

Base de Data

JP Momo & Sttv & Brunoir

16 décembre 2014

Chapitre 1

Fonctions et procédures stockées

1.1 Partie PL/SQL

Donc comme vous êtes tous des GROSSES mer..., je vais vous expliquer ce que sont les fonctions et les procédures en PL/SQL. Tout d'abord, à quoi servent les fonctions et les procédures STOCKEES ?

Elles servent à enregistrer des programmes dans le noyau Oracle. Elles peuvent être utilisées par tout le monde (selon les droits) et sont stockées sous forme de *pseudo-code*, c'est-à-dire qu'elles ne sont compilées qu'UNE SEULE FOIS. C'est génial non ?

Trêve de bavardages, on va voir leurs syntaxes.

Une procédure en PL/SQL se déclare de la manière suivante :

```
1  — declaration d'une procedure
2  CREATE [OR REPLACE] PROCEDURE nom_procedure ([liste des parametres]) IS |AS
3      [declaration des variables]
4  BEGIN
5      — corps de la procedure
6  EXCEPTION — facultatif
7      — definition des exceptions
8  END;
```

Bon ce n'est pas très clair mais on va voir ça petit à petit.

Donc, la première ligne correspond à une déclaration de procédure basique avec son nom et ses différents paramètres (ne pas oublier le IS ou le AS).

La deuxième ligne correspond à la déclaration des variables.

A partir du BEGIN, on écrit ce que va faire notre procédure. EXCEPTION est facultatif mais correspond à la zone d'exception du code.

Enfin le END, bah END.

En ce qui concerne les paramètres dans une procédure (ou une fonction), il faut leur donner un nom (gicLo), spécifié si c'est un paramètre **d'entrée** (IN, donc non modifiable) ou **de sortie** (OUT, modifiable par la procédure) ou les deux et enfin spécifié son type (une procédure ou une fonction n'a pas obligatoirement de paramètres).

```
1  CREATE OR REPLACE PROCEDURE numAuteur (nom IN VARCHAR2 prenom IN VARCHAR2
    num OUT INTEGER)
```

Ici, la procédure numAuteur qui donne le numéro d'un auteur, prend en paramètres **d'entrée**, de type VARCHAR 2, *nom* et *prenom*. En paramètre **de sortie**, de type INTEGER, on a *num*.

Voici un exemple de procédure stockée.

```
1  /* on definit la procedure unTitre qui renvoie le nom d'un film
2  en entree, on a le numero du film et du realisateur
3  en sortie, le titre du film */
4  CREATE OR REPLACE PROCEDURE unTitre (numFilm IN INTEGER, realisateur IN
      INTEGER, titreFilm OUT VARCHAR2) IS
5  BEGIN
6      /* on selectionne le titre du film DANS titreFilm (le parametre de
          sortie)
7      sinon ca ne marche pas */
8      SELECT f.titre INTO titreFilm
9      FROM ens2004.film f
10     WHERE f.numFilm = numFilm
11     AND f.realisateur = realisateur;
12 END;
```

Pas besoin d'expliquer ce qui différencie une fonction d'une procédure. Donc syntaxe.

```
1  — declaration d'une fonction
2  CREATE [OR REPLACE] FUNCTION nom_fonction([liste des parametres])
3      RETURN type_retour IS |AS
4      [variable de retour]
5      [declaration des variables]
6  BEGIN
7      — corps de la fonction
8      RETURN variable_retour;
9  EXCEPTION \\facultatif
10     — definition des exceptions
11 END;
```

La principale différence c'est que l'on spécifie la variable que l'on va renvoyé. En ce qui concerne les paramètres d'une fonction, c'est la même chose qu'une procédure SAUF que l'on préfère ne spécifier que des paramètres d'entrées (IN).

Vu que vous êtes trop fort, on peut passer directement à un exemple ofc.

```
1  /* on definit la fonction numFilm qui renvoie le numero d'un film
2  en entree, on a le titre du film et le numero du realisateur */
3  CREATE OR REPLACE FUNCTION numFilm (titreFilm IN VARCHAR2, realisateur IN
      INTEGER)
4      RETURN INTEGER IS
5      numFilm INTEGER;
6  BEGIN
7      — on selectionne le numero du film dans numFilm
8      SELECT f.numFilm INTO numFilm
9      FROM ens2004.film f
10     WHERE f.titre = titreFilm
11     AND f.realisateur = realisateur
12     — et on retourne numFilm qui contient le numero du film
13     RETURN numFilm;
14 END;
```

1.2 Partie Java

C'est bien beau de définir des fonctions ou des procédures stockées mais faut quand même les utiliser un jour.

