Block types in PL/SQL

Different Types of Blocks in PL/SQL.

PL/SQL stands for procedural language-standard query language. It is a significant member of Oracle programming tool set which is extensively used to code server side programming. Similar to SQL language PL/SQL is also a case-insensitive programming language.

Blocks

Generally a program written in PL/SQL language is divided into blocks. We can say blocks are basic programming units in PL/SQL programming language.
PL/SQL Blocks contain set of instructions for oracle to execute, display information to the screen, write data to file, call other programs, manipulate data and many more.

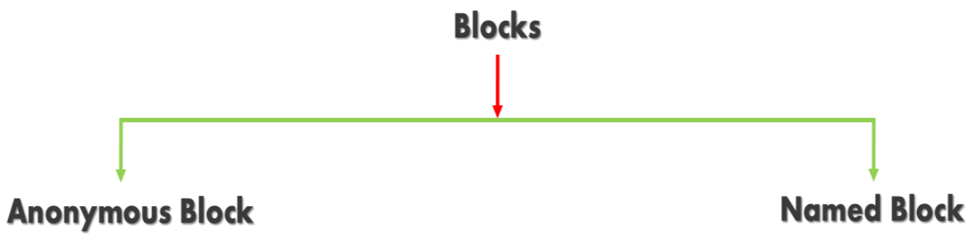
Does Blocks supports DDL statements?

Yes, PL/SQL blocks support all DML statements and using Native Dynamic SQL (NDS) or they can run DDL statements using the build in DBMS_SQL package.

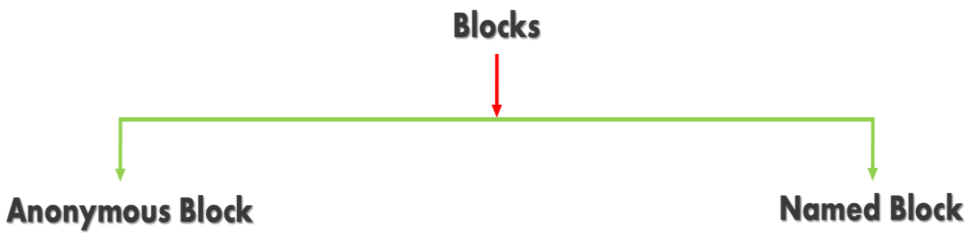
Types of PL/SQL Blocks

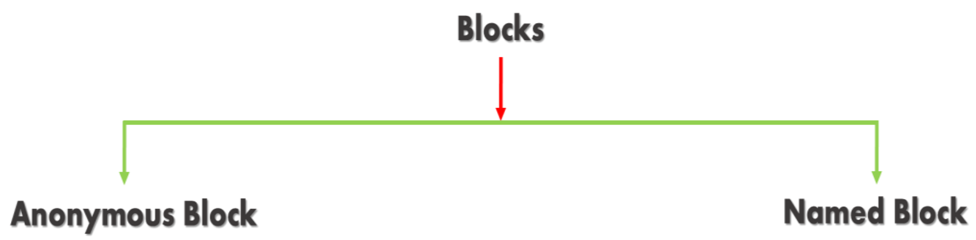
There are two types of blocks in PL/SQL

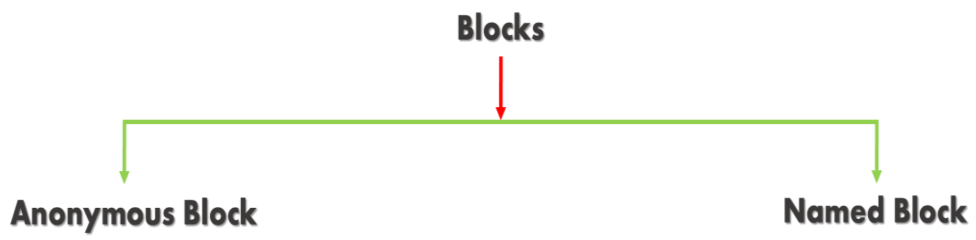
1. Anonymous Block



2. Named Block







Anonymous Block

As the title suggests these anonymous blocks do not have any names as a result they cannot be stored in database and referenced later.

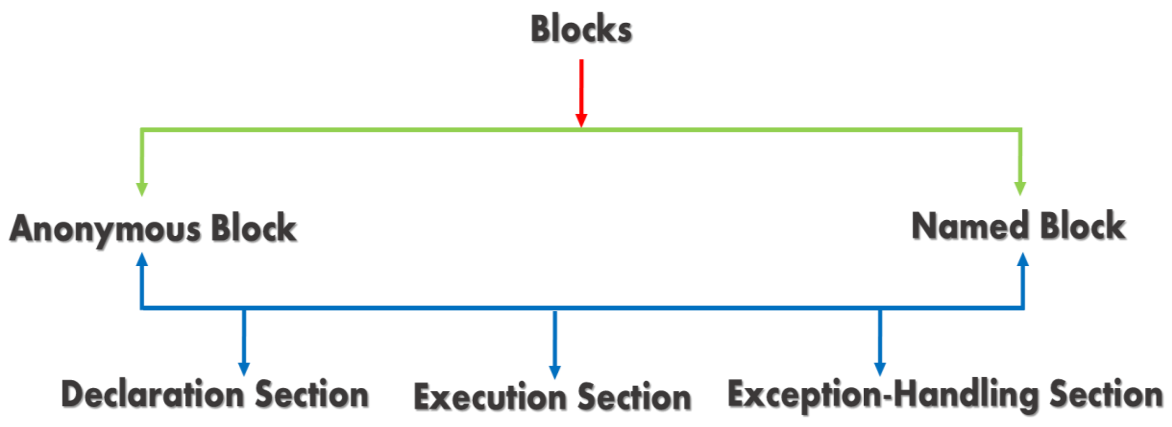
Named Block

On the other hand Named PL/SQL blocks are the one that have names and are used when creating subroutines such as procedures, functions and packages. These subroutines then can be stored in the database and referenced by their name later.

Both type of PL/SQL blocks are further divided into 3 different sections which are:

3. The Declaration Section
4. The Execution Section and
5. The Exception-handling Section

The Execution Section is the only mandatory section of block whereas Declaration and Exception Handling sections are optional.



Basic prototype of Anonymous PL/SQL Block

```
DECLARE
   Declaration Statements
BEGIN
   Executable statements
Exception
   Exception handling statements
END;
```

Declaration Section

This is the first section of PL/SQL block which contains definition of PL/SQL identifiers such as variables, Constants, cursors and so on. You can say this is the place where all local variables used in the program are defined and documented.

Example 1

```
DECLARE
   Var_first_name VARCHAR2(30);
   Var_last_name VARCHAR2(30);
   Con_flag CONSTANT NUMBER:=0;
```

The above example shows declaration section of an anonymous block. It begins with keyword declare and contains two variables var_first_name and var_last_name and one constant con_flag. Notice that semicolon terminates each declaration.

Execution Section

This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section. The content of this section must be complete to allow the block to compile. By complete I mean complete set of instruction for the PL/SQL engine must be between BEGIN and END keyword.

The execution Section of any PL/SQL block always begins with the Keyword BEGIN and ends with the Keyword END.

This is the only mandatory section in PL/SQL block. This section supports all DML commands and SQL*PLUS built-in functions and using Native Dynamic SQL (NDS) or using DMBS_SQL built-in package it also supports DDL commands.

Example 2

```
BEGIN
   SELECT first_name, last_name INTO var_first_name, var_last_name
   FROM employees WHERE employee_id =100;
   DBMS_OUTPUT.PUT_LINE(‘Employee Name ’||var_first_name||’ ‘||var_last_name);
END;
```

This is very simple program where I fetched the value of first name and last name column from employees table where employee id is 100 and stored it into the variable var_first_name and var_last_name which we declared in our first example.

Exception-Handling Section

This is the last section of PL/SQL block which is optional like the declaration block. This section contains statements that are executed when a runtime error occurs within the block.

Runtime error occurs while the program is running and cannot be detected by the PL/SQL compiler. When a runtime error occurs, controlled is pass to the exception handling section of the block the error is evaluated and specific exception is raised.

Example 3

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE ('No Employee Found with '||employee_id);
```

VARIABLES in PL/SQL

Proper way of Declaring and Initializing variables in PL/SQL.

Variables are place holders in the computer's main memory which hold some data. Every variable has a name which is user defined, also a data type that defines the nature of data a variable can hold and the total amount of space they can have in main memory along with some value. Like every other programming language in PL/SQL also we first need to declare a variable before using it.

Variable Declaration

All the variable declaration must be done in ***Declare Section*** of the PL/SQL block. As soon as you declare a variable, the compiler will allocate the memory according to the data type to that variable. Though you can assign value to the variable either in declare section or in execution section of your PL/SQL block but the ***declaration must be done in declare section***.

Example 1

Here is a simple program to understand this.

```
SET SERVEROUTPUT ON;
DECLARE
   Test_var1 NUMBER; -- Declaring variable Test_var
BEGIN
   Test_var1:= 10;
   DBMS_OUTPUT.PUT_LINE (Test_var1);
END;
```

This is a very simple program in which I first declared a variable by the name of test_var1 which has data type number in declaration section and later in execution section I initialized it and assigned a numeric value 10 and then using DBMS_OUTPUT statement I displayed the value of this variable.

Assignment Operator (:=)

If you have noticed in the above program that unlike conventional assignment operators in other programming language which is Equal to (=) operator, here we used **colon (:)** with **equal to (=)** operator for assigning the value to the variable.
Yes in PL/SQL the ***combo of colon and equal to operator (:=) works as assignment operator*** which is very different from the other programming languages.

Note: There is no space between colon and equal to operator.

Variable Initialization

Two main questions which I am going to address in this section are

6. Where can we initialize the variables?
7. Different ways of initializing variables in PL/SQL program.

So let's start with the first question which is where can we initialize the variables in PL/SQL program?

Though it's mandatory to declare all the variables of your programs in declaration section of your PL/SQL block but initializing them and assigning them some value in execution section is not mandatory. This means that you can initialize or say assign values to your variables anywhere in your program.
You can initialize a variable in declaration section while creating it or you can initialize the variable in execution section as we did in the example 1. This is the answer to the first question which I have given you in as simple a language as possible.

Now the second question is what are the different ways of initializing a variable in PL/SQL program?

Variable initialization means assigning some value to the variable which you previously declared. There are two ways of assigning value to the variable.

8. First is the direct way of giving value to the variable. We have seen this demonstrated in the previous example where we assign integer 10 to the variable test_var1.
9. Second way is by fetching value from the column of a row of a table and assigning that value to the variable.

So let's see some examples and try to understand the above concept.

Example 2.
Declaring variable in declaration section and assigning value by direct way.

```
DECLARE
    var_test1 VARCHAR2(30) := 'RebellionRider'; --Declare & initialize the variable at same time
BEGIN
  DBMS_OUTPUT.PUT_LINE(var_test1);
END;
```

A very simple program where I declared a variable by the name of var_test with data type VARCHAR2 and data width 30 also I initialized this variable and assigned a string 'RebellionRider' right after declaring it in the declaration section.

How To Initialize Variable using SELECT…INTO statement.

In the previous tutorial we learnt about variables as well as how to declare and initialize them. There we saw two different examples of direct initialization. Here in this tutorial we will see another way of initializing the variable using SELECT INTO statement.

The SELECT INTO statement retrieves data from one or more database tables, and assigns the selected values to variables or collections.

Syntax

```
SELECT column1, column2…. Column n INTO variable1, variable2… Variable n FROM table_name
WHERE ;
```

Now let's see some examples of initializing a variable by fetching values from tables of your database. For the demonstration I will use the Employees table of HR sample Schema.

Example 1.

```
DECLARE
    v_salary NUMBER(8);
```

As I mentioned in my previous tutorial, that every variable must be declared prior to its use and we can only declare a variable in declaration section of PL/SQL block. In the above demonstration I declared a variable by the name of v_salary which has data type NUMBER and Data width 8. One thing you must take care while declaring variable here is that the data type and data width of your variable and the column whose value you want to fetch must match.

