eBook Gratuit

APPRENEZ plsql

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#plsql

Table des matières

À propos 1

Chapitre 1: Commencer avec plsql 2

Remarques 2

Examples 2

Définition de PLSQL 2

Bonjour le monde 3

À propos de PLSQL 4

Différence entre% TYPE et% ROWTYPE 4

Créer ou remplacer une vue 4

Créer une table 5

Chapitre 2: Boucle 6

Syntaxe 6

Examples 6

Boucle simple 6

WHILE Boucle 6

FOR boucle 7

Chapitre 3: Collecte en vrac 9

Examples 9

Traitement des données en vrac 9

Chapitre 4: Collections et disques 10

Examples 10

Utiliser une collection comme type de retour pour une fonction fractionnée 10

Chapitre 5: Déclaration IF-THEN-ELSE 11

Syntaxe 11

Examples 11

SI DONC 11

IF-THEN-ELSE 11

IF-THEN-ELSIF-ELSE 12

Chapitre 6: Déclencheurs 13

Introduction 13

Syntaxe 13
Examples 13
Avant le déclencheur INSERT ou UPDATE 13
Chapitre 7: Des curseurs 15
Syntaxe 15
Remarques 15
Examples 15
Curseur paramétrable "FOR loop" 15
Curseur implicite "FOR loop" 15
Travailler avec SYS_REFCURSOR 16
fonction renvoyant un curseur 16
et comment l'utiliser: 16
Manipulation d'un CURSEUR 17
Chapitre 8: Gestion des exceptions 18
Introduction 18
Examples 18
Gestion des exceptions 18
Syntaxe 18
Exceptions définies en interne 19
Exceptions prédéfinies 20
Exceptions définies par l'utilisateur 21
Définir une exception personnalisée, l'élever et voir d'où elle provient 21
Gestion des exceptions d'erreur de connexion 23
Chapitre 9: Gestion des exceptions 25
Introduction 25
Examples 25
Gestion des exceptions d'erreur de connexion 25
Définir une exception personnalisée, l'élever et voir d'où elle provient 26
Chapitre 10: Les fonctions 28
Syntaxe 28
Examples 28
Générer un GUID 28

Fonctions d'appel ... 28
Chapitre 11: Modèle d'affectation et langue ... 30
Examples ... 30
Modèle d'affectation en PL / SQL ... 30
Chapitre 12: Paquets ... 32
Syntaxe ... 32
Examples ... 33
Utilisation du paquet ... 34
Surcharge ... 34
Restrictions sur la surcharge ... 35
Définissez un en-tête et un corps de package avec une fonction ... 35
Chapitre 13: Procédure PLSQL ... 37
Introduction ... 37
Examples ... 37
Syntaxe ... 37
Bonjour le monde ... 37
Paramètres In / Out ... 38
Chapitre 14: Types d'objet ... 39
Remarques ... 39
Examples ... 39
BASE_TYPE ... 39
MID_TYPE ... 40
LEAF_TYPE ... 41
Accéder aux objets stockés ... 42
Crédits ... 44

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: plsql

It is an unofficial and free plsql ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official plsql.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec plsql

Remarques

Cette section fournit une vue d'ensemble de ce qu'est plsql et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les sujets importants dans plsql, et établir un lien avec les sujets connexes. La documentation de plsql étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

Définition de PLSQL

PL / SQL (Langage procédural / Langage de requête structuré) est l'extension procédurale d'Oracle Corporation pour SQL et la base de données relationnelle Oracle. PL / SQL est disponible dans Oracle Database (depuis la version 7), la base de données en mémoire TimesTen (depuis la version 11.2.1) et IBM DB2 (depuis la version 9.7).

L'unité de base en PL / SQL s'appelle un bloc composé de trois parties: une partie déclarative, une partie exécutable et une partie de construction d'exception.

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

Déclarations - Cette section commence par le mot-clé DECLARE. C'est une section facultative et définit toutes les variables, curseurs, sous-programmes et autres éléments à utiliser dans le programme.

Commandes exécutables - Cette section est placée entre les mots-clés BEGIN et END. Il s'agit d'une section obligatoire. Il comprend les instructions PL / SQL exécutables du programme. Il doit avoir au moins une ligne de code exécutable, qui peut être juste une commande NULL pour indiquer que rien ne doit être exécuté.

Gestion des exceptions - Cette section commence par le mot-clé EXCEPTION. Cette section est à nouveau facultative et contient des exceptions qui traitent les erreurs dans le programme.

Chaque instruction PL / SQL se termine par un point-virgule (;). Les blocs PL / SQL peuvent être imbriqués dans d'autres blocs PL / SQL en utilisant BEGIN et END.

En bloc anonyme, seule la partie exécutable du bloc est requise, les autres parties ne sont pas

indispensables. Vous trouverez ci-dessous un exemple de code anonyme simple, qui ne fait rien d'autre que de produire des rapports d'erreur.

```
BEGIN
    NULL;
END;
/
```

Les instructions exécutables manquantes entraînent une erreur, car PL / SQL ne prend pas en charge les blocs vides. Par exemple, l'exécution du code ci-dessous entraîne une erreur:

```
BEGIN
END;
/
```

L'application va générer une erreur:

```
END;
*
ERROR at line 2:
ORA-06550: line 2, column 1:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
( begin case declare exit for goto if loop mod null pragma
raise return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge pipe purge
```

Symbole "*" dans la ligne sous le mot clé "END;" signifie que le bloc qui se termine par ce bloc est vide ou mal construit. Chaque bloc d'exécution a besoin d'instructions à faire, même s'il ne fait rien, comme dans notre exemple.

Bonjour le monde

```
set serveroutput on

DECLARE
   message constant varchar2(32767):= 'Hello, World!';
BEGIN
   dbms_output.put_line(message);
END;
/
```

Le `set serveroutput on` commandes `set serveroutput on` est requis dans les clients SQL * Plus et SQL Developer pour activer la sortie de `dbms_output` . Sans la commande, rien ne s'affiche.

La `end;` line signale la fin du bloc PL / SQL anonyme. Pour exécuter le code à partir de la ligne de commande SQL, vous devrez peut-être taper `/` au début de la première ligne vide après la dernière ligne du code. Lorsque le code ci-dessus est exécuté à l'invite SQL, il produit le résultat suivant:

```
Hello, World!

PL/SQL procedure successfully completed.
```

À propos de PLSQL

PL / SQL signifie Extensions du langage procédural à SQL. PL / SQL est disponible uniquement en tant que "technologie habilitante" au sein d'autres produits logiciels; il n'existe pas en tant que langage autonome. Vous pouvez utiliser PL / SQL dans la base de données relationnelle Oracle, dans le serveur Oracle et dans les outils de développement d'applications côté client, tels que Oracle Forms. Voici quelques exemples d'utilisation de PL / SQL:

1. Pour construire des procédures stockées .
2. Pour créer des déclencheurs de base de données.
3. Pour implémenter la logique côté client dans votre application Oracle Forms.
4. Pour lier une page d'accueil World Wide Web à une base de données Oracle.

Différence entre% TYPE et% ROWTYPE.

`%TYPE` : Utilisé pour déclarer un champ du même type que celui d'une colonne de la table spécifiée.

```
DECLARE
        vEmployeeName   Employee.Name%TYPE;
BEGIN
        SELECT Name
        INTO   vEmployeeName
        FROM   Employee
        WHERE  RowNum = 1;

        DBMS_OUTPUT.PUT_LINE(vEmployeeName);
END;
/
```

% ROWTYPE: Utilisé pour déclarer un enregistrement avec les mêmes types que ceux trouvés dans la table, la vue ou le curseur spécifié (= plusieurs colonnes).

```
DECLARE
        rEmployee     Employee%ROWTYPE;
BEGIN
        rEmployee.Name := 'Matt';
        rEmployee.Age := 31;

        DBMS_OUTPUT.PUT_LINE(rEmployee.Name);
        DBMS_OUTPUT.PUT_LINE(rEmployee.Age);
END;
/
```

Créer ou remplacer une vue

Dans cet exemple, nous allons créer une vue.

