Bases de données objet

Les types utilisés dans les BDO sont
- les types standards existant dans les BD classiques : VARCHAR, NUMBER…
- les types « distincts » : définis par le concepteur de la base pour distinguer des types de données qui s'appuient sur le même type de base.
- les types utilisateurs: utilisés comme des types standards

On appelle aussi ces types utilisateurs des types objet car ils ont une structure complexe et peuvent contenir des opérations (méthodes).

Exemple de création de type objet

```
CREATE TYPE t_adresse AS OBJECT (
        num NUMBER,
        rue VARCHAR(30),
        ville VARCHAR(20),
        codepostal CHAR(5)
) ;
```

On peut ensuite utiliser ce type objet (ou type utilisateur)
- soit pour définir une table relationnelle standard
- soit pour définir une table relationnelle objet (ou table objet relationnelle)
- soit pour définir d'autres types objet qui contiennet cette structure.

Exemple d'objet de type t_adresse

```
t_adresse (2, 'Bd Lavoisier', 'ANGERS', '49000')
<constructeur de type> ( <valeurs des différents champs du type t_adresse>)
```

Utilisation du type dans un autre type

```
CREATE TYPE t_personne AS OBJECT (
nom VARCHAR(30),
prenom VARCHAR(30),
adresse t_adresse
) ;
```

Utilisation d'un type utilisateur dans une table relationnelle standard

Dans une table relationnelle, on l'utilise comme un type prédéfini standard.

Exemple

```
CREATE TABLE employes
(
num NUMBER,
dept NUMBER,
salaire NUMBER,
adresse t_adresse,          -- type objet
nom VARCHAR(30),
PRIMARY KEY num             -- on peut définir des contraintes habituelles sur la table
) ;
```

Insertion dans une table relationnelle (on procède comme habituellement)

```
INSERT INTO employes
 VALUES (1000, 15, 2000, t_adresse (2, 'Bd Lavoisier', 'ANGERS','49000'), 'toto') ;
 -- On utilise le constructeur d'objet t_adresse pour écrire l'adresse
```

Interrogation dans une table relationnelle (on procède aussi comme habituellement)

```
SELECT * FROM employes ;

-- On obtient la table suivante :
```

num	dept	salaire	adresse (num, rue, ville, cp)	nom
1000	15	2000	t_adresse (2, 'Bd Lavoisier', 'ANGERS', '49000')	toto

```
SELECT e.adresse FROM employes e ;
-- Préciser un alias de table
```

adresse (num, rue, ville, cp)
t_adresse (2, 'Bd Lavoisier', 'ANGERS', '49000')

```
SELECT e.adresse.num FROM employes e ;
```

adresse.num
2

Utilisation d'un type objet dans une table relationnelle

Un table objet relationnelle est une table qui contient des éléments de type objet. Chaque élément est identifié par un numéro appelé OID (Object Identifier).

Exemple avec le type t_adresse

```
CREATE TABLE adresses OF t_adressse ;
```

On a une relation dont chaque élément est un objet de type t_adresse. On peut voir la relation de deux manières différentes :

vision objet

t_adresse(…)
OID1 | num1 | rue1 | ville1 | cp1
OID2 | num2 | rue2 | ville2 | cp2
…

vision relationnelle

	num	rue	ville	cp
OID1	num1	rue1	ville1	cp1
OID2	num2	rue2	ville2	cp2
…	…	…	…	…

Remarque

On peut écrire des contraintes comme en relationnel standard lors de la création de la table, on peut définir des clés, etc.

CREATE TABLE adresses OF t_adresse (ville DEFAULT 'ANGERS') ;

Insertion dans une table objet relationnelle

On a deux manières d'insérer des n-uplets :
- soit avec le constructeur de type (vision objet)

INSERT INTO adresses VALUES (t_adresse(30, 'Bd Foch', 'ANGERS', '49000')) ;
-- on insère un objet avec constructeur de type et valeurs des champs du type objet
- soit en précisant chacun des champs (vision relationnelle)

INSERT INTO adresses VALUES (30, 'Bd Foch', 'ANGERS', '49000') ;
-- on précise les valeurs des différents attributs de la table relationnelle

Les deux requêtes sont équivalentes :
- on insère un n-uplet
- un OID lui est attribué lors de l'insertion