```
BEGIN
    SELECT salary INTO v_salary FROM employees
    WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE (v_salary);
END;
```

This is the execution section of our anonymous block. This section contains our select statement. This select statement is retrieving salary of the employee whose employee id is 100 from employees table and storing it into the variable v_salary.

The variable v_salary which we declare above in the declaration section is capable of holding single data at a time thus make sure your select statement must return only single data. This you can ensure by using WHERE clause of your SELECT statement as I did in this example.

Let's put all the parts together and see the complete anonymous block.

```
DECLARE
    v_salary NUMBER(8);
BEGIN
    SELECT salary INTO v_salary FROM employees
    WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE (v_salary);
END;
```

Example 2. Fetch data from multiple column and store it into multiple variables.

Suppose along with Salary you also want to display the first name of the employee using PL/SQL. In this case we will need two different variables as we want to fetch data from two different columns of the table first name and salary. Let's see the example

```
DECLARE
    v_salary NUMBER(8);
    v_fname VARCHAR2 (20);
BEGIN
    SELECT first_name, salary INTO v_fname, v_salary FROM employees
    WHERE employee_id =100;
        DBMS_OUTPUT.PUT_LINE(v_fname||' has salary '||v_salary);
END;
```

In this query we have two variables v_salary which will hold the value from salary column and v_fname which will hold the value from first name column of employees table. The select statement is very similar to the previous one except the one extra column with first name and variable v_fname.

This select statement will return the first name and salary of the employee whose employee id is 100 and then those values will be stored into our variable v_ fname and v_salary. Few things which you must take care here are:

10. As I explained a while ago that variable v_fname and v_salary can hold only one data at a time thus make sure your select statement will return data from one row at a time. You can ensure this by using WHERE clause.
11. The value from first name and salary columns will be stored into variable v_fname and v_salary respectively hence you should always make sure that the data type and data width of the variable matches that of the columns.

Anchored Datatype (% Type)

Anchored data types are those data type which you assign to a variable based on a database object. They are called anchored data type because unlike the variable data type it is not dependent on that of any underlying object.

Syntax of Anchored Datatype.

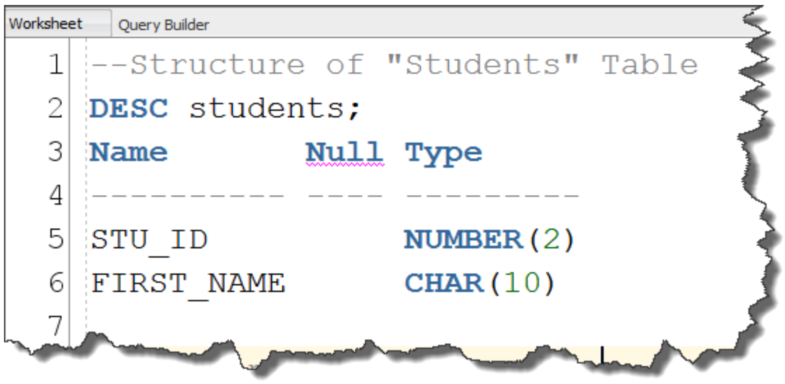
```
variable_name typed-attribute%type
```

Where variable name is user defined name given to a variable and type attribute can be anything such as previously declared PL/SQL variable or column of a table. And at the end %type is the direct reference to the underlying database object.

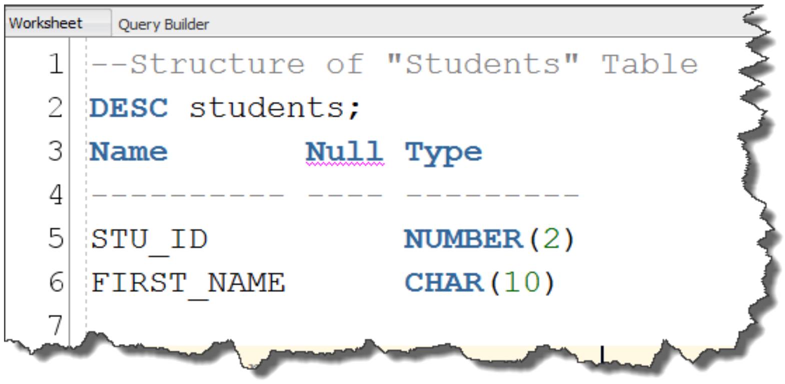
Examples of Anchored Datatype.

For the demonstration I have created a table by the name of Students which has two columns

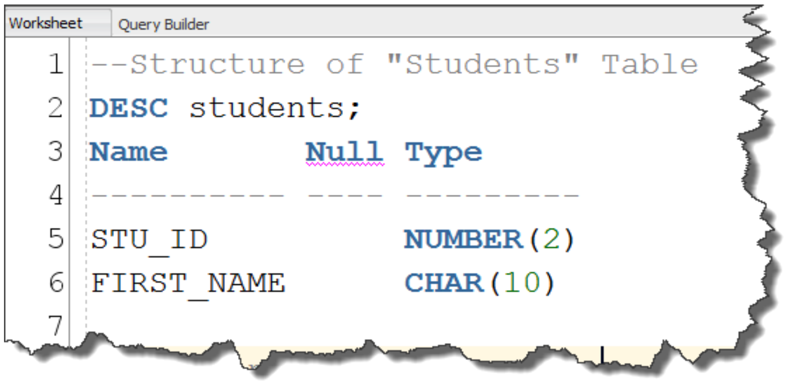
Stu_id with data type Number and data width 2 and First_name with data type varchar2 and data width 8.



```
--Structure of "Students" Table
DESC students;
Name       Null Type
---------- ---- ---------
STU_ID          NUMBER(2)
FIRST_NAME      CHAR(10)
```

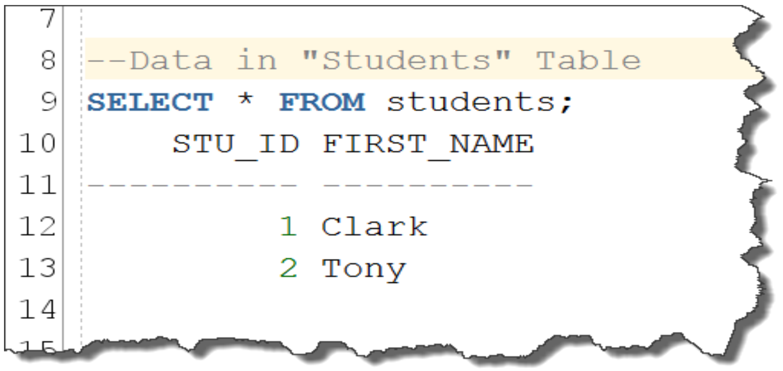

Worksheet Query Builder

```
--Structure of "Students" Table
DESC students;
Name       Null Type
---------- ---- ---------
STU_ID          NUMBER(2)
FIRST_NAME      CHAR(10)
```


Worksheet Query Builder

```
--Structure of "Students" Table
DESC students;
Name       Null Type
---------- ---- ---------
STU_ID          NUMBER(2)
FIRST_NAME      CHAR(10)
```

I have also inserted two rows into this table.



```
--Data in "Students" Table
SELECT * FROM students;
    STU_ID FIRST_NAME
---------- ----------
         1 Clark
         2 Tony
```

How to Declare a variable with Anchored Datatype

Next I will write an anonymous block where I will declare a variable with anchored data type and then initialize that variable by fetching value from this table.
So let's do it.

```
SET SERVEROUTPUT ON;
DECLARE
  v_fname    students.first_name%TYPE;
```

Here In the declaration section I have declared a variable by the name of **v_fname** with identical data type as the column **First Name** of table **Students**. This means that the data type of variable v_fname will be the varchar2 with data width 8. This is the data type and data width of the column first name of our table students.
So let's add execution section to this anonymous PL/SQL block and initialize this variable v_fname by fetching data from the table students.

```
SET SERVEROUTPUT ON;
DECLARE
  v_fname    students.first_name%TYPE;
BEGIN
  SELECT first_name INTO v_fname FROM students WHERE stu_id =1;
  DBMS_OUTPUT.PUT_LINE (v_fname);
END;
```

Here in this execution block I have a Select… Into statement using which I am fetching first name of the student whose stu_id is 1 and storing it into our variable v_fname.
That's how we declare a variable with anchored data type.

Proper way of Declaring and Initializing Constants in PL/SQL.

Like several other programming languages, the constant in PL/SQL is also a user defined identifier whose value remains unchanged throughout the program. Like variables in PL/SQL constants also need to be declared prior to their use. Furthermore you can only declare them in the declaration section of your PL/SQL block.

Syntax

PL/SQL has its own way of declaring a constant. To learn how to declare a constant in PL/SQL let's quickly take a look at the syntax. This is our syntax:

```
constant_name CONSTANT datatype (data-width) := value;
```

First you need to give a valid name to your constant followed by keyword CONSTANT that indicates the declaration of a constant in your program. Then you have to specify the data type and data width for your constant followed by the assignment operator and the value which you want to assign to your constant.

Note here. You must initialize a constant at its declaration. You have to initialize your constant at the time of its creation in declaration section of your PL/SQL block. You cannot initialize it anywhere else.

Example 1

First example will demonstrate how to declare and initialize a constant.

```
SET SERVEROUTPUT ON;
DECLARE
  v_pi CONSTANT NUMBER(7,6) := 3.141592;
BEGIN
  DBMS_OUTPUT.PUT_LINE (v_pi);
END;
/
```

This is a simple example of Constant declaration and initialization. Here in declaration section I have declared a constant v_pi and initialized it with the approximate value of pi. In the execution section we have our DBMS output statement which is displaying the value stored into our constant.

This is the proper way of declaring and initializing a constant in PL/SQL. We have two more attributes of PL/SQL constants to discuss which are “DEFAULT” and “NOT NULL”.

DEFAULT

You can use default keyword instead of assignment operator to initialize the constant in PL/SQL. Let’s do an example and see how to initialize a constant using DEFAULT keyword.

```
DECLARE
  v_pi CONSTANT NUMBER(7,6) DEFAULT 3.1415926;
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_pi);
END;
/
```

Same code just this time I used keyword DEFAULT instead of assignment operator for initializing the constant.

NOT NULL

Next attribute is NOT NULL. Using this attribute you can impose NOT NULL constraint while declaring constants as well as variables. This will prevent you from assigning NULL values to your constants or variables.
To impose not null constraint simply write NOT NULL keyword before the Keyword default or before assignment operator in case you have used it. Let me show you how

```
DECLARE
  v_pi CONSTANT NUMBER(7,6) NOT NULL DEFAULT 3.1415926;
BEGIN
  DBMS_OUTPUT.PUT_LINE (v_pi);
END;
/
```

How To Create, Declare, Initialize and Display Bind Variables in PL/SQL

There are two types of variables in Oracle database.

12. User variables. Discussed in PL/SQL Tutorial 2
13. Bind variables a.k.a Host variables.

Unlike user variables which can only be declared inside the declaration section of PL/SQL block you can declare bind variable anywhere in the host environment and that is the reason why we also refer bind variables as host variable.

Definition.

Bind variables in Oracle database can be defined as the variables that we create in SQL* PLUS and then reference in PL/SQL.

Oracle Docs

How To Declare a Bind Variable (Variable command)

Let's see how to create or say declare a bind variable. We can declare a bind variable using VARIABLE command. Variable command declares the bind variable which you can refer in PL/SQL. Also as I said earlier in this tutorial that in order to declare bind variables we do not need to write any PL/SQL block or section. Let's do an example and declare the our first bind variable

```
VARIABLE  v_bind1  VARCHAR2 (10);
```

See how easy it is to declare a bind variable in oracle database! You simply have to write a command which starts with keyword VARIABLE followed by the name of your bind variable which is completely user defined along with the data type and data width. That's how we declare a bind variable in Oracle database.
Did you notice that I didn't write any PL/SQL block or section here to declare this bind variable which is very unlike the user variable.

Other Uses of Variable Command.

Declaring the bind variable is the first use of this variable command there are few other uses of it also. Let's see what those are:

1. List all the bind variable declared in the session.

Yes using Variable command you can display the list of all the bind variables you have declared in the session. To display the list of all the bind variables you simply have to write the keyword variable and execute. Doing so will return list of all the bind variables.
Let's see.