Une vue est principalement utilisée comme un moyen simple d'extraire des données de plusieurs tables.

Exemple 1:
Afficher avec une sélection sur une table.

```
CREATE OR REPLACE VIEW LessonView AS
       SELECT     L.*
       FROM       Lesson L;
```

Exemple 2:
Afficher avec une sélection sur plusieurs tables.

```
CREATE OR REPLACE VIEW ClassRoomLessonView AS
       SELECT     C.Id,
                  C.Name,
                  L.Subject,
                  L.Teacher
       FROM       ClassRoom C,
                  Lesson L
       WHERE      C.Id = L.ClassRoomId;
```

Pour appeler ces vues dans une requête, vous pouvez utiliser une instruction select.

```
SELECT * FROM LessonView;
SELECT * FROM ClassRoomLessonView;
```

Créer une table

Ci-dessous, nous allons créer une table avec 3 colonnes.
L' `Id` colonne doit être rempli, nous le définissons donc `NOT NULL` .
Sur la colonne `Contract` nous ajoutons également une coche afin que la seule valeur autorisée soit "Y" ou "N". Si un insert est terminé et que cette colonne n'est pas spécifiée pendant l'insertion, un "N" est inséré par défaut.

```
CREATE TABLE Employee (
        Id            NUMBER NOT NULL,
        Name          VARCHAR2(60),
        Contract      CHAR DEFAULT 'N' NOT NULL,
        ---
        CONSTRAINT p_Id PRIMARY KEY(Id),
        CONSTRAINT c_Contract CHECK (Contract IN('Y','N'))
);
```

Lire Commencer avec plsql en ligne: https://riptutorial.com/fr/plsql/topic/1962/commencer-avec-plsql

Chapitre 2: Boucle

Syntaxe

1. BOUCLE
2. [déclarations];
3. EXIT WHEN [condition pour la boucle de sortie];
4. FIN BOUCLE;

Examples

Boucle simple

```
DECLARE
v_counter NUMBER(2);

BEGIN
  v_counter := 0;
  LOOP
    v_counter := v_counter + 1;
    dbms_output.put_line('Line number' || v_counter);

    EXIT WHEN v_counter = 10;
  END LOOP;
END;
```

WHILE Boucle

La boucle WHILE est exécutée jusqu'à ce que la condition de fin soit remplie. Exemple simple:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start, first value of our iteration

  WHILE v_counter < 10 LOOP --exit condition

    dbms_output.put_line('Current iteration of loop is ' || v_counter); --show current
iteration number in dbms script output
    v_counter := v_counter + 1; --incrementation of counter value, very important step

  END LOOP; --end of loop declaration
END;
```

Cette boucle sera exécutée jusqu'à ce que la valeur courante de la variable v_counter soit inférieure à dix.

Le résultat:

```
Current iteration of loop is 0
Current iteration of loop is 1
Current iteration of loop is 2
Current iteration of loop is 3
Current iteration of loop is 4
Current iteration of loop is 5
Current iteration of loop is 6
Current iteration of loop is 7
Current iteration of loop is 8
Current iteration of loop is 9
```

La chose la plus importante est que notre boucle commence par la valeur "0", donc la première ligne de résultats est "L'itération actuelle de la boucle est 0".

FOR boucle

Loop FOR fonctionne sur des règles similaires à celles d'autres boucles. La boucle FOR est exécutée avec un nombre exact de fois et ce nombre est connu au début - les limites inférieure et supérieure sont directement définies dans le code. À chaque étape de cet exemple, la boucle est incrémentée de 1.

Exemple simple:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start, first value of our iteration, execute of variable

  FOR v_counter IN 1..10 LOOP --The point, where lower and upper point of loop statement is
declared - in this example, loop will be executed 10 times, start with value of 1

    dbms_output.put_line('Current iteration of loop is ' || v_counter); --show current
iteration number in dbms script output

  END LOOP; --end of loop declaration
END;
```

Et le résultat est:

```
Current iteration of loop is 1
Current iteration of loop is 2
Current iteration of loop is 3
Current iteration of loop is 4
Current iteration of loop is 5
Current iteration of loop is 6
Current iteration of loop is 7
Current iteration of loop is 8
Current iteration of loop is 9
Current iteration of loop is 10
```

Loop FOR a une propriété supplémentaire, qui fonctionne en sens inverse. Utiliser le mot supplémentaire 'REVERSE' dans la déclaration de la limite inférieure et supérieure de la boucle permet de le faire. Chaque exécution de la boucle décrémente la valeur de v_counter de 1.

Exemple:

```
DECLARE
v_counter NUMBER(2); --declaration of counter variable

BEGIN
  v_counter := 0; --point of start

  FOR v_counter IN REVERSE 1..10 LOOP

    dbms_output.put_line('Current iteration of loop is ' || v_counter); --show current
iteration number in dbms script output

  END LOOP; --end of loop declaration
END;
```

Et le résultat:

```
Current iteration of loop is 10
Current iteration of loop is 9
Current iteration of loop is 8
Current iteration of loop is 7
Current iteration of loop is 6
Current iteration of loop is 5
Current iteration of loop is 4
Current iteration of loop is 3
Current iteration of loop is 2
Current iteration of loop is 1
```

Lire Boucle en ligne: https://riptutorial.com/fr/plsql/topic/6157/boucle

Chapitre 3: Collecte en vrac

Examples

Traitement des données en vrac

les collections locales ne sont pas autorisées dans les instructions select. La première étape consiste donc à créer une collection de niveaux de schéma. Si la collection n'est pas au niveau du schéma et qu'elle est utilisée dans les instructions SELECT, cela provoquerait "PLS-00642: les types de collection locaux ne sont pas autorisés dans les instructions SQL"

```
CREATE OR REPLACE TYPE table1_t IS OBJECT (
a_1 INTEGER,
a_2 VARCHAR2(10)
);
```

- Autorisations concédées sur la collecte afin de pouvoir les utiliser publiquement dans la base de données

```
     GRANT EXECUTE ON table1_t TO PUBLIC;
     CREATE OR REPLACE TYPE table1_tbl_typ IS TABLE OF table1_t;
     GRANT EXECUTE ON table1_tbl_typ TO PUBLIC;
```

--tirer les données de la table dans la collection, puis parcourir la collection et imprimer les données.

```
    DECLARE
     table1_tbl table1_tbl_typ;
    BEGIN
     table1_tbl := table1_tbl_typ();
      SELECT table1_t(a_1,a_2)
      BULK COLLECT INTO table1_tbl
      FROM table1 WHERE ROWNUM<10;

     FOR rec IN (SELECT a_1 FROM TABLE(table1_tbl))--table(table1_tbl) won't give error)
     LOOP
       dbms_output.put_line('a_1'||rec.a_1);
       dbms_output.put_line('a_2'||rec.a_2);
     END LOOP;
    END;
/
```

Lire Collecte en vrac en ligne: https://riptutorial.com/fr/plsql/topic/6855/collecte-en-vrac

Chapitre 4: Collections et disques

Examples

Utiliser une collection comme type de retour pour une fonction fractionnée

Il faut déclarer le type; ici `t_my_list` ; une collection est une `TABLE OF` *`something`*

```
CREATE OR REPLACE TYPE t_my_list AS TABLE OF VARCHAR2(100);
```