Je suppose que vous avez tous suivi le magnifique cours sur JDBC, donc on va directement voir le code Java. Donc comment appeler nos fonctions et procédures en Java. Il suffit d'utiliser ce que l'on appelle l'interface CallableStatement. Comme l'indique son nom, elle appelle les procédures et les fonctions stockées sur Oracle. Comment ça marche?(pas le site ofc) Il suffit d'indiquer le nom de la fonction ou de la procédure lors de l'initialisation de l'objet CallableStatement grâce à la méthode prepareCall() de l'interface Connection (fallait suivre le cours sur JDBC).

Il existe deux cas : soit la procédure ou la fonction comporte des paramètres lors de l'appel soit elle n'en comporte pas. EXEMPLE!!!

```
1 // procedure sans parametres
2 CallableStatement cst = co.prepareCall("{call nom_procedure()}");
3 // procedure avec parametres (2 ici)
4 CallableStatement cst1 = co.prepareCall("{call nom_procedure(?, ?)}");
5 // fonction sans parametres
6 CallableStatement cst2 = co.prepareCall("{? = call nom_fonction()}");
7 // fonction avec parametres (2 ici)
8 CallableStatement cst3 = co.prepareCall("{? = call nom_fonction(?, ?)}");
```

Dans les cas, où il y a des paramètres, il faut qu'on les définissent pour Java (bah oui c'est pas de la gicMa). Pour les paramètres d'entrée (IN), il faut utiliser la méthode set<type>(<rang>, <valeur>). On va la définir pour les plus mauvais d'entre vous :

- <type> correspond au type que l'on va définir pour notre paramètre
- <rang> correspond au rang du paramètre à positionner (il faut respecter l'ordre de la procédure ou de la fonction stockée)
- <valeur> correspond à la valeur que va prendre notre paramètre

En ce qui concerne les paramètres de sortie, il faut spécifier, dans un premier temps, son type. Pour cela, on utilise la méthode registerOutParameter(<rang>, <type>) :

- <rang> même chose que les paramètres d'entrée
- <type> le type de notre paramètre

Si on reprend la procédure unTitre et la fonction numFilm, voilà ce que ça donne :

```
1 // pour la procedure unTitre
2 // on a 2 parametres en entree (INTEGER) et 1 en sortie (VARCHAR2)
3 CallableStatement cst = co.prepareCall("{call unTitre(?, ?, ?)}");
4 cst.setInt(1, '50'); // numFilm est a la premiere place
5 cst.setInt(2, '45'); // realisateur est a la deuxieme place
6 cst.registerOutParameter(3, java.sql.Types.VARCHAR); // titreFilm a la
   troisieme
7
8 // pour la fonction numFilm
9 // on a 2 parametres en entree (VARCHAR2 et INTEGER)
10 CallablaStatement cst1 = co.prepareCall("{? = call numFilm(?, ?)}");
11 // il faut respecter l'ordre des ? et dans la fonction SQL
12 cst1.setString(2, 'TITANIQUE');
13 cst1.setInt(3, '69');
14 cst1.registerOutParameter(1, java.sql.Types.INTEGER);
```

Vous avez compris? Cependant, on n'a toujours pas exécuter notre procédure ou fonction. C'est très simple THREAD, il suffit d'utiliser la méthode execute(). (Vu que c'est un booléen, on préfère la stocker dans une

variable). Et enfin, il faut pouvoir récupérer les valeurs de sortie (si il y en a).

Pour cela, on utilise la méthode `get<type>(<rang>)` qui fonctionne un peu près comme `set<type>(<rang>, <valeur>)`.