```
VARIABLE;
```

Execute the above command and that will show you the list of all the bind variables that you have declared in your session.

2. To see the definition of bind variable.

Variable command can also show you the definition of any bind variable created in the session. By definition I mean the data type and data width of the variable. To see the definition of the bind variable you have to write the keyword VARIABLE followed by the name of the bind variable in question.

Let's do an example and see the definition of this bind variable v_Bind2.

```
Variable v_bind2;
```

Execution of above command will show you the definition of bind variable v_bind2.
So these were the few uses of Variable command in Oracle database. If you know any other uses then do tweet and tell me at @RebellionRider.

Restriction: If you are creating a bind variable of NUMBER datatype then you can not specify the precision and scale.

Initialize the Bind Variable

As we have now declared the bind variable next we have to initialize it. We have several different ways of initializing the bind variable. Let see what those are.

1. You can initialize the bind variable using "Execute" command.

Execute command is like a wrapper which works as an execution section of PL/SQL block. Let's see how it works. Let's initialize our bind variable v_bind1 with a string RebellionRider.

```
Exec  :v_bind1 := 'Rebellion Rider';
```

This statement starts with keyword Exec which is the starting 4 alphabets of Keyword Execute. You can either write whole keyword Execute or just the starting 4 alphabets "Exec" both will work fine. This is followed by the name of our bind variable which is v_bind1. After that we have assignment operator followed by the string Rebellion Rider, as it's a string thus it's enclosed in single quotes.
That's the first way of initializing the bind variable,

2. Initialize the bind variable by explicitly writing execution section of PL/SQL block.

If you do not like shortcuts and are willing to do some hard work of writing a few extra lines of code then this is for you.

```
SET SERVEROUTPUT ON;
BEGIN
   :v_bind1 := 'Manish Sharma';
END;
/
```

This is a simple execution block where I initialized the bind variable v_bind1 with the string Manish Sharma. That is how we initialize the bind variable in Oracle Database or in PL/SQL.

Referencing the Bind Variable

Manish why did you put the colon sign (:) before the name of bind variable (:v_bind1) while initializing it? Glad you asked.
Unlike user variables which you can access simply by writing their name in your code, you use colon before the name of bind variable to access them or in other words you can reference bind variable in PL/SQL by using a colon (:) followed immediately by the name of the variable as I did in the previous section.

Display The Current Value of Bind variable.

There are 3 ways of displaying the value held by bind variable in PL/SQL or say in Oracle Database.

14. Using DBMS OUTPUT package.
15. Using Print command
16. Setting Autoprint parameter on

Introduction of Conditional Control Statements in PL/SQL

Statements which allow you to control the execution flow of the program depending on a condition. In other words the statements in the program are not necessarily executed in a sequence rather one or other group of statements are executed depending on the evaluation of a condition.

In Oracle PL/SQL we have two types of conditional control statements which are

17. IF statements and
18. CASE statements

Both these statements can be further divided into different forms. For example IF statements has 3 different forms

19. IF THEN
20. IF THEN ELSE
21. IF THEN ELSEIF
    And CASE statement has 2 different forms such as
22. SIMPLE CASE and
23. SEARCHED CASE

Simple IF-THEN Control Statements in PL/SQL

IF-THEN is the most basic kind of conditional statements in PL/SQL that enables you to specify only a single group of action to be taken. You can also say that this specific group of action is taken only when a condition is evaluated to be true.

Correct me, if I am wrong, I think IF conditional statement is used in almost every programming language. If you disagree then tweet me @RebellionRider and tell me which programming languages do not support IF statements.

IF-THEN Structure

Before jumping into the tutorial let's quickly understand the structure/syntax of the IF-THEN statement.

```
IF condition THEN
   Statement1;
   …
   Statement N;
END IF;
```

In the starting we have the keyword IF which marks the beginning of IF-THEN block followed by a valid condition or a valid expression which will get evaluated followed by another keyword THEN. Similarly the reserved phrase END IF marks the ending of IF-THEN block. In between we have a sequence of executable statements which will get executed only if the condition evaluated to be true otherwise the whole IF-THEN block will be skipped.

Working

When an IF-THEN statement is executed a condition is evaluated to either True or False. If the condition evaluates to true, control is passed to the first executable statement of the IF-THEN construct. If the condition evaluates to false, control is passed to the first executable statement after the END-IF statement.

Examples 1

```
SET SERVEROUTPUT ON;
DECLARE
   v_num NUMBER := 9;
BEGIN
   IF v_num < 10 THEN
      DBMS_OUTPUT.PUT_LINE('Inside The IF');
   END IF;
   DBMS_OUTPUT.PUT_LINE('outside The IF');
END;
/
```

This is a very simple PL/SQL anonymous block. In the declaration section I have declared a variable v_num with data type NUMBER and initialized it with integer 9.
Let's come to our execution section. Here as you can see we have 2 DBMS_OUTPUT statements one is inside the IF-THEN block and another is outside it.
The first DBMS_OUTPUT statement will execute only if the condition of our IF-THEN block is evaluated as true otherwise it will be skipped but the 2nd DBMS_OUTPUT statement which is outside the IF-THEN block will execute every time you execute this PL/SQL block.
This means that if the condition is true then both the string INSDIE THE IF and OUTSIDE THE IF will be printed otherwise only OUTSIDE THE IF will be printed.

Examples 2

```
DECLARE
   v_website     VARCHAR2(30)     :=     'RebellionRider.com';
   v_author      VARCHAR2(30)     :=     'Manish';
BEGIN
   IF v_website ='RebellionRider.com' AND v_author= 'Manish' THEN
```

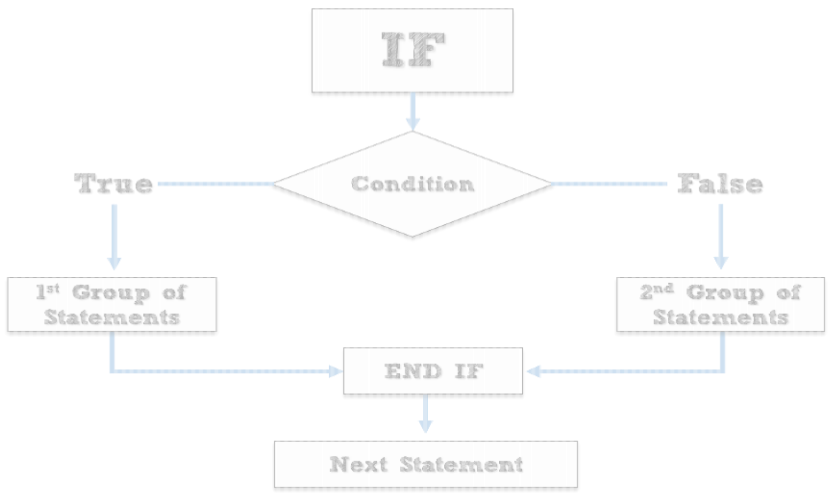
```
      DBMS_OUTPUT.PUT_LINE('Everything is Awesome :)');
    END IF;
    DBMS_OUTPUT.PUT_LINE('Give this Video a Thumbs Up');
  END;
  /
```

This one is slightly different than the previous example. Here we used logical AND operator in the condition. If this condition is evaluated to be true then both strings from both DBMS_OUTPUT statements will be printed otherwise only the string from 2nd DBMS_OUTPUT statement will be displayed back to you. This example is for showing how you can check multiple conditions in a single go using logical operator. You can even use logical OR instead of logical AND operator.

IF-THEN-ELSE Control Statements in PL/SQL

The previous tutorial was all about IF-THEN statement in Oracle PL/SQL. There we learnt that a simple IF-THEN statement enables us to specify the sequence of statements to be executed only if the condition is evaluated to be true. In case this condition is evaluated to be false then no special action is to be taken except to proceed with execution of the program.
To overcome this drawback in oracle PL/SQL we have IF-THEN-ELSE statement widely pronounced as IF-ELSE statement.

With IF-THEN-ELSE in PL/SQL we have two groups of executable statements, one which gets executed if the condition is evaluated to be true and another group gets executed if the condition is evaluated to be false. Once the IF-THEN-ELSE construct gets completed the next statement right after IF-THEN-ELSE block is executed.



Syntax

```
  IF condition THEN
    Statement 1;
  ELSE
```

```
    Statement 2;
  END IF;
    Statement 3
```

And here is the syntax as you can see IF-THEN-ELSE construct starts with the keyword IF and ends with the reserved phrase END IF. Followed by IF keyword we have to specify a valid condition which will get evaluated. If this condition is evaluated to be TRUE then control is passed to the statement 1, and if this condition is evaluated to be false then control will jump over to statement 2. Once the construct is completed then statement 3 is executed.

Example

Here in this example we will take a numeric input from the user and will check whether a user has entered an even number or an odd number using IF THEN ELSE statement. This is going to be a very easy example.

```
SET SERVEROUTPUT ON;
DECLARE
    v_num NUMBER := &enter_a_number;
BEGIN
    IF MOD (v_num, 2) = 0 THEN
        DBMS_OUTPUT.PUT_LINE (v_num || ' Is Even');
    ELSE
        DBMS_OUTPUT.PUT_LINE (v_num ||' is odd');
    END IF;
    DBMS_OUTPUT.PUT_LINE ('IF THEN ELSE Construct complete ');
END;
```

In the declaration section of this block I have declared a variable v_num with data type NUMBER and using substitution operator I am taking values from the user.
Next in the execution section of the block we have our IF THEN ELSE construct where we are checking whether the number entered by the user is even or odd. For this test I have used MOD function of Oracle library as our IF condition which will divide the first parameter by the second parameter and return the remainder or say Modulus. If this modulus is zero then number will be even otherwise number will be odd. Thus if the modulus comes as zero then control will jump over first DBMS_OUTPUT statement and will print the given string but if modulus is not zero then control will jump over the 2nd DBMS_OUTPUT statement and will display its string. Once this is complete then control will come out from the IF THEN ELSE construct and will print the third DBMS_OUTPUT statement

IF-THEN-ELSIF Control Statements in PL/SQL

In the previous tutorial we saw that IF ELSE condition gives us the provision of executing statements not only when the condition is evaluated to be true but also when the condition is evaluated to be false. But using IF ELSE statement we can only check one condition at a time, there is no provision for checking multiple conditions. This becomes its major drawback.
To overcome this we have IF THEN ELSIF condition in Oracle PL/SQL. Using this statement you can check multiple conditions unlike the IF conditions discussed in the previous tutorials.

Syntax

```
IF CONDITION 1 THEN
  STATEMENT 1;
ELSIF CONDITION 2 THEN
  STATEMENT 2;
ELSIF CONDITION 3 THEN
  STATEMENT 3;
  ...
ELSE
  STATEMENT N;
END IF;
```

Similar to the other IF conditions, keyword IF marks the beginning and reserved phrase END IF marks the ending of the block. Make sure to put a white space in between END and IF of ending phrase. In this ELSIF construct we have multiple conditions. CONDITION 1 through CONDITION N are a sequence of conditions that have to be evaluated for TRUE or FALSE. These conditions are mutually exclusive. This means that if condition 1 is evaluated to be TRUE then statement 1 is executed and control will jump over the first executable statement outside the ELSIF construct and rest of the conditions will be ignored. If condition 1 is evaluated to be false then the compiler will jump inside and check the rest ELSIF conditions to look for the one which is true. If it finds any then it will execute the corresponding statements otherwise it will run the else statement.

In simple words the IF THEN ELSIF statement is responsible for running the first statement for which the condition is true. Once this is done the rest of the conditions are not evaluated. In case none of the conditions are true, then the else_statements run provided that they exist; otherwise no action is taken by the IF THEN ELSIF statement.

Example