Voici la fonction. Notez le `()` utilisé comme une sorte de constructeur et les mots-clés `COUNT` et `EXTEND` qui vous aident à créer et à développer votre collection.

```
CREATE OR REPLACE
FUNCTION cto_table(p_sep in Varchar2, p_list IN VARCHAR2)
  RETURN t_my_list
AS
--- this function takes a string list, element being separated by p_sep
--                                                          as separator
  l_string VARCHAR2(4000) := p_list || p_sep;
  l_sep_index PLS_INTEGER;
  l_index PLS_INTEGER := 1;
  l_tab t_my_list     := t_my_list();
BEGIN
  LOOP
    l_sep_index := INSTR(l_string, p_sep, l_index);
    EXIT
  WHEN l_sep_index = 0;
    l_tab.EXTEND;
    l_tab(l_tab.COUNT) := TRIM(SUBSTR(l_string,l_index,l_sep_index - l_index));
    l_index            := l_sep_index + 1;
  END LOOP;
  RETURN l_tab;
END cto_table;
/
```

Ensuite, vous pouvez voir le contenu de la collection avec la fonction `TABLE()` de SQL; il peut être utilisé comme une liste dans une instruction SQL `IN ( ..)` :

```
select * from A_TABLE
 where A_COLUMN in ( TABLE(cto_table('|','a|b|c|d')) )
--- gives the records where A_COLUMN in ('a', 'b', 'c', 'd') --
```

Lire Collections et disques en ligne: https://riptutorial.com/fr/plsql/topic/9779/collections-et-disques

Chapitre 5: Déclaration IF-THEN-ELSE

Syntaxe

- SI [condition 1] ALORS
- [instructions à exécuter lorsque la condition 1 est TRUE];
- ELSIF [condition 2] PUIS
- [instructions à exécuter lorsque la condition 2 est TRUE];
- AUTRE
- [instructions à exécuter lorsque les deux conditions 1 et 2 sont FALSE];
- FIN SI;

Examples

SI DONC

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 1;

  IF v_num1 > v_num2 THEN
     dbms_output.put_line('v_num1 is bigger than v_num2');
  END IF;
END;
```

IF-THEN-ELSE

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 10;

  IF v_num1 > v_num2 THEN
     dbms_output.put_line('v_num1 is bigger than v_num2');
  ELSE
    dbms_output.put_line('v_num1 is NOT bigger than v_num2');
  END IF;
END;
```

IF-THEN-ELSIF-ELSE

```
DECLARE
v_num1 NUMBER(10);
v_num2 NUMBER(10);

BEGIN
  v_num1 := 2;
  v_num2 := 2;

  IF v_num1 > v_num2 THEN
     dbms_output.put_line('v_num1 is bigger than v_num2');
  ELSIF v_num1 < v_num2 THEN
    dbms_output.put_line('v_num1 is NOT bigger than v_num2');
  ELSE
    dbms_output.put_line('v_num1 is EQUAL to v_num2');
  END IF;
END;
```

Lire Déclaration IF-THEN-ELSE en ligne: https://riptutorial.com/fr/plsql/topic/5871/declaration-if-then-else

Chapitre 6: Déclencheurs

Introduction

Introduction:

Les déclencheurs sont un concept utile en PL / SQL. Un déclencheur est un type spécial de procédure stockée qui ne nécessite pas d'être appelé explicitement par l'utilisateur. C'est un groupe d'instructions, qui est automatiquement déclenché en réponse à une action de modification de données spécifique sur une table ou une relation spécifique, ou lorsque certaines conditions spécifiées sont satisfaites. Les déclencheurs aident à maintenir l'intégrité et la sécurité des données. Ils facilitent le travail en prenant automatiquement les mesures nécessaires.

Syntaxe

- CREATE [OU REPLACE] TRIGGER nom_transaction
- AVANT UPDATE [ou INSERT] [ou DELETE]
- ON nom_table
- [POUR CHAQUE RANG]
- DÉCLARER
- - déclarations de variables
- COMMENCER
- - code de déclenchement
- EXCEPTION
- QUAND ...
- -- gestion des exceptions
- FIN;

Examples

Avant le déclencheur INSERT ou UPDATE

```
CREATE OR REPLACE TRIGGER CORE_MANUAL_BIUR
  BEFORE INSERT OR UPDATE ON CORE_MANUAL
  FOR EACH ROW
BEGIN
  if inserting then
    -- only set the current date if it is not specified
    if :new.created is null then
      :new.created := sysdate;
    end if;
  end if;

  -- always set the modified date to now
  if inserting or updating then
    :new.modified := sysdate;
  end if;
```

```
end;
/
```

Lire Déclencheurs en ligne: https://riptutorial.com/fr/plsql/topic/7674/declencheurs

Chapitre 7: Des curseurs

Syntaxe

- Curseur *nom_curseur* est *votre_sélection_de_sélection*
- Curseur *nom_curseur* (*paramètre* TYPE) est-ce que *votre_select_statement_using_param*
- FOR x in (*your_select_statement*) LOOP ...

Remarques

Les curseurs déclarés sont difficiles à utiliser et vous devriez préférer les boucles `FOR` dans la plupart des cas. Ce qui est très intéressant dans les curseurs par rapport aux boucles `FOR` simples, c'est que vous pouvez les paramétrer.

Il est préférable d'éviter les boucles avec PL / SQL et les curseurs plutôt que d'utiliser Oracle SQL. Cependant, pour les personnes habituées au langage procédural, il peut être beaucoup plus facile à comprendre.

Si vous voulez vérifier si un enregistrement existe, puis faire des choses différentes selon que l'enregistrement existe ou non, il est logique d' utiliser des instructions `MERGE` dans des requêtes SQL ORACLE pures au lieu d'utiliser des boucles de curseur. (Veuillez noter que `MERGE` est uniquement disponible dans les versions Oracle> = 9i).

Examples

Curseur paramétrable "FOR loop"

```
DECLARE
  CURSOR c_emp_to_be_raised(p_sal emp.sal%TYPE) IS
    SELECT * FROM emp WHERE  sal < p_sal;
BEGIN
  FOR cRowEmp IN c_emp_to_be_raised(1000) LOOP
    dbms_Output.Put_Line(cRowEmp .eName ||' ' ||cRowEmp.sal||'... should be raised ;)');
  END LOOP;
END;
/
```

Curseur implicite "FOR loop"

```
BEGIN
  FOR x IN (SELECT * FROM emp WHERE sal < 100) LOOP
    dbms_Output.Put_Line(x.eName ||' '||x.sal||'... should REALLY be raised :D');
  END LOOP;
END;
/
```

- Le premier avantage est qu'il n'y a pas de déclaration fastidieuse à faire (pensez à cette

horrible chose "CURSOR" que vous aviez dans les versions précédentes)
- deuxième avantage est que vous construisez d'abord votre requête select, puis quand vous avez ce que vous voulez, vous pouvez immédiatement accéder aux champs de votre requête (`x.<myfield>`) dans votre boucle PL / SQL
- La boucle ouvre le curseur et récupère un enregistrement à la fois pour chaque boucle. A la fin de la boucle, le curseur est fermé.
- Les curseurs implicites sont plus rapides car le travail de l'interpréteur augmente à mesure que le code s'allonge. Le moins de code, moins l'interprète doit travailler.