Si on reprend les exemples précédents :

```
1 // pour la procedure unTitre
2 CallableStatement cst = co.prepareCall("{call unTitre(?, ?, ?)}");
3 cst.setInt(1, '50');
4 cst.setInt(2, '45');
5 cst.registerOutParameter(3, java.sql.Types.VARCHAR);
6 boolean success = cst.execute(); // on execute notre procedure
7 String titreFilm = cst.getString(3); // on recupere dans titreFilm le titre
   renvoie
8 cst.close(); // on oublie pas de fermer (c'est pour etre propre)
9
10 // pour la fonction numFilm
11 CallablaStatement cst1 = co.prepareCall("{? = call numFilm(?, ?)}");
12 cst1.setString(2, 'TITANIQUE');
13 cst1.setInt(3, '69');
14 cst1.registerOutParameter(1, java.sql.Types.INTEGER);
15 boolean success1 = cst1.execute(); // on execute notre procedure
16 int numFilm = cst1.getInt(1); // on recupere dans numFilm le numero du film
17 cst1.close();
```

Chapitre 2

Concurrence d'accès

La concurrence d'accès est un principe, en base de données, qui consiste en la gestion de données. Pour être plus clair, en reprenant le cours, si plusieurs utilisateurs manipulent les mêmes données en même temps, il faut arbitrer entre :

- la disponibilité de l'information : ne pas bloquer tout le monde parce que une personne travaille
- la cohérence de l'information : ne pas rendre les données incohérentes en tenant compte de plusieurs demandes en concurrence en même temps.

2.1 Accès concurrents

Pour cela, on va définir plusieurs termes importants pour la fiche et pour tout le reste.

Transaction : unité logique de traitement. Pour résumer grossièrement, ça correspond à la modification ou l'interrogation d'une base de données (requêtes, fonctions, procédures, etc...).

Le début d'une transaction se définit par un ordre SQL(SELECT, UPDATE, INSERT, etc...) ou la fin de la transaction précédente. La fin d'une transaction est défini par l'instruction COMMIT ou ROLLBACK. L'instruction COMMIT consiste à valider la transaction en cours et donc rendre les modifications persistantes.

Quant à ROLLBACK, elle annule la transaction en cours et toutes les modifications effectués lors de cette dernière.

Accès concurrents : plusieurs utilisateurs accèdent, en même temps, à la même donnée dans la base.

Base cohérente : contraintes d'intégrités respectées(en gros les règles sur les tables). Si lors d'une transaction une BD passe d'un état cohérent à un autre état cohérent, l'intégrité des données est sauvegardée.

Consistance des données : Si SGBD (Système de gestion de base de données) garantit que les données utilisés par une transaction ne sont pas modifiées par des requêtes d'autres transactions pendant cette même transaction.

Bon c'est un peu trop théorique mais y aura sûrement ça au DS (non je rigole). Pour mieux comprendre la consistance des données, on va aller ouvrir les verrous.

Pour assurer la consistance de nos données, il faudra verrouiller les données durant la durée de la transaction. Il en existe 3 :

- verrous partagés (shared lock) : pour lire les données avec l'intention d'y faire des mises à jour.
- verrous exclusifs (exclusive lock) : pour modifier des données.
- verrous globaux (global lock) : pour bloquer un ensemble de données, généralement une table toute entière.

Il y a une autre notion à connaître c'est la collision. si 2 transactions accèdent à la même donnée en même temps, il y a collision avec un GROS CAMION et donc une perte de cohérence.

2.2 Utilisation et problèmes liés aux verrous

Quand est ce qu'un verrou est posé ?

Il est posé jusqu'à la fin de la transaction en cours. Le seul moyen de relâcher les verrous posés par une transaction est par le biais des instructions COMMIT et ROLLBACK (lors de la création d'une table aussi, il y a un COMMIT implicite).

Pour mieux comprendre cette histoire de verrous, on va reprendre les exemples du cours et les commenter.

Pour une collision de type : perte de mise à jour :

T₁ et T₂ modifient simultanément la quantité *qte*.