```
DECLARE
  v_Place VARCHAR2(30) := '&Enter Place';
BEGIN
  IF v_Place = 'Metropolis' THEN
    DBMS_OUTPUT.PUT_LINE('This City Is Protected By Superman');
  ELSIF v_Place = 'Gotham' THEN
    DBMS_OUTPUT.PUT_LINE('This City is Protected By Batman');
  ELSIF v_Place = 'Amazon' THEN
    DBMS_OUTPUT.PUT_LINE('This City is protected by Wonder Woman');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Please Call Avengers');
  END IF;
DBMS_OUTPUT.PUT_LINE('Thanks For Contacting us');
END;
```

This is a very simple Example where I have declared a variable v_place which has data type varchar2 and we are taking input in this variable from the user. As you can see here I have used substitution operator (&). Don’t forget to enclose this input string along with ampersand operator (or substitution operator &) inside single quotes as variable is of varchar2 data type. <

In the execution section we have ELSIF construct. Where we have one IF condition one else condition and 2 ELSIF conditions. Every condition is accompanied with a DBMS_OUTPUT statement which will get executed if the respective condition is evaluated to be true. Otherwise the DBMS_OUTPUT statement in ELSE section will run. Once this block is executed then control will come out and execute the first executable statement outside the ELSIF construct which is our last DBMS_OUTPUT statement.

Introduction Of Iterative statements and Simple Loop Explained

With the previous tutorial we finished our Conditional Control Statement Series and learnt the concept of different types of IF conditions in Oracle PL/SQL. That was a very important topic from the examination's perspective. Another topic which is again very important for your exam and could help you in getting good percentage is Iterative Statements.

Iterative statements famously known as Loops in Programming language. It executes block of statements or a part of a program several times. [**Click Here To Tweet This**]

Types of Loops in Oracle PL/SQL

There are 4 types of Loops in Oracle PL/SQL

24. Simple Loop
25. While Loop
26. Numeric For Loop and
27. Cursor For loop

In this series we will focus on the first 3 types of loops. The last type which is "Cursor For Loop" will be discussed with Cursor in the future Tutorial. Having said that let's start today's tutorial with Simple Loop.

Simple Loop

Simple loop is the most basic loop in Oracle PL/SQL

Syntax

```
LOOP
   Statement 1;
   Statement 2;
   …
   Statement 3;
END LOOP;
```

Here keyword LOOP marks the beginning and phrase END LOOP marks the ending of the loop. In between we have a sequence of executable statements.
As you can see in this syntax that unlike conventional loops here we do not have update statements or for that matter exist conditions which will terminate the loop. May be that is why we call this a simple loop.

Example 1

```
DECLARE
  v_counter NUMBER :=0;
  v_result NUMBER;
BEGIN
  LOOP
    v_counter := v_counter+1;
    v_result := 19*v_counter;
    DBMS_OUTPUT.PUT_LINE('19'||' x '||v_counter||' = '|| v_result);
  END LOOP;
END;
```

Here in this example as you can see we do not have any exit statement to terminate the loop. This means that if we execute this program then the execution will keep on printing till we halt it manually.

In this case Oracle PL/SQL gives us two clauses to terminate the loop

28. Exit
29. Exit When

Exit clause will terminate the loop when Exit condition is evaluated to be true. The exit condition is evaluated with the help of Simple IF THEN condition which we discussed in PL/SQL Tutorial 8. So let's see how you can use this exit statement in this example.

Example 2 Terminate Loop with EXIT

```
DECLARE
  v_counter NUMBER :=0;
  v_result NUMBER;
BEGIN
  LOOP
    v_counter := v_counter+1;
    v_result := 19*v_counter;
    DBMS_OUTPUT.PUT_LINE('19'||' x '||v_counter||' = '|| v_result);

    IF v_counter >=10 THEN
      EXIT;
    END IF;

  END LOOP;
END;
```

You simply have to add this IF THEN block either right above the phrase END LOOP or immediately below the keyword loop. What this IF THEN block will do? This block will keep an eye on your counter and will tell the control to exit the loop when counter either becomes greater than or equal to 10. Which means loop will execute for 10 times.

Example 3 Terminate the Loop with EXIT WHEN Clause

Second way of terminating the loop is by using EXIT WHEN clause. Using this clause you can replace this whole IF THEN block with a simple single statement

```
DECLARE
  v_counter NUMBER :=0;
  v_result NUMBER;
BEGIN
  LOOP
    v_counter := v_counter+1;
    v_result := 19*v_counter;
    DBMS_OUTPUT.PUT_LINE('19'||' x '||v_counter||' = '|| v_result);

    EXIT WHEN i_counter>=10;

  END LOOP;
END;
```

While Loop In PL/SQL

Concept of While Loop is not new to the programming world and almost all the programming languages support this concept. While loop, is not the exception to other loops. It also executes block of statements several times but this loop is best usable when number of iterations to be performed are unknown. [Click Here To Tweet This]

Syntax

```
WHILE condition LOOP
  Statement 1;
  Statemen 2;
  …
  Statement 3;
END LOOP;
```

The keyword WHILE marks the beginning of the loop followed by word CONDITION which will serve your test condition. This will get evaluated either to be true or to be false and at the end of our first line we have another keyword which is LOOP. The statements 1 through N are sequence of executable statements which define the body of the loop. And at the end we have a reserved phrase END LOOP which indicates the ending of the while loop.

In order to execute the body of the loop the test condition needs to be true. If this condition is evaluated to be true then the control will jump inside the loop and execute whatever statements it has. This iteration will continue until the test condition becomes false. As soon as the test condition is evaluated to be false the control will come out of the loop and execute the statement which immediately follows the loop.

Examples

Example 1

```
DECLARE
  v_counter NUMBER :=1;
  v_result NUMBER ;
```

```
BEGIN
  WHILE v_counter <= 10
  LOOP
    v_result := 9 *v_counter;
    DBMS_OUTPUT.PUT_LINE('9'||' x '||v_counter||' = '||v_result);
    v_counter := v_counter+1;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('out');
END;
/
```

Example 2: Boolean Expression as test Condition

```
DECLARE
  v_test BOOLEAN := TRUE;
  v_counter NUMBER := 0;
BEGIN
  WHILE v_test LOOP
    v_counter := v_counter+1;
    DBMS_OUTPUT.PUT_LINE( v_counter );
    IF v_counter = 10 THEN
      v_test := FALSE;
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('This Statement is outside the loop and will always execute');
END;
/
```

With the Boolean expression in loop we have to write the code which will change its value to false and terminate the loop. Failing to do so can make your loop an infinity loop. In the above program the simple IF THEN block inside the loop body will change the value of the Boolean expression v_test and set it on false when counter becomes equal to 10 this till terminate the loop and bring the control over the first statement immediately outside the loop body.

Program to Print multiplication table of 19 using Boolean Expression In While Loop

As promised in my video here is the code for printing multiplication table of 19 using Boolean expression in while loop.

```
DECLARE
   v_test BOOLEAN := TRUE;
   v_counter NUMBER :=1;
   v_result NUMBER;
BEGIN
   WHILE v_test LOOP
      v_result := 19 * v_counter;
      DBMS_OUTPUT.PUT_LINE('19'||' x '||v_counter||' = '||v_result);
      -- Loop Termination Code
      IF v_counter =10 THEN
          v_test := FALSE;
      END IF;
      v_counter := v_counter +1;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('Outside the loop');
END;
/
```

Numeric FOR Loop In PL/SQL

The simplicity and easy to use behavior of FOR loop has won the hearts of millions and has become the most widely used loop in programming. In PL/SQL we have two types of FOR loops:

30. Numeric FOR loop and
31. Cursor FOR loop.

FOR loop allows you to execute the block of statements repeatedly for a fixed number of time whereas WHILE loop is better suited when the number of iterations are unknown. [**Click Here To Tweet This**]

This tutorial will concentrate on Numeric “FOR LOOP”. We’ll leave the Cursor FOR loop for the future when we will learn the concepts of Cursor.

Syntax

```
FOR        loop_counter      IN       [REVERSE]       lower       limit..       upper_limit       LOOP
   Statement1;
   Statement                                                                                      2;
   …
   Statement                                                                                      3;
END                                                                                            LOOP;
```

For the in-depth explanation of the above syntax please watch my video. There I have explained the same in detail. Now let’s see some examples.

Examples of Numeric FOR Loop In Oracle PL/SQL.

Example 1: FOR loop

```
SET                              SERVEROUTPUT                              ON;
BEGIN
   FOR              v_counter              IN              1..10              LOOP
```

```
        DBMS_OUTPUT.PUT_LINE(v_counter);
    END LOOP;
END;
```

Here we only have the execution section and inside that we have our FOR loop which will print the value of v_counter variable from 1 to 10.

Have you noticed that we didn't declare the v_counter variable anywhere in the program? Even we don't have the declaration section here in this code. This is because variable v_counter is an implicit index integer variable which gets declared automatically with the definition of FOR loop. Moreover the variable v_counter will increment by 1 with each iteration automatically by FOR loop construct thus you do not need to write update statement (v_counter := v_counter +1) explicitly. As a matter of fact if you will try to write the update statement in the "FOR loop" then you will get an error.

Example 2: FOR Loop with IN REVERSE keyword.

Now suppose you want to print the counting, same as we did in the previous example but this time in reverse order. To do so you don't have to change the loop definition or even you don't have to add any extra line of codes, PL/SQL block will be same as of the previous example. You just have to add one keyword REVERSE immediately after IN keyword in FOR LOOP definition. Rest of the code will remain the same as that of the previous example.

```
BEGIN
    FOR v_counter IN REVERSE 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(v_counter);
    END LOOP;
END;
/
```

This code will give you counting from 1 to 10 in reverse order on execution.

Example 3: Multiplication Table using Numeric FOR loop