Travailler avec SYS_REFCURSOR

`SYS_REFCURSOR` peut être utilisé comme type de retour lorsque vous devez manipuler facilement une liste renvoyée non pas par une table, mais plus précisément par une fonction:

fonction renvoyant un curseur

```
CREATE OR REPLACE FUNCTION list_of (required_type_in IN VARCHAR2)
   RETURN SYS_REFCURSOR
IS
   v_ SYS_REFCURSOR;
BEGIN
   CASE required_type_in
      WHEN 'CATS'
      THEN
         OPEN v_ FOR
           SELECT nickname FROM (
                select 'minou' nickname from dual
      union all select 'minâ'           from dual
      union all select 'minon'          from dual
           );
      WHEN 'DOGS'
      THEN
         OPEN v_ FOR
              SELECT dog_call FROM (
                select 'bill'   dog_call from dual
      union all select 'nestor'          from dual
      union all select 'raoul'           from dual
           );
   END CASE;
   -- Whit this use, you must not close the cursor.
   RETURN v_;
END list_of;
/
```

et comment l'utiliser:

```
DECLARE
   v_names   SYS_REFCURSOR;
   v_        VARCHAR2 (32767);
BEGIN
   v_names := list_of('CATS');
```

```
   LOOP
      FETCH v_names INTO v_;
      EXIT WHEN v_names%NOTFOUND;
      DBMS_OUTPUT.put_line(v_);
   END LOOP;
   -- here you close it
   CLOSE v_names;
END;
/
```

Manipulation d'un CURSEUR

- Déclarez le curseur pour numériser une liste d'enregistrements
- L'ouvrir
- Récupère l'enregistrement actuel dans des variables (cette position d'incrémentation)
- Utilisez `%notfound` pour détecter la fin de la liste
- N'oubliez pas de fermer le curseur pour limiter la consommation de ressources dans le contexte actuel

-

```
DECLARE
  CURSOR curCols IS -- select column name and type from a given table
         SELECT column_name, data_type FROM all_tab_columns where table_name='MY_TABLE';
  v_tab_column all_tab_columns.column_name%TYPE;
  v_data_type all_tab_columns.data_type%TYPE;
  v_ INTEGER := 1;
BEGIN
  OPEN curCols;
  LOOP
     FETCH curCols INTO v_tab_column, v_data_type;
     IF curCols%notfound OR v_ > 2000 THEN
       EXIT;
     END IF;
     dbms_output.put_line(v_||':Column '||v_tab_column||' is of '|| v_data_type||' Type.');
     v_:= v_ + 1;
  END LOOP;

  -- Close in any case
  IF curCols%ISOPEN THEN
     CLOSE curCols;
  END IF;
END;
/
```

Lire Des curseurs en ligne: https://riptutorial.com/fr/plsql/topic/5303/des-curseurs

Chapitre 8: Gestion des exceptions

Introduction

Oracle produit diverses exceptions. Vous serez peut-être surpris de voir à quel point il peut être fastidieux d'arrêter votre code avec un message peu clair. Pour améliorer la capacité de votre code PL / SQL à être corrigé facilement, il est nécessaire de gérer les exceptions au niveau le plus bas. Ne cachez jamais une exception "sous le tapis", sauf si vous êtes ici pour garder votre morceau de code uniquement pour vous et pour que personne d'autre ne le maintienne.

Les erreurs prédéfinies .

Examples

Gestion des exceptions

1. Qu'est-ce qu'une exception?

   Une exception dans PL / SQL est une erreur créée lors de l'exécution du programme.

   Nous avons trois types d'exceptions:

   - Exceptions définies en interne
   - Exceptions prédéfinies
   - Exceptions définies par l'utilisateur

2. Qu'est-ce qu'une gestion des exceptions?

   La gestion des exceptions est une possibilité de garder notre programme en cours d'exécution même s'il apparaît une erreur d'exécution résultant, par exemple, d'erreurs de codage, de pannes matérielles.

Syntaxe

La syntaxe générale pour la section des exceptions:

```
declare
    declaration Section
begin
    some statements

exception
    when exception_one then
        do something
    when exception_two then
        do something
    when exception_three then
        do something
    when others then
```

```
        do something
end;
```

Une section d'exception doit figurer à la fin du bloc PL / SQL. PL / SQL nous donne la possibilité d’implanter des blocs, puis chaque bloc peut avoir sa propre section d’exception, par exemple:

```
create or replace procedure nested_blocks
is
begin
    some statements
    begin
        some statements

    exception
        when exception_one then
            do something
    end;
exception
    when exception_two then
        do something
end;
```

Si l'exception est déclenchée dans le bloc imbriqué, elle doit être gérée dans la section d'exception interne, mais si la section d'exception interne ne gère pas cette exception, cette exception ira dans la section d'exception du bloc externe.

Exceptions définies en interne

Une exception définie en interne n'a pas de nom, mais elle a son propre code.

Quand l'utiliser?

Si vous savez que votre opération de base de données peut générer des exceptions spécifiques, celles qui n'ont pas de nom, alors vous pouvez leur donner des noms afin que vous puissiez écrire des gestionnaires d'exceptions spécifiques pour eux. Sinon, vous ne pouvez les utiliser qu'avec d' `others` gestionnaires d'exceptions.

Syntaxe

```
declare
    my_name_exc exception;
    pragma exception_init(my_name_exc,-37);
begin
    ...
exception
    when my_name_exc then
        do something
end;
```

`my_name_exc exception;` c'est la déclaration de nom d'exception.

`pragma exception_init(my_name_exc,-37);` attribuer un nom au code d'erreur de l'exception définie en interne.

Exemple

Nous avons un emp_id qui est une clé primaire dans la table emp et une clé étrangère dans la table dept. Si nous essayons de supprimer emp_id quand il a des enregistrements enfants, une exception sera lancée avec le code -2292.

```
create or replace procedure remove_employee
is
    emp_exception exception;
    pragma exception_init(emp_exception,-2292);
begin
    delete from emp where emp_id = 3;
exception
    when emp_exception then
        dbms_output.put_line('You can not do that!');
end;
/
```

> La documentation Oracle indique: "Une exception définie en interne avec un nom déclaré par l'utilisateur est toujours une exception définie en interne, et non une exception définie par l'utilisateur."

Exceptions prédéfinies

Les exceptions prédéfinies sont des exceptions définies en interne, mais elles ont des noms. La base de données Oracle soulève automatiquement ce type d'exceptions.

Exemple

```
create or replace procedure insert_emp
is
begin
    insert into emp (emp_id, ename) values ('1','Jon');

exception
    when dup_val_on_index then
        dbms_output.put_line('Duplicate value on index!');
end;
/
```

Vous trouverez ci-dessous des exemples, des noms d’exceptions avec leurs codes:

Nom d'exception	Code d'erreur
AUCUNE DONNÉE DISPONIBLE	-1403
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
ROWTYPE_MISMATCH	-6504
TOO_MANY_ROWS	-1422

Nom d'exception	Code d'erreur
ZERO_DIVIDE	-1476

Liste complète des noms d'exception et de leurs codes sur le site Web Oracle.

Exceptions définies par l'utilisateur

Comme son nom l'indique, les exceptions définies par l'utilisateur sont créées par les utilisateurs. Si vous souhaitez créer votre propre exception, vous devez:

1. Déclarer l'exception
2. Soulevez-le de votre programme
3. Créez un gestionnaire d'exceptions approprié pour l'attraper.

Exemple

Je veux mettre à jour tous les salaires des travailleurs. Mais s'il n'y a pas de travailleurs, faites une exception.

```
create or replace procedure update_salary
is
    no_workers exception;
    v_counter number := 0;
begin
    select count(*) into v_counter from emp;
    if v_counter = 0 then
        raise no_workers;
    else
        update emp set salary = 3000;
    end if;

    exception
        when no_workers then
            raise_application_error(-20991,'We don''t have workers!');
end;
/
```

Qu'est-ce que cela signifie `raise` ?