Temps	État de la base	Transaction T ₁	Transaction T ₂
t ₀	qte=1000		
t ₁		Lire qte	
t ₂		qte ← qte + 3000	
t ₃	qte=4000	Écrire qte COMMIT	
t ₄			Lire qte
t ₅			qte ← qte + 500
t ₆			Écrire qte
	qte=4500		COMMIT

Donc dans cet exemple, 2 transactions veulent modifier simultanément la quantité *qte*. Le but étant d'avoir une quantité *qte* égale à 4500. Donc ici on a l'ordre des opérations à faire pour chacune des transactions. Cependant, il faut voir comment fonctionnent les verrous dans cet exemple.

Temps	Etat de la base	Transaction T ₁	Transaction T ₂
t ₀	qte=1000		
t ₁		Lire qte	
t ₂			Lire qte
t ₃		qte ← qte + 3000	
t ₄	qte=4000	Écrire qte COMMIT	
t ₅			qte ← qte + 500
t ₆	qte=1500		Écrire qte COMMIT

À la fin, *qte* : 1000 ? ou 1500 ? ou 4500 ?

Dans ce tableau, à t₀, *qte* = 1000. Ensuite à t₁, la transaction T₁ lit la valeur de *qte* (donc 1000), de même pour la transaction T₂ à l'instant t₂ (*qte* = 1000).

A l'instant t₃, T₁ modifie la valeur de *qte* (*qte* = *qte* + 3000) et à t₄ enregistre cette modification par l'instruction COMMIT. Donc *qte* = 4000. A t₅, T₂ modifie aussi la valeur de *qte* et enregistre cette valeur. Cependant on s'attendrait à avoir *qte* = 4500 mais non bande de gogols. Cette erreur intervient à l'instant t₂. En effet, T₂ lit la valeur actuel de *qte*, c'est-à-dire 1000. Donc sa modification de la valeur *qte* ne prend pas en compte la modification de T₁. En effet, il y a une collision et donc une mauvaise gestion des verrous. La solution pour avoir *qte* = 4500 est la suivante.

Temps	État base	Transaction T ₁	Transaction T ₂
t ₀	qte=1000		
t ₁		Verrou exclusif sur qte + Lire qte	
t ₂			Attente pour poser verrou exclusif sur qte
t ₃		qte ← qte + 3000	
t ₄	qte=4000	Écrire qte COMMIT	
t ₅			Verrou exclusif sur qte
t ₆			Lire qte
t ₇			qte ← qte + 500
t ₈	qte=4500		Écrire qte COMMIT

Dans cette version, T1 pose un verrou exclusif lors de la lecture de qte, c'est-à-dire, que T1 va appliquer une modification sur cette valeur et que T2 doit attendre la fin de ce verrou (COMMIT ou ROLLBACK). Après que T1 ait modifier la valeur et fait un COMMIT (qte = 4000), c'est au tour de T2 d'appliquer un verrou exclusif et de modifier la valeur de qte. Et comme par gicMa, qte = 4500, car la valeur à laquelle accède T2 a été modifié et validé par T1.

Pour les collisions de type : lecture impropre et lecture non reproductible se référerait au cours parce que j'ai trop la flemme. Si vous avez compris cette exemple, vous comprendrez sûrement les autres (je vous conseille d'aller voir le cours pour tester vos connaissances ofc).

Vu que vous êtes chimiques, on va faire une suite d'exemples :

```

1  — l'utilisateur 1 change le nom d'un film (numero 15 et realisateur 58)
2  — on a ici l'application d'un verrou exclusif
3  UPDATE ens2004.film
4  SET titre = 'MABITE'
5  WHERE numFilm = 15
6  AND realisateur = 58;
7
8  — l'utilisateur 1 et 2 executent la meme requete d'affichage
9  SELECT * FROM ens2004.film;
10 — A ce moment la, la modification du film n'etant pas valide, le titre du
    film affiche est toujours l'ancien
11
12 — l'utilisateur 1 fait un COMMIT
13 COMMIT;
14
15 SELECT * FROM ens2004.film;
16 — La valeur etant modifie, l'affichage contiendra la nouvelle valeur c'est-a-
    dire MA...

```

Il existe plusieurs niveaux de cohérence pour une transaction. Une donnée est dite salie si elle a été modifiée par une transaction non confirmée. En gros, par un COMMIT. Pour une transaction T, on peut exiger que T satisfasse une ou plusieurs des propriétés (c'est pas vraiment important mais c'est toujours utile) :

1. : T ne modifie pas des données salies par d'autres transactions.
2. : T ne confirme pas ses changements avant la fin de la transaction.

