**Collections et enregistrements**

Ces types de données n'existent qu'en PL/SQL   
  
**Les collections**   
  
Une collection est un ensemble ordonné d'éléments de même type.   
Elle est indexée par une valeur de type numérique ou alphanumérique  
Elle ne peut avoir qu'une seule dimension (en créant des collections de collections on peut obtenir des tableaux à plusieurs dimensions)   
  
On peut distinguer trois types différents de collections :

* **Les tables (INDEX-BY TABLES) qui peuvent être indicées par des variables numériques ou alpha-numériques**
* **Les tables imbriquées(NESTED TABLES) qui sont indicées par des variables numériques et peuvent être lues et écrites directement depuis les colonnes d'une table**
* **Les tableaux de type VARRAY, indicés par des variables numériques, dont le nombre d'éléments maximum est fixé dès leur déclaration et peuvent être lus et écrits directement depuis les colonnes d'une table**

Les collections de type NESTED TABLE et VARRAY doivent-être initialisées après leur déclaration, à l'aide de leur constructeur qui porte le même nom que la collection  
(elles sont assignées à NULL lors de leur déclaration. Il est donc possible de tester leur nullité)

**Les enregistrements**   
  
Il est composé de champs qui peuvent être de type différent

**Déclarations et initialisation**

* **Les collections de type NESTED TABLE et INDEX-BY TABLES**

Elles sont de taille dynamique et il n'existe pas forcément de valeur pour toutes les positions   
  
Déclaration d'une collection de type NESTED TABLE  
**TYPE nom type IS TABLE OF type élément [NOT NULL] ;**   
  
Déclaration d'une collection de type INDEX BY  
**TYPE nom type IS TABLE OF type élément [NOT NULL] INDEX BY index\_by\_type ;**   
**index\_by\_type** représente l'un des types suivants :

* BINARY\_INTEGER
* PLS\_INTEGER(9i)
* VARCHAR2(taille)
* LONG

declare

-- collection de type nested table

TYPE NES\_TAB is table of varchar2(100) ;

-- collection de type index by

TYPE IND\_TAB is table of number index by binary\_integer ;

tab1 NES\_TAB ;

tab2 IND\_TAB ;

Begin

tab1 := NES\_TAB('Lundi','Mardi','Mercredi','Jeudi' ) ;

for i in 1..10 loop

tab2(i):= i ;

end loop ;

dbms\_output.put\_line(tab1(1)||' : '||tab2(2));

End;

/

* **Les collections de type VARRAY**

Ce type de collection possède une dimension maximale qui doit être précisée lors de sa déclaration  
Elle possède une longueur fixe et donc la suppression d'éléments ne permet pas de gagner de place en mémoire  
Ses éléments sont numérotés à partir de la valeur 1   
  
Déclaration d'une collection de type VARRAY  
**TYPE nom type IS VARRAY (taille maximum) OF type élément [NOT NULL] ;**

declare

-- collection de type VARRAY

TYPE VAR\_TAB is VARRAY(30) of varchar2(100) ;

tab1 VAR\_TAB := VAR\_TAB('','','','','','','','','','');

Begin

for i in 1..10 loop

tab1(i):= to\_char(i) ;

end loop ;

dbms\_output.put\_line(tab1(3));

End;

/

* **Les enregistrements**

**TYPE nom type IS RECORD ( nom\_champ type\_élément [[ NOT NULL] := expression ] [, ….] ) ;**  
**Nom\_variable nom\_type ;**   
  
Comme pour la déclaration des variables, il est possible d'initialiser les champs lors de leur déclaration

declare

-- Record --

TYPE REC\_EMP IS RECORD (

Num emp.empno%TYPE,

Nom emp.ename%TYPE,

Job emp.job%TYPE );

R\_EMP REC\_EMP ; -- variable enregistrement de type REC\_EMP

Begin

R\_EMP.Num := 1 ;

R\_EMP.Nom := 'JP' ;

R\_EMP.job := 'INFO' ;

dbms\_output.put\_line(R\_EMP.NUM||' : '||R\_EMP.NOM||' : '||R\_EMP.JOB);

End;

/

Bien sûr il est possible de gérer des tableaux d'enregistrements

declare

-- Record --

TYPE REC\_EMP IS RECORD (

Num emp.empno%TYPE,

Nom emp.ename%TYPE,

Job emp.job%TYPE );

-- Table de records --

TYPE T\_REC\_EMP IS TABLE OF REC\_EMP index by binary\_integer ;

t\_rec T\_REC\_EMP ; -- variable tableau d'enregistrements

Begin

t\_rec(1).Num := 1 ;

t\_rec(1).Nom := 'JP' ;

t\_rec(1).job := 'INFO' ;

t\_rec(2).Num := 2 ;

t\_rec(2).Nom := 'MAF' ;

t\_rec(2).job := 'RECH' ;

dbms\_output.put\_line(t\_rec(1).NOM||' : '||t\_rec(2).NOM);

End;

/

Les éléments d'un enregistrement peuvent être des objets, des collections ou d'autres enregistrements.

Declare

TYPE Temps IS RECORD

( heures SMALLINT,

minutes SMALLINT,

secondes SMALLINT );

TYPE Vol IS RECORD

( numvol PLS\_INTEGER,

Numavion VARCHAR2(15),

Commandant Employe, -- type objet

Passagers ListClients, -- type nested table

depart Temps, -- type record

arrivee Temps -- type record

);

Begin

...

End ;

A la différence des types VARRAY et NESTED TABLE, les types RECORD ne peuvent pas être créés et stockés dans la base.

**Initialisation des collections**   
  
Les collections de type NESTED TABLE et VARRAY doivent être initialisées avant toute utilisation (à l'exception des collections de type INDEX-BY TABLE).   
  
Declare

-- Déclaration d'un type tableau VARRAY de 30 éléments de type Varchar2(100)

TYPE TYP\_VAR\_TAB is VARRAY(30) of varchar2(100) ;

-- Déclaration et initialisation d'une variable de type TYP\_VAR\_TAB

tab1 TYP\_VAR\_TAB := TYP\_VAR\_TAB('','','','','','','','','','');

Il n'est pas obligatoire d'initialiser tous les éléments d'une collection. On peut même n'en initialiser aucun. Dans ce cas l'appel de la méthode se fait sans argument.   
  
tab1 TYP\_VAR\_TAB := TYP\_VAR\_TAB();   
  
Une collection non initialisée n'est pas vide mais NULL.