Les exceptions sont déclenchées automatiquement par le serveur de base de données en cas de besoin, mais si vous le souhaitez, vous pouvez générer explicitement toute exception à l'aide de `raise` .

Procédure `raise_application_error(error_number,error_message);`

- error_number doit être compris entre -20000 et -20999
- message d'erreur_message à afficher en cas d'erreur.

Définir une exception personnalisée, l'élever et voir d'où elle provient

Pour illustrer cela, voici une fonction qui a 3 comportements différents "erronés"

- le paramètre est complètement stupide: nous utilisons une expression définie par l'utilisateur
- le paramètre a une faute de frappe: nous utilisons l'erreur standard `NO_DATA_FOUND` Oracle
- une autre affaire, mais pas traitée

N'hésitez pas à l'adapter à vos normes:

```
DECLARE
  this_is_not_acceptable EXCEPTION;
  PRAGMA EXCEPTION_INIT(this_is_not_acceptable, -20077);
  g_err varchar2 (200) := 'to-be-defined';
  w_schema all_tables.OWNER%Type;

  PROCEDURE get_schema( p_table in Varchar2, p_schema out Varchar2)
  Is
    w_err varchar2 (200) := 'to-be-defined';
  BEGIN
    w_err := 'get_schema-step-1:';
    If (p_table = 'Delivery-Manager-Is-Silly') Then
      raise this_is_not_acceptable;
    end if;
    w_err := 'get_schema-step-2:';
    Select owner Into p_schema
      From all_tables
     where table_name like(p_table||'%');
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- handle Oracle-defined exception
     dbms_output.put_line('[WARN]'||w_err||'This can happen. Check the table name you
entered.');
  WHEN this_is_not_acceptable THEN
    -- handle your custom error
     dbms_output.put_line('[WARN]'||w_err||'Please don''t make fun of the delivery manager.');
  When others then
     dbms_output.put_line('[ERR]'||w_err||'unhandled exception:'||sqlerrm);
     raise;
  END Get_schema;

BEGIN
  g_err := 'Global; first call:';
  get_schema('Delivery-Manager-Is-Silly', w_schema);
  g_err := 'Global; second call:';
  get_schema('AAA', w_schema);
  g_err := 'Global; third call:';
  get_schema('', w_schema);
  g_err := 'Global; 4th call:';
  get_schema('Can''t reach this point due to previous error.', w_schema);

EXCEPTION
  When others then
     dbms_output.put_line('[ERR]'||g_err||'unhandled exception:'||sqlerrm);
  -- you may raise this again to the caller if error log isn't enough.
--  raise;
END;
/
```

Donner sur une base de données régulière:

```
[WARN]get_schema-step-1:Please don't make fun of the delivery manager.
```

```
[WARN]get_schema-step-2:This can happen. Check the table name you entered.
[ERR]get_schema-step-2:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
[ERR]Global; third call:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
```

Rappelez-vous que cette exception concerne *les* cas *rares* . J'ai vu des applications qui ont soulevé une exception à chaque accès, simplement pour demander le mot de passe de l'utilisateur, en disant "non connecté" ... tant de déchets de calcul.

Gestion des exceptions d'erreur de connexion

Chaque erreur Oracle standard est associée à un numéro d'erreur. Il est important de prévoir ce qui pourrait mal tourner dans votre code. Ici, pour une connexion à une autre base de données, cela peut être:

- `-28000` compte est verrouillé
- `-28001` mot de passe expiré
- `-28002` période de grâce
- `-1017` mauvais utilisateur / mot de passe

Voici un moyen de tester ce qui ne va pas avec l'utilisateur utilisé par le lien de la base de données:

```
declare
  v_dummy number;
begin
  -- testing db link
  execute immediate 'select COUNT(1) from dba_users@pass.world' into v_dummy ;
  -- if we get here, exception wasn't raised: display COUNT's result
  dbms_output.put_line(v_dummy||' users on PASS db');

EXCEPTION
  -- exception can be referred by their name in the predefined Oracle's list
    When LOGIN_DENIED
    then
        dbms_output.put_line('ORA-1017 / USERNAME OR PASSWORD INVALID, TRY AGAIN');
    When Others
    then
  -- or referred by their number: stored automatically in reserved variable SQLCODE
        If  SQLCODE = '-2019'
        Then
          dbms_output.put_line('ORA-2019 / Invalid db_link name');
        Elsif SQLCODE = '-1035'
        Then
          dbms_output.put_line('ORA-1035 / DATABASE IS ON RESTRICTED SESSION, CONTACT YOUR
DBA');
        Elsif SQLCODE = '-28000'
        Then
          dbms_output.put_line('ORA-28000 / ACCOUNT IS LOCKED. CONTACT YOUR DBA');
        Elsif SQLCODE = '-28001'
        Then
          dbms_output.put_line('ORA-28001 / PASSWORD EXPIRED. CONTACT YOUR DBA FOR CHANGE');
        Elsif SQLCODE  = '-28002'
        Then
```

```
          dbms_output.put_line('ORA-28002 / PASSWORD IS EXPIRED, CHANGED IT');
        Else
   -- and if it's not one of the exception you expected
          dbms_output.put_line('Exception not specifically handled');
          dbms_output.put_line('Oracle Said'||SQLCODE||':'||SQLERRM);
        End if;
END;
/
```

Lire Gestion des exceptions en ligne: https://riptutorial.com/fr/plsql/topic/6050/gestion-des-exceptions

Chapitre 9: Gestion des exceptions

Introduction

Oracle produit diverses exceptions. Vous serez peut-être surpris de voir à quel point il peut être fastidieux d'arrêter votre code avec un message peu clair. Pour améliorer la capacité de votre code PL / SQL à être corrigé facilement, il est nécessaire de gérer les exceptions au niveau le plus bas. Ne cachez jamais une exception "sous le tapis", sauf si vous êtes ici pour garder votre morceau de code uniquement pour vous et pour que personne d'autre ne le maintienne.

Les erreurs prédéfinies .

Examples

Gestion des exceptions d'erreur de connexion

Chaque erreur Oracle standard est associée à un numéro d'erreur. Il est important d'anticiper ce qui pourrait mal tourner dans votre code. Ici, pour une connexion à une autre base de données, il peut être:

- `-28000` compte est verrouillé
- `-28001` mot de passe expiré
- `-28002` période de grâce
- `-1017` mauvais utilisateur / mot de passe

Voici un moyen de tester ce qui ne va pas avec l'utilisateur utilisé par le lien de la base de données:

```
declare
  v_dummy number;
begin
  -- testing db link
  execute immediate 'select COUNT(1) from dba_users@pass.world' into v_dummy ;
  -- if we get here, exception wasn't raised: display COUNT's result
  dbms_output.put_line(v_dummy||' users on PASS db');

EXCEPTION
  -- exception can be referred by their name in the predefined Oracle's list
    When LOGIN_DENIED
    then
        dbms_output.put_line('ORA-1017 / USERNAME OR PASSWORD INVALID, TRY AGAIN');
    When Others
    then
  -- or referred by their number: stored automatically in reserved variable SQLCODE
        If  SQLCODE = '-2019'
        Then
          dbms_output.put_line('ORA-2019 / Invalid db_link name');
        Elsif SQLCODE = '-1035'
        Then
          dbms_output.put_line('ORA-1035 / DATABASE IS ON RESTRICTED SESSION, CONTACT YOUR
```

```
DBA');
        Elsif SQLCODE = '-28000'
        Then
          dbms_output.put_line('ORA-28000 / ACCOUNT IS LOCKED. CONTACT YOUR DBA');
        Elsif SQLCODE = '-28001'
        Then
          dbms_output.put_line('ORA-28001 / PASSWORD EXPIRED. CONTACT YOUR DBA FOR CHANGE');
        Elsif SQLCODE  = '-28002'
        Then
          dbms_output.put_line('ORA-28002 / PASSWORD IS EXPIRED, CHANGED IT');
        Else
   -- and if it's not one of the exception you expected
          dbms_output.put_line('Exception not specifically handled');
          dbms_output.put_line('Oracle Said'||SQLCODE||':'||SQLERRM);
        End if;
END;
/
```