```
DECLARE
    v_result NUMBER;
BEGIN
    FOR v_counter IN 1..10 LOOP
        v_result:= 19*v_counter;
        DBMS_OUTPUT.PUT_LINE(v_result);
    END LOOP;
END;
/
```

In this example we need one extra variable to store the result of the multiplication thus we declared a variable v_result with NUMBER data type. In the execution section we have our "FOR loop" and this time inside the loop we have only two statements. First is an arithmetic expression which will perform the multiplication of our table and will store the result in v_result variable. Second is the output statement which will display you the result in a formatted manner.

Introduction Of Triggers

With the previous tutorial we finished the series of Iterative statements or looping statements in Oracle PL/SQL so now it's time to move on to a little more advance topic in PL/SQL. Thus today's tutorial is about Introduction of Triggers in Oracle PL/SQL. In this tutorial I will try to explain you the concepts of triggers, and try to give you the answers of almost all the questions which you might face in your **Certification Exam** or in your **Interview**.

Definition of Triggers in Database

Triggers are named PL/SQL blocks which are stored in the database or we can also say that they are specialized stored programs which execute implicitly when a triggering event occurs [Click Here To Tweet This] which means we cannot call and execute them directly instead they only get triggered by events in the database.

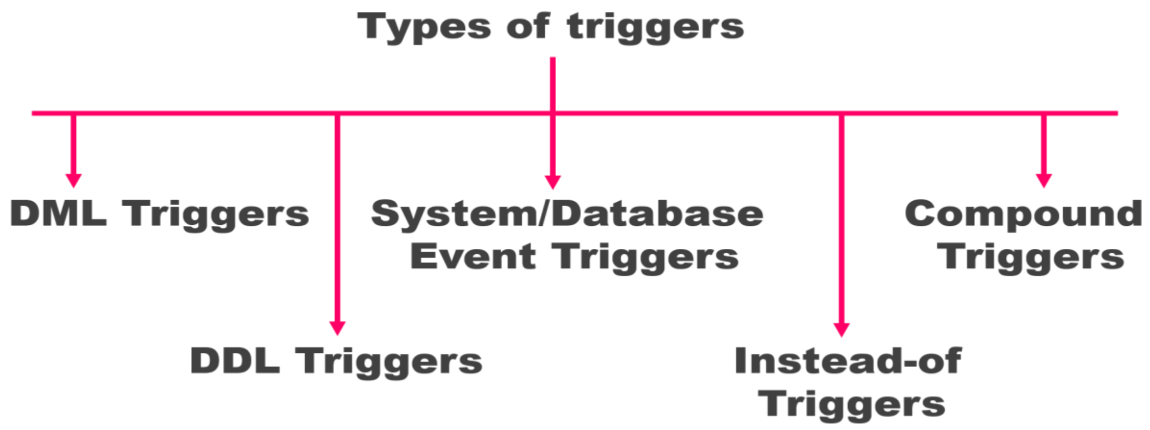
Events Which Fires the Triggers

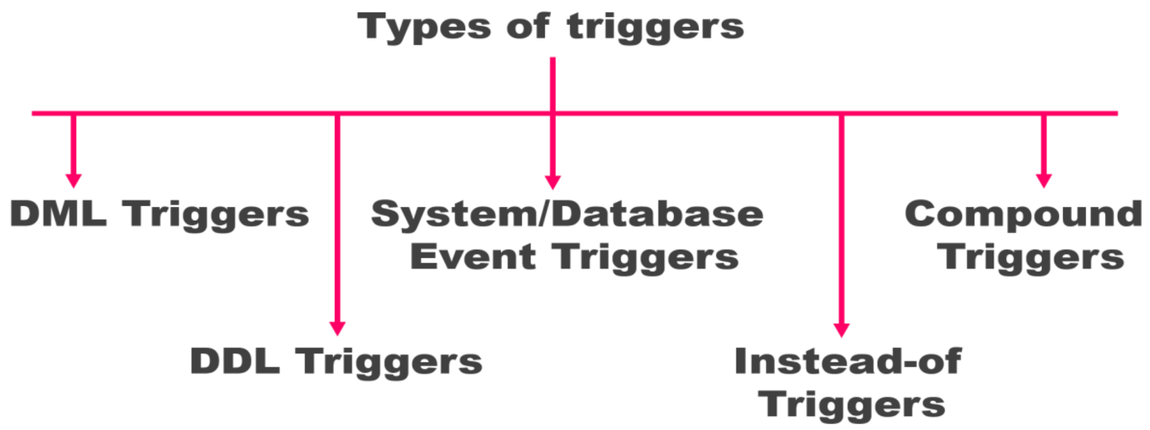
These events can be anything such as

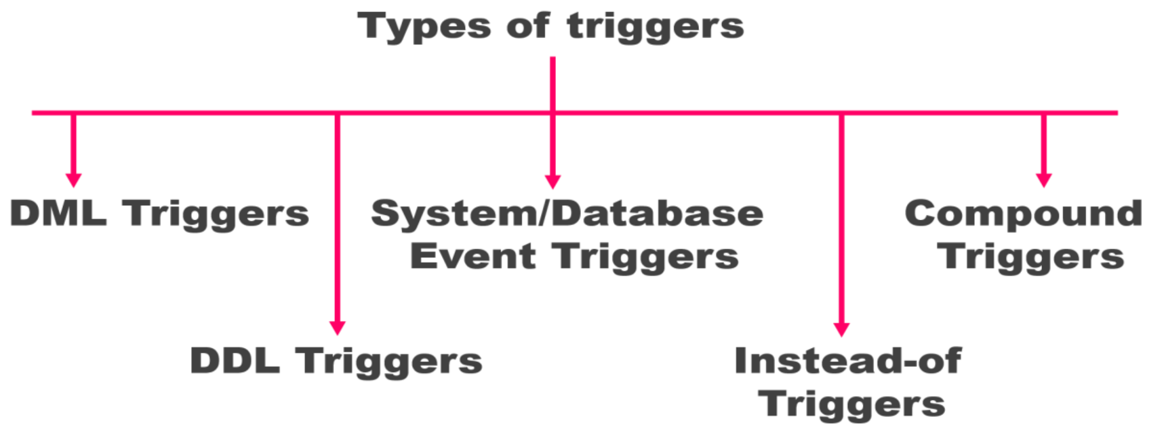
32. **A DML Statement**. For example Update, Insert or Delete, executing on any table of your database. You can program your trigger to execute either BEFORE or AFTER executing your DML statement. For example you can create a trigger which will get fired Before the Update statement. Or you can create a trigger which will get triggered after the execution of your INSERT DML statement.
33. Next type of triggering statement can be a **DDL Statement** such as CREATE or ALTER. These triggers can also be executed either BEFORE or AFTER the execution of your DDL statement. These triggers are generally used by DBAs for auditing purposes and they really come in handy when you want to keep an eye on the various changes on your schema such as who created the object or which user. Just like some cool spy tricks.
34. **A system event**. Yes, you can create a trigger on a system event and by system event I mean shut down or startup of your database.
35. Another type of triggering event can be **User Events**such as log off or log on onto your database. You can create a trigger which will either execute before or after the event and record the information such as time of event occur, the username who created it.

Types of Triggers

There are 5 types of triggers in oracle database in which 3 of them are based on the triggering event which are discussed in the previous section.







36. **Data Manipulation Language Triggers or DML triggers**

37. As the name suggests these are the triggers which depend on DML statements such as Update, Insert or Delete and they get fired either before or after them. Using DML trigger you can control the behavior of your DML statements. You can audit, check, replace or save values before they are changed. Automatic Increment of your Numeric primary key is one of the most frequent tasks of these types of triggers.

38. **Data Definition Language Triggers or DDL triggers.**

39. Again as the name suggests these are the type of triggers which are created over DDL statements such as CREATE or ALTER and get fired either before or after execution of your DDL statements. Using this type of trigger you can monitor the behavior and force rules on your DDL statements.

40. **System or Database Event triggers.**

41. Third type of triggers is system or database triggers. These are the type of triggers which come into action when some system event occurs such as database log on or log off. You can use these triggers for auditing purposes for example keeping an eye on information of system access like say who connects with your database and when. Most of the time System or Database Event triggers work as Swiss Knife for DBAs and help them in increasing the security of the data.

42. **Instead-of Trigger**

43. This is a type of trigger which enables you to stop and redirect the performance of a DML statement. Often this type of trigger helps you in managing the way you write to non-updatable views. You can also see the application of business rules by INSTEAD OF triggers where they insert, update or delete rows directly in tables that are defining updatable views. Alternatively, sometimes the INSTEAD OF triggers are also seen inserting, updating or deleting rows in designated tables that are otherwise unrelated to the view.

44. **Compound triggers**

45. These are multi-tasking triggers that act as both statement as well as row-level triggers when the data is inserted, updated or deleted from a table. You can capture information at four timing points using this trigger:
(a) before the firing statement;
(b) prior to change of each row from the firing statement;
(c) post each row changes from the firing statement;
(d) after the firing statement.
All these types of triggers can be used to audit, check, save and replace the values even before they are changed right when there is a need to take action at the statement as well as row event levels.

Syntax

```
CREATE [OR REPLACE] TRIGGER Ttrigger_name
{BEFORE|AFTER} Triggering_event ON table_name
[FOR EACH ROW]
[FOLLOWS another_trigger_name]
[ENABLE/DISABLE]
[WHEN condition]
DECLARE
  declaration statements
BEGIN
  executable statements
EXCEPTION
  exception-handling statements
END;
```

For the detailed explanation of the syntax I would suggest you to watch the video tutorial. There I have explained each and every clause of the syntax in detail.

Uses of triggers.

46. Using triggers we can enforce business rules that can't be defined by using integrity constants.
47. Using triggers we can gain strong control over the security.
48. We can also collect statistical information on the table access.
49. We can automatically generate values for derived columns such as auto increment numeric primary key.
50. Using triggers you can prevent invalid transaction.

Restriction on Triggers

51. Maximum size of the trigger body must not exceed 32,760 bytes because triggers' bodies are stored in LONG datatypes columns.
52. A trigger may not issue transaction control statements or TCL statements such as COMMIT, ROLLBACK or SAVEPOINT. All operations performed when the trigger fires, become part of a transaction. Therefore whenever this transaction is rolled back or committed it leads to the respective rolling back or committing of the operations performed.
53. Any function or procedure called by a trigger may not issue a transactional control statement unless it contains an autonomous transaction.
54. Declaring LONG or LONG RAW variable is not permissible in the body of the trigger.

DML Trigger with Examples

As the name suggests these are the triggers which execute on DML events or say depend on DML statements such as Update, Insert or Delete. Using DML trigger you can control the behavior of your DML statements. [**Click Here To Tweet This**]

Since the theory has already been discussed in the previous tutorial hence I won't bore you further. You can refer to the previous tutorial "Introduction to Triggers" anytime.

Examples

In order to demonstrate the creation process of DML trigger we need to first create a table.

```
CREATE TABLE superheroes (
  sh_name VARCHAR2 (15)
);
```

I have created this table with the name SUPERHEROES which has only one column sh_name with data type varchar2 and data width 15. Now I will write a DML trigger which will work on this table. So the table is created. Now let's do some examples which will help you in understanding the concepts more clearly.

Before that a simple tip: Always remember to set your server output ON otherwise the output message returned from your trigger will not be displayed back to you.

```
SET SERVEROUTPUT ON;
```

Example 1. Before Insert Trigger

In the first example we will see how to create a trigger over Insert DML. This trigger will print a user defined message every time a user inserts a new row in the superheroes table.

```
CREATE OR REPLACE TRIGGER bi_Superheroes
BEFORE INSERT ON superheroes
FOR EACH ROW
ENABLE
DECLARE
  v_user VARCHAR2 (15);
BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_OUTPUT.PUT_LINE('You Just Inserted a Row Mr.'|| v_user);
END;
/
```

On successfully compiling, this trigger will show you a string along with the user name who performed the "Insert" DML on superheroes table. Thus you check this trigger by Inserting a row in Superheroes table.

```
INSERT INTO superheroes VALUES ('Ironman')
```

Example 2: Before Update Trigger.

Update Trigger is the one which will execute either before or after Update DML. The creation process of an Update trigger is the same as that of Insert Trigger. You just have to replace Keyword INSERT with UPDATE in the 2nd Line of the above example.

```
CREATE OR REPLACE TRIGGER bu_Superheroes
BEFORE UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
  v_user VARCHAR2 (15);
BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_OUTPUT.PUT_LINE('You Just Updated a Row Mr.'|| v_user);
END;
/
```

On successfully compiling, this trigger will print a user defined string with the username of the user who updated the row. You can check this trigger by writing an update DML on the superheroes table.

```
UPDATE superheroes SET SH_NAME = ‘Superman’ WHERE SH_NAME='Ironman'
```

Example 3: Before Delete Trigger

Similar to Insert and Update DML you can write a trigger over Delete DML. This trigger will execute either before or after a user deletes a row from the underlying table.

```
CREATE OR REPLACE TRIGGER bu_Superheroes
BEFORE DELETE ON superheroes
```

```
FOR EACH ROW
ENABLE
DECLARE
  v_user VARCHAR2 (15);
BEGIN
  SELECT user INTO v_user FROM dual;
  DBMS_OUTPUT.PUT_LINE('You Just Deleted a Row Mr.'|| v_user);
END;
/
```

You can check the working of this trigger by executing a DELETE DML on underlying table which is superheroes.

```
  DELETE FROM superheroes WHERE sh_name = 'Superman';
```

Above three examples showed you 3 different triggers for 3 different DML events on one table. Don't you think that if we can cover all these 3 events in just 1 trigger then it will be a great relief? If you think so then my dear friend I have some good news for you. Let me show you how we can achieve this feat.

INSERT, UPDATE, DELETE All in One DML Trigger Using IF-THEN-ELSIF

```
CREATE OR REPLACE TRIGGER tr_superheroes
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
  v_user VARCHAR2(15);
BEGIN
SELECT
  user INTO v_user FROM dual;
IF INSERTING THEN
  DBMS_OUTPUT.PUT_LINE('one line inserted by '||v_user);
ELSIF DELETING THEN
  DBMS_OUTPUT.PUT_LINE('one line Deleted by '||v_user);
ELSIF UPDATING THEN
  DBMS_OUTPUT.PUT_LINE('one line Updated by '||v_user);
END IF;
END;
/
```

Using this one trigger you can achieve the same results as that of the above three triggers. I have explained every single line of this trigger along with the other three triggers in detail in my Video tutorial. I highly suggest you to watch that tutorial.

Trigger For Auditing

Next we will write a trigger on the source table superheroes and will store the data into the auditing table sh_audit.

