonction **`IS OF(nomType)`** permet de tester le type d'un objet. La directive `ONLY` permet de spécifier un niveau précis dans la hiérarchie des types.

```
SELECT *
FROM Etudiant e
WHERE VALUE(e) IS OF (prof_type);
```

```
SELECT *
FROM Etudiant e
WHERE VALUE(e) IS OF (ONLY etudiant_type);
```

La fonction **`TREAT(nomObjet AS nomType)`** permet de caster un objet (pour par exemple, pouvoir utiliser certaines méthodes d'un sous-type)

```
SELECT e.nom, TREAT(VALUE(e) AS prof_type).salaire
FROM Etudiant e
WHERE VALUE(e) IS OF (prof_type);
```