Définir une exception personnalisée, l'élever et voir d'où elle provient

Pour illustrer cela, voici une fonction qui a 3 comportements différents "erronés"

- le paramètre est complètement stupide: nous utilisons une expression définie par l'utilisateur
- paramètre a une faute de frappe: nous utilisons l'erreur standard `NO_DATA_FOUND` Oracle
- une autre affaire, mais pas traitée

N'hésitez pas à l'adapter à vos normes:

```
DECLARE
  this_is_not_acceptable EXCEPTION;
  PRAGMA EXCEPTION_INIT(this_is_not_acceptable, -20077);
  g_err varchar2 (200) := 'to-be-defined';
  w_schema all_tables.OWNER%Type;

  PROCEDURE get_schema( p_table in Varchar2, p_schema out Varchar2)
  Is
    w_err varchar2 (200) := 'to-be-defined';
  BEGIN
    w_err := 'get_schema-step-1:';
    If (p_table = 'Delivery-Manager-Is-Silly') Then
      raise this_is_not_acceptable;
    end if;
    w_err := 'get_schema-step-2:';
    Select owner Into p_schema
      From all_tables
     where table_name like(p_table||'%');
  EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- handle Oracle-defined exception
     dbms_output.put_line('[WARN]'||w_err||'This can happen. Check the table name you
entered.');
  WHEN this_is_not_acceptable THEN
    -- handle your custom error
     dbms_output.put_line('[WARN]'||w_err||'Please don''t make fun of the delivery manager.');
  When others then
     dbms_output.put_line('[ERR]'||w_err||'unhandled exception:'||sqlerrm);
     raise;
  END Get_schema;
```

```
BEGIN
  g_err := 'Global; first call:';
  get_schema('Delivery-Manager-Is-Silly', w_schema);
  g_err := 'Global; second call:';
  get_schema('AAA', w_schema);
  g_err := 'Global; third call:';
  get_schema('', w_schema);
  g_err := 'Global; 4th call:';
  get_schema('Can''t reach this point due to previous error.', w_schema);

EXCEPTION
  When others then
     dbms_output.put_line('[ERR]'||g_err||'unhandled exception:'||sqlerrm);
  -- you may raise this again to the caller if error log isn't enough.
--  raise;
END;
/
```

Donner sur une base de données régulière:

```
[WARN]get_schema-step-1:Please don't make fun of the delivery manager.
[WARN]get_schema-step-2:This can happen. Check the table name you entered.
[ERR]get_schema-step-2:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
[ERR]Global; third call:unhandled exception:ORA-01422: exact fetch returns more than requested
number of rows
```

Rappelez-vous que cette exception concerne *les* cas *rares* . J'ai vu des applications qui ont soulevé une exception à chaque accès, simplement pour demander un mot de passe utilisateur, en disant "non connecté" ... tant de déchets de calcul.

Lire Gestion des exceptions en ligne: https://riptutorial.com/fr/plsql/topic/9480/gestion-des-exceptions

Chapitre 10: Les fonctions

Syntaxe

- CREATE [OR REPLACE] FUNCTION nom_fonction [(paramètre [, paramètre])]

  RETOUR return_datatype

  IS | COMME

  [déclaration_section]

  BEGIN executable_section

  [EXCEPTION exception_section]

  END [nom_fonction];

Examples

Générer un GUID

```
Create Or Replace Function Generateguid
Return Char Is
    V_Guid Char(40);
Begin
    Select Substr(Sys_Guid(),1,8)||'-'||Substr(Sys_Guid(),9,4)||'-'
                        ||Substr(Sys_Guid(),13,4)||'-'||Substr(Sys_Guid(),17,4)||'-'
                        ||Substr(Sys_Guid(),21) Into V_Guid
                        From Dual;
    Return V_Guid;
Exception
    When Others Then
    dbms_output.put_line('Error '|| SQLERRM);
End Generateguid;
```

Fonctions d'appel

Il y a plusieurs façons d'utiliser les fonctions.

Appeler une fonction avec une déclaration d'affectation

```
DECLARE
    x NUMBER := functionName(); --functions can be called in declaration section
BEGIN
    x := functionName();
END;
```

Appeler une fonction dans l'instruction IF

```
IF functionName() = 100 THEN
    Null;
END IF;
```

Appeler une fonction dans une instruction SELECT

```
SELECT functionName() FROM DUAL;
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/plsql/topic/4005/les-fonctions

Chapitre 11: Modèle d'affectation et langue

Examples

Modèle d'affectation en PL / SQL

Tous les langages de programmation nous permettent d'affecter des valeurs aux variables. Habituellement, une valeur est affectée à la variable, située à gauche. Le prototype de l'ensemble des opérations d'affectation dans un langage de programmation contemporain ressemble à ceci:

```
left_operand assignment_operand right_operand instructions_of_stop
```

Cela affectera un opérande de droite à l'opérande gauche. En PL / SQL, cette opération ressemble à ceci:

```
left_operand := right_operand;
```

L'opérande gauche **doit toujours être une variable** . Un opérande droit peut être une valeur, une variable ou une fonction:

```
set serveroutput on
declare
  v_hello1 varchar2(32767);
  v_hello2 varchar2(32767);
  v_hello3 varchar2(32767);
  function hello return varchar2 is begin return 'Hello from a function!'; end;
begin
   -- from a value (string literal)
  v_hello1 := 'Hello from a value!';
   -- from variable
  v_hello2 := v_hello1;
  -- from function
  v_hello3 := hello;

  dbms_output.put_line(v_hello1);
  dbms_output.put_line(v_hello2);
  dbms_output.put_line(v_hello3);
end;
/
```

Lorsque le bloc de code est exécuté dans SQL * Plus, la sortie suivante est imprimée dans la console:

```
Hello from a value!
Hello from a value!
Hello from a function!
```

Il y a une fonctionnalité en PL / SQL qui nous permet d'attribuer "de droite à gauche". Il est possible de le faire dans l'instruction SELECT INTO. Prototype de cet instrument, vous trouverez

ci-dessous:

```
SELECT [ literal | column_value ]

INTO local_variable

FROM [ table_name | aliastable_name ]

WHERE comparison_instructions;
```

Ce code affectera un caractère littéral à une variable locale:

```
set serveroutput on
declare
  v_hello varchar2(32767);
begin
  select 'Hello world!'
  into v_hello
  from dual;

  dbms_output.put_line(v_hello);
end;
/
```

Lorsque le bloc de code est exécuté dans SQL * Plus, la sortie suivante est imprimée dans la console:

```
Hello world!
```

L'attribution "de droite à gauche" **n'est pas un standard** , mais c'est une fonctionnalité précieuse pour les programmeurs et les utilisateurs. Généralement, il est utilisé lorsque le programmeur utilise des curseurs dans PL / SQL - cette technique est utilisée lorsque nous voulons renvoyer une seule valeur scalaire ou un ensemble de colonnes dans une ligne de curseur à partir du curseur SQL.

Lectures complémentaires:

- Affectation de valeurs aux variables

Lire Modèle d'affectation et langue en ligne: https://riptutorial.com/fr/plsql/topic/6959/modele-d-affectation-et-langue

Chapitre 12: Paquets

Syntaxe

- CREATE [OU REPLACE] PACKAGE nom_package

  [AUTHID {CURRENT_USER | DEFINER}]

  {IS | COMME}

  [PRAGMA SERIALLY_REUSABLE;]

  [collection_type_definition ...]

  [record_type_definition ...]

  [sous-type_definition ...]

  [collection_declaration ...]

  [const_declaration ...]

  [exception_declaration ...]

  [object_declaration ...]

  [record_declaration ...]

  [variable_declaration ...]

  [curseur_spec ...]

  [function_spec ...]

  [procedure_spec ...]

  [call_spec ...]

  [PRAGMA RESTRICT_REFERENCES (assertions) ...]

  END [nom_package];

- CREATE OR REPLACE PACKAGE PackageName IS

  FUNCTION FunctionName (paramètre1 IN VARCHAR2, paramètre 2 EN NUMBER) RETURN VARCHAR2;

  END PackageName;

- CREATE [OU REPLACE] PACKAGE BODY nom_package

{IS | COMME}

[PRAGMA SERIALLY_REUSABLE;]

[collection_type_definition ...]

[record_type_definition ...]

[sous-type_definition ...]

[collection_declaration ...]

[const_declaration ...]

[exception_declaration ...]

[object_declaration ...]

[record_declaration ...]

[variable_declaration ...]

[cursor_body ...]

[function_spec ...]

[procedure_spec ...]

[call_spec ...]

END [nom_package];

- CREATE OR REPLACE PODAGE BODY PackageName EST

  FUNCTION FunctionName (paramètre1 IN VARCHAR2, paramètre 2 EN NUMBER) RETURN VARCHAR2 IS

  déclarations

  COMMENCER

  déclarations à exécuter

  Variable varchar2

  END FunctionName;

  END PackageName;

Examples

Utilisation du paquet

Les packages dans PLSQL sont un ensemble de procédures, de fonctions, de variables, d'exceptions, de constantes et de structures de données. Généralement, les ressources d'un package sont liées entre elles et accomplissent des tâches similaires.

Pourquoi utiliser des packages

- La modularité
- Meilleure performance / Funtionality

Parties d'un paquet

Spécification - Parfois appelé en-tête de package. Contient des déclarations de variables et de types et les signatures des fonctions et des procédures qui sont dans le paquet qui sont **publiques** à appeler de l' extérieur du paquet.

Package Body - Contient le code et **les** déclarations **privées** .

La spécification du package doit être compilée avant le corps du package, sinon la compilation du corps du package signalera une erreur.

Surcharge

Les fonctions et les procédures des packages peuvent être surchargées. Le package suivant **TEST** a deux procédures appelées **print_number** , qui se comportent différemment selon les paramètres avec lesquels elles sont appelées.

```
create or replace package TEST is
  procedure print_number(p_number in integer);
  procedure print_number(p_number in varchar2);
end TEST;
/
create or replace package body TEST is

  procedure print_number(p_number in integer) is
  begin
    dbms_output.put_line('Digit: ' || p_number);
  end;

  procedure print_number(p_number in varchar2) is
  begin
    dbms_output.put_line('String: ' || p_number);
  end;

end TEST;
/
```

Nous appelons les deux procédures. Le premier avec le paramètre entier, le second avec varchar2.

```
set serveroutput on;
```

```
-- call the first procedure
exec test.print_number(3);
-- call the second procedure
exec test.print_number('three');
```

La sortie du script ci-dessus est:

```
SQL>
Digit: 3
PL/SQL procedure successfully completed
String: three
PL/SQL procedure successfully completed
```

Restrictions sur la surcharge

Seuls les sous-programmes ou les méthodes de type locales ou empaquetées peuvent être surchargés. Par conséquent, vous ne pouvez pas surcharger les sous-programmes autonomes. En outre, vous ne pouvez pas surcharger deux sous-programmes si leurs paramètres formels ne diffèrent que par leur nom ou leur mode de paramètre.

Définissez un en-tête et un corps de package avec une fonction.

Dans cet exemple, nous définissons un en-tête de package et un corps de package avec une fonction.
Après cela, nous appelons une fonction du package qui renvoie une valeur de retour.

En-tête du paquet :

```
CREATE OR REPLACE PACKAGE SkyPkg AS

       FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
       RETURN VARCHAR2;

END;
/
```

Corps du colis :

```
CREATE OR REPLACE PACKAGE BODY SkyPkg AS

        FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
        RETURN VARCHAR2
        AS
               vColour VARCHAR2(100) := NULL;
        BEGIN
               IF vPlanet = 'Earth' THEN
                       vColour := 'Blue';
               ELSIF vPlanet = 'Mars' THEN
                       vColour := 'Red';
               END IF;

               RETURN vColour;
        END;
```

```
END;
/
```

Appel de la fonction depuis le corps du package :

```
DECLARE
        vColour VARCHAR2(100);
BEGIN
        vColour := SkyPkg.GetSkyColour(vPlanet => 'Earth');
        DBMS_OUTPUT.PUT_LINE(vColour);
END;
/
```

Lire Paquets en ligne: https://riptutorial.com/fr/plsql/topic/4764/paquets

Chapitre 13: Procédure PLSQL

Introduction

La procédure PLSQL est un groupe d'instructions SQL stockées sur le serveur pour être réutilisées. Cela augmente les performances car les instructions SQL ne doivent pas être recompilées à chaque exécution.

Les procédures stockées sont utiles lorsque le même code est requis par plusieurs applications. Les procédures stockées éliminent la redondance et simplifient le code. Lorsque le transfert de données est requis entre le client et le serveur, les procédures peuvent réduire les coûts de communication dans certaines situations.

Examples

Syntaxe

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
  < declarations >
BEGIN
  < procedure_body >
EXCEPTION                       -- Exception-handling part begins
  <exception handling goes here >
   WHEN exception1 THEN
       exception1-handling-statements
END procedure_name;
```

- nom-procédure spécifie le nom de la procédure.
- L'option [OU REMPLACER] permet de modifier une procédure existante.
- La liste de paramètres facultative contient le nom, le mode et les types de paramètres. IN représente que la valeur sera passée de l'extérieur et OUT représente que ce paramètre sera utilisé pour renvoyer une valeur en dehors de la procédure. Si aucun mode n'est spécifié, le paramètre est supposé être en mode IN.
- Dans la section de déclaration, nous pouvons déclarer des variables qui seront utilisées dans la partie corps.
- procedure-body contient la partie exécutable.
- Le mot-clé AS est utilisé à la place du mot-clé IS pour créer une procédure autonome.
- La section d'exception traitera les exceptions de la procédure. Cette section est facultative.

Bonjour le monde

La procédure simple suivante affiche le texte "Hello World" dans un client qui prend en charge `dbms_output` .

```
CREATE OR REPLACE PROCEDURE helloworld
```

```
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

Vous devez l'exécuter à l'invite SQL pour créer la procédure dans la base de données ou exécuter la requête ci-dessous pour obtenir le même résultat:

```
SELECT 'Hello World!' from dual;
```

Paramètres In / Out

PL / SQL utilise les mots-clés IN, OUT, IN OUT pour définir ce qui peut arriver à un paramètre passé.

IN spécifie que le paramètre est en lecture seule et que la valeur ne peut pas être modifiée par la procédure.

OUT spécifie que le paramètre est en écriture seule et qu'une procédure peut lui attribuer une valeur, mais ne fait pas référence à la valeur.

IN OUT spécifie que le paramètre est disponible pour référence et modification.