```
CREATE OR REPLACE TRIGGER superheroes_audit
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
  v_user varchar2 (30);
  v_date varchar2(30);
BEGIN
  SELECT user, TO_CHAR(sysdate, 'DD/MON/YYYY HH24:MI:SS') INTO v_user, v_date FROM dual;

  IF INSERTING THEN
    INSERT INTO sh_audit (new_name,old_name, user_name, entry_date, operation)
    VALUES(:NEW.SH_NAME, Null , v_user, v_date, 'Insert');

  ELSIF DELETING THEN
    INSERT INTO sh_audit (new_name,old_name, user_name, entry_date, operation)
    VALUES(NULL,:OLD.SH_NAME, v_user, v_date, 'Delete');

  ELSIF UPDATING THEN
    INSERT INTO sh_audit (new_name,old_name, user_name, entry_date, operation)
    VALUES(:NEW.SH_NAME, :OLD.SH_NAME, v_user, v_date,'Update');
  END IF;
END;
/
```

I would highly suggest you to watch the YouTube Video Tutorial on the same topic since there I have explained the working of this particular trigger line by line in detail.

On successful compilation this trigger will insert a row containing auditing data such as the data inserted, updated and deleted from the source table superheroes along with the username who tampered the data as well as the date and time when it was done and also the name of DML statement executed by user to tamper the data of your table.

Pseudo Records (New/Old)

If you will carefully see the Insert statements used in the IF-THEN-ELSIF statements in the above code, we used some Pseudo Records such as ':New' or ':Old' followed by the name of the column of our source table sh_name.

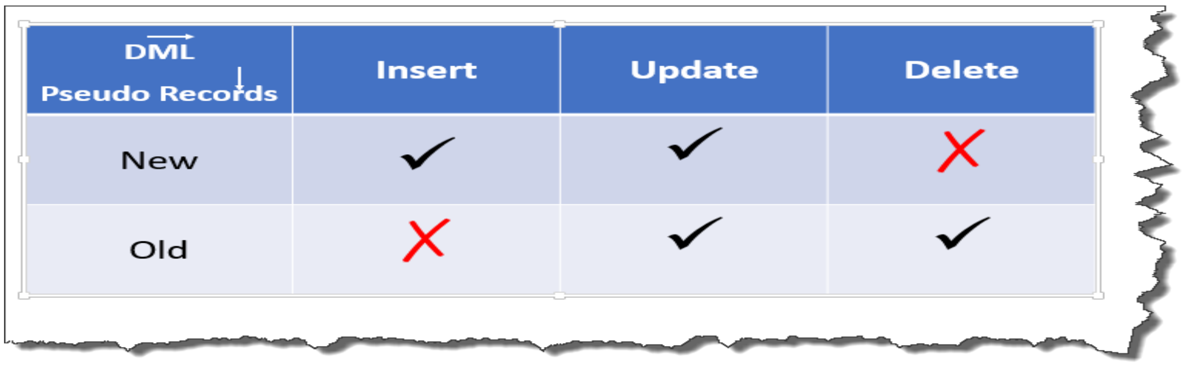
These Psuedo Records helps us in fetching data from the sh_name column of the underlying source table 'Superheroes' and storing it into the audit table sh_audit.

Pseudo Record '**: NEW**', allows you to access a row currently being processed. In other words, when a row is being inserted or updated into the superheroes table. Whereas Pseudo Record '**: OLD**' allows you to access a row which is already being either Updated or Deleted from the superheroes table.

In order to fetch the data from the source table, you have to first write the proper Pseudo Record (New/Old) followed by dot (.) and the name of the column of the source table whose value you want to fetch. For example in our case we want to fetch the data from sh_name column which belongs to our source table *Superheroes*. Thus we will write "**:New.sh_name**" for fetching the current value and to fetch the previously stored value we will write "**:OLD.sh_name**". Once the values are fetched the INSERT dml will store these values into the respective columns of the audit table.

Restriction on Pseudo Record

- For an INSERT trigger, OLD contain no values, and NEW contain the new values.
- For an UPDATE trigger, OLD contain the old values, and NEW contain the new values.
- For a DELETE trigger, OLD contain the old values, and NEW contain no values.



DML → Pseudo Records ↓	Insert	Update	Delete
New	✓	✓	✗
Old	✗	✓	✓

Once you execute and compile this trigger then you can take it on a test run by writing DML statements on the underlying source table 'Superheroes'. For example you can try Inserting a row in superheroes table and then check the audit table whether there is some data or not.

```
INSERT INTO superheroes VALUES ('Superman');
```

Similarly you can write Update and Delete DML statements on Superheroes table.

```
UPDATE SUPERHEROES SET SH_NAME = 'Ironman' WHERE SH_NAME='Superman';
Or
DELETE FROM superheroes WHERE SH_NAME = 'Ironman';
```

As soon as you execute any of these DML statements on the underlying table superheroes, the trigger will execute in the background and insert the audit data into the audit table sh_audit.

Synchronized Table Backup Using DML Trigger

Recently we learnt how to audit a table using DML Triggers in Oracle database now we will see how we can make a synchronized backup copy of a table using the same. By synchronized backup copy I mean the backup table gets automatically populated or updated with the main table simultaneously. [**Tweet This**]

For the demonstration we will require two identical tables; one which will serve as your main table that will accept the data from your database user and the second which will be your backup table. I will use the Superheroes table which we have been using since the beginning of this DML trigger series as our main table.

```
CREATE TABLE    superheroes(
  Sh_name    VARCHAR2(30)
);
```

Next we will have to create an identical table to this one which will work as our backup table.

Let’s create this backup table.

```
CREATE TABLE superheroes_backup
AS
SELECT * FROM superheroes WHERE 1=2;
```

The above command will create the identical table just like the main table superheroes only without data.

Next we have to write the trigger which will insert, update or delete the rows from the backup table when someone does the same with our main table.

```
CREATE or REPLACE trigger Sh_Backup
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
BEGIN
  IF INSERTING THEN
    INSERT INTO superheroes_backup (SH_NAME) VALUES (:NEW.SH_NAME);
  ELSIF DELETING THEN
    DELETE FROM superheroes_backup WHERE SH_NAME =:old.sh_name;
  ELSIF UPDATING THEN
    UPDATE superheroes_backup
    SET SH_NAME =:new.sh_name WHERE SH_NAME =:old.sh_name;
  END IF;
END;
/
```

After successful execution of the trigger, changes from the main table will get reflected on the backup table too. For detail explanation please watch the video on the same topic. Before ending up this blog a quick disclaimer, though you can write this trigger for any table but I would not advise you to use such trigger on those tables that involve heavy data input, deletion and updation. This is chiefly due to the performance reasons.

Schema and Database Auditing

DDL triggers are the triggers which are created over DDL statements such as CREATE, DROP or ALTER. Using this type of trigger you can monitor the behavior and force rules on your DDL statements. [**Tweet This]**

For detailed theory on the subject I would suggest you to read Introduction of Triggers in Oracle database. In this tutorial we will concentrate on the particular part where we will learn how to create a DDL trigger for auditing the schema/user and the whole database.

In order to proceed ahead and start writing the trigger first we need a table in which we can journal the auditing information created by the trigger.

```
  CREATE TABLE schema_audit
  (
    ddl_date          DATE,
    ddl_user          VARCHAR2(15),
    object_created    VARCHAR2(15),
    object_name       VARCHAR2(15),
    ddl_operation     VARCHAR2(15)
);
```

In case of schema/user auditing using DDL trigger creates this table in the same schema which you are auditing and in case of Database auditing using DDL trigger create this table in sys or system schema (sys or system both schemas can be used to perform database auditing).

DDL Trigger for Schema Auditing

First you need to log on to the database using the schema which you want to audit. For example suppose you want to create the DDL trigger to audit the HR schema then log on to your database using the HR schema.

Then Write, Execute and Compile the below trigger.

```
  CREATE OR REPLACE TRIGGER hr_audit_tr
  AFTER DDL ON SCHEMA
  BEGIN
    INSERT INTO schema_audit VALUES
    (
      sysdate,
      sys_context('USERENV','CURRENT_USER'),
      ora_dict_obj_type,
      ora_dict_obj_name,
      ora_sysevent
    );
  END;
  /
```

If you will notice carefully the second line of the code ("AFTER DDL ON SCHEMA") indicates that this trigger will work on the schema in which it is created. On successful compilation this trigger will insert the respective information such as the date when the DDL is executed, username who executed the DDL, type of database object created, name of the object given by the user at the time of its creation and the type of DDL into the table which we created earlier.

DDL Trigger for Database Auditing.

Similar to the schema auditing with some minor changes in the above trigger you can audit your database too. But for that first you need to logon to the database using either SYS user or SYSTEM user.

After doing that you have to create the above shown table under the same user so that your trigger can dump the auditing data without any read and write errors.

```
  CREATE OR REPLACE TRIGGER db_audit_tr
  AFTER DDL ON DATABASE
  BEGIN
    INSERT INTO schema_audit VALUES
    (
        sysdate,
        sys_context('USERENV','CURRENT_USER'),
        ora_dict_obj_type,
        ora_dict_obj_name,
        ora_sysevent
    );
END;
/
```

If you notice the second line of this code carefully then you will find that we have replaced the keyword Schema with the keyword Database which indicates that this trigger will work for the whole database and will perform the underlying work.

To create a trigger on database we require ADMINISTER DATABASE TRIGGER system privilege. All the administrative users such as sys or system already has these privileges by default that is the reason we created this database auditing DDL trigger using these users. Though you can create the same trigger with any user by granting the same privileges to them but that is not advisable because of your database security reasons.

Hope you enjoyed this tutorial and learnt something. You can help your friends or other people by sharing this blog on your social or by any means you find convenient. Thanks have a great day.

Schema Level Database Logon Trigger

Database event triggers also known as system event triggers come into action when some system event occurs such as database log on, log off, start up or shut down.[Tweet This] These types of triggers are majorly used for monitoring activity of the system events and have been proved quite a powerful tool for a DBA.

Types of Database Event Triggers.

55. Schema Level Event Triggers
56. Database Level Event Triggers

Schema level event triggers can work on some specific schemas while the database event triggers have database wide scope. In other words database event triggers can be created to monitor the system event activities of either a specific user/schema or the whole database.

Object/System Privileges

Schema level event triggers can be created by any user of your database who has CREATE TRIGGER system privilege while the database event trigger can only be created by privileged user such as SYS or SYSTEM who has 'Administrative Database Trigger' System Privileges.

Syntax

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE | AFTER database_event ON database/schema
BEGIN
   PL/SQL Code
END;
/
```

Please watch the video tutorial for detailed explanation of the syntax.

Example. Schema Level Event Trigger.

Suppose user HR is a control freak and wants to monitor its every log on and log off activity. In this case what HR can do is, create event triggers on Log on and log off database event in its own schema.

Step 1: Connect to the database

Connect to the database using the user/schema in which you want to create the trigger. For the demonstration I will connect using my HR user.

```
C:/> Conn hr/hr
```

Or if you are using SQL Developer then read here on how to connect to the database using the same.

Step 2: Create a Table

Next you will need a table to store the logon and logoff data.

```
CREATE TABLE hr_evnt_audit
(
  event_type VARCHAR2(30),
  logon_date DATE,
  logon_time VARCHAR2(15),
  logof_date DATE,
  logof_time VARCHAR2(15)
);
```

Step3: Write the trigger Logon Schema Event Trigger.

Now you are connected to the database using the desired user and also have the table ready to store the data. The only thing which is left is the trigger.

This trigger will fire every time HR user logs on to the database and respective values will be stored into the table which we just created in the step 2.

```
CREATE OR REPLACE TRIGGER hr_lgon_audit
AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO hr_evnt_audit VALUES(
    ora_sysevent,
    sysdate,
    TO_CHAR(sysdate, 'hh24:mi:ss'),
    NULL,
    NULL
  );
  COMMIT;
END;
/
```

That is how we create a LogOn event trigger in Oracle Database. You can watch the video tutorial on my YouTube channel on the same topic where I created and tested the trigger live. And Stay tuned as in the next tutorial we will see how to create a logoff event trigger in Oracle Database.

LogOff Database Event Trigger

Welcome to the next tutorial of PL/SQL Triggers in Oracle Database. In this tutorial we will learn how to write:

57. Schema Level Logoff System Event Trigger and
58. Database Level Logoff System Event Trigger

For this tutorial, knowledge of System/Database Event trigger is required which we have already discussed in the previous tutorial.

Unlike logon database event trigger which fires ***after*** a valid logon attempt by a user on the database, logoff triggers execute ***before*** the user logs off from the database. Logoff trigger can be proved as a versatile tool for a DBA as well as for a database user. [Click Here To Tweet This]

Schema Level Logoff System Event Trigger

As I explained in the previous tutorial that a schema level trigger is one which worked only for a specific schema in which it is created or designed to work for. Any user of the database who has "***Create Trigger***" system privilege can design and create this trigger.