3. : T ne lit pas des données salies pas d'autres transactions.
4. : D'autres transactions ne salissent pas des données lues par T avant que T soit terminée.

Pour chacune de ces propriétés, un niveau de cohérence est attribué à T :

- niveau 0 si T vérifie 1 : donc pas de problèmes de perte de mise à jour.
- niveau 1 si T vérifie 1 et 2 : si une transaction est annulée (ROLLBACK), il n'est pas nécessaire de défaire explicitement les modifications antérieures à l'annulation.
- niveau 2 si T vérifie 1 et 2 et 3 : pas de problèmes de perte de mise à jour et de lecture impropre.
- niveau 3 si T vérifie 1 et 2 et 3 et 4 : pas de problèmes de perte de mise à jour, de lecture impropre et assure la reproductibilité des lectures, donc une isolation totale de la transaction. (je vous avais dit de voir le cours)

Sur Oracle les verrous sont invoqués automatiquement dans des cas spécifiques, mais on peut le faire manuellement aussi. Ce tableau résume bien les différents verrous que l'on peut mettre.

Opération SQL	Verrou ligne	Verrou table
SELECT	Non	Aucun
INSERT	Oui	RX
UPDATE	Oui	RX
DELETE	Oui	RX
SELECT... FOR UPDATE	Oui	RS
LOCK TABLE ... IN :		
ROW SHARE MODE	Non	RS
ROW EXCLUSIVE MODE	Non	RX
SHARE MODE	Non	S
SHARE EXCLUSIVEMODE	Non	SRX
EXCLUSIVE MODE	Non	X

- RX : Row exclusif, interdiction de modification et de lecture sur la ligne que l'on modifie pour tout autre utilisateur
- RS : Row share, interdiction de modification seulement sur la ligne que l'on lit
- S : Share, interdiction de modification sur la table que l'on lit
- X : Exclusif, interdiction de modification et lecture sur la table que l'on modifie

Problème possible lié au verrou

Deadlock : On rencontre ce problème lorsque un utilisateur attend la fin du verrou d'un autre alors que celui ci attend la fin du verrou de l'un.

Par Exemple je pose un verrou sur la ligne 1 d'une table puis un random pose un verrou sur la ligne 2. Je souhaite récupérer la valeur de la ligne 2 → je suis mis en attente par le random, sauf que ce putain de random attend que moi j'enlève mon verrou sur la ligne 1. Du coup on se retrouve en attente tout les deux comme des cons. → Pour éviter ce problème il faut que l'ordre dans laquelle le random et moi posons les verrous soit le même. Je prend la ligne 1, il prend la ligne 1, je prend la ligne 2, il prend la ligne 2 etc... Cette solution est appelé **Règle de Havender**. Allez savoir pourquoi...

Prenons deux utilisateurs lambda :

```

1  — l'utilisateur 1 change le nom du film (numero 15 et realisateur 58)
2  UPDATE ens2004.film
3  SET titre = 'TD32'
4  WHERE numFilm = 15
5  AND realisateur = 58;
6
7  — l'utilisateur 2 change le nom d'un autre film (numero 32 et realisateur 96)
8  UPDATE ens2004.film
9  SET titre = 'BOGOSS'
10 WHERE numFilm = 32
11 AND realisateur = 96;
12
13 — l'utilisateur 1 change le nom du film qui est en train d'etre modifie par l
   — utilisateur 2 (numero 32 et realisateur 96)
14 UPDATE ens2004.film
15 SET titre = 'TD321'
16 WHERE numFilm = 32
17 AND realisateur = 96;
18
19 — l'utilisateur 2 change le nom du film qui est en train d'etre modifie par l
   — utilisateur 1 (numero 15 et realisateur 58)
20 UPDATE ens2004.film
21 SET titre = 'BOGOSS1'
22 WHERE numFilm = 15
23 AND realisateur = 58;

```

Dans cet exemple, on ne fait pas de COMMIT. A partir de la troisième instruction, Oracle va tourner en boucle (il attend le COMMIT de l'utilisateur 2) et va afficher sur l'utilisateur 1 blocage fatale de même à la quatrième instruction. Cette erreur est due au fait que deux utilisateurs veulent modifier des données qui sont modifiées par l'un et par l'autre mais n'ont fait aucune validation ou annulation.

