FREE eBook

LEARNING plsql

#plsql

Table of Contents

About

Chapter 1: Getting started with plsql

Remarks

This section provides an overview of what plsql is, and why a developer might want to use it.

It should also mention any large subjects within plsql, and link out to the related topics. Since the Documentation for plsql is new, you may need to create initial versions of those related topics.

Examples

Definition of PLSQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension for SQL and the Oracle relational database. PL/SQL is available in Oracle Database (since version 7), TimesTen in-memory database (since version 11.2.1), and IBM DB2 (since version 9.7).

The basic unit in PL/SQL is called a block, which is made up of three parts: a declarative part, an executable part, and an exception-building part.

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

Declarations - This section starts with the keyword DECLARE. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

Executable Commands - This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.

Exception Handling - This section starts with the keyword EXCEPTION. This section is again optional and contains exception(s) that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END.

In anonymous block, only executable part of block is required, other parts are not nessesary. Below is example of simple anonymous code, which does not do anything but perform without error reporting.

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

Missing excecutable instruction leads to an error, becouse PL/SQL does not support empty blocks. For example, excecution of code below leads to an error:

```
BEGIN
END;
/
```

Application will raise error:

```
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
```

Symbol " * " in line below keyword "END;" means, that the block which ends with this block is empty or bad constructed. Every execution block needs instructions to do, even if it does nothing, like in our example.

Hello World

```
set serveroutput on

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

Command `set serveroutput on` is required in SQL*Plus and SQL Developer clients to enable the output of `dbms_output`. Without the command nothing is displayed.

The `end;` line signals the end of the anonymous PL/SQL block. To run the code from SQL command line, you may need to type `/` at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces the following result:

```
Hello, World!

PL/SQL procedure successfully completed.
```

About PLSQL

PL/SQL stands for Procedural Language extensions to SQL. PL/SQL is available only as an "enabling technology" within other software products; it does not exist as a standalone language. You can use PL/SQL in the Oracle relational database, in the Oracle Server, and in client-side application development tools, such as Oracle Forms. Here are some of the ways you might use PL/SQL:

1. To build stored procedures. .
2. To create database triggers.
3. To implement client-side logic in your Oracle Forms application.
4. To link a World Wide Web home page to an Oracle database.

Difference between %TYPE and %ROWTYPE.

`%TYPE`: Used to declare a field with the same type as that of a specified table's column.

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

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

%ROWTYPE: Used to declare a record with the same types as found in the specified table, view or cursor (= multiple columns).

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

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

Create or replace a view

In this example we are going to create a view.
A view is mostly used as a simple way of fetching data from multiple tables.

Example 1:
View with a select on one table.

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

Example 2:
View with a select on multiple 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;
```

To call this views in a query you can use a select statement.

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

Create a table

Below we are going to create a table with 3 columns.
The column `Id` must be filled is, so we define it `NOT NULL`.
On the column `Contract` we also add a check so that the only value allowed is 'Y' or 'N'. If an insert in done and this column is not specified during the insert then default a 'N' is inserted.

```
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'))
);
```

Read Getting started with plsql online: https://riptutorial.com/plsql/topic/1962/getting-started-with-plsql

Chapter 2: Assignments model and language

Examples

Assignments model in PL/SQL

All programming languages allow us to assign values to variables. Usually, a value is assigned to variable, standing on left side. The prototype of the overall assignment operations in any contemporary programming language looks like this:

```
left_operand assignment_operand right_operand instructions_of_stop
```

This will assign right operand to the left operand. In PL/SQL this operation looks like this:

```
left_operand := right_operand;
```

Left operand **must be always a variable**. Right operand can be value, variable or function:

```
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;
/
```

When the code block is executed in SQL*Plus the following output is printed in console:

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

There is a feature in PL/SQL that allow us to assign "from right to the left". It's possible to do in SELECT INTO statement. Prototype of this instrunction you will find below:

```
SELECT [ literal | column_value ]

INTO local_variable
```

```
FROM [ table_name | aliastable_name ]

WHERE comparison_instructions;
```

This code will assign character literal to a local variable:

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

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

When the code block is executed in SQL*Plus the following output is printed in console:

```
Hello world!
```

Asignment "from right to the left" **is not a standard**, but it's valuable feature for programmers and users. Generally it's used when programmer is using cursors in PL/SQL - this technique is used, when we want to return a single scalar value or set of columns in the one line of cursor from SQL cursor.

