Chapitre 3 : Oracle 8 et l'objet

Maîtrise informatique
F. Bannay

December 11, 2001

Contents

1 Introduction **3**

2 Les types de base **3**

3 Les objets **3**
3.1 Déclaration d'un type objet 3
3.2 Utilisation par une table objet-relationnelle 4
3.2.1 Déclaration d'une table objet-relationnelle 4
3.2.2 Insertion dans une table objet-relationnel 4
3.2.3 Interrogation 5
3.3 Utilisation d'un type objet dans un autre type objet ou dans une table relationnelle 5
3.3.1 Déclaration : 5
3.3.2 Insertion dans une table qui utilise un type 6
3.3.3 Interrogation 6
3.4 Utilisation d'un type objet par l'intermédiaire de pointeurs 6
3.4.1 Déclaration 6
3.4.2 Insertion 7
3.4.3 Interrogation 7
3.4.4 Types incomplets 8

4 Comment gérer l'héritage? **9**

5 Les collections imbriquées **9**
5.1 Les tables imbriquées 9
5.1.1 Déclaration 10
5.1.2 Utilisation 10
5.1.3 Insertion 10
5.1.4 Interrogation 11

5.2 Les tableaux fixes 12
5.2.1 Déclaration 12
5.2.2 Insertion 12
5.2.3 Interrogation 13
5.3 Manipulation des collections en PL/SQL 13
5.3.1 Accès à un élément par son indice 13
5.3.2 Méthodes prédéfinies 14

6 Les méthodes **15**
6.1 Les fonctions 15
6.1.1 Création 15
6.1.2 Utilisation et protection de la BD 15
6.2 Procedure 16
6.2.1 Création 16
6.2.2 Utilisation 17
6.2.3 Surcharge 17
6.3 Les fonctions MAP et ORDER 18
6.3.1 La fonction de type MAP 18
6.3.2 La fonction de type ORDER 19

7 Nouvelles vues du dictionnaire de données **19**

8 Nouveauté du SELECT **20**

1 Introduction

Face à la montée de la philosophie objet, Oracle a réagit en proposant des fonctionnalités objets. Ces fonctionnalités objets sont évidemment une extension d'Oracle, mais l'énorme machinerie a un peu de mal a suivre. Les extensions proposées sont assez lourdes à mettre en œuvre par un utilisateur...
Une des grosses lacunes d'Oracle vis-à-vis de l'objet est la non gestion de l'héritage. Cependant, Oracle propose la définition de type utilisateurs, l'implémentation de méthodes, la gestion de collections imbriquées (assez lourde).
O2 était un système de base de données orienté objet, c'est-à-dire qu'il est basé sur un langage objet que l'on a étendu pour obtenir des fonctionnalités BD (la concurrence d'accès, la persistance des données, la gestion des pannes).
Ici, on est dans le cas inverse : on part d'un système de gestion de bases de données et on lui ajoute des fonctionnalités objet. Déjà on change de forme normale : on passe en NF2 (non first normal form), c'est-à-dire que les attributs des tables ne sont pas forcément de type atomique, mais on peut admettre des objets ou des collections d'objet (en fait ce sont des tables d'objets). Pour Oracle les collections imbriquées sont des *tables imbriquées*.

2 Les types de base

Sous Oracle, en SQL2, on avait accès (seulement) aux types suivants :

CHAR(taille) : chaîne de caractères (taille : défaut 1, max 255) (ex: 'toto')

VARCHAR(taille) : chaîne de caractères de longueur variable (1, 2000)

LONG : idem mais jusqu'à 2 gigaoctets

RAW et LONG RAW : données binaires (taille idem que varchar et long)

NUMBER(p, s) : nombre total de chiffres (précision) p(1 à 38) et nombre de chiffre après la virgule (échelle) s (-84 à 127)

ROWID : valeurs de la pseudo-colonne ROWID (adresse physique de chaque enregistrement)

DATE : date entre le 1er janvier -4712 et le 31 décembre 4712

Remarque le type BOOLEAN n'est pas un type SQL, mais c'est un type PL/SQL.