Problème de la tasse à café.... : Il s'agit du fait que la saisie d'un utilisateur peut potentiellement mettre indéfiniment le programme en pause (Par exemple, je vais prendre une tasse de thé parce que j'aime le thé et faire chier les gens), ce qui provoque dans le cas où on a posé un verrou avant la demande de saisie, cela verrouille la cible indéfiniment et donc bloqué d'autre utilisateur. → Pour éviter ce problème, il faut poser le verrou après la saisie tout simplement.

Chapitre 3

Base de données Objet

Objet ? Wut ? C'est quoi encore cette partie ?
Ne vous inquiétez pas ! La partie objet de la base de données n'est juste qu'une simple révision de vos cours d'informatique objet mais avec ce merveilleux langage SQL.

3.1 Créer un type

A quoi sa sert ? Bah sa sert à définir un type objet qui est abstrait. Il est défini pas l'utilisateur pour décrire la structure (ses attributs) et le comportement (ses méthodes).

Sa fameuse syntaxe :

```
1 CREATE [ OR REPLACE ] TYPE nomType AS OBJECT  
2 ( nomAttribut type [ , nomAttribut type ... ] );
```

Prenons un exemple : Nous voulons créer un type adresse avec une rue, une ville, un code postal et le nom de la ville :

```
1 CREATE TYPE obAdresseTy AS OBJECT(  
2 Rue VARCHAR2(50) ,  
3 Ville VARCHAR2(30) ,  
4 CodePostal CHAR(5) ,  
5 Pays VARCHAR2(15) );
```

Comme vous le constatez, nous créons un type comme si c'était une table avec des attributs, la nature de l'attribut et la taille de l'attribut. **AS OBJECT** permet de définir le type comme un objet ; c'est à dire comme un constructeur en Java qui va être utilisé dans d'autres types et tables quand c'est nécessaire.

Maintenant nous voulons créer un type personne qui possède un nom, un prénom et le type adresse qu'on a créé plus tôt.

```
1 CREATE TYPE obPersonneTy AS OBJECT(  
2 Nom VARCHAR2(30) ,  
3 Prenom VARCHAR2(30) ,  
4 Adresse obAdresseTy );
```

Vous pouvez le remarquer aussi qu'un attribut est de nature un type. Cet attribut permet d'appeler le type qu'on souhaite pour éviter de réécrire tous les attributs d'adresse. En gros l'attribut adresse va appeler tous les attributs du type adresse. Plus simple non ?

Pour supprimer un type rien de spécial. Mais à faire si il n'est plus utilisé !

```
1 DROP TYPE nomObjetType;
```

Une erreur apparaîtra si vous voulez supprimer un type qui est encore présent dans une table utilisée. Pour cela il faut supprimer la table puis le type.

3.2 Créer une table avec des types

Après avoir pris le temps de créer ces jolis types, nous voulons créer une table qui comporte la colonne personne à l'intérieur accompagné d'un login et de la date d'inscription.

```
1 CREATE TABLE obClient (  
2 login CHAR(20) PRIMARY KEY ,  
3 client obPersonneTy ,  
4 dateIns DATE NOT NULL);
```

Le code ci-dessus nous montre que la table peut appeler un type. A quoi sa sert ? Bah sa sert à raccourcir le code et d'éviter d'appeler tous les attributs qu'on a besoin. De manière plus abstraite, le table va appeler tous les attributs de type Personne qui va appeler tous les attributs de type Adresse. En gros nous avons économiser 7 lignes de création d'attributs qui va quand même apparaître dans la table. Elle n'est pas belle la vie ?

REMARQUE : Dans une table, il faut toujours avoir un attribut qui a le rôle de clé primaire (Genius gnégné) MAIS un type ne peut pas être une clé primaire (Je t'ai troué le cul là hein!).