Further Reading:

- Assigning Values to Variables

Read Assignments model and language online: https://riptutorial.com/plsql/topic/6959/assignments-model-and-language

Chapter 3: Bulk collect

Examples

Bulk data Processing

local collections are not allowed in select statements. Hence the first step is to create a schema level collection. If the collection is not schema level and being used in SELECT statements then it would cause "PLS-00642: local collection types not allowed in SQL statements"

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

--Grant permissions on collection so that it could be used publically in database

```
     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;
```

--fetching data from table into collection and then loop through the collection and print the data.

```
    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;
/
```

Read Bulk collect online: https://riptutorial.com/plsql/topic/6855/bulk-collect

Chapter 4: Collections and Records

Examples

Use a collection as a return type for a split function

It's necessary to declare the type; here `t_my_list`; a collection is a `TABLE OF` *`something`*

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

Here's the function. Notice the `()` used as a kind of constructor, and the `COUNT` and `EXTEND` keywords that help you create and grow your 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;
/
```

Then you can see the content of the collection with the `TABLE()` function from SQL; it can be used as a list inside a SQL `IN ( ..)` statement:

```
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') --
```

Read Collections and Records online: https://riptutorial.com/plsql/topic/9779/collections-and-records

Chapter 5: Cursors

Syntax

- Cursor *cursor_name* Is *your_select_statement*
- Cursor *cursor_name*(param TYPE) Is *your_select_statement_using_param*
- FOR x in (*your_select_statement*) LOOP ...

Remarks

Declared Cursors are difficult to use, and you should prefer `FOR` loops in most cases. What's very interesting in cursors compared to simple `FOR` loops, is that you can parameterize them.

It's better to avoid doing loops with PL/SQL and cursors instead of using Oracle SQL anyway. However, For people accustomed to procedural language, it can be far easier to understand.

If you want to check if a record exists, and then do different things depending on whether the record exists or not, then it makes sense to use `MERGE` statements in pure ORACLE SQL queries instead of using cursor loops. (Please note that `MERGE` is only available in Oracle releases >= 9i).

Examples

Parameterized "FOR loop" Cursor

```
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;
/
```

Implicit "FOR loop" cursor

```
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;
/
```

- First advantage is there is no tedious declaration to do (think of this horrible "CURSOR" thing you had in previous versions)
- second advantage is you first build your select query, then when you have what you want, you immediately can access the fields of your query (`x.<myfield>`) in your PL/SQL loop

- The loop opens the cursor and fetches one record at a time for every loop. At the end of the loop the cursor is closed.
- Implicit cursors are faster because the interpreter's work grows as the code gets longer. The less code the less work the interpreter has to do.

Working with SYS_REFCURSOR

`SYS_REFCURSOR` can be used as a return type when you need to easily handle a list returned not from a table, but more specifically from a function:

function returning a cursor

```
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;
/
```

and how to use it:

```
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;
/
```

Handling a CURSOR

- Declare the cursor to scan a list of records
- Open it
- Fetch current record into variables (this increments position)
- Use `%notfound` to detect end of list
- Don't forget to close the cursor to limit resources consumption in current context

--

```
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;
/
```

Read Cursors online: https://riptutorial.com/plsql/topic/5303/cursors

Chapter 6: Exception Handling

Introduction

Oracle produces a variety of exceptions. You may be surprised how tedious it can be to have your code stop with some unclear message. To improve your PL/SQL code's ability to get fixed easily it is necessary to handle exceptions at the lowest level. Never hide an exception "under the carpet", unless you're here to keep your piece of code for you only and for no one else to maintain.

The predefined errors.

Examples

Exception handling

1. What is an exception?

   Exception in PL/SQL is an error created during a program execution.

   We have three types of exceptions:

   - Internally defined exceptions
   - Predefined exceptions
   - User-defined exceptions

2. What is an exception handling?

   Exception handling is a possibility to keep our program running even if appear runtime error resulting from for example coding mistakes, hardware failures.We avoid it from exiting abruptly.

Syntax

The general syntax for exception section:

```
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;
```

An exception section has to be on the end of the PL/SQL block. PL/SQL gives us the opportunity to nest blocks, then each block may have its own exception section for example:

```
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;
```

If exception will be raised in the nested block it should be handled in the inner exception section, but if inner exception section does not handle this exception then this exception will go to exception section of the external block.

Internally defined exceptions

An internally defined exception doesn't have a name, but it has its own code.

When to use it?

If you know that your database operation might raise specific exceptions those which don't have names, then you can give them names so that you can write exception handlers specifically for them. Otherwise, you can use them only with `others` exception handlers.

Syntax

```
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;` that is the exception name declaration.