3 Les objets

3.1 Déclaration d'un type objet

Avec le nouvel SQL3, l'utilisateur peut définir ses propres types, qui seront ensuite utilisable dans les tables comme des types standards. Ces types sont appelés types objets car - ils

peuvent être associés à des méthodes, - les éléments des tables basées sur ces types sont accessibles par leurs OID, - ces types objets peuvent avoir des attributs qui sont à leur tour des types objets et ainsi de suite. Ils permettent l'exécution de requêtes parcourant des chemins d'objets (du genre x.pays.capitale.nom).
Exemple de syntaxe de la création d'un type objet :

```
CREATE TYPE type_adresse AS OBJECT
(norue NUMBER,
rue VARCHAR(20),
ville VARCHAR(30),
codepostal VARCHAR(10)
)
/
```

On pourra ensuite utiliser le type objet :

- soit pour déclarer une collection d'objets : appelée table objet-relationnelle
- soit dans un autre type objet
- soit dans une table relationnelle

Cette déclaration permettra d'utiliser le *constructeur de type associé*, afin de créer un objet de ce type.
Ex : `type_adresse(17,'rue des roses','Toulouse','31500')` devient une nouvelle valeur que l'on pourra insérer dans une colonne de type type_adresse.

3.2 Utilisation par une table objet-relationnelle

3.2.1 Déclaration d'une table objet-relationnelle

Définition 3.1 *Une table* objet-relationnelle *est une table qui contient des éléments ayant un type-objet. Les éléments de ces tables sont des objets, ils pourront être référencés grâce à leurs OID.*

Ex: Si l'on veut créer une collection d'objet de type adresse, on peut utiliser une table *objet-relationnelle.*

```
CREATE TABLE mesadresses OF type_adresse;
```

Des contraintes peuvent être posées sur les tables.
Ex:

```
CREATE TABLE mesadresses OF type_adresse
(ville DEFAULT 'Angers' CHECK (ville IN('Toulouse','Angers','Paris')),
CONSTRAINT pkadr PRIMARY KEY(norue,rue,codepostal));
```

3.2.2 Insertion dans une table objet-relationnel

L'insertion peut se faire simplement par les valeurs des attributs de l'objet, quand ils sont de type simple, ou encore en utilisant le constructeur du type.

Ex : `INSERT INTO mesadresses VALUES (17, 'rue des roses', 'Toulouse', '31500');`
cette instruction est équivalente à
`INSERT INTO mesadresses VALUES (type_adresse(17, 'rue des roses', 'Toulouse', '31500'));`

3.2.3 Interrogation

Accès aux OID Les élements de cette table sont des objets et ont des OID qu'il est possible de récupérer avec la fonction REF :
Ex : `SELECT REF(m) FROM mesadresses m;` (ça donne un code superlong fait de chiffres et de nombres)

Accès aux valeurs des attributs des objets Les valeurs des attributs des objets sont accessibles directement comme si la table contenait directement les colonnes correspondantes :
Ex : `SELECT * FROM mesadresses;`
identique à `SELECT m.norue, m.rue, m.ville, m.codepostal FROM mesadresses m;`

NORUE	RUE	VILLE	CODEPOSTAL
17	rue des roses	Toulouse	31500
2	bd Foch	Angers	49100

accès aux objets On peut accéder aux "vraies valeurs" des éléments de la table, i.e, aux objets :
Ex: `SELECT VALUE(m) FROM mesadresses m;`

VALUE(m)(NORUE,RUE, VILLE,CODEPOSTAL
TYPE_ADRESSE(17,'rue des roses','Toulouse','31500')
TYPE_ADRESSE(2,'bd Foch', 'Angers','49100')

Remarque : on ne peut jamais utiliser le surnom de la table tout seul, par exemple, `SELECT m FROM mesadresses m;` est interdit.

3.3 Utilisation d'un type objet dans un autre type objet ou dans une table relationnelle

3.3.1 Déclaration :

L'utilisation d'un type objet se fait comme l'utilisation des types standards.

- Ex : utilisation par un autre type-objet
```
CREATE TYPE type_etudiant AS OBJECT
(numero NUMBER,
bac VARCHAR(5),
adr type_adresse)
/
```

- Ex : utilisation dans une table relationnelle
```
CREATE TABLE lesemployes
(numero NUMBER,
dept VARCHAR(5),
salaire NUMBER,
adr type_adresse);
```

3.3.2 Insertion dans une table qui utilise un type

Il faut utiliser le constructeur du type afin d'insérer toutes les valeurs dudit type.
Ex: `INSERT INTO lesemployes VALUES(1,'INFO',12000, type_adresse(2,'rue des capucines','Angers','49000'));`

3.3.3 Interrogation

Accès à tous les champs : Ex: `SELECT * FROM lesemployes;`

NUMERO	DEPT	SALAIRE	ADR(NORUE, RUE, VILLE, CODEPOSTAL)
1	INFO	12000	TYPE_ADRESSE(2, 'rue des capucines', 'Angers','49000')

Accès à un champs :