Declare

TYPE TYP\_VAR\_TAB is VARRAY(30) of varchar2(100) ;

tab1 TYP\_VAR\_TAB; -- collection NULL

L'initialisation d'une collection peut se faire dans la section instructions, mais dans tous les cas, elle ne pourra pas être utilisée avant d'avoir été initialisée.

Declare

TYPE TYP\_VAR\_TAB is VARRAY(30) of varchar2(100) ;

tab1 TYP\_VAR\_TAB ; -- collection automatiquement assignée à NULL

Begin

-- La collection est assignée à NULL mais n'est pas manipulable --

If Tab1 is null Then -- Test OK

Dbms\_output.put\_line('COLL Nulle');

End if ;

Tab1 := TYP\_VAR\_TAB('','','','','','','','','','');

-- La collection est manipulable –

Dbms\_output.put\_line(tab1(1)) ;

End ;

COLL Nulle

**5.2. Accès aux éléments d'une collection**

La syntaxe d'accès à un élément d'une collection est la suivante :   
  
**Nom\_collection( indice )**   
  
L'indice doit être un nombre valide compris entre -2\*\*31 et 2\*\*31  
Pour une collection de type NESTED TABLE, l'indice doit être un nombre valide compris entre 1 et 2\*\*31  
Pour une collection de type VARRAY, l'indice doit être un nombre valide compris entre 1 et la taille maximum du tableau  
Dans le cas d'une collection de type INDEX-BY Varchar2 ou Long, l'indice représente toute valeur possible du type concerné.  
  
Indice peut être un littéral, une variable ou une expression

Declare

Type TYPE\_TAB\_EMP IS TABLE OF Varchar2(60) INDEX BY BINARY\_INTEGER ;

emp\_tab TYPE\_TAB\_EMP ;

Begin

For i in 0..5 Loop

emp\_tab( i+1 ) := 'Emp ' || ltrim( to\_char( i ) ) ;

Dbms\_output.put\_line(emp\_tab(i+1));

End loop ;

Dbms\_output.put\_line(emp\_tab(4));

End ;

/

Declare

Type TYPE\_TAB\_JOURS IS TABLE OF PLS\_INTEGER INDEX BY VARCHAR2(20) ;

jour\_tab TYPE\_TAB\_JOURS ;

Begin

jour\_tab( 'LUNDI' ) := 10 ;

jour\_tab( 'MARDI' ) := 20 ;

jour\_tab( 'MERCREDI' ) := 30 ;

Dbms\_output.put\_line(jour\_tab(‘MARDI’));

End ;

/

Il est possible d'assigner une collection à une autre à condition qu'elles soient de même type

Declare

Type TYPE\_TAB\_EMP IS TABLE OF EMP%ROWTYPE INDEX BY BINARY\_INTEGER ;

Type TYPE\_TAB\_EMP2 IS TABLE OF EMP%ROWTYPE INDEX BY BINARY\_INTEGER ;

tab1 TYPE\_TAB\_EMP := TYPE\_TAB\_EMP( ... );

tab2 TYPE\_TAB\_EMP := TYPE\_TAB\_EMP( ... );

tab3 TYPE\_TAB\_EMP2 := TYPE\_TAB\_EMP2( ... );

Begin

tab2 := tab1 ; -- OK

tab3 := tab1 ; -- Illégal : types différents

...

End ;

(10g)  
Les collections de même type peuvent être comparées en égalité ou différence

DECLARE

TYPE Colors IS TABLE OF VARCHAR2(64);

primaries Colors := Colors('Blue','Green','Red');

rgb Colors := Colors('Red','Green','Blue');

traffic\_light Colors := Colors('Red','Green','Amber');

BEGIN

-- On peut utiliser = ou !=, mais pas < ou >.

-- Notez que ces 2 collections sont égales même si leurs membres sont dans un ordre différent.

IF primaries = rgb THEN

dbms\_output.put\_line('OK, PRIMARIES et RGB ont les mêmes membres.');

END IF;

IF rgb != traffic\_light THEN

dbms\_output.put\_line('RGB et TRAFFIC\_LIGHT ont des membres différents.');

END IF;

END;

(10g)  
Il est possible d'appliquer certains opérateurs sur des tables imbriquées

DECLARE

TYPE nested\_typ IS TABLE OF NUMBER;

nt1 nested\_typ := nested\_typ(1,2,3);

nt2 nested\_typ := nested\_typ(3,2,1);

nt3 nested\_typ := nested\_typ(2,3,1,3);

nt4 nested\_typ := nested\_typ(1,2,4);

reponse BOOLEAN;

combien NUMBER;

PROCEDURE verif(test BOOLEAN DEFAULT NULL, quantite NUMBER DEFAULT NULL) IS

BEGIN

IF test IS NOT NULL THEN

dbms\_output.put\_line(CASE test WHEN TRUE THEN 'True' WHEN FALSE THEN 'False' END);

END IF;

IF quantite IS NOT NULL THEN

dbms\_output.put\_line(quantite);

END IF;

END;

BEGIN

reponse := nt1 IN (nt2,nt3,nt4); -- true, nt1 correspond à nt2

verif(test => reponse);

reponse := nt1 SUBMULTISET OF nt3; -- true, tous les éléments correspondent

verif(test => reponse);

reponse := nt1 NOT SUBMULTISET OF nt4; -- true

verif(test => reponse);

combien := CARDINALITY(nt3); -- nombre d'éléments dans nt3

verif(quantite => combien);

combien := CARDINALITY(SET(nt3)); -- nombre d'éléments distincts

verif(quantite => combien);

reponse := 4 MEMBER OF nt1; -- false, aucun élément ne correspond

verif(test => reponse);

reponse := nt3 IS A SET; -- false, nt3 a des éléments dupliqués

verif(test => reponse);

reponse := nt3 IS NOT A SET; -- true, nt3 a des éléments dupliqués

verif(test => reponse);

reponse := nt1 IS EMPTY; -- false, nt1 a des éléments

verif(test => reponse);

END;

**5.3. Méthodes associées aux collections**

Les méthodes sont des fonctions ou des procédures qui s'appliquent uniquement aux collections.  
L'appel de ces méthodes s'effectue en préfixant le nom de la méthode par le nom de la collection   
  
**Nom\_collection.nom\_méthode[(paramètre, …)]**   
  
Les méthodes ne peuvent pas être utilisées à l'intérieur de commandes SQL   
  
Seule la méthode EXISTS peut être utilisée sur une collection NULL.  
L'utilisation de toute autre méthode sur une collection NULL provoque l'exception COLLECTION\_IS\_NULL

**EXISTS(indice)**   
  
Cette méthode retourne la valeur TRUE si l'élément indice de la collection existe et retourne la valeur FALSE dans le cas contraire   
  
Cette méthode doit être utilisée afin de s'assurer que l'on va réaliser une opération conforme sur la collection   
  
Le test d'existence d'un élément qui n'appartient pas à la collection ne provoque pas l'exception SUBSCRIPT\_OUTSIDE\_LIMIT mais retourne simplement FALSE

**COUNT**   
  
Cette méthode retourne le nombre d'éléments de la collection y compris les éléments NULL consécutifs à des suppressions  
Elle est particulièrement utile pour effectuer des traitements sur l'ensemble des éléments d'une collection.