`pragma exception_init(my_name_exc,-37);` assign name to the error code of internally defined exception.

Example

We have an emp_id which is a primary key in emp table and a foreign key in dept table. If we try to remove emp_id when it has child records, it will be thrown an exception with 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;
/
```

> Oracle documentation says: "An internally defined exception with a user-declared name is still an internally defined exception, not a user-defined exception."

Predefined exceptions

Predefined exceptions are internally defined exceptions but they have names. Oracle database raise this type of exceptions automatically.

Example

```
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;
/
```

Below are examples exceptions name with theirs codes:

Exception Name	Error Code
NO_DATA_FOUND	-1403
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
ROWTYPE_MISMATCH	-6504
TOO_MANY_ROWS	-1422
ZERO_DIVIDE	-1476

Full list of exception names and their codes on Oracle web-site.

User defined exceptions

As the name suggest user defined exceptions are created by users. If you want to create your own exception you have to:

1. Declare the exception
2. Raise it from your program
3. Create suitable exception handler to catch him.

Example

I want to update all salaries of workers. But if there are no workers, raise an 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;
/
```

What does it mean `raise`?

Exceptions are raised by database server automatically when there is a need, but if you want, you can raise explicitly any exception using `raise`.

Procedure `raise_application_error(error_number,error_message);`

- error_number must be between -20000 and -20999
- error_message message to display when error occurs.

Define custom exception, raise it and see where it comes from

To illustrate this, here is a function that has 3 different "wrong" behaviors

- the parameter is completely stupid: we use a user-defined expression
- the parameter has a typo: we use Oracle standard `NO_DATA_FOUND` error
- another, but not handled case

Feel free to adapt it to your standards:

```
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;
/
```

Giving on a regular database:

```
[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
```

Remember that exception are here to handle *rare* cases. I saw applications who raised an exception at every access, just to ask for the user password, saying "not connected"... so much computation waste.

Handling connexion error exceptions

Each standard Oracle error is associated with an error number. It's important to anticipate what could go wrong in your code. Here for a connection to another database, it can be:

- `-28000` account is locked
- `-28001` password expired
- `-28002` grace period
- `-1017` wrong user / password

Here is a way to test what goes wrong with the user used by the database link:

```
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;
/
```

Read Exception Handling online: https://riptutorial.com/plsql/topic/6050/exception-handling

Chapter 7: Exception Handling

Introduction

Oracle produces a variety of exceptions. You may be surprised how tedious it can be to have your code stop with some unclear message. To improve your PL/SQL code's ability to get fixed easily it is necessary to handle exceptions at the lowest level. Never hide an exception "under the carpet", unless you're here to keep your piece of code for you only and for no one else to maintain.

The predefined errors.

Examples

Handling connexion error exceptions

Each standard Oracle error is associated with an error number. Its important to anticipate what could go wrong in your code. Here for a connection to another database it can be:

- `-28000` account is locked
- `-28001` password expired
- `-28002` grace period
- `-1017` wrong user / password

Here is a way to test what goes wrong with the user used by the database link:

```
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;
/
```

Define custom exception, raise it and see where it comes from

To illustrate this, here is a function that has 3 different "wrong" behaviors

- parameter is completely stupid: we use a user-defined expression
- parameter has a typo: we use Oracle standard `NO_DATA_FOUND` error
- another, but not handled case

Feel free to adapt it to your standards:

```
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;
/
```

Giving on a regular database:

```
[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
```

Remember that exception are here to handle *rare* cases. I saw applications who raised an exception at every access, just to ask for user password, saying "not connected"... so much computation waste.