Interrogation dans une table objet relationnelle

On peut accéder aux valeurs comme dans le cas du relationnel standard.

```
SELECT * FROM adresses ;
SELECT a.num, a.rue, a.ville, a.cp FROM adresses a ;
-- a est un alias de table
```

On peut accéder aux objets.

```
SELECT VALUE(a) FROM adresse a ;
-- a est un alias de table et VALUE est un mot clé pour récupérer les objets
```

On peut accéder aux OID

```
SELECT REF(a) FROM adresse a ;
-- a est un alias de table et REF est un mot clé pour récupérer les
```

En accédant aux valeurs des attributs

num	Rue	ville	cp
2	Bd Lavoisier	ANGERS	49000
…	…	…	…

En accédant aux objets

VALUE (a) (num, rue, ville, cp)
t_adresse(2, 'Bd Lavoisier', 'ANGERS', 49000)
…

En accédant aux OID

On obtient une relation de référence où chaque référence correspond à un code assez long

fait de chiffres et de lettres.

Remarque

Dans le cas d'une table relationnelle standard utilisant des types objets, on n'aura pas d'OID pour les n-uplets donc pas de SELECT REF(...) ou de SELECT VALUE(…)

Utilisation des références pour représenter les informations dans les relations

Exemple

TABLE employes

num	Dept	salaire	Adresse	nom
1	15	2000	2 Bd Lavoisier ANGERS 49000	toto
6	13	1000	2 Bd Lavoisier ANGERS 49000	titi
9	12	3000	10 Bd Foch ANGERS 49000	tata
…	…	…	…	…

Plutôt que cette représentation qui pose des problèmes de redondance, de maintien de cohérence lors des mises à jours, on préfèrera la représentation suivante :

TABLE employes (avec des OID d'objets de la table adresses)

Num	dept	salaire	Adresse	nom
1	15	2000	ABC1234	toto
6	13	1000	ABC1234	titi
9	12	3000	XYZ9999	tata

TABLE adresses — table objet relationnelle

	num	rue	ville	Cp
ABC1234	2	Bd Lavoisier	ANGERS	49000
XYZ9999	10	Bd Foch	ANGERS	49000

- Pas de changement pour l'interrogation des relations.
- Par contre, en terme de représentation, un objet est conservé en un seul exemplaire dans une autre table et non pas à l'intérieur de chaque n-uplet qui l'utilise.

Exemple : définition de types utilisant des références (ou pointeurs)

```
CREATE TYPE t_ville AS OBJECT (
nom VARCHAR(10),
population NUMBER ) ;

CREATE TABLE pays (
nom VARCHAR(30),
capitale REF t_ville,
population NUMBER ) ;
```

Ou en créant un autre type objet :

```
CREATE TYPE t_pays AS OBJECT (
nom VARCHAR(30),
capitale REF t_ville,
population NUMBER );

CREATE TABLE pays OF t_pays ; CREATE TABLE villes OF t_ville ;
```

Un objet tout seul n'a pas d'OID, c'est un objet d'une table objet relationnelle qui possède un OID.

Insertion de références dans les tables objet relationnelles

TABLE pays

	nom	capitale *(référence)*	population
AAA111	FRANCE	ABC123	60 000 000
AAA222	ITALIE	XYZ999	57 000 000

TABLE villes

	nom	population
ABC123	Paris	2 000 000
XYZ999	Rome	2 700 000

```
INSERT INTO villes VALUES ( 'Paris', 2000000 ) ;
INSERT INTO villes VALUES ( 'Rome', 2700000 );
INSERT INTO pays (nom, population) VALUES ( 'FRANCE', 60000000 ) ;
```

Pour modifier le n-uplet correspondant à la France :

```
UPDATE pays SET capitale = ( SELECT REF(v) FROM villes v WHERE v.nom = 'Paris' )
WHERE nom = 'FRANCE' ;
```

Pour insérer le n-uplet correspondant à l'Italie :

```
INSERT INTO pays SELECT 'ITALIE', REF(v), 57000000 FROM villes v
WHERE v.nom = 'Rome' ;
```

Interrogation de tables utilisant des références

Les données sont interrogées comme si elles étaient physiquement dans la table.

Exemples

```
SELECT * FROM pays ;
 -- le résultat est la table pays ci-dessus

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

nom	capitale.nom	Population
FRANCE	Paris	60 000 000
ITALIE	Rome	57 000 000

Autre fonction sur les objets : DEREF qui renvoie un objet à partir de sa référence.