- `SELECT e.adr.norue FROM lesemployes e;`

ADR.NORUE
2

- `SELECT e.adr FROM lesemployes e;`

ADR(NORUE, RUE, VILLE, CODEPOSTAL)
TYPE_ADRESSE(2, 'rue des capucines', 'Angers', '49000')

Attention, les valeurs des types utilisés n'ont pas d'OID car ils n'appartiennent pas à des tables objets-relationnels. On ne peut donc pas utiliser REF ou VALUE (REF(e.adr) ou VALUE(e.adr) n'existe pas).
De plus, si l'on veut réutiliser des objets, il vaut mieux qu'ils soient présents dans une seule table et que dans les autres tables, on ne garde que leurs OID et pas leurs valeurs entières. C'est pour cela que l'on peut déclarer des types ou des tables qui utilisent non pas des types utilisateurs mais des pointeurs vers des objets de ces types.

3.4 Utilisation d'un type objet par l'intermédiaire de pointeurs

3.4.1 Déclaration

Si l'on a déclaré un type objet, il est possible de créer des tables qui contiennent non pas l'objet tout entier mais une référence à cet objet (son OID), lors de l'interrogation on pourra naviguer de la même façon que si l'objet était physiquement présent, mais cela prendra moins de place au stockage et surtout cela évitera les redondances sources d'erreurs!

L'utilisation d'un pointeur permet de représenter des associations 1-1 (comme avec une clef étrangère). Cette référence à un objet se fait grâce au mot clé REF.

ex :
```
CREATE TYPE type_ville AS OBJECT
(nom VARCHAR(30),
population NUMBER
)
/
CREATE TABLE Lespays
(nom VARCHAR(30),
capitale REF type_ville,
population NUMBER);
```

ou si l'on veut créer des objets pays :
```
CREATE TYPE type_pays AS OBJECT
(nom VARCHAR(30),
capitale REF type_ville,
population NUMBER)
/
CREATE TABLE Lespays OF type_pays;
```

3.4.2 Insertion

La valeur par défaut d'un pointeur est NULL.

Ex :
```
CREATE TABLE Lesvilles OF type_ville;
INSERT INTO Lesvilles VALUES('Paris',2 000 000);
INSERT INTO Lesvilles VALUES('Rome',2 700 000);
INSERT INTO Lespays(nom,population) VALUES('France',58 700 000);
```
identique à `INSERT INTO Lespays VALUES('France',NULL, 58 700 000);`

Si l'on veut insérer la référence à l'objet qui représente la ville de Paris, il faut utiliser un select dans Lesvilles pour l'obtenir :
```
UPDATE Lespays SET capitale=(SELECT REF(v) FROM Lesvilles v WHERE v.nom='Paris')
WHERE nom='France';
```
Tout ceci aurait pu être fait directement ainsi :
```
INSERT INTO Lespays SELECT 'Italie', REF(v), 57 200 000 FROM Lesvilles v WHERE
v.nom='Rome';
```

3.4.3 Interrogation

La sélection des objets dont seule la référence est dans la table se fait comme si l'objet était entièrement dans la table.

Ex : `SELECT * FROM Lespays;`

NOM	CAPITALE	POPULATION
France	(...)80020AC063B72182E91884A56C6E034080020AC063B	58 700 000
Italie	(...)80020AC063C72182E91884A56C6E034080020AC063B	57 200 000

```
SELECT p.nom, p.population, p.capitale.nom FROM lespays p;
```

NOM	POPULATION	CAPITALE.NOM
France	58700000	Paris
Italie	57200000	Rome

La fonction DEREF renvoie l'objet (REF renvoie son adresse) à partir de son adresse (à partir de l'objet).

```
SELECT DEREF (p.capitale) FROM Lespays p; => type_ville(...)
SELECT VALUE(v) FROM Lesvilles v; => type_ville(...)
SELECT REF(v) FROM Lesvilles v; => OID
SELECT DEREF(REF(v)) FROM Lesvilles v; identique à VALUE(v)
```

3.4.4 Types incomplets

Un objet peut référencer un objet du même type que lui.
Ex:`CREATE TYPE type_employe AS OBJECT`

```
(nom VARCHAR(30),
prenom VARCHAR(30),
superieur REF type_employe,
salaire NUMBER)
/
```

Mais, on ne peut déclarer un pointeur vers un objet d'un certain type, que si ce type est déclaré. Comment faire si l'on veut que les villes pointent vers les pays et les pays vers les villes? Utiliser un type incomplet.
Ex: `DROP TABLE Lespays; DROP TABLE Lesvilles;` (on le fait en premier car ces tables utilisent le type type_ville) `DROP TYPE type_ville;`

```
CREATE TYPE type_ville AS OBJECT (nom VARCHAR(30),
population NUMBER, pays REF type_pays
)
/
```

Warning: Type created with compilation errors.
SQL> show errors
Errors for TYPE TYPE_VILLE:
LINE/COL ERROR
—— ————————————————————————

0/0 PL/SQL: Compilation unit analysis terminated
4/6 PLS-00201: identifier 'TYPE_PAYS' must be declared

```
CREATE TYPE type_pays AS OBJECT
(nom VARCHAR(30),
capitale REF type_ville,
population NUMBER)
/
```

Le type est créé, il faut ensuite relancer la création du type_ville.