**LIMIT**   
  
Cette méthode retourne le nombre maximum d'éléments permis d'une collection  
Elle n'est utile que pour les collections de type VARRAY et retourne NULL pour les collections des autre types

**FIRST**   
  
Cette méthode retourne le plus petit indice d'une collection.  
Elle retourne NULL si la collection est vide  
Pour une collection de type VARRAY cette méthode retourne toujours 1

**LAST**   
  
Cette méthode retourne le plus grand indice d'une collection.  
Elle retourne NULL si la collection est vide  
Pour une collection de type VARRAY cette méthode retourne la même valeur que la méthode COUNT

**PRIOR(indice)**   
  
Cette méthode retourne l'indice de l'élément précédent l'indice donné en argument  
Elle retourne NULL si indice est le premier élément de la collection

**NEXT(indice)**   
  
Cette méthode retourne l'indice de l'élément suivant l'indice donné en argument  
Elle retourne NULL si indice est le dernier élément de la collection

**EXTEND[(n[,i])]**   
  
Cette méthode permet d'étendre une collection par ajout de nouveaux éléments  
  
Elle dispose de 3 syntaxes différentes

* **EXTEND**

Un seul élément NULL est ajouté à la collection

* **EXTEND(n)**

n éléments NULL sont ajoutés à la collection

* **EXTEND(n,i)**

n éléments sont ajoutés à la collection. Chaque élément ajouté contient une copie de la valeur contenue dans l'élément d'indice i

**TRIM[(n)]**   
  
Cette méthode permet de supprimer un ou plusieurs éléments situés à la fin d'une collection  
Elle dispose de 2 formes de syntaxe différentes   
  
**TRIM**   
  
Le dernier élément de la collection est supprimé

**TRIM(n)**   
  
Les n derniers éléments de la collection sont supprimés

Si le nombre d'éléments que l'on veut supprimer est supérieur au nombre total d'éléments de la collection,  
une exception SUBSCRIPT\_BEYOND\_COUNT est générée

**DELETE[(n[,m])]**   
  
Cette méthode permet de supprimer un, plusieurs, ou la totalité des éléments d'une collection  
Elle dispose de 3 formes de syntaxe différentes

* **DELETE**

Suppression de tous les éléments d'une collection

* **DELETE(n)**

Suppression de l'élément d'indice n de la collection

On peut observer que l'élément d'indice 5 de la collection, une fois supprimé, ne peut plus être affiché.  
Il convient, lorsque l'on supprime un ou plusieurs éléments d'une collection des tester l'existence d'une valeur avant de la manipuler

Il est important de noter le décalage entre la valeur retournée par la méthode COUNT et celle retournée par la méthode LAST  
Dans l'exemple précédent LAST retourne la plus grande valeur d'indice de la collection soit 7, alors que COUNT retourne le nombre d'éléments de la collection soit 6   
  
Méfiez-vous de l'erreur facile consistant à penser que COUNT = LAST

* **DELETE(n,m)**

Suppression des l'éléments dont les indices sont compris entre n et m (inclus) Si m est plus grand que n, aucun élément n'est supprimé

**Pour les collections de type VARRAY on ne peut supprimer que le dernier élément**Si l'élément à supprimer n'existe pas, aucune exception n'est générée  
L'espace mémoire assigné aux éléments supprimés est conservé. Il est tout à fait permis de réassigner une nouvelle valeur à ces éléments.

**EXEMPLES :**

DECLARE

type t\_liste is varray(5) of integer;

L1 t\_liste:=t\_liste(5,9,12);

begin

if L1.exists(4) then

dbms\_output.put\_line('Ok, existe');

else

dbms\_output.put\_line('Aie, n''existe pas');

end if;

dbms\_output.put\_line(L1.count);

for i in 1..L1.limit loop

dbms\_output.put\_line(i);

end loop;

dbms\_output.put\_line(L1.first);

dbms\_output.put\_line(L1.last);

dbms\_output.put\_line(L1(L1.prior(2)));

dbms\_output.put\_line(L1(L1.next(2)));

L1.extend;

L1(4):=16;

dbms\_output.put\_line(L1(4));

L1.extend(1);

L1(5):=20;

dbms\_output.put\_line(L1(5));

dbms\_output.put\_line(L1.count);

L1.trim;

dbms\_output.put\_line(L1.count);

-- dbms\_output.put\_line(L1(5)); -- erreur subscript\_beyond\_count

L1.trim(2);

dbms\_output.put\_line(L1.count);

-- dbms\_output.put\_line(L1(4)); -- erreur subscript\_beyond\_count

-- L1.trim(8); -- erreur subscript\_beyond\_count

L1.delete;

dbms\_output.put\_line(L1.count);

EXCEPTION

WHEN SUBSCRIPT\_BEYOND\_COUNT then

dbms\_output.put\_line('Vous voulez supprimer trop d''éléments');

end;

/

**Principales exceptions**

Declare

TYPE TYP\_TAB is table of varchar2(100) ;

tab TYP\_TAB ;

lc$valeur varchar2(100) ;

Begin

tab(1) := 'Lundi' ; -- ORA-06531: Référence à un ensemble non initialisé

tab := TYP\_TAB( 'lundi','mardi','mercredi' ) ;

tab.EXTEND(4,1) ;

tab.DELETE(4,6) ;

lc$valeur := tab(4) ; -- ORA-01403: Aucune donnée trouvée

tab(0) :='lunmanche' ; -- ORA-06532: Indice hors limites

tab(22) :='marcredi' ; -- ORA-06533: Valeur de l'indice trop grande

lc$valeur := tab(999999999999999999) ; -- ORA-01426: dépassement numérique

lc$valeur := tab(NULL) ; -- ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur:

la valeur de clé de la table d'index est NULL.