```
SELECT DEREF ( p.capitale ) FROM pays p ;

-- le résultat est t_ville('Paris', 2 000 000), t_ville('Rome', 2 700 000)

SELECT DEREF ( REF(v) ) FROM villes v ;

-- on obtient les objets de la table villes. C'est équivalent à la requête suivante :

SELECT VALUE ( v ) FROM villes v ;
```

Les collections imbriquées

Exemple

nom	prénom	liste des diplômes
…	…	□□□□□□
…	…	□□□□□□□

Il existe deux types de collections imbriquées :
- les tables imbriquées (NESTED TABLE) dont ne fixe pas la taille à priori
- les tableaux fixes (VARRAY)

Les tables imbriquées

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

Attention, sous Oracle, il ne peut y avoir qu'un seul niveau d'imbrication (c'est-à-dire qu'il ne peut pas y avoir de table imbriquée dans un élément d'une table imbriquée)

Déclarations de types et de tables utilisant des tables imbriquées

```
-- type des éléments de la table imbriquée
CREATE TYPE t_employe AS OBJECT (
num_insee VARCHAR(20),
nom VARCHAR(30),
age NUMBER
) ;

-- type pour une table imbriquée
-- t_employes est le nom du type de la table imbriquée
-- t_employe est le type des éléments de la table
CREATE TYPE t_employes AS TABLE OF t_employe ;

-- table relationnelle standard
```

```
-- tab_emp  est le nom physique de la table imbriquée (ne sert jamais dans les requêtes)
CREATE TABLE departements (
        num_dep  NUMBER,
         budget NUMBER,
        employes t_employes
) NESTED TABLE employes  STORE AS tab_emp  ;
```

num_dep	budget	employes
1	100 000	□□□□□□□
13	50 000	□□□

Création des tables objet relationnelles utilisant des tables imbriquées

```
CREATE TYPE t_departement  AS OBJECT (
num_dep  NUMBER,
budget NUMBER,
employes t_employes
) ;

-- table objet relationnelle
-- STORE AS table_name à préciser lors de la création de la table
CREATE TABLE departements OF t_departement
NESTED TABLE employes  STORE AS tab_emp  ;
```

Insertion dans une relation utilisant une table imbriquée

Pour inséser un élément qui comporte une table imbriquée, on peut, comme pour les autres objets, utiliser le constructeur de type.

```
INSERT INTO departements VALUES (1, 200 000, t_employes(t_employe(12345, 'toto', 25),
t_employe(2222, 'titi',  28))) ;
-- t_employes  est le constructeur de type de la table employé
-- t_employe  est le constructeur de type des éléments  de la table imbriquée.
-- (12345, 'toto',  25) et (2222, 'titi',  28) sont les valeurs  des attributs  de l'élément

INSERT INTO departements VALUES (2, 100 000, t_employes()) ;
-- insertion d'une table imbriquée vide
INSERT INTO departements (numdep,  budget) VALUES (4, 100 000) ;
```

	num_dep	budget	employes
OID	1	200 000	□□
OID	2	100 000	⊏
OID	4	100 000	null

Insertion dans une table imbriquée

Il existe une commande particulière pour insérer dans une table imbriquée : THE

```
INSERT INTO THE ( SELECT employes FROM departements WHERE numdep = 1 )
VALUES ( t_employe (789, 'tutu', 20) ) ;
-- ( SELECT … ) est la table imbriquée dans laquelle on insère l'élément
-- employes est l'attribut de la table imbriquée
-- numdep = 1 est la condition du n-uplet dont on veut modifier la table
-- t_employe (789, 'tutu', 20) est l'élément à insérer dans la table imbriquée
```

Attention : on ne peux insérer que dans une seule table imbriquée.
Que se passe-t-il avec les requêtes suivantes ?