Si l'on veut faire les choses correctement, il faut d'abord déclarer le type_pays :`CREATE TYPE type_pays;` puis créer le type ville et enfin le type pays.
Maintenant les types sont interdépendants pour les détruire il faut forcer la destruction (auparavant, il faudra détruires les tables qui dépendent de ces types) : `DROP TYPE type_pays FORCE; DROP TYPE type_ville FORCE;`

4 Comment gérer l'héritage?

Puisqu'Oracle ne gère pas l'héritage, il faut le faire manuellement en utilisant une association 1-1. C'est-à-dire soit au moyen d'une clef étrangère soit avec un pointeur comme dans l'exemple ci-dessous.
Ex :

```
CREATE TYPE type_personne AS OBJECT
(nom VARCHAR(20),
prenom VARCHAR(20),
datenais DATE)
/
CREATE TYPE etudiant AS OBJECT
(tp REF type_personne,
netu NUMBER,
affectation VARCHAR(30))
/
```

Le champ tp sert à simuler l'héritage. Ainsi, à partir d'un étudiant on peut accéder à son nom, son prénom et sa date de naissance. Si l'on désire connaître les nom, prénom, numéro d'étudiant et affectation des étudiants, on peut ensuite exécuter la requête suivante. `SELECT e.tp.nom, e.tp.prenom, e.tp.datenais, e.netu, e.affectation FROM Lesetudiants;`

5 Les collections imbriquées

Il y a deux types de collection imbriquée sous Oracle 8, les tables imbriquées (nested tables) et les tableaux fixes (VARRAYs). Ces deux collections doivent contenir des éléments de mêmes types (elles doivent être homogènes).
La gestion des collections imbriquées permet de représenter des associations 1-N.

5.1 Les tables imbriquées

Définition 5.1 *Une table imbriquée est une collection* illimitée, non ordonnée *d'éléments de même type.*

Sous Oracle, un seul niveau d'imbrication est permis dans les tables imbriquées.

5.1.1 Déclaration

On utilise l'instruction TABLE OF.

Ex:
```
CREATE TYPE type_employe AS OBJECT
(ninsee VARCHAR(13), age NUMBER, nom VARCHAR(30))
/
CREATE TYPE type_employes AS TABLE OF type_employe
/
```

5.1.2 Utilisation

Elle peut s'utiliser dans des tables relationnelles ou objet-relationnelles.

Ex:
```
CREATE TABLE Lesdepartements
(numdep VARCHAR(11), budget NUMBER, emps type_employes)
NESTED TABLE emps STORE AS tabemp;
```
ou encore
```
CREATE TYPE departement AS OBJECT
(numdep VARCHAR(11), budget NUMBER, emps type_employes)
/
CREATE TABLE Lesdepartements OF departement
NESTED TABLE emps STORE AS tabemp;
```

L'instruction NESTED TABLE sert à préciser que la table mère (ici, Lesdepartements) contient une collection et dans quel attribut (ici, emps), le STORE AS est obligatoire et exige un nom de stockage pour la table imbriquée. Ce nom ne sera jamais utilisé par ailleurs, il est juste nécessaire à la création de la table mais n'apparaît jamais dans les requêtes.

5.1.3 Insertion

Utilisation du constructeur de type de la table imbriquée La déclaration d'une table imbriquée définit automatiquement un constructeur de type associé. Ce constructeur de type permet d'insérer les valeurs une à une.

Ex :
```
INSERT INTO Lesdepartements VALUES
('informatique', 100 000, type_employes(type_employe('23456789012', 32,'Titi'),
type_employe('112345666',20,'Tutu')));
INSERT INTO Lesdepartements VALUES ('mathématique',200 000,type_employes( ));
INSERT INTO Lesdepartements(numdep,budget) VALUES('physique', 200 000);
```
Attention, la dernière instruction, ne crée pas de place pour la table imbriquée des employés, qui ne pourront pas être insérés par la suite!