End ;

**5.4. Utilisation des collections avec les données issues de la base**

Prenons l'exemple d'une table des entêtes de factures qui stocke également les lignes des factures   
  
Définissons le type ligne de facture (TYP\_LIG\_FAC)

CREATE TYPE TYP\_LIG\_FAC AS OBJECT (

numlig Integer,

code Varchar2(20),

Pht Number(6),

Tva Number(3),

Qte Integer );

Définissons le type TYP\_TAB\_LIG\_FAC comme étant une collection d'éléments du type TYP\_LIG\_FAC

CREATE TYPE TYP\_TAB\_LIG\_FAC AS TABLE OF TYP\_LIG\_FAC ;

Création de la table des factures

CREATE TABLE FACTURE (

numero Number(9),

numcli Number(6),

date\_fac Date,

ligne TYP\_TAB\_LIG\_FAC )

NESTED TABLE ligne STORE AS ligne\_table ;

Chaque élément présent dans la colonne ligne est une collection de type NESTED TABLE qui va permettre de stocker les différentes lignes de la facture   
  
Insertion de données dans la table FACTURE

Begin

Insert into FACTURE

values( 1, 1214, sysdate,

Typ\_tab\_lig\_fac( Typ\_lig\_fac( 1, 'Oracle 9i', 999, 55, 3 ),

Typ\_lig\_fac( 1, 'Forms 9i', 899, 55, 3 ),

Typ\_lig\_fac( 1, 'Reports 9i', 699, 55, 3 )

)

);

Insert into FACTURE

values( 2, 1215, sysdate,

Typ\_tab\_lig\_fac( Typ\_lig\_fac( 1, 'Oracle 9i', 999, 55, 1 ),

Typ\_lig\_fac( 1, 'Forms 9i', 899, 55, 1 ),

Typ\_lig\_fac( 1, 'Reports 9i', 699, 55, 1 )

)

);

End ;

/

commit;

Modification d'une facture

Declare

Tab\_lig Typ\_tab\_lig\_fac := Typ\_tab\_lig\_fac(

Typ\_lig\_fac( 1, 'Forms 9i', 899, 55, 2 ),

Typ\_lig\_fac( 1, 'Reports 9i', 699, 55, 2 )

);

Begin

Update FACTURE Set ligne = Tab\_lig Where numero = 2 ;

End;

/

commit;

Utilisation d'un type enregistrement (RECORD) pour sélectionner une ligne de la table FACTURE ainsi que toutes les lignes rattachées via la NESTED TABLE

Declare

TYPE Fact\_rec IS RECORD

( numero facture.NUMERO%type,

numcli facture.NUMCLI%type,

date\_fac facture.DATE\_FAC%type,

lignes facture.LIGNE%type ) ;

rec\_fact Fact\_rec ;

Cursor C\_Fact is select \* from facture ;

Begin

Open C\_Fact ;

Loop

Fetch C\_Fact into rec\_fact ;

Exit when C\_Fact%NOTFOUND ;

For i IN 1..rec\_fact.lignes.last Loop

dbms\_output.put\_line( 'Numcli/Numfac ' || rec\_fact.numcli || '/' || rec\_fact.numero || ' Ligne ' || rec\_fact.lignes(i).numlig || ' code ' || rec\_fact.lignes(i).code || ' Qté ' || To\_char(rec\_fact.lignes(i).qte) ) ;

End loop ;

End loop ;

End ;

/

Le champ lignes de l'enregistrement est déclaré de type LIGNE%type donc de type TYP\_LIG\_FAC.  
On récupère dans un enregistrement l'entête de la facture ainsi que toutes les colonnes des lignes attachées.   
  
Une variable de type RECORD peut être utilisée dans une clause RETURNING INTO

Declare

TYPE Emp\_rec IS RECORD

( empno emp.empno%type,

empname emp.ename%type,

salaire emp.sal%type

);

emp\_info Emp\_rec ;

Begin

Select empno, ename, sal Into emp\_info From EMP where empno = 7499 ;

dbms\_output.put\_line( 'Ancien salaire pour ' || emp\_info.empno || ' : ' || To\_char(emp\_info.salaire) ) ;

Update EMP set sal = sal \* 1.1 Where empno = 7499

RETURNING empno, ename, sal INTO emp\_info ;

dbms\_output.put\_line( 'Nouveau salaire pour ' || emp\_info.empno || ' : ' || To\_char(emp\_info.salaire) ) ;

End ;

/

Ancien salaire pour 7499 : 1760

Nouveau salaire pour 7499 : 1936

**5.5. Traitements en masse des collections**

Les collections permettent le traitement des données en " masse "   
  
Elles permettent de charger les données d'une table, de les traiter puis de les enregistrer dans la base afin de limiter les interactions coûteuses entre le moteur PL/SQL et le moteur SQL,  
les collections peuvent être traitées intégralement grâce à la copie des données par blocs   
  
Cette copie des données par blocs autorise un ordre SQL à traiter toute la collection grâce aux instructions BULK COLLECT et FORALL.   
FORALL pour INSERT, UPDATE ou DELETE.

Pour s'en convaincre, analysons le code suivant :

create table test(a number, B number);

truncate table test;

alter system flush shared\_pool;

alter system flush buffer\_cache;

-- Test boucle FOR

Declare

TYPE TYP\_TAB\_NUM IS TABLE OF TEST.B%TYPE INDEX BY PLS\_INTEGER ;

TYPE TYP\_TAB\_CAR IS TABLE OF TEST.A%TYPE INDEX BY PLS\_INTEGER ;

tab1 TYP\_TAB\_NUM ;

tab2 TYP\_TAB\_CAR ;

LN$T1 number ;

LN$T2 number ;

LN$T3 number ;

begin

For i in 1..300000 Loop

tab1(i) := i ;

tab2(i) := ltrim( to\_char( i ) ) ;

End loop ;

Select to\_char( sysdate, 'SSSSS' ) into LN$T1 from dual ;

For i in 1..300000 Loop

Insert into TEST( A, B ) Values( tab2(i), tab1(i) ) ;

End loop ;

Select to\_char( sysdate, 'SSSSS' ) into LN$T2 from dual ;

dbms\_output.put\_line( 'Temps d''exécution en secondes' ) ;

dbms\_output.put\_line( 'For ' || to\_char(LN$T2 - LN$T1) ) ;

End ;