Example: How To Create Schema Level Logoff event Trigger In Oracle PL/SQL

Let's write a trigger to audit the logoff

Step 1: Logon to the database

Logon to the database using any user such as HR, SH, OE or any other you want.

```
C:\> SQLPLUS hr/hr
```

Step 2: Create a table.

Create a table to dump the data generated by your schema level logoff trigger.

```
CREATE TABLE hr_evnt_audit
(
   event_type VARCHAR2(30),
   logon_date DATE,
   logon_time VARCHAR2(15),
   logof_date DATE,
   logof_time VARCHAR2(15)
);
```

Step 3: Write the trigger.

Below written trigger will execute every time user HR logs off from the database.

```
CREATE OR REPLACE TRIGGER log_off_audit
BEFORE LOGOFF ON SCHEMA
BEGIN
   INSERT INTO hr_evnt_audit VALUES
   (
      ora_sysevent,
      NULL,
      NULL,
      SYSDATE,
      TO_CHAR(sysdate, 'hh24:mi:ss')
    );
    COMMIT;
  END;
  /
```

On successful compilation this trigger will fire every time the user, using which it was created, logs off from the database and that user in our case is HR.

As I said above that this trigger is bound to work only for the user in which it is created. What if we want to keep track of all the logoff activities of all the users of the database? In such a scenario we can write a database level system event trigger.

Database Level System/Database Event Trigger.

An often asked question in the interview is what are the differences between Schema Level and Database Level system Event triggers? There can be variety of answers for this question but major differences are as follows.

As the name suggests Database event trigger fires for the entire database or you can say that it fires for all the schemas created under the database in which these triggers are created which is unlike Schema Level System Event trigger which runs for a specific schema only.

Also database level system event trigger can only be created by high privileged users such as sys or system or any user who has ***ADMINISTER DATABASE TRIGGER*** system privilege where Schema level system event trigger can be created by any user of the database on its own schema which has Create Trigger System Privilege.

Example: How To Create Database Level Logoff event Trigger In Oracle PL/SQL

Creation process of Database Level Logoff Event Trigger is pretty similar to the trigger which we just saw except for a few minute changes.

Step 1: Logon to the database

As only the user with **ADMINISTER DATABASE TRIGGER** system privilege can create a database level event trigger thus we need to make sure that this time we should log on to the database using one of these users.

```
  C:\> SQLPLUS / as SYSDBA
```

Step 2: Create a Table

Again in order to store the audit data we need to create a table where this trigger will journal the entries of all the users. The structure of the table will be pretty similar to the above one except one extra column for storing the username with the help of which we can clearly identify the details and avoid the confusion.

```
  CREATE TABLE db_evnt_audit
  (
    User_name VARCHAR2(15),
    event_type VARCHAR2(30),
    logon_date DATE,
    logon_time VARCHAR2(15),
    logof_date DATE,
    logof_time VARCHAR2(15)
);
```

Step 3: Write the Database Level logoff system event trigger.

Following Trigger will keep an eye on the logoff activity of the user of the database.

```
CREATE OR REPLACE TRIGGER db_lgof_audit
BEFORE LOGOFF ON DATABASE
BEGIN
    INSERT INTO db_evnt_audit
    VALUES
    (
        user,
        ora_sysevent,
        NULL,
        NULL,
        SYSDATE,
        TO_CHAR(sysdate, 'hh24:mi:ss')
    );
END;
/
```

Code is very similar to the previous one except that this time this trigger will execute for all the users of the database. This we are making sure by using the keyword DATABASE instead of SCHEMA in the second line of the trigger unlike the previous trigger.

There can be various ways of taking advantage of these triggers; it depends on your creativity. All the above examples are meant to teach you the proper way of creating a System/Database Event Trigger.

Startup and Shutdown Database Event Trigger

In the series of Database Event Triggers so far we have learned how to create and work with database event Logonand Logoff triggers. Now the only options which are left are Database event Startup and Shutdown Triggers. Today in this blog we will learn how to create these triggers with some easy to understand examples.

Startup Trigger.

Startup triggers execute during the startup process of the database.[Tweet This] In order to create a database event trigger for shutdown and startup events we either need to logon to the database as a **user with DBA privileges**such as sys or we must possess the **ADMINISTER DATABASE TRIGGER** system privilege.

Suggested Reading: System Privileges

Examples

Step1: Logon to the database

In order to create a trigger on Startup Database Event first we will have to logon to our database using the user SYS with DBA privileges.

Step 2: Create a Table

To store the data generated by the execution of trigger we will require a table.

```
CREATE TABLE startup_audit
(
  Event_type VARCHAR2(15),
  event_date DATE,
  event_time VARCHAR2(15)
);
```

Step 3: Create the database Event Startup Trigger

In this step we will create a trigger which will execute every time the database in which it is created starts up.

```
CREATE OR REPLACE TRIGGER startup_audit
AFTER STARTUP ON DATABASE
BEGIN
```

```
    INSERT INTO startup_audit VALUES
    (
        ora_sysevent,
        SYSDATE,
        TO_CHAR(sysdate, 'hh24:mm:ss')
    );
END;
/
```

On successful execution this trigger will insert a row of data each time database starts up.

Shutdown Triggers

SHUTDOWN triggers execute before database shutdown processing is performed. Similar to the startup trigger, only a user with DBA role or ADMINISTER DATABASE TRIGGER system privilege can create a shutdown trigger.

Example.

First 2 steps of creating a database event shutdown triggers are same as that of the startup trigger which we saw above.

```
CREATE OR REPLACE TRIGGER tr_shutdown_audit
BEFORE SHUTDOWN ON DATABASE
BEGIN
    INSERT INTO startup_audit VALUES
    (
        ora_sysevent,
        SYSDATE,
        TO_CHAR(sysdate, 'hh24:mm:ss')
    );
END;
/
```

Table used in this trigger is the same one which we created during the coding of the Startup trigger above.

SHUTDOWN triggers execute only when the database is shut down using NORMAL or IMMEDIATE mode. They do not execute when the database is shut down using ABORT mode or when the database crashes.

You can also use shutdown database event triggers for gathering your database system statistics. Here is an example

```
CREATE OR REPLACE TRIGGER before_shutdown
BEFORE SHUTDOWN ON DATABASE
BEGIN
    gather_system_stats;
END;
/
```

Instead-oF Insert Trigger

So far we have learnt DML, DDL and System Event Triggers in Oracle Database. Today in this tutorial we will explore the concepts of Instead-of trigger in Oracle Database. This blog has been written keeping in mind the certification exam as well as the job interview. Hope you will enjoy reading it. If so then click here to tweet and share.

Instead Of Trigger

Instead-of triggers in oracle database provide a way of modifying views that cannot be modified directly through the DML statements. By using Instead-of triggers, you can perform Insert, Update, Delete and Merge operations on a view in oracle database.

Restriction on Instead-of View.

Instead-of triggers can control Insert, Delete, Update and Merge operations of the View, not the table. Yes you heard it right, you can write an instead-of trigger on Views only and not on tables in Oracle database. That is the restriction that you have to comply with. Along with this you even have to comply with every general restriction that is imposed on all types of triggers. We have discussed those in the Introduction of triggers section.

Uses of Instead-of trigger.

Since an Instead-of trigger can only be used with views therefore we can use them to make a non-updatable view updatable as well as to override the default behavior of views that are updatable.

Syntax of instead-of trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF operation
ON view_name
FOR EACH ROW
BEGIN
    ---Your SQL Code—
END;
/
```

For line-by-line detailed explanation of the above syntax please refer my video tutorial. There I have explained all the clause and keyword of the syntax in detail.

So you asked…

When does an Instead-of trigger fires – Before or After the triggering event?

If you noticed carefully then you'll see that we do not have either BEFORE or AFTER clause in the syntax. This is because unlike other triggers, instead-of trigger executes neither BEFORE nor AFTER but instead of a triggering event. That is why we do not need to specify either BEFORE or AFTER clause.

Examples

Instead-of Insert Trigger

Instead-of trigger can be best demonstrated using a View joining two or more tables. Thus in this example I will create two simple tables and will then create a view over them. After that I will create an Instead of trigger for Insert operation on this view.

Step1: Create Tables

Table 1- trainer

```
CREATE TABLE trainer
  (
    full_name VARCHAR2(20)
  );
```

Table 2- Subject

```
CREATE TABLE subject
  (
    subject_name VARCHAR2(15)
  );
```

Insert dummy data into the above tables

```
INSERT INTO trainer VALUES ('Manish Sharma');

INSERT INTO subject VALUES ('Oracle');
```

Step 2: Create View

In this step I will create a view which will show you the combined result of the data from the two tables above.

```
CREATE VIEW vw_rebellionrider AS
SELECT full_name, subject_name FROM trainer, subject;
```

This is a non-updatable view which you can confirm by executing any DML statement over it. Error as a result of DML operation on this view will be your confirmation.

Step 3: Create Instead-of Insert Trigger

Next I will create an Instead-of Insert trigger over the view vw_rebellionrider that we created in step 2.

```
CREATE OR REPLACE TRIGGER tr_Io_Insert
INSTEAD OF INSERT ON vw_rebellionrider
FOR EACH ROW
BEGIN
  INSERT INTO trainer (full_name) VALUES (:new.full_name);
  INSERT INTO subject (subject_name) VALUES (:new.subject_name);
END;
/
```

On successful execution, this trigger will insert a new row of data into both the underlying tables of the view vw_rebellionrider. You can confirm that by executing an insert DML over the view.

Instead-oF UPDATE Trigger

Similar to Instead-of Insert trigger, which we discussed in the previous tutorial, Instead-of Update trigger overrides default behavior of Update DML operation executed on the view. Execution of Update DML on a complex view is restricted because of the involvement of multiple tables over which your view is created. To override this restriction we can take the help of Instead-Of Update trigger.

Instead-Of Update Trigger

Instead-of update trigger will override the default behavior of your update operation when you execute the update statement and will let you update the data of the underlying tables over which your view is created.

Instead-Of Update trigger Examples:

Tables (Trainer and Subject) and View (VW_RebellionRider) used in this example are the same as the ones we created in the previous tutorial.

```
CREATE OR REPLACE TRIGGER io_update
INSTEAD OF UPDATE ON vw_rebellionrider
FOR EACH ROW
BEGIN
   UPDATE trainer SET FULL_NAME = :new.full_name
   WHERE FULL_NAME = :old.full_name;

   UPDATE subject SET subject_NAME = :new.subject_name
   WHERE subject_NAME = :old.subject_name;
END;
/
```

On successful execution this trigger will let you execute an Update DML on the view.

Instead-oF DELETE Trigger

Welcome to the last tutorial on the “INSTEAD-OF” triggers in Oracle database. Till so far in this series we have seen how to create INSTEAD-OF INSERT trigger and INSTEAD-OF UPDATE trigger. The only trigger that is left now is the INSTEAD-OF DELETE trigger which we will cover in today’s tutorial.

Similar to other instead-of triggers which we have seen in previous tutorial, using INSTEAD-OF DELETE we can override the standard action of Delete DML on a view.

Instead-of Delete trigger Example.

In this example I will again use the View VW_RebellionRider which we created earlier and have consistently used in this Instead-of trigger series so far.

Needless to say that executing DELETE DML on this view will return an error because of its non-updatable nature. Thus the only way to perform DELETE DML on this view is by using an Instead of trigger. Let's quickly create one.