NUMDEP	BUDGET	EMPS(NINSEE, AGE, NOM)
informatik	100000	type_employes(type_employe('23456789012', 32, 'Titi'), type_employe('112345666', 20, 'Tutu'))
mathimatik	200000	type_employes()
physique	200000	

Utilisation de THE Si une table imbriquée a été créé pour un certain tuple, on peut y insérer des éléments en sélectionnant la table imbriquée que l'on veut mettre à jour au moyen de THE.
Ex : `INSERT INTO THE (SELECT emps FROM Lesdepartements`
`WHERE numdep='informatik') VALUES('1234556667',34,'Tata');`
`INSERT INTO THE (SELECT emps FROM Lesdepartements WHERE numdep='mathématik')`
`VALUES('1234556668',48,'Bidule');`
`INSERT INTO THE (SELECT emps FROM Lesdepartements WHERE numdep='physique')`
`VALUES('1234556669',23,'Machin');`
ERROR at line 1:
ORA-22908: reference to NULL table value

Les deux premières insertions ajoutent les employés, la troisième ne fonctionne pas car la table imbriquée des employés du département de physique n'existe pas. On peut la créer avec un UPDATE :
`UPDATE Lesdepartements SET emps=type_employes() WHERE numdep='physique';`
Ensuite les insertions fonctionneront.

5.1.4 Interrogation

Le champ correspondant à la table imbriquée sera affiché grâce à son constructeur de type:
Ex: `SELECT * FROM Lesdepartements;` donne le tableau du paragraphe 5.1.3.
Pour sélectionner dans une table imbriquée, il faut utiliser THE suivi d'une requête renvoyant le champs table imbriqué, cette expression `THE (SELECT ... WHERE ...)` s'utilise comme une table standard :
Ex: `SELECT nom, age FROM THE (SELECT emps FROM Lesdepartements`
`WHERE numdep='informatik');`
Attention : la requête qui suit le THE ne doit renvoyer qu'une seule table imbriquée, il faut donc ABSOLUMENT un WHERE qui désigne le tuple pour lequel on veut la table imbriquée.
Limite : On ne peut pas faire un WHERE qui concernerait successivement chacune des tables imbriquées :
`SELECT e.nom, e.age, d.numdep FROM lesdepartements d, THE (SELECT d1.emps FROM`
`lesdepartements d1 WHERE d1.numdep=d.numdep) e;`
ERROR at line 2:
ORA-03001: unimplemented feature
Pour exécuter ce genre de requête il faut utiliser la commande CURSOR :
`SELECT d.numdep, CURSOR(select e.nom, e.age FROM TABLE(emps) e)`
`FROM lesdepartements d;`
Pb: l'affichage est très moche !!
Remarque : le `THE(SELECT ... WHERE ...)` jouant le rôle d'une nouvelle table, il peut aussi(en plus du INSERT et du SELECT vus précédemment) être utilisé dans un DELETE ou un UPDATE :

Ex : `DELETE FROM THE (SELECT emps FROM Lesdepartements WHERE numdep='informatik')`
`WHERE nom='Tata';`
`UPDATE THE (SELECT emps FROM Lesdepartements WHERE numdep='physique')`
`SET nom='machin'` /*modifie tous les tuples de la table imbriquée*/

5.2 Les tableaux fixes

Définition 5.2 *Un tableau fixe (VARRAY) est une collection* limitée, ordonnée *d'éléments de même type.*

- *Un tableau fixe (VARRAY) permet plusieurs niveaux d'imbrication,*
- *Les différents éléments du VARRAY ne sont pas accessibles en SQL mais en PL/SQL (il n'y a pas d'équivalent du "THE").*

5.2.1 Déclaration

On doit déclarer d'emblée combien de tuples maximum contiendra le tableau fixe.
Ex: `CREATE TYPE type_vadresses AS VARRAY(2) OF type_adresse` /
Un tableau fixe peut-être utilisé dans une table relationnelle ou objet-relationnelle.
Ex:
```
CREATE TABLE Lesetudiants
(netu VARCHAR(15),
nom VARCHAR(11),
adrs type_vadresses);
```
Les manipulations sur le VARRAY en entier se font comme si c'était un attribut simple grâce au constructeur de type.