/

--Temps d'exécution en secondes

--For 14

truncate table test;

alter system flush shared\_pool;

alter system flush buffer\_cache;

-- Test boucle FORALL

Declare

TYPE TYP\_TAB\_NUM IS TABLE OF TEST.B%TYPE INDEX BY PLS\_INTEGER ;

TYPE TYP\_TAB\_CAR IS TABLE OF TEST.A%TYPE INDEX BY PLS\_INTEGER ;

tab1 TYP\_TAB\_NUM ;

tab2 TYP\_TAB\_CAR ;

LN$T1 number ;

LN$T2 number ;

LN$T3 number ;

begin

For i in 1..300000 Loop

tab1(i) := i ;

tab2(i) := ltrim( to\_char( i ) ) ;

End loop ;

Select to\_char( sysdate, 'SSSSS' ) into LN$T1 from dual ;

Forall i in 1..300000

Insert into TEST( A, B ) Values( tab2(i), tab1(i) ) ;

Select to\_char( sysdate, 'SSSSS' ) into LN$T2 from dual ;

dbms\_output.put\_line( 'Temps d''exécution en secondes' ) ;

dbms\_output.put\_line( 'Forall ' || to\_char(LN$T2 - LN$T1) ) ;

End ;

/

--Temps d'exécution en secondes

--Forall 1

Les deux parties de code réalisent exactement la même opération soit l'insertion de 300000 lignes dans une table.  
Cependant les temps d'exécutions respectifs sont sans commune mesure  
La différence s'explique uniquement par la charge de travail générée par les passages entre le moteur PL/SQL et le moteur SQL   
  
**BULK COLLECT**   
  
**(Select)(Fetch)(execute immediate) … BULK COLLECT Into nom\_collection [,nom\_collection, …] [LIMIT nombre\_lignes] ;**   
  
Ce mot clé demande au moteur SQL de retourner l'ensemble des lignes lues dans une ou plusieurs collections avant de rendre la main au moteur PL/SQL.  
Cette fonctionnalité réduit donc considérablement les allers-retours entre les deux moteurs.   
  
Dans le cas d'une instruction FETCH, la clause optionnelle LIMIT permet de restreindre le nombre de lignes ramenées.  
Ce nombre de lignes doit être exprimé sous forme de littéral ou de variable   
  
Dans l'exemple suivant, on alimente la collection par paquet de 3 lignes

Declare

TYPE TYP\_TAB\_EMP IS TABLE OF EMP.EMPNO%Type ;

Temp\_no TYP\_TAB\_EMP ;

Cursor C\_EMP is Select empno From EMP ;

Pass Pls\_integer := 1 ;

Begin

Open C\_EMP ;

Loop

Fetch C\_EMP BULK COLLECT into Temp\_no LIMIT 3 ;

For i In Temp\_no.first..Temp\_no.last Loop

dbms\_output.put\_line( 'Pass ' || to\_char(Pass) || ' Empno= ' || Temp\_no(i) ) ;

End loop ;

Pass := Pass + 1 ;

Exit When C\_EMP%NOTFOUND ;

End Loop ;

End ;

/

Pass 1 Empno= 7369

Pass 1 Empno= 7499

Pass 1 Empno= 7521

Pass 2 Empno= 7566

Pass 2 Empno= 7654

Pass 2 Empno= 7698

Pass 3 Empno= 7782

Pass 3 Empno= 7788

Pass 3 Empno= 7839

Pass 4 Empno= 7844

Pass 4 Empno= 7876

Pass 4 Empno= 7900

Pass 5 Empno= 7902

Pass 5 Empno= 7934

On peut également utiliser le mot clé LIMIT pour effectuer par tranches, des opérations coûteuses pour le UNDO SEGMENT

Declare

TYPE TYP\_TAB\_EMP IS TABLE OF EMP.EMPNO%Type ;

Temp\_no TYP\_TAB\_EMP ;

Cursor C\_EMP is Select empno From EMP ;

Begin

Open C\_EMP ;

Loop

Fetch C\_EMP BULK COLLECT into Temp\_no LIMIT 3 ;

Forall i In **Temp\_no.first..Temp\_no.last**

Update EMP set SAL = Round(SAL \* 2) Where empno = Temp\_no(i) ;

Commit ;

Temp\_no.DELETE ;

Exit When C\_EMP%NOTFOUND ;

End Loop ;

End ;

/

**Utilisation de INDICES OF à la place de FIRST et LAST**

Declare

TYPE TYP\_TAB\_EMP IS TABLE OF EMP.EMPNO%Type ;

Temp\_no TYP\_TAB\_EMP ;

Cursor C\_EMP is Select empno From EMP ;

Begin

Open C\_EMP ;

Loop

Fetch C\_EMP BULK COLLECT into Temp\_no LIMIT 3 ;

Forall i In **indices of temp\_no**

Update EMP set SAL = Round(SAL / 2) Where empno = Temp\_no(i) ;

Commit ;

Temp\_no.DELETE ;

Exit When C\_EMP%NOTFOUND ;

End Loop ;

End ;

/

**Pour connaitre le nombre de lignes traitées : %BULK\_ROWCOUNT (oter LIMIT 3)**

Declare

TYPE TYP\_TAB\_EMP IS TABLE OF EMP.EMPNO%Type ;

Temp\_no TYP\_TAB\_EMP ;

Cursor C\_EMP is Select empno From EMP ;

nb integer:=0;

Begin

Open C\_EMP ;

Loop

Fetch C\_EMP BULK COLLECT into Temp\_no ;

Forall i In indices of temp\_no

Update EMP set SAL = Round(SAL / 2) Where empno = Temp\_no(i) ;

Commit ;

**FOR i in temp\_no.first..temp\_no.last loop**

**nb:=nb+SQL%BULK\_ROWCOUNT(i);**

**END LOOP;**

**DBMS\_OUTPUT.PUT\_LINE(nb);**

Temp\_no.DELETE ;

Exit When C\_EMP%NOTFOUND ;

End Loop ;

End ;

/

Le mot clé BULK COLLECT peut également être utilisé pour récupérer les résultats d'un ordre DML, lorsqu'il est associé à la clause RETURNING INTO.