Maintenant qu'on a perdu du temps pour créer des tables, il faut savoir les remplir (de sp... spécialité maison). Regardez bien le code :

```
1 INSERT INTO obClient  
2 VALUES ( 'toto' ,  
3 obPersonneTy ( 'Duval' , 'Thomas' ,  
4 obAdresseTy( 'rue des dunes' , 'Trouville' , '76666' , 'France' ) ) ,  
5 TO DATE( '01/05/07' , 'DD/MM/YY' ) );
```

Comme vous le constatez, pour donner des valeurs à une table c'est la même procédure SQL MAIS nous avons mis un type dans notre table et pour compléter les attributs de ce type, il faut juste l'appeler.

ATTENTION : Si un type possède un autre type dans ses attributs, il faut faire la même opération : appeler le type et compléter ses attributs.

Après avoir bien remplis (hohoho), faut savoir rechercher les données d'un type. C'est la même procédure mais faut faire un appel spécial pour le type.

Pour chercher le nom d'un client :

```
1 SELECT p.client.Nom FROM obClient p;
```

Pour que le code marche à tous les coups, il faut donner une "abréviation" à notre table. Pourquoi ? Cela permet au code SQL de ne pas se confondre lors de l'exécution. Ici, nous avons mis comme abréviation la lettre p pour le client. Cela permettra de pouvoir cibler plus facilement ce qu'on cherche. Nous allons cibler dans ce cas l'attribut client qui est de type Personne dans la table et plus précisément nous voulons l'attribut nom du type Personne.

Pour chercher la ville d'un client ;

```
1 SELECT p.client.Adresse.Ville FROM obClient p;
```

C'est la même procédure mais pour ce cas, nous voulions adresse. Pour le chercher, il faut passer par l'attribut client de type Personne qui contient Adresse qui celui la contient l'attribut ville.

Maintenant pour modifier un type dans un table, c'est la même méthode mais faut spécifier l'appel tout simplement.

Modifions la rue d'un client tiens :

```
1 UPDATE obClient p
2 SET p.client.adresse.rue='15 rue des vagues'
3 WHERE UPPER (p.client.nom)='DUVAL';
```

Comme prévu, la méthode SQL pour modifier ne change pas mais faut juste faire la spécification comme dans la recherche.

3.3 Créer une table avec un seul type d'objet

La méthode ne change pas, juste spécifier le type d'objet à utiliser et donner une clé primaire.

```
1 CREATE TABLE nomTable OF nomTypeObjet;
```

ATTENTION : Si nous voulons créer une table avec un seul type d'objet MAIS le type concerné possède un attribut venant d'un autre type, là c'est la merde. Mais SQL a tout pensé, faut juste donner une référence à l'attribut qui possède le type.

Regardez le code ça sera plus simple à comprendre.

```
1 CREATE TYPE obIndividuTy AS OBJECT (
2   NumIndividu NUMBER(5) ,
3   Nom VARCHAR2(30) ,
4   Prenom VARCHAR2(30) );
5
6 CREATE TABLE obIndividu OF obIndividuTy
7   (NumIndividu PRIMARY KEY);
8
9 CREATE TYPE obFilmTy AS OBJECT (
10  NumFilm NUMBER(5) ,
11  Titre VARCHAR2(50) ,
12  realisateur REF obIndividuTy );
13
14 CREATE TABLE obFilm OF obFilmTy
15   (NumFilm PRIMARY KEY);
```

Comme vous le voyez, la table obFilm n'est que de type obFilmTy. Cela veut dire que cette table ne peut posséder que les attribut du type Film. Le problème c'est qu'il a le type Individu qui vient foutre la merde. Pourquoi il a un REF ?? Le REF a pour fonctionnalité de faire référence a un autre type qui possède aussi une table (ici c'est Individu). Donc avec le REF l'affaire est réglée.

Maintenant pour donner des valeurs, c'est autre chose. Il faut d'abord donner les attributs du type. Ici il faut d'abord donner les attributs de Individu avant de donner les attributs Film.