5.2.2 Insertion

Utilisation du constructeur de type soit sans paramètres, soit avec le bon nombre de paramètres (NULL si besoin).
ex :
```
INSERT INTO Lesetudiants VALUES('12345', 'pouet', type_vadresses());
INSERT INTO Lesetudiants VALUES ('12346', 'tuut', type_vadresses (type_adresse
(15, 'rue toto', 'Angers', 49100), NULL));
```

Remarque : On peut modifier/effacer *tout* un VARRAY grâce à la commande UPDATE (puisque l'attribut VARRAY se manipule comme un attribut simple.
Ex:
```
UPDATE Lesetudiants
SET adrs= type_vadresses( type_adresse(16, 'rue pouet', 'Angers', 49100), NULL))
WHERE numetu='12346';
```
Pour modifier des valeurs dans un VARRAY existant, il faut utiliser un bloc PL/SQL car on ne dispose pas du THE.
Ex:
```
DECLARE
lesadr type_vadresses;
```

```
BEGIN
SELECT adrs INTO lesadr FROM lesetudiants WHERE netu='12346';
lesadr(2):=type_adresse(123,'av Gallieni','Les ponts de Cé',49300);
UPDATE Lesetudiants SET adrs=lesadr WHERE netu='12346';
END;
.
/
```

Cette commande n'aurait pas marché avec l'autre étudiant(ERROR at line 5: ORA-06533: Subscript beyond count) car son VARRAY est déclaré vide au départ, deux solutions :

- il faudrait d'abord le déclarer avec deux valeurs (éventuellement vides) pour pouvoir le mettre à jour ensuite.
- soit utiliser la méthode extend cf. paragraphe 5.3.2

5.2.3 Interrogation

Le champs correspondant au VARRAY sera affiché grâce à son constructeur de champs (c'est la totalité de la collection qui sera affichée):
Ex: `SELECT * FROM etudiants;`

NETU	NOM	ADRS(NORUE, RUE, VILLE, CODEPOSTAL)
12345	pouet	type_vadresses()
12346	tuut	type_vadresses(type_adresse(15, 'rue toto', 'Angers', '49100'), NULL)

Il n'est pas possible d'accéder à des attributs du VARRAY.

5.3 Manipulation des collections en PL/SQL

5.3.1 Accès à un élément par son indice

Pour les deux types de collections on peut accéder aux éléments par leurs indices (bien que la table imbriquée soit théoriquement non ordonnée), en écrivant le nom de la collection suivi de l'indice entre parenthèses.
ex: Remplacement du deuxième employé en informatique.

```
DECLARE
lesemps type_employes;
BEGIN
SELECT emps INTO lesemps FROM lesdepartements WHERE numdep='informatique';
lesemps(2):=type_employe('123456',12,'trucmuche');
UPDATE lesdepartements SET emps=lesemps WHERE numdep='informatique';
END;
.
/
```

5.3.2 Méthodes prédéfinies

Les méthodes suivantes sont disponibles et fonctionnent sur les tables imbriquées ou les tableaux fixes (sauf LIMIT). Elles ne peuvent être utilisées que sur une variable d'un programme PL/SQL (jamais à l'intérieur d'une requête).

- EXISTS(x) : TRUE si le xième élément existe
- COUNT : nombre d'éléments de la collection
- LIMIT : nombre maxi d'éléments d'un VARRAY
- FIRST/LAST : indices du premier/dernier élément de la collection
- PRIOR(x)/NEXT(x) : indice de l'élément juste avant/après x
- TRIM(x) : supprime x éléments à partir de la fin
- EXTEND[(x)] ajoute un/x élément NULL
- EXTEND(x,y) ajoute x copies du yième élément
- DELETE supprime toute la collection
- DELETE(x[,y]) supprime le xième/ les éléments de x à y

Exemple : l'ajout d'un élément à la table imbriquée des employés du département de physique peut se faire en PL/SQL.

```
DECLARE
lesemps type_employes;
BEGIN
SELECT emps INTO lesemps FROM lesdepartements WHERE numdep='physique';
lesemps(2):=type_employe('123456',12,'trucmuche');
UPDATE lesdepartements SET emps=lesemps WHERE numdep='physique';
END;
.
/
```

Ceci provoquera une erreur(ERROR at line 5: ORA-06533: Subscript beyond count) car la collection contient moins de 2 éléments. Il faut d'abord ajouter un élément vide puis on peut le modifier.

```
DECLARE
lesemps type_employes;
BEGIN
SELECT emps INTO lesemps FROM lesdepartements WHERE numdep='physique';
lesemps.extend;
lesemps(lesemps.last):=type_employe('123456',12,'trucmuche');
```

```
UPDATE lesdepartements SET emps=lesemps WHERE numdep='physique';
END;
.
/
```

Remarque : les fonctions VALUE, REF et DEREF ne peuvent pas s'utiliser hors des requêtes SQL, si on veut les utiliser dans un programme PL/SQL, il faut qu'elles soient incluses dans un SELECT du programme.
Notons que la définition de méthodes sur les collection est plus objet que ce qui est fait dans le système O_2, puisque dans le système O_2 on ne dispose que de fonctions applicables à des collections mais pas de méthodes : nombre d'éléments d'une collection O_2 : *count(coll)* alors qu'en PL/SQL *coll.count*.