Declare

TYPE TYP\_TAB\_EMPNO IS TABLE OF EMP.EMPNO%Type ;

TYPE TYP\_TAB\_NOM IS TABLE OF EMP.ENAME%Type ;

Temp\_no TYP\_TAB\_EMPNO ;

Tnoms TYP\_TAB\_NOM ;

Begin

Delete From EMP where sal < 100

RETURNING empno, ename BULK COLLECT INTO Temp\_no, Tnoms ;

For i in Temp\_no.first..Temp\_no.last Loop

dbms\_output.put\_line( 'Employé viré : ' || To\_char( Temp\_no(i) ) || ' ' || Tnoms(i) ) ;

End loop ;

End ;

/

Declare

\*

ERREUR à la ligne 1 :

ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur

ORA-06512: à ligne 9

Une attention particulière doit être portée sur l'utilisation des méthodes appliquées aux collections, notamment FIRST.   
  
Dans l'exemple suivant, la procédure tombe en erreur car aucune ligne n'est retournée.  
Dans ce cas nom\_collection.FIRST ne vaut pas zéro mais NULL

Il convient de tester la nullité d'une méthode avant de l'utiliser :

Declare

TYPE TYP\_TAB\_EMPNO IS TABLE OF EMP.EMPNO%Type ;

TYPE TYP\_TAB\_NOM IS TABLE OF EMP.ENAME%Type ;

Temp\_no TYP\_TAB\_EMPNO ;

Tnoms TYP\_TAB\_NOM ;

Begin

Delete From EMP where sal < 100

RETURNING empno, ename BULK COLLECT INTO Temp\_no, Tnoms ;

If Temp\_no.first is not null Then

For i in Temp\_no.first..Temp\_no.last Loop

dbms\_output.put\_line( 'Employé viré : ' || To\_char( Temp\_no(i) ) || ' ' || Tnoms(i) ) ;

End loop ;

else

dbms\_output.put\_line('Personne');

End if ;

End ;

/

**FORALL**   
  
**FORALL index IN borne\_inférieure..borne\_supérieure [SAVE EXCEPTION] ordre\_sql**   
  
Bien que l'instruction FORALL précise une borne début et une borne fin, il ne peut pas être inclus dans une boucle FOR … LOOP   
L'instruction FORALL ne peut pas être utilisée dans le code PL/SQL coté client   
L'ordre SQL doit être INSERT, UPDATE ou DELETE en relation avec au moins une collection   
Il doit exister des éléments dans la collection pour toutes les valeurs d'indice de l'instruction FORALL   
Il n'est pas possible d'exprimer l'indice sous forme d'un calcul   
FORALL ne peut traiter qu'un seul ordre SQL à la fois sous peine de générer un message d'erreur :

Declare

TYPE TYP\_TAB\_NUM IS TABLE OF TEST.B%TYPE INDEX BY PLS\_INTEGER ;

TYPE TYP\_TAB\_CAR IS TABLE OF TEST.A%TYPE INDEX BY PLS\_INTEGER ;

tab1 TYP\_TAB\_NUM ;

tab2 TYP\_TAB\_CAR ;

begin

For i in 1..1000 Loop

tab1(i) := i ;

tab2(i) := ltrim( to\_char( i ) ) ;

End loop ;

Forall i in 1..1000

Insert into TEST( A, B ) Values( tab2(i), tab1(i) ) ;

**Delete from TEST where A = tab2(i) ;**

End ;

/

Delete from TEST where A = tab2(i) ;

\*

ERREUR à la ligne 14 :

ORA-06550: Ligne 14, colonne 40 :

PLS-00201: l'identificateur 'I' doit être déclaré

ORA-06550: Ligne 14, colonne 35 :

PL/SQL: ORA-00904: : identificateur non valide

ORA-06550: Ligne 14, colonne 8 :

PL/SQL: SQL Statement ignored

La variable d'index I n'est plus connue en sortie de l'instruction FORALL   
  
A partir d'oracle9i, les copies d'informations par blocs peuvent être effectuées directement dans les collections d'enregistrements

Declare

TYPE TYP\_TAB\_TEST IS TABLE OF TEST%ROWTYPE ;

tabrec TYP\_TAB\_TEST ;

CURSOR C\_test is select A, B From TEST ;

nb integer;

begin

-- chargement de la collection depuis la table --

Select A, B BULK COLLECT into tabrec From TEST ;

-- insertion de lignes à partir de la collection --

Forall i in tabrec.first..tabrec.last

insert into TEST values tabrec(i) ;

-- mise à jour des données de la collection --

For i in tabrec.first..tabrec.last Loop

tabrec(i).B := tabrec(i).B \* 2 ;

End loop ;

-- utilisation du curseur --

Open C\_test ;

Fetch C\_test BULK COLLECT Into tabrec ;

dbms\_output.put\_line(tabrec.count);

Close C\_test ;

End ;

/

Il n'est pas possible de mettre à jour une ligne complète avec une collection d'enregistrements en conjonction avec l'instruction FORALL

Forall i in tabrec.first..tabrec.last

update TEST set row = tabrec(i) where A = tabrec(i).A ; -- INVALIDE

Pour cela il faut utiliser une boucle For..Loop

For i in tabrec.first..tabrec.last loop

update TEST set row = tabrec(i) where A = tabrec(i).A ;

End loop ;

Ou utiliser des collections simples

Declare

TYPE TYP\_TAB\_A IS TABLE OF TEST.A%TYPE ;

TYPE TYP\_TAB\_B IS TABLE OF TEST.B%TYPE ;

taba TYP\_TAB\_A ;

tabb TYP\_TAB\_B ;

CURSOR C\_test is select A, B From TEST ;

Begin

-- utilisation du curseur --

Open C\_test ;

Fetch C\_test BULK COLLECT Into taba, tabb ;

Close C\_test ;

-- mise à jour des données de la collection --

For i in taba.first..taba.last Loop

tabb(i) := tabb(i) \* 2 ;

End loop ;

-- mise à jour des lignes de la table --

Forall i in taba.first..taba.last

update TEST set B = tabb(i) where A = taba(i) ;

End ;

**La gestion des erreurs avec la commande FORALL**   
  
Sans gestion particulière des exceptions dans le bloc PL/SQL, toute erreur d'exécution provoque l'annulation de toutes  
les instructions réalisées par l'instruction FORALL (ROLLBACK)  
En présence d'une section Exception dans le bloc PL/SQL, il sera possible de décider si l'on conserve les modifications valides jusqu'à l'erreur (COMMIT)  
ou si l'on annule toute l'instruction FORALL (ROLLBACK)   
  