Read Exception Handling online: https://riptutorial.com/plsql/topic/9480/exception-handling

Chapter 8: Functions

Syntax

- CREATE [OR REPLACE] FUNCTION function_name [(parameter [,parameter])]

  RETURN return_datatype

  IS | AS

  [declaration_section]

  BEGIN executable_section

  [EXCEPTION exception_section]

  END [function_name];

Examples

Generate 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;
```

Calling Functions

There are a few ways to use functions.

Calling a function with an assignment statement

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

Calling a function in IF statement

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

Calling a function in a SELECT statement

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

Read Functions online: https://riptutorial.com/plsql/topic/4005/functions

Chapter 9: IF-THEN-ELSE Statement

Syntax

- IF [condition 1] THEN
- [statements to execute when condition 1 is TRUE];
- ELSIF [condition 2] THEN
- [statements to execute when condition 2 is TRUE];
- ELSE
- [statements to execute when both condition 1 & condition 2 are FALSE];
- END IF;

Examples

IF-THEN

```
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;
```

Read IF-THEN-ELSE Statement online: https://riptutorial.com/plsql/topic/5871/if-then-else-statement

Chapter 10: Loop

Syntax

1. LOOP
2. [statements];
3. EXIT WHEN [condition for exit loop];
4. END LOOP;

Examples

Simple Loop

```
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 Loop

The WHILE loop is executed untill the condition of end is fulfilled. Simple example:

```
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;
```

This loop will be executed untill current value of variable v_counter will be less than ten.

The result:

```
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
```

The most important thing is, that our loop starts with '0' value, so first line of results is 'Current iteration of loop is 0'.

FOR Loop

Loop FOR works on similar rules as other loops. FOR loop is executed exact number of times and this number is known at the beginning - lower and upper limits are directly set in code. In every step in this example, loop is increment by 1.

Simple example:

```
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;
```

And the result is:

```
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 has additional property, which is working in reverse. Using additional word 'REVERSE' in declaration of lower and upper limit of loop allow to do that. Every execution of loop decrement value of v_counter by 1.

Example:

```
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;
```

And the result:

```
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
```

Read Loop online: https://riptutorial.com/plsql/topic/6157/loop

Chapter 11: Object Types

Remarks

It is important to note that an object body may not always be necessary. If the default constructor is sufficient, and no other functionality needs to be implemented then it should not be created.

A default constructor is the Oracle supplied constructor, which consists of all attributes listed in order of declaration. For example, an instance of BASE_TYPE could be constructed by the following call, even though we do not explicitly declare it.

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

Examples

BASE_TYPE

Type declaration:

```
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 body:

```
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

Type declaration:

```
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 body:

```
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

Type declaration:

```
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 Body:

```
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;
```

Accessing stored objects

```
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'));
```

Returns object reference:

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

Returns object reference, pushing all to subtype

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

Returns a string descriptor of each object, by 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;
```

Read Object Types online: https://riptutorial.com/plsql/topic/7699/object-types

Chapter 12: Packages

Syntax

- CREATE [OR REPLACE] PACKAGE package_name

  [AUTHID {CURRENT_USER | DEFINER}]

  {IS | AS}

  [PRAGMA SERIALLY_REUSABLE;]

  [collection_type_definition ...]

  [record_type_definition ...]

  [subtype_definition ...]

  [collection_declaration ...]

  [constant_declaration ...]

  [exception_declaration ...]

  [object_declaration ...]

  [record_declaration ...]

  [variable_declaration ...]

  [cursor_spec ...]

  [function_spec ...]

  [procedure_spec ...]

  [call_spec ...]

  [PRAGMA RESTRICT_REFERENCES(assertions) ...]

  END [package_name];
- CREATE OR REPLACE PACKAGE PackageName IS

  FUNCTION FunctionName(parameter1 IN VARCHAR2, paramter2 IN NUMBER) RETURN VARCHAR2;

  END PackageName;
- CREATE [OR REPLACE] PACKAGE BODY package_name

{IS | AS}

[PRAGMA SERIALLY_REUSABLE;]

[collection_type_definition ...]

[record_type_definition ...]

[subtype_definition ...]

[collection_declaration ...]

[constant_declaration ...]

[exception_declaration ...]

[object_declaration ...]

[record_declaration ...]

[variable_declaration ...]

[cursor_body ...]

[function_spec ...]

[procedure_spec ...]

[call_spec ...]

END [package_name];

- CREATE OR REPLACE PACKAGE BODY PackageName IS

  FUNCTION FunctionName(parameter1 IN VARCHAR2, paramter2 IN NUMBER) RETURN VARCHAR2 IS

  declarations

  BEGIN

  statements to execute

  RETURN *varchar2 variable*

  END FunctionName;

  END PackageName;

Examples

Package Usage

Packages in PLSQL are a collection of procedures, functions, variables, exceptions, constants, and data structures. Generally the resources in a package are related to each other and accomplish similar tasks.

Why Use Packages

- Modularity
- Better Performance/ Funtionality

Parts of a Package

Specification - Sometimes called a package header. Contains variable and type declarations and the signatures of the functions and procedures that are in the package which are **public** to be called from outside the package.

Package Body - Contains the code and **private** declarations.

The package specification must be compiled before the package body, otherwise the package body compilation will report an error.

Overloading

Functions and procedures in packages can be overloaded. The following package **TEST** has two procedures called **print_number**, which behave differently depending on parameters they are called with.