6 Les méthodes

Les types objets peuvent admettre des méthodes qui sont soit des fonctions soit des procédures.
La déclaration de toutes les méthodes peut se faire au moment de la déclaration du type. Si l'on veut ajouter des méthodes à un type objet il faut utiliser l'instruction ALTER TYPE.
Le corps des méthodes peut faire référence à l'objet SELF de type le type objet et qui représente l'objet sur lequel on effectue la méthode.

6.1 Les fonctions

6.1.1 Création

La création d'une fonction se fait en deux temps : la spécification puis l'implémentation.
Ex:

```
CREATE TYPE type_personne AS OBJECT
(nom VARCHAR(10),
datenais DATE,
MEMBER FUNCTION age RETURN NUMBER)
/
CREATE TYPE BODY type_personne IS
MEMBER FUNCTION age RETURN number IS n NUMBER; /* déclaration des paramètres*/
BEGIN
n:=TRUNC((SYSDATE-SELF.datenais)/365);
RETURN n;
END age;
END;
/
```

6.1.2 Utilisation et protection de la BD

Une fonction peut s'utiliser dans une requête ou dans un programme PL/SQL :

Ex : `SELECT p.age() FROM Lespersonnes p;`
Le danger, c'est que la méthode puisse modifier la base de données. Oracle déclenchera une erreur(ERROR at line 1: ORA-06571: Function AGE does not guarantee not to update database) si on lance le SELECT précédent parce que lorsque l'on a défini la méthode on n'a pas dit qu'elle n'allait rien modifier.
Il faudra préciser que la méthode de change rien dans la BD, on peut ajouter

- WNDS : pas de modif de la BD
- RNDS : pas de lecture de la BD

Ex:
```
CREATE TYPE type_personne AS OBJECT
(nom VARCHAR(10),datenais DATE,
MEMBER FUNCTION age RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(age, WNDS))
/
```

6.2 Procedure

6.2.1 Création

Les procédures se créent de la même façon que les fonctions :
Ex:
```
ALTER TYPE type_personne REPLACE AS OBJECT
(nom VARCHAR(10),
datenais DATE,
MEMBER FUNCTION age RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(age, WNDS),
MEMBER PROCEDURE modif_datenais (nouvdate IN DATE))
/
```
On est obligé de réécrire *tout* le type dans le ALTER TYPE ou dans le BODY quand on veut y ajouter une procédure ou une fonction.
Ex:
```
CREATE OR REPLACE TYPE BODY type_personne IS
MEMBER FUNCTION age RETURN number IS
n NUMBER; /* déclaration des paramètres*/
BEGIN
SELECT TRUNC((SYSDATE-SELF.datenais)/365) INTO n
FROM DUAL;
RETURN n;
END age;
MEMBER PROCEDURE modif_datenais (nouvdate IN DATE) IS
/* déclaration des paramètres */
BEGIN
UPDATE lespersonnes p SET datenais=nouvdate WHERE VALUE(p)=SELF;
END modif_datenais;
```

```
END;
/
```

6.2.2 Utilisation

L'appel d'une procédure peut se faire dans un programme PL/SQL mais pas dans une requête.

Ex:
```
DECLARE
toto type_personne;
BEGIN
select VALUE(p) into toto
FROM lespersonnes p
WHERE nom='Dupont';
toto.modif_datenais('8-DEC-1967');
END;
.
/
```

6.2.3 Surcharge

Il peut y avoir un polymorphisme sur le nombre et le type des paramètres :

Ex :
```
CREATE TYPE type_toto AS OBJECT
(x NUMBER,
y VARCHAR(5),
d DATE,
MEMBER PROCEDURE modif(nouvy IN VARCHAR), /* types non contraints !!*/
MEMBER PROCEDURE modif(nouvd IN DATE),
MEMBER PROCEDURE modif (nouvd IN DATE, nouvx IN NUMBER))
/
CREATE OR REPLACE TYPE BODY type_toto is
MEMBER PROCEDURE modif(nouvy IN VARCHAR) is /* types non contraints !!*/
BEGIN
self.y:=SUBSTR(nouvy,1,5);
end modif;
MEMBER PROCEDURE modif(nouvd IN DATE) is
begin
self.d:=nouvd;
end modif;
MEMBER PROCEDURE modif (nouvd IN DATE, nouvx IN NUMBER) is
begin
self.d:=nouvd;
self.x:=nouvx;
end modif;
```

```
end;
/
CREATE TABLE lestotos OF type_toto;
DECLARE
x type_toto:=type_toto(1,'youpi','21-FEB-2000');
BEGIN
x.modif('LALA');
x.modif(TO_DATE('3/12/00','dd/mm/yy')); /* si on utilise pas le todate il prend
ça comme un VARCHAR*/
x.modif('4-DEC-00',2);
INSERT INTO lestotos VALUES(x);
END;
.
/
```