Il est également possible, lors d'erreur sur une instruction, de sauvegarder l'information concernant l'exception et de poursuivre le processus  
Cette fonctionnalité est implémentée par l'ajout du mot clé **SAVE EXCEPTION** dans l'instruction FORALL  
Toutes les exceptions levées en cours d'exécution sont sauvées dans l'attribut %BULK\_EXCEPTIONS, qui est une collection d'enregistrements.  
Chaque enregistrement est composé de 2 champs :   
  
**%BULK\_EXCEPTIONS(n).ERROR\_INDEX** qui contient l'indice de l'itération qui a provoqué l'erreur  
**%BULK\_EXCEPTIONS(n).ERROR\_CODE** qui contient le code d'erreur   
  
Le nombre total d'exceptions générées par l'instruction FORALL est contenu dans l'attribut SQL%BULK\_EXCEPTIONS.COUNT

Create table test (a number, b number) ;

Insert into test values (1,1);

Insert into test values (2,2);

Insert into test values (3,3);

Insert into test values (4,4);

Insert into test values (5,5);

Commit;

Declare

TYPE TYP\_TAB IS TABLE OF Number ;

tab TYP\_TAB := TYP\_TAB( 2, 0, 1, 3, 0, 4, 5 ) ;

nb\_err Pls\_integer ;

Begin

Forall i in tab.first..tab.last SAVE EXCEPTIONS

delete from TEST where B = 5 / tab(i) ;

Exception

When others then

nb\_err := SQL%BULK\_EXCEPTIONS.COUNT ;

dbms\_output.put\_line( to\_char( nb\_err ) || ' Erreurs ' ) ;

For i in 1..nb\_err Loop

dbms\_output.put\_line( 'Indice ' || to\_char( SQL%BULK\_EXCEPTIONS(i).ERROR\_INDEX ) || ' Erreur Oracle : '

|| to\_char( SQL%BULK\_EXCEPTIONS(i).ERROR\_CODE ) ) ;

End loop ;

End ;

/

2 Erreurs

Indice 2 Erreur Oracle : 1476

Indice 5 Erreur Oracle : 1476

**L'attribut %BULK\_ROWCOUNT**   
  
En plus des attributs de curseur %FOUND, %NOTFOUND, %ISOPEN et %ROWCOUNT, le curseur implicite SQL dispose d'un attribut spécifique  
à l'instruction FORALL : %BULK\_ROWCOUNT   
  
Il s'agit d'une collection de type INDEX BY TABLE pour laquelle l'élément d'indice n contient le nombre de lignes affectées par l'exécution de l'ordre SQL numéro n  
Si aucune ligne n'est affectée par l'instruction numéro n alors l'attribut SQL%BULK\_ROWCOUNT(n) vaut 0

create table test\_nb (code number);

Declare

TYPE tab1 IS TABLE OF Number ;

t1 tab1 := tab1(4,8,3) ;

nb integer:=0 ;

Begin

Forall i in t1.first..t1.last

insert into TEST\_nb values (t1(i));

FOR i in t1.first..t1.last loop

nb:=nb+SQL%BULK\_ROWCOUNT(i);

end loop;

dbms\_output.put\_line(nb) ;

End ;

/

**Les collections et enregistrements en paramètres des procédures et fonctions**

Le passage de ces types d'objets se fait de façon identique qu'avec les types scalaires.   
  
Soit le package suivant :

CREATE OR REPLACE PACKAGE PKG\_TEST IS

TYPE TYP\_TAB\_EMP IS TABLE OF EMP%ROWTYPE ;

TYPE TYP\_TAB\_EMP2 IS TABLE OF EMP%ROWTYPE INDEX BY PLS\_INTEGER ;

PROCEDURE Affiche\_lignes ( Temp IN TYP\_TAB\_EMP ) ;

PROCEDURE Affiche\_ligne ( Temp IN OUT EMP%ROWTYPE ) ;

PROCEDURE Affiche\_ligne\_nocopy ( Temp IN OUT NOCOPY EMP%ROWTYPE ) ;

PROCEDURE Modifie\_lignes ( Temp IN OUT TYP\_TAB\_EMP ) ;

PROCEDURE Modifie\_lignes\_nocopy ( Temp IN OUT NOCOPY TYP\_TAB\_EMP ) ;

PROCEDURE Modifie\_lignes2 ( Temp IN OUT TYP\_TAB\_EMP2 ) ;

PROCEDURE Modifie\_lignes\_nocopy2 ( Temp IN OUT NOCOPY TYP\_TAB\_EMP2 ) ;

FUNCTION Affiche\_lignes Return TYP\_TAB\_EMP ;

END;

/

CREATE OR REPLACE PACKAGE BODY PKG\_TEST IS

GN$Lig pls\_integer := 0 ;

PROCEDURE Affiche\_lignes ( Temp IN TYP\_TAB\_EMP ) IS

Begin

GN$Lig := 0 ;

For i IN Temp.first..Temp.last Loop

GN$Lig := GN$Lig + 1 ;

If GN$Lig <= 10 Then

dbms\_output.put\_line(Rpad(Temp(i).ename,25)||' -> '||To\_char(Temp(i).sal));

End if ;

End loop ;

End Affiche\_lignes ;

PROCEDURE Affiche\_ligne ( Temp IN OUT EMP%ROWTYPE ) IS

Begin

GN$Lig := GN$Lig + 1 ;

Temp.sal := Temp.sal \* 1.1 ;

If GN$Lig <= 10 Then

dbms\_output.put\_line(Rpad(Temp.ename,25)||' -> '||To\_char( Temp.sal ) ) ;

End if ;

End Affiche\_ligne ;

PROCEDURE Affiche\_ligne\_nocopy ( Temp IN OUT NOCOPY EMP%ROWTYPE) IS

Begin

GN$Lig := GN$Lig + 1 ;

Temp.sal := Temp.sal \* 1.1 ;

If GN$Lig <= 10 Then

dbms\_output.put\_line(Rpad( Temp.ename,25)||' -> '||To\_char(Temp.sal )) ;

End if ;

End Affiche\_ligne\_nocopy ;

PROCEDURE Modifie\_lignes ( Temp IN OUT TYP\_TAB\_EMP ) IS

Begin

GN$Lig := 0 ;

For i IN Temp.first..Temp.last Loop

Temp(i).sal := Temp(i).sal \* 1.1 ;

Affiche\_ligne( Temp(i) ) ;

End loop ;

End Modifie\_lignes ;

PROCEDURE Modifie\_lignes\_nocopy ( Temp IN OUT NOCOPY TYP\_TAB\_EMP) IS

Begin

GN$Lig := 0 ;