```
INSERT INTO THE ( SELECT employes FROM departements WHERE numdep = 2 )
VALUES ( t_employe(5432, 'lulu', 27) ) ;
-- on insère l'élément

INSERT INTO THE ( SELECT employes FROM departements WHERE numdep = 4 )
VALUES ( t_employe(987, 'nono', 50) ) ;
-- ça ne marche pas car la table imbriquée n'exite pas
```

Comment faire pour que ça marche ?

Il faut créer la table imbriquée au préalable.

1. On met à jour le n-uplet pour rendre l'insertion possible :
UPDATE departements SET employes = t_employes() WHERE numdep = 4 ;
puis on exécute la requête d'insertion.

2. On met à jour en insérant l'élément :
UPDATE departements SET employes = t_employes(t_employe(987, 'nono', 50) WHERE numdep = 4 ;

Interrogation des tables utilisant des tables imbriquées

Pour la table objet-relationnelle

```
SELECT * FROM departements ;
```

On obtient les valeurs des attributs comme habituellement avec les tables imbriquées sous forme de constructeurs de type.

Pour les tables imbriquées, on utilise THE comme pour l'insertion

```
SELECT e.nom FROM THE ( SELECT employes FROM departements WHERE numdep = 1
) e ;
-- on doit utiliser un alias de table (e)
-- e.nom est un attribut du type t_employe
-- ( SELECT … ) est une table imbriquée
```

Remarque

Ici aussi, il faut sélectionner une seule table imbriquée.

Comment récupérer les informations de plusieurs tables imbriquées simultanément ?

On utilise la commande CURSOR, par exemple :

SELECT d.numdep, CURSOR (SELECT e.nom FROM TABLE (employes) e) FROM departements d ;

Les tableaux fixes (VARRAY)

C'est une collection limitée, ordonnée d'éléments de même type.

Remarque : Un tableau fixe permet :

- d'avoir plusieurs niveaux d'imbrication (contrairement aux NESTED TABLE qui nécessitent l'utilisation de références)
- d'accéder aux éléments par leur numéro

Mais on ne peut pas accéder à un élément particulier du VARRAY dans une requête SQL standard, il faut utiliser un bloc PL/SQL (langage procédural qui intègre les requêtes SQL) contrairement aux NESTED TABLE qui se comportent comme des tables relationnelles standard.

Déclaration d'un VARRAY

Comme c'est une collection de taille limitée, il faut déclarer d'emblée la taille du tableau fixe.

Exemple

```
CREATE TYPE tvadresses AS VARRAY(2) OF tadresse ;
-- tadresse est le type des éléments du VARRAY, type utilisateur défini précédemment
-- 2 est la taille du VARRAY
-- tvadresses est le type VARRAY défini
```

On peut ensuite utiliser le type ainsi créé pour définir des tables relationnelles ou objet-relationnelles.

Exemple

```
CREATE TABLE etudiants (
noine VARCHAR(10),
nom VARCHAR(30),
adresses tvadresses) ; -- ici, une table relationnelle
```

Manipulation d'un VARRRAY

On peut manipuler un VARRAY de deux manières :

- dans une requête SQL, on manipule le VARRAY entier
- dans un bloc PL/SQL, on manipule des éléments particuliers du VARRAY

Insertion dans un VARRAY

On utilise le constructeur de type avec autant d'éléments que l'on veut (en respectant le nombre maximal d'éléments de VARRAY)

Exemple

```
INSERT INTO etudiants VALUES ('12345', 'Jacques',
tvadresses(tadresse('Elysée','Paris')));
-- on a un VARRAY qui contient un seul élément
INSERT INTO etudiants VALUES ('9999', 'Dom', tvadresses(tadresse('Matignon',
'Paris'),null));
-- on a un VARRAY qui contient les deux éléments du VARRAY dont un n'est pas
renseigné
INSERT INTO etudiants VALUES ('2340', 'Nico', null);
-- aucun VARRAY n'est créé
```

Remarque

Si on veut travailler sur le VARRAY de Nico, il faut bien sûr en créer un. Si on veut travailler dans une requête SQL, il faut travailler sur l'ensemble du VARRAY.

Exemple

```
UPDATE etudiants SET adresses = tvadresses(tadresse('Elysee', 'Paris'), tadresse('Bitz',
'Correze')) WHERE nom = 'Jacques' ;
```

Si on veut travailler sur un seul élément, il faut utiliser un bloc PL/SQL.

Exemple : pour ajouter ajouter une nouvelle adresse à l'étudiant nommé Dom.