Résultat : 2 LALA 04-DEC-00

6.3 Les fonctions MAP et ORDER

Un type peut contenir soit une fonction MAP soit une fonction ORDER soit ni l'une ni l'autre.
Ces fonctions sont utiles pour créer de nouveaux types ordonnés, elles fournissent des opérateurs implicites de comparaison.

6.3.1 La fonction de type MAP

La fonction de type MAP doit renvoyer un résultat d'un type scalaire standard comme DATE, NUMBER, VARCHAR, CHAR ou REAL et ne doit admettre aucun argument.
Ex : on peut créer le type rationnel et avoir une fonction MAP qui associe à un objet de ce type le réel qui lui correspond, ce réel sera utilisé pour les comparaisons entre rationnels.

```
CREATE TYPE type_rationnel AS OBJECT
(numerateur INTEGER, denominateur INTEGER,
MAP MEMBER FUNCTION conversion RETURN real)
/
CREATE TYPE BODY type_rationnel IS
MAP MEMBER FUNCTION conversion RETURN REAL IS
BEGIN RETURN numerateur/denominateur;
END conversion;
END;
/
```

La fonction de type MAP sera appelée implicitement lors d'une comparaison de deux objets avec les signes =, <, >, <=, >=.
Ex:
```
DECLARE
r1 type_rationnel:=type_rationnel(4,3);
```

```
r2 type_rationnel:=type_rationnel(5,2);
BEGIN
if (r1 > r2) /* utilisation implicite de la fonction conversion */
then...
```

Ce type de fonction peut aussi être appelé explicitement par un SELECT ou encore dans un ORDER BY ou un GROUP BY.
Ex : `ORDER BY r.conversion()`

6.3.2 La fonction de type ORDER

Cette fonction doit comparer le SELF au paramètre fourni et renvoyer

- -1/+1 : si SELF inférieur/supérieur au paramètre
- 0 : si identiques

Ex :
```
CREATE TYPE rationnel AS OBJECT
(numerateur INTEGER,
denominateur INTEGER,
ORDER MEMBER FUNCTION compare(r rationnel) RETURN integer)
/
CREATE TYPE BODY rationnel IS
ORDER MEMBER FUNCTION compare(r rationnel) RETURN integer IS
s,t REAL;
BEGIN
s:= SELF.numerateur/SELF.denominateur;
t:= r.numerateur/r.denominateur;
IF s < t THEN RETURN -1;
ELSIF s=t THEN RETURN 0;
ELSE RETURN +1;
END compare;
END;
/
```

Cette fonction est utilisée implicitement lors d'un ORDER BY.
Ex : `SELECT * FROM Lesrationnels r ORDER BY VALUE(r);`

7 Nouvelles vues du dictionnaire de données

- USER_TYPES : types utilisateurs
- USER _DEPENDENCIES : dépendances entre types via REF
- USER_TYPE_METHODS : méthodes des types abstraits
- USER_OBJECT_TABLES : tables objets-relationnelles

La commande de description d'une table, s'applique aux types et aux tables objets-relationnelles.
Ex: `DESC type_rationnel`

Name	Null?	Type
NUMERATEUR		NUMBER(38)
DENOMINATEUR		NUMBER(38)

METHOD
MAP MEMBER FUNCTION CONVERSION RETURNS NUMBER

8 Nouveauté du SELECT

Le champs FROM peut contenir un autre SELECT. Ceci permet l'utilisation de résultats intermédiaires.
Ex : On veut connaître le pourcentage de chercheurs dans chaque département. On dispose de la table Leschercheurs dans laquelle on a le champ dept et les champs nom, prenom des chercheurs. Résultat affichant le nom du département et le pourcentage de chercheurs par rapport au nombre total de chercheurs.

```
SELECT a.dept "Departement", a.nch/b.total*100 "%Chercheurs"
FROM (SELECT dept, COUNT(*) nch
FROM Leschercheurs GROUP BY dept) a,
(SELECT COUNT(*) total FROM Leschercheurs b;
```