Le code pour individu :

```
1 INSERT INTO obIndividu VALUES
2 (1996, 'LECONTE', 'PATRICE');
3 INSERT INTO obIndividu VALUES
4 (2871, 'SPIELBERG', 'STEVEN');
5 INSERT INTO obIndividu VALUES
6 (2987, 'PAGNOL', 'MARCEL');
```

Ici c'est simple on ajoute des données dans la table obIndividu. Ensuite pour donner les attributs Film, c'est plus compliqué. Pour insérer les attributs, il faut donner la référence à l'individu qu'on souhaite.

```
1 INSERT INTO obFilm
2 SELECT 43, 'LE MARI DE LA COIFFEUSE',
3 REF(b) FROM obIndividu b
4 WHERE NumIndividu=1996;
5 INSERT INTO obFilm
6 SELECT 84, 'LA LISTE DE SCHINDLER',
7 REF(a) FROM obIndividu a
8 WHERE NumIndividu=2871;
9 INSERT INTO obFilm
10 SELECT 18, 'CESAR',
11 REF(c) FROM obIndividu c
12 WHERE NumIndividu=2987;
```

Ce code nous explique que lorsqu'on ajoute un film, il faut faire référence (avec REF) sur l'individu qu'on souhaite car sinon, le film ne contient aucun réalisateur, c'est similaire à NullPointerException de mes c...

3.4 Création d'une spécification et d'un corps par des méthodes

Une méthode permet de manipuler des types et de définir leur comportement à n'importe quel moment. Une méthode ne peut cependant effectuer des modifications sur la base de données.

Pour créer un type avec sa spécification :

```
1 CREATE [OR REPLACE] TYPE nomType AS OBJECT
2 (nomAttribut type [, nomAttribut type ? ] ,
3 MEMBER specification de procedure | fonction
4 [MEMBER specification de procedure | fonction ? ]
5 [MAP MEMBER specification de fonction ]
6 [ , PRAGMA RESTRICT REFERENCES (
7 methode, mode [ , mode ? ] ) );
```

MAP : pour définir un critère de tri. mode : pour augmenter/restreindre les possibilités des méthodes (WNDS, WNPS, RNDS, RNPS).

Exemple avec le type Personne et une fonction qui détermine si la Personne habite en Ile de France ou en Province :

```

1 CREATE [OR REPLACE] TYPE obPersonneTy AS OBJECT
2   Nom VARCHAR2(30) ,
3   Prenom VARCHAR2(30) ,
4   Adresse obAdresseTy)
5 MEMBER FUNCTION parisProvince RETURN VARCHAR2
6 [ , PRAGMA RESTRICT REFERENCES(
7   parisProvince ,WNDS,RNDS) );

```

WNDS : pas de modification de la base de données RNDS : ne doit pas consulter la base de données.

Maintenant faut définir le corps :

```

1 CREATE [OR REPLACE] TYPE BODY nomType IS |AS
2 MEMBER corps de la procedure | fonction
3 [MEMBER corps de la procedure | fonction ? ]
4 [MAP MEMBER corps de la fonction de tri ]

```

Dans notre exemple nous voulons définir la fonction qui détermine où la Personne habite :

```

1 CREATE OR REPLACE TYPE BODY obPersonneTy IS
2 MEMBER FUNCTION parisProvince RETURN VARCHAR2 IS
3   pp VARCHAR2(8)
4   BEGIN
5       if (SUBSTR(adresse.CodePostal,1,2) IN
6           ('77','78','75','91','92','93','94','95'))
7           then pp:='idf';
8           else pp:='province';
9       end if;
10      return pp
11      END parisProvince;
12 END;

```

Nous pouvons faire pareil avec une fonction de tri.
Spécification :

```

1 CREATE OR REPLACE TYPE obAdresseTy AS OBJECT
2   Rue VARCHAR2(50) ,
3   Ville VARCHAR2(30) ,
4   Code Postal CHAR(5)
5   Pays VARCHAR2(15) ,
6   MAP MEMBER FUNCTION triAdresse RETURN VARCHAR2);

```

Corps :

```

1 CREATE OR REPLACE TYPE BODY obAdresseTy IS
2   MAP MEMBER FUNCTION triAdresse RETURN VARCHAR2 IS
3   BEGIN
4       RETURN CodePostal || Ville;
5   END triAdresse;
6 END;

```

Recherche :

```
1 SELECT p.client.Nom,  
2        p.client.parisProvince situation ,  
3        p.client.adresse.codePostal code FROM obClient p  
4 ORDER BY p.client.adresse;
```