```
-- déclaration des variables
DECLARE lesadr tvadresses;
-- instructions
BEGIN
        -- initialisation de la variable lesadr
        SELECT adresses INTO lesadr FROM etudiants WHERE nom = 'Dom' ;
        -- modification de la deuxième valeur
        lesadr(2) := tadresse('Hôtel Martinez', 'Cannes') ;
        -- mise à jour de la relation
        UPDATE etudiants SET adresses = lesadr WHERE nom = 'Dom' ;
END ;
```

Autre exemple

```
INSERT INTO etudiants VALUES ('54321', 'Lionel', tvadresses()) ;
DECLARE lesadr tvadresses ;
BEGIN
        SELECT adresses INTO lesadr FROM etudiants WHERE nom = 'Lionel' ;
        lesadr.extend; -- on ajoute un élément au VARRAY
        lesadr(1) := tadresse('Bon Repo', 'Ile de Ré') ;
        UPDATE etudiants SET adresses = lesadr WHERE nom = 'Lionel' ;
END ;
```

-- ici ça ne marche pas car le tableau fixe est vide, tout à l'heure, ça marchait pour Dom car on avait le deuxième élément (non renseigné mais il existait). Ici, il faut donc créer la place du premier élément avant de le traiter. On aurait pu aussi mettre à jour le VARRAY dans une requête SQL standard puis exécuter le bloc PL/SQL

Il existe plusieurs fonctions prédéfinies sur les tableaux fixes (utilisables seulement dans des blocs PL/SQL) :
- limit : nombre minimum d'un VARRAY
- last : indice du dernier élément
- extend : ajout d'un élément au tableau
- extend(n) : ajout de n élément au tableau

Exemple : Ajout d'un élément sans savoir combien d'éléments comporte le tableau

```
DECLARE lesadr tvadresses ;
 BEGIN
        SELECT adresses INTO lesadr FROM etudiants WHERE nom = 'Lionel' ;
        lesadr.extend;
        lesadr(lesadr.last) := tadresse('Bon Repo', 'Ile de Ré') ; -- valeur du dernier élément
        UPDATE etudiants SET adresses = lesadr WHERE nom = 'Lionel' ;
END ;
```

Les méthodes

Une méthode (ou opération) est la modélisation d'une action applicable sur un objet, carac- térisée par un en-tête appelé signature définissant son nom, ses paramètres d'appel et de re- tour, et qui permet de modifier l'état de l'objet ou de renvoyer un résultat. En relationnel ob- jet, les types peuvent admettre soit des fonctions soit des procédures.

On déclare les méthodes :
- soit au début lors de la déclaration de l'objet
- soit plus tard avec commande ALTER TYPE

Le corps de la méthode correspond aux opérations effectuées sur l'objet, il peut faire référence à l'objet concerné grâce à SELF.
il y a deux sortes de méthodes : MEMBER et STATIC.
• Les méthodes membre s'appliquent aux instances du TDU. Elles sont le moyen pour accéder aux données d'une instance du TDU (qui est fournie par l'argument SELF).
• Un méthode statique s'applique à un TDU, mais pas à sesinstances, et n'a pas de paramètre SELF.

Exemple : type objet contenant une fonction

```
CREATE TYPE tpersonne  AS OBJECT (
        nom VARCHAR(10), datenaiss date,
        MEMBER FUNCTION age RETURN number
        -- le type contient une fonction de nom age sans paramètre  qui retourne un
        nombre
)

-- implémentation des méthodes
CREATE TYPE BODY tpersonne  AS MEMBER FUNCTION age RETURN number IS n
number ;
BEGIN
```

```
n := TRUNC((SYSDATE - SELF.datenaiss)) ; RETURN n ;
END age ;
END ;
```

On peut utiliser une fonction dans un bloc PL/SQL ou une requête SQL. Exemple : avec

```
personne table objet-relationnelle de type tpersonne
SELECT p.age() FROM personnes p ;
```

Protection de la base de données

Remarque
Quand on crée une fonction, Oracle demande de préciser l'influence de la fonction sur la base de données (en particulier, on doit préciser que l'on ne modifie rien dans la base) : il faut rajouter dans la déclaration de la fonction une instruction en ce sens
PRAGMA RES- TRICT_REFERENCES(<nom_fonction>, <restriction d'accès>)

Exemple