For i IN Temp.first..Temp.last Loop

Temp(i).sal := Temp(i).sal \* 1.1 ;

Affiche\_ligne\_nocopy( Temp(i) ) ;

End loop ;

End Modifie\_lignes\_nocopy ;

PROCEDURE Modifie\_lignes2 ( Temp IN OUT TYP\_TAB\_EMP2 ) IS

Begin

GN$Lig := 0 ;

For i IN Temp.first..Temp.last Loop

Temp(i).sal := Temp(i).sal \* 1.1 ;

Affiche\_ligne( Temp(i) ) ;

--dbms\_output.put\_line(Rpad(Temp(i).ename,25)||'->'||To\_char(Temp(i).sal));

End loop ;

End Modifie\_lignes2 ;

PROCEDURE Modifie\_lignes\_nocopy2 ( Temp IN OUT NOCOPY TYP\_TAB\_EMP2) IS

Begin

GN$Lig := 0 ;

For i IN Temp.first..Temp.last Loop

Temp(i).sal := Temp(i).sal \* 1.1 ;

Affiche\_ligne\_nocopy( Temp(i) ) ;

--dbms\_output.put\_line(Rpad(Temp(i).ename,25)||'->'||To\_char(Temp(i).sal));

End loop ;

End Modifie\_lignes\_nocopy2 ;

FUNCTION Affiche\_lignes Return TYP\_TAB\_EMP

IS

Tlignes PKG\_TEST.TYP\_TAB\_EMP ;

Cursor C\_EMP is Select \* From EMP ;

Begin

Open C\_EMP ;

Fetch C\_EMP BULK COLLECT into Tlignes ;

Close C\_EMP ;

Return( Tlignes ) ;

End ;

END;

/

**Collections en argument de procédure**   
  
Nous allons maintenant écrire un bloc PL/SQL anonyme qui utilise la fonction Affiche\_lignes()

Declare

Tlignes PKG\_TEST.TYP\_TAB\_EMP ;

Cursor C\_EMP is Select \* From EMP ;

Begin

Open C\_EMP ;

Fetch C\_EMP BULK COLLECT into Tlignes ;

Close C\_EMP ;

PKG\_TEST.Affiche\_lignes( Tlignes ) ;

End ;

/

Une variable Tlignes est déclarée avec le type PKG\_TEST.TYP\_TAB\_EMP  
qui est une collection d'enregistrements de type EMP%ROWTYPE   
  
Cette collection est alimentée par le curseur C\_EMP et transmise à la procédure qui en affiche une partie du contenu.   
Vous pouvez également ne transmettre qu'une ligne de la collection

Declare

Tlignes PKG\_TEST.TYP\_TAB\_EMP ;

Cursor C\_EMP is Select \* From EMP ;

Begin

Open C\_EMP ;

Fetch C\_EMP BULK COLLECT into Tlignes ;

Close C\_EMP ;

PKG\_TEST.Affiche\_ligne( Tlignes(2) ) ;

End ;

/

--exécuter dans sqlplus

ALLEN -> 1760

La procédure Affiche\_ligne() reçoit en argument une variable de type EMP%ROWTYPE.  
Dans l'appel, seule la deuxième ligne de la collection lui est transmise.   
  
**Pour pouvoir modifier la valeur des éléments, la collection doit être transmise en mode IN OUT**

Declare

Tlignes PKG\_TEST.TYP\_TAB\_EMP ;

Cursor C\_EMP is Select \* From EMP ;

Begin

Open C\_EMP ;

Fetch C\_EMP BULK COLLECT into Tlignes ;

Close C\_EMP ;

PKG\_TEST.Modifie\_lignes( Tlignes ) ;

End ;

/

Lors du passage de collections ou éléments de collection à des procédures ou fonctions, les arguments sont passés par copie.  
Une copie de l'objet est faite dans un espace mémoire particulier.  
Dans le cas où des collections très volumineuses sont passées en argument, où de nombreux appels sont effectués avec des collections ou éléments de collection, il est préférable de passer les arguments en mode **NOCOPY**.  
En effet dans ce mode, aucun espace mémoire supplémentaire n'est alloué puisque la variable est passée par référence (Le pointeur sur la variable)  
Voir la procédure PKG\_TEST.Affiche\_ligne\_nocopy()   
  
**Collections en retour de fonction**   
  
Soit la fonction PKG\_TEST.Affiche\_lignes

Appelée par le bloc PL/SQL suivant :

Declare

Tlignes PKG\_TEST.TYP\_TAB\_EMP ;

Begin

Tlignes := PKG\_TEST.Affiche\_lignes ;

For i IN Tlignes.first..Tlignes.last Loop

dbms\_output.put\_line( Rpad( Tlignes(i).ename, 25 ) || ' --> ' || To\_char( Tlignes(i).sal ) ) ;

End loop ;

End ;

/

La fonction Affiche\_lignes retourne une collection de type PKG\_TEST.TYP\_TAB\_EMP,  
ouvre un curseur sur la table EMP pour remplir la collection et la retourne au bloc PL/SQL appelant qui peut en afficher les valeurs   
  
**Enregistrements**   
  
Lorsque l'on veut appeler une fonction retournant un type RECORD, il faut respecter la syntaxe suivante pour référencer un champ de l'enregistrement :   
  
**Nom\_fonction(argument).nom\_champ**

Declare

TYPE Emp\_rec IS RECORD( empno emp.empno%type,

salaire emp.sal%type );

salaire emp.sal%type ;

Function xx(ln$numemp in emp.EMPNO%type) return Emp\_rec is

emp\_info EMp\_rec ;

Begin

Select empno, sal into emp\_info.empno, emp\_info.salaire

from emp where empno = ln$numemp ;

return emp\_info ;

End ;

Begin

salaire := xx(7499).salaire ;

dbms\_output.put\_line( 'Salaire de 7499 : ' || to\_char(salaire) ) ;

End ;

/

Salaire de 7499 : 1760

**Exceptions :**

DECLARE

type liste is table of real;

L1 liste;

begin

L1(1):=5;

EXCEPTION

when **collection\_is\_null** then

dbms\_output.put\_line('Initialisation obligatoire');

end;

/

DECLARE

type liste is table of real;

L1 liste;

begin

L1:=liste(5,8,21,45);

L1(5):=56;

EXCEPTION

when **subscript\_beyond\_count** then

dbms\_output.put\_line('En dehors de la plage');

end;

/