```
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;
/
```

We call both procedures. The first with integer parameter, the second with varchar2.

```
set serveroutput on;
-- call the first procedure
exec test.print_number(3);
```

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

The output of the above script is:

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

Restrictions on Overloading

Only local or packaged subprograms, or type methods, can be overloaded. Therefore, you cannot overload standalone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode

Define a Package header and body with a function.

In this example we define a package header and a package body wit a function.
After that we are calling a function from the package that return a return value.

Package header:

```
CREATE OR REPLACE PACKAGE SkyPkg AS

       FUNCTION GetSkyColour(vPlanet IN VARCHAR2)
       RETURN VARCHAR2;

END;
/
```

Package body:

```
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;
/
```

Calling the function from the package body:

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

Read Packages online: https://riptutorial.com/plsql/topic/4764/packages

Chapter 13: PLSQL procedure

Introduction

PLSQL procedure is a group of SQL statements stored on the server for reuse. It increases the performance because the SQL statements do not have to be recompiled every time it is executed.

Stored procedures are useful when same code is required by multiple applications. Having stored procedures eliminates redundancy, and introduces simplicity to the code. When data transfer is required between the client and server, procedures can reduce communication cost in certain situations.

Examples

Syntax

```
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;
```

- procedure-name specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure. If no mode is specified, parameter is assumed to be of IN mode.
- In the declaration section we can declare variables which will be used in the body part.
- procedure-body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.
- exception section will handle the exceptions from the procedure. This section is optional.

Hello World

The following simple procedure displays the text "Hello World" in a client that supports `dbms_output`.

```
CREATE OR REPLACE PROCEDURE helloworld
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
```

```
/
```

You need to execute this at the SQL prompt to create the procedure in the database, or you can run the query below to get the same result:

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

In/Out Parameters

PL/SQL uses IN, OUT, IN OUT keywords to define what can happen to a passed parameter.

IN specifies that the parameter is read only and the value cannot be changed by the procedure.

OUT specifies the parameter is write only and a procedure can assign a value to it, but not reference the value.

IN OUT specifies the parameter is available for reference and modification.

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


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

The variables passed in the above example need to be typed as they are defined in the procedure parameter section.

Read PLSQL procedure online: https://riptutorial.com/plsql/topic/2580/plsql-procedure

Chapter 14: Triggers

Introduction

Introduction:

Triggers are a useful concept in PL/SQL. A trigger is a special type of stored procedure which does not require to be explicitly called by the user. It is a group of instructions, which is automatically fired in response to a specific data modification action on a specific table or relation, or when certain specified conditions are satisfied. Triggers help maintain the integrity, and security of data. They make the job convenient by taking the required action automatically.

Syntax

- CREATE [OR REPLACE] TRIGGER trigger_name
- BEFORE UPDATE [or INSERT] [or DELETE]
- ON table_name
- [FOR EACH ROW]
- DECLARE
- -- variable declarations
- BEGIN
- -- trigger code
- EXCEPTION
- WHEN ...
- -- exception handling
- END;

Examples

Before INSERT or UPDATE trigger

```
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;
/
```

Read Triggers online: https://riptutorial.com/plsql/topic/7674/triggers

Credits

S. No	Chapters	Contributors
1	Getting started with plsql	Community, Dinidu, JDro04, m.misiorny, Prashant Mishra, Tenzin, user272735
2	Assignments model and language	m.misiorny, user272735
3	Bulk collect	Prashant Mishra
4	Collections and Records	J. Chomel
5	Cursors	dipdapdop, J. Chomel, Jucan
6	Exception Handling	Ice, J. Chomel, jiri.hofman, Tony Andrews, Zug Zwang
7	Functions	JDro04, Jon Clements, user3216906
8	IF-THEN-ELSE Statement	massko
9	Loop	m.misiorny, massko
10	Object Types	HepC
11	Packages	JDro04, jiri.hofman, StewS2, Tenzin
12	PLSQL procedure	Dinidu, Doruk, Harjot, JDro04, Kekar, William Robertson
13	Triggers	Harjot, jiri.hofman