```
CREATE TYPE t_personne AS OBJECT (
  Nom VARCHAR ( 10 ) ,
  datenaiss DATE ,
MEMBER FUNCTION age RETURN NUMBER ;
 PRAGMA RESTRICT_REFERENCES ( age, WNDS ))
 -- WNDS signifie qu'on n'écrit pas dans la base
)
```

Paramètres des méthodes

De manière générale, dans les méthodes, les paramètres sont déclarés de la manière suivante
< nom_paramètre > [IN | OUT | IN OUT] <type_du_paramètres> Les paramètres sont séparés par des virgules.
Les fonctions ne comportent que des paramètres IN (c'est-à-dire des données).

Les procédures

Elles sont définies de la même manière que les fonctions. On peut :
- soit les définir lors de la création du type
- soit les ajouter par la suite avec la commande ALTER TYPE

Exemple Pour rajouter une procédure au type défini précédemment

```
ALTER TYPE t_personne AS OBJECT (
Nom VARCHAR ( 10 ) ,
datenaiss DATE ,
MEMBER FUNCTION age RETURN NUMBER ; PRAGMA RESTRICT_REFERENCES( age,
WNDS ) ;
MEMBER PROCEDURE modifnaiss ( newdate IN DATE )
) /
-- Puis on réécrit le corps de l'implémentation
CREATE OR REPLACE TYPE BODY t_personne AS MEMBER FUNCTION age RETURN
```

```
NUMBER  IS
        … END age ;
MEMBER PROCEDURE modifnaiss  ( newdate  IN DATE  ) BEGIN
        SELF.datenaiss := newdate ;
        END modifnaiss ;
END
/
```

Il faut redonner l'ensemble de l'implémentation du corps des méthodes. L'appel d'une procédure se fait dans un bloc PL / SQL (jamais dans un SELECT, contrairement aux fonctions).

Exemple

```
CREATE TABLE personne  OF t_personne ;
INSERT INTO personne  VALUES ( 'Jacques',  '14-JUL-1900'  ) ;
INSERT INTO …
…
-- Pour changer la date de naissance de Jacques :
DECLARE toto t_personne ;
BEGIN
-- on initialise toto
SELECT VALUE  ( p ) INTO toto FROM personne p WHERE p.nom = 'Jacques' ;
-- la date de naissance de toto est modifiée
toto.modifnaiss('29-NOV-1932') ;
-- on met à jour la base
DELETE FROM  personne  WHERE  nom = 'Jacques' ; INSERT INTO personne  VALUES (
toto ) ;
END ;
```

Si on veut que l'appel à la procédure modifie la base, il faut procéder différemment :

```
CREATE OR REPLACE TYPE BODY AS MEMBER …
MEMBER PROCEDURE modifnaiss  ( newdate  IN DATE  ) IS BEGIN
UPDATE personne SET datenaiss  = newdate  WHERE VALUE  ( p ) = SELF ;
 END modifnaiss ;
END ;

DECLARE toto t_personne ;
BEGIN
     SELECT VALUE  ( p ) INTO toto FROM personne p WHERE p.nom = 'Jacques' ;
      -- l'appel à la procédure modifie directement la table de la base de données
       toto.modifnaiss('29-NOV-1932') ;
       END ;
```

Surcharge

On peut définir une même méthode avec des paramètres différents

Exemple

```
CREATE TYPE t_etudiant AS OBJECT (
noine            NUMBER ,
nom              VARCHAT  ( 30 ) ,
MEMBER PROCEDURE modif ( newnumero IN NUMBER  ) ;
MEMBER PROCEDURE modif ( newnom IN VARCHAR  )
) /
CREATE TYPE BODY t_etudiant AS MEMBER PROCEDURE modif (newnumero IN
NUMBER  ) IS
BEGIN
      SELF.noine := newnumero ;
END modif ;
MEMBER PROCEDURE modif (newnom IN VARCHAR  ) IS
BEGIN
      SELF.nom := newnom ;
END modif ;
END ;
/
```

On a une surcharge de la procédure modif, lors de l'utilisation de la procédure, c'est le type de la valeur passée comme paramètre qui détermine quelles sont les instructions qui seront exécutées.

Héritage:

L'héritage simple entre types (type inheritance) est supporté au moyen de la clause UNDER de l'énoncé CREATE TYPE.

- UNDER dit qu'un TDU Y hérite d'un autre TDU X.
- Y est un sous-type de X et X est le super-type de Y.
- Seulement un type NOT FINAL peut avoir un sous-type.