```
PROCEDURE procedureName(x IN INT, strVar IN VARCHAR2, ans OUT VARCHAR2)
...
...
END procedureName;


procedureName(firstvar, secondvar, thirdvar);
```

Les variables passées dans l'exemple ci-dessus doivent être saisies telles qu'elles sont définies dans la section des paramètres de procédure.

Lire Procédure PLSQL en ligne: https://riptutorial.com/fr/plsql/topic/2580/procedure-plsql

Chapitre 14: Types d'objet

Remarques

Il est important de noter qu'un corps d'objet peut ne pas toujours être nécessaire. Si le constructeur par défaut est suffisant et qu'aucune autre fonctionnalité ne doit être implémentée, il ne doit pas être créé.

Un constructeur par défaut est le constructeur fourni par Oracle, qui comprend tous les attributs répertoriés par ordre de déclaration. Par exemple, une instance de BASE_TYPE pourrait être construite par l'appel suivant, même si nous ne la déclarons pas explicitement.

```
l_obj := BASE_TYPE(1, 'Default', 1);
```

Examples

BASE_TYPE

Déclaration de type:

```
CREATE OR REPLACE TYPE base_type AS OBJECT
(
   base_id     INTEGER,
   base_attr   VARCHAR2(400),
   null_attr   INTEGER, -- Present only to demonstrate non-default constructors
   CONSTRUCTOR FUNCTION base_type
   (
      i_base_id INTEGER,
      i_base_attr VARCHAR2
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_base_id RETURN INTEGER,
   MEMBER FUNCTION get_base_attr RETURN VARCHAR2,
   MEMBER PROCEDURE set_base_id(i_base_id INTEGER),
   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2),
   MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE NOT FINAL
```

Type de corps:

```
CREATE OR REPLACE TYPE BODY base_type AS
   CONSTRUCTOR FUNCTION base_type
   (
      i_base_id INTEGER,
      i_base_attr VARCHAR2
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      RETURN;
```

```
   END base_type;

   MEMBER FUNCTION get_base_id RETURN INTEGER IS
   BEGIN
      RETURN self.base_id;
   END get_base_id;

   MEMBER FUNCTION get_base_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.base_attr;
   END get_base_attr;

   MEMBER PROCEDURE set_base_id(i_base_id INTEGER) IS
   BEGIN
      self.base_id := i_base_id;
   END set_base_id;

   MEMBER PROCEDURE set_base_attr(i_base_attr VARCHAR2) IS
   BEGIN
      self.base_attr := i_base_attr;
   END set_base_attr;

   MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN 'BASE_ID ['||self.base_id||']; BASE_ATTR ['||self.base_attr||']';
   END to_string;
END;
```

MID_TYPE

Déclaration de type:

```
CREATE OR REPLACE TYPE mid_type UNDER base_type
(
   mid_attr DATE,
   CONSTRUCTOR FUNCTION mid_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_mid_attr RETURN DATE,
   MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE),
   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE NOT FINAL
```

Type de corps:

```
CREATE OR REPLACE TYPE BODY mid_type AS
   CONSTRUCTOR FUNCTION mid_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
```

```
      self.base_attr := i_base_attr;
      self.mid_attr := i_mid_attr;
      RETURN;
   END mid_type;

   MEMBER FUNCTION get_mid_attr RETURN DATE IS
   BEGIN
      RETURN self.mid_attr;
   END get_mid_attr;

   MEMBER PROCEDURE set_mid_attr(i_mid_attr DATE) IS
   BEGIN
      self.mid_attr := i_mid_attr;
   END set_mid_attr;

   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
   IS
   BEGIN
      RETURN (SELF AS base_type).to_string || '; MID_ATTR [' || self.mid_attr || ']';
   END to_string;
END;
```

LEAF_TYPE

Déclaration de type:

```
CREATE OR REPLACE TYPE leaf_type UNDER mid_type
(
   leaf_attr VARCHAR2(1000),
   CONSTRUCTOR FUNCTION leaf_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE,
      i_leaf_attr VARCHAR2
   ) RETURN SELF AS RESULT,
   MEMBER FUNCTION get_leaf_attr RETURN VARCHAR2,
   MEMBER PROCEDURE set_leaf_attr(i_leaf_attr VARCHAR2),
   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2
) INSTANTIABLE FINAL
```

Type de corps:

```
CREATE OR REPLACE TYPE BODY leaf_type AS
   CONSTRUCTOR FUNCTION leaf_type
   (
      i_base_id   INTEGER,
      i_base_attr VARCHAR2,
      i_mid_attr  DATE,
      i_leaf_attr VARCHAR2
   ) RETURN SELF AS RESULT
   IS
   BEGIN
      self.base_id := i_base_id;
      self.base_attr := i_base_attr;
      self.mid_attr := i_mid_attr;
      self.leaf_attr := i_leaf_attr;
      RETURN;
```

```
   END leaf_type;

   MEMBER FUNCTION get_leaf_attr RETURN VARCHAR2 IS
   BEGIN
      RETURN self.leaf_attr;
   END get_leaf_attr;

   MEMBER PROCEDURE set_leaf_attr(i_leaf_attr VARCHAR2) IS
   BEGIN
      self.leaf_attr := i_leaf_attr;
   END set_leaf_attr;

   OVERRIDING MEMBER FUNCTION to_string RETURN VARCHAR2 IS
   BEGIN
      RETURN (SELF AS mid_type).to_string || '; LEAF_ATTR [' || self.leaf_attr || ']';
   END to_string;
END;
```

Accéder aux objets stockés

```
CREATE SEQUENCE test_seq START WITH 1001;

CREATE TABLE test_tab
(
   test_id  INTEGER,
   test_obj base_type,
   PRIMARY KEY (test_id)
);

INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.nextval, base_type(1,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.nextval, base_type(2,'BASE_TYPE'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.nextval, mid_type(3, 'MID_TYPE',SYSDATE - 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.nextval, mid_type(4, 'MID_TYPE',SYSDATE + 1));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.nextval, leaf_type(5, 'LEAF_TYPE',SYSDATE - 20,'Maple'));
INSERT INTO test_tab (test_id, test_obj)
VALUES (test_seq.nextval, leaf_type(6, 'LEAF_TYPE',SYSDATE + 20,'Oak'));
```

Renvoie la référence d'objet:

```
SELECT test_id
      ,test_obj
  FROM test_tab;
```

Renvoie la référence d'objet, en poussant tous les sous-types

```
SELECT test_id
      ,TREAT(test_obj AS mid_type) AS obj
  FROM test_tab;
```

Renvoie un descripteur de chaîne de chaque objet, par type

```
SELECT test_id
      ,TREAT(test_obj AS base_type).to_string() AS to_string -- Parenthesis are needed after
the function name, or Oracle will look for an attribute of this name.
  FROM test_tab;
```

Lire Types d'objet en ligne: https://riptutorial.com/fr/plsql/topic/7699/types-d-objet

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec plsql	Community, Dinidu, JDro04, m.misiorny, Prashant Mishra, Tenzin, user272735
2	Boucle	m.misiorny, massko
3	Collecte en vrac	Prashant Mishra
4	Collections et disques	J. Chomel
5	Déclaration IF-THEN-ELSE	massko
6	Déclencheurs	Harjot, jiri.hofman
7	Des curseurs	dipdapdop, J. Chomel, Jucan
8	Gestion des exceptions	Ice, J. Chomel, jiri.hofman, Tony Andrews, Zug Zwang
9	Les fonctions	JDro04, Jon Clements, user3216906
10	Modèle d'affectation et langue	m.misiorny, user272735
11	Paquets	JDro04, jiri.hofman, StewS2, Tenzin
12	Procédure PLSQL	Dinidu, Doruk, Harjot, JDro04, Kekar, William Robertson
13	Types d'objet	HepC