```
CREATE OR REPLACE TRIGGER io_delete
INSTEAD OF DELETE ON vw_RebellionRider
FOR EACH ROW
BEGIN
   DELETE FROM trainer WHERE FULL_NAME = :old.FULL_NAME;
   DELETE FROM subject WHERE SUBJECT_NAME= :old.SUBJECT_NAME;
END;
/
```

On successful execution this trigger will allow you to execute DELETE DML on the view.

PL/SQL Cursors in Oracle Database

What is a database cursor? I guess, is the question which majority of us have faced at least once in our life either during our college studies, or job interview or while doing oracle certification. Database cursor is an important topic from oracle certification as well as from job interview perspective. Thus I am writing this blog by taking both the perspectives in mind so that you will get good marks in your exams as well as ace your Job interview.

What is Database Cursor?

Cursor is a pointer to a memory area called **context area**. This context area is a memory region inside the Process Global Area or PGA assigned to hold the information about the processing of a SELECT statement or DML Statement such as INSERT, DELETE, UPDATE or MERGE.

A quick tip:

Refrain from saying that cursor is a pointer to the data stored in the database. Saying this in your interview will definitely put you in an indefinite queue of candidates who never receive a call back. Because cursor is the pointer to the memory area called context area not to the data of the database.

What is the Context Area?

Let's dig a little deeper and see what the context area is?

The context area is a special memory region inside the Process Global Area or PGA which helps oracle server in processing an SQL statement by holding the important information about that statement.

This information includes:

- Rows returned by a query.
- Number of rows processed by a query.
- A pointer to the parsed query in the shared pool.

Using cursor you can control the context area as it is a pointer to the same.

A quick tip:

Parsing an SQL statement is the term used for the process that includes the transferring of information to the server, whereby the SQL statement is evaluated as being valid.

Advantages of Cursors

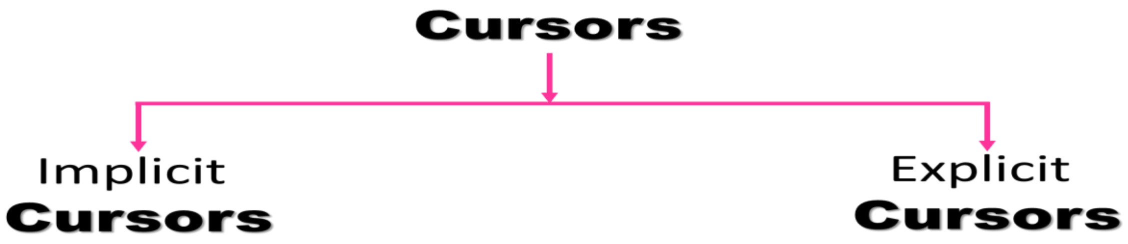
There are two main advantages of a cursor:

59. Cursor is names thus you can reference it in your program whenever you want.
60. Cursor allows you to fetch and process rows returned by a SELECT statement by a row at a time.

Types of cursors in oracle database:

There are two types of cursors in oracle database:

61. Implicit cursor
62. Explicit cursor



Implicit Cursors in Oracle Database

As the name suggests these are the cursors which are automatically created by the oracle server every time an SQL DML statement is executed. User cannot control the behavior of these cursors. Oracle server creates an implicit cursor in the background for any PL/SQL block which executes an SQL statement as long as an explicit cursor does not exist for that SQL statement.

Oracle server associates a cursor with every DML statement. Each of the Update & Delete statements has cursors which are responsible to identify those set of rows that are affected by the operation. Also the implicit cursor fulfills the need of a place for an Insert statement to receive the data that is to be inserted into the database.

Info Byte:

The Most recently opened cursor is called SQL Cursor.

Explicit Cursor in oracle database

In contrast to implicit cursors, we have explicit cursors. Explicit cursors are user defined cursors which means user has to create these cursors for any statement which returns more than one row of data. Unlike implicit cursor user has full control of explicit cursor. An explicit cursor can be generated only by naming the cursor in the declaration section of the PL/SQL block.

Advantages of Explicit Cursor:

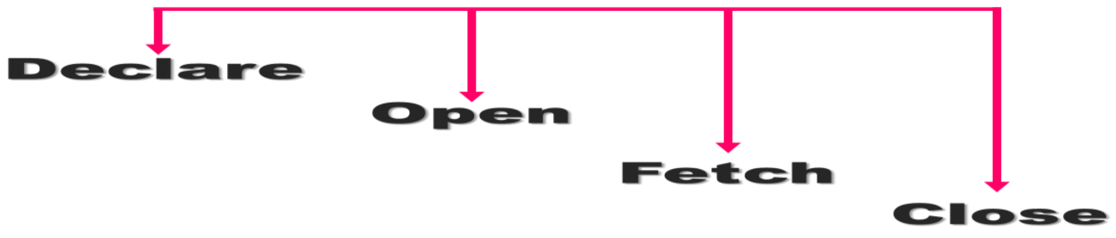
63. Since Explicit cursors are user defined hence they give you more programmatic control on your program.
64. Explicit cursors are more efficient as compared to implicit cursors as in latters case it is hard to track data errors.

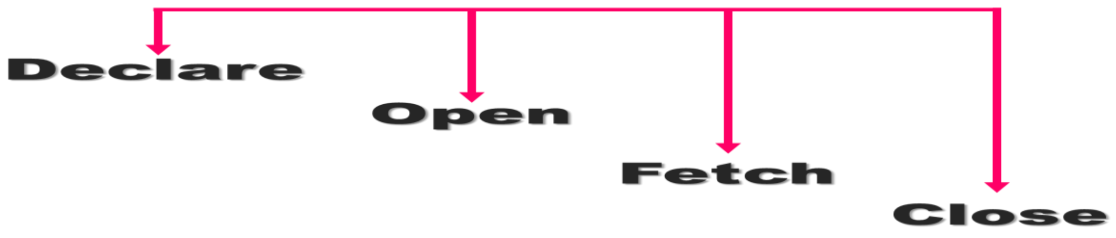
Steps for creating an Explicit Cursor

To create an explicit cursor you need to follow 4 steps. These 4 steps are:

- Declare
- Open
- Fetch
- Close

In case of implicit cursors oracle server performs all these steps automatically for you.





Info Byte:

Unless the complete cycle of declaring, opening, fetching and closing has been performed, you can't use a cursor.

Declare: How To Declare a Database Cursor?

Declaring a cursor means initializing a cursor into memory. You define explicit cursor in declaration section of your PL/SQL block and associate it with the SELECT statement.

Syntax

```
CURSOR cursor_name IS select_statement;
```

Open: How to Open a Database Cursor?

Whenever oracle server comes across an 'Open Cursor' Statement the following four steps take place automatically.

65. All the variables including bind variables in the WHERE clause of a SELECT statement are examined.
66. Based on the values of the variables, the active set is determined, and the PL/SQL engine executes the query for that cursor. Variables are examined at cursor open time.
67. The PL/SQL engine identifies the active set of data.
68. The active set pointer sets to the first row.

Active set: Rows from all the involved tables that meet the WHERE clause criteria.

Syntax

```
OPEN cursor_name;
```

Fetch: How to retrieve data from cursor?

The process of retrieving the data from the cursor is called fetching. Once the cursor is declared and opened then you can retrieve the data from it. Let's see how.

Syntax

```
FETCH cursor_name INTO PL/SQL variable;
Or
FETCH cursor_name INTO PL/SQL record;
```

What happens when you execute fetch command of a cursor?

- The use of a FETCH command is to retrieve one row at a time from the active set. This is usually done inside a loop. The values of each row in the active set can then be stored in the corresponding variables or PL/SQL record one at a time, performing operations on each one successively.
- After completion of each FETCH, the active set pointer is moved forward to the next row. Therefore, each FETCH returns successive rows of the active set, until the entire set is returned. The last FETCH does not assign values to the output variables as they still contain their previous values.

Close: How To Close a Database Cursor?

Once you are done working with your cursor it's advisable to close it. As soon as the server comes across the closing statement of a cursor it will relinquish all the resources associated with it.

Syntax

```
CLOSE cursor_name;
```

Info Byte:

You can no longer fetch from a cursor once it's closed. Similarly it is impossible to close a cursor once it is already closed. Either of these actions will result in an Oracle error.

Basic Programming Structure of the Cursor

Here is the basic programming structure of the cursor in oracle database.

```
DECLARE
   CURSOR cursor_name IS select_statement;
BEGIN
   OPEN cursor_name;
   FETCH cursor_name INTO PL/SQL variable [PL/SQL record];
```

```
    CLOSE cursor_name;
  END;
  /
```

Cursor Attributes

Oracle provides six attributes which work in correlation with cursors. These attributes are:

69. %FOUND
70. %NOTFOUND
71. %ISOPEN
72. %ROWCOUNT
73. %BULK_ROWCOUNT
74. %BULK_EXCEPTIONS

First three attributes 'Found', 'NotFound' and 'IsOpen' are Boolean attributes whereas the last one 'RowCount' is a numeric attribute.

Let's quickly take a look at all these attributes.

%Found

Cursor attribute 'Found' is a Boolean attribute which returns TRUE if the previous FETCH command returned a row otherwise it returns FALSE.

%NotFound

'Not Found' cursor attribute is also a Boolean attribute which returns TRUE only when previous FETCH command of the cursor did not return a row otherwise this attribute will return FALSE.

%IsOpen

Again 'Is Open' Cursor attribute is a Boolean attribute which you can use to check whether your cursor is open or not. It returns TRUE if the cursor is open otherwise it returns FALSE.

%RowCount

Row count cursor attribute is a numeric attribute which means it returns a numeric value as a result and that value will be the number of records fetched from a cursor at that point in time.

%BULK_ROWCOUNT

'Bulk RowCount' cursor attribute Returns the number of records modified by the FORALL statement for each collection element

%BULK_EXCEPTIONS

Similar to 'Bulk RowCount' the 'Bulk EXCEPTIONS' cursor attribute Returns exception information for rows modified by the FORALL statement for each collection element

Create Explicit Cursor In Oracle Database

Cursor is a pointer to a memory area called context area. This we have already learnt with all the other details in the previous tutorial. Today in this blog we will learn how to create an explicit database cursor.

As we have already learnt that whenever we execute a DML statement, the oracle server creates an implicit cursor in the background. As these cursors are created by oracle server itself thus user does not have much programmatic control on them. In case if you want to control your own DMLs then you need to write an explicit cursor.

So let's quickly see how you can create your own database cursor in oracle database.

```
SET SERVEROUTPUT ON;
DECLARE
  v_name VARCHAR2(30);
  --Declare Cursor
  CURSOR  cur_RebellionRider  IS
  SELECT  first_name  FROM  EMPLOYEES
  WHERE  employee_id  <  105;
BEGIN
  OPEN cur_RebellionRider;
  LOOP
    FETCH cur_RebellionRider INTO v_name;
    DBMS_OUTPUT.PUT_LINE (v_name);
    EXIT WHEN cur_RebellionRider%NOTFOUND;
  END LOOP;--Simple Loop End
  CLOSE cur_RebellionRider;
END;
/
```

We used EMPLOYEE table of HR sample Schema for creating the above explicit cursor. You can watch my Video Tutorial on The Same Topic for line by line explanation of the above code.

Declaration of your cursor can only be done in the "Declaration" section of the PL/SQL block and the rest of the steps of explicit cursor creation cycle must be done in the execution section of a PL/SQL block.

Parameterized Cursor In Oracle Database

In previous PL/SQL tutorial we saw how to create a simple explicit cursor. No doubt that explicit cursor has certain advantages over implicit cursor and can increase the efficiency of DML statements by giving more programmatic controls in user's hands. Now let's take a step ahead and learn how to create a parameterized explicit cursor a.k.a. cursor parameter.

Knowledge of cursor creation cycle and simple cursor creation is a must to understand the concept of cursor parameter thus I will highly suggest you to read the previous tutorial on Introduction to cursor and Simple Cursor Creation.

What is Parameterized cursor?

Unlike simple explicit cursor, parameterized cursors accept values as parameter. You specify the list of parameters separated by comma (,) while declaring the cursor and supply the corresponding argument for each parameter in the list while opening the cursor.

Definition:

Cursor parameter can be appropriately defined as an explicit cursor that accepts arguments from the user in the form of parameter.

Syntax of Parameterized Cursor in Oracle Database

```
CURSOR cur _ name (parameter list) IS SELECT statement;
```

Syntax of declaring a cursor parameter is pretty similar to that of the simple cursor except the addition of parameters enclosed in the parenthesis.

```
OPEN cur _ name (argument list)
```

You have to provide corresponding arguments for each parameter that are specified during the declaration. Rest of the steps are the same as that of the simple cursor.

There are few things which you have to take care of while specifying the parameters in your explicit cursor.

- In case of multiple parameters, always separate parameters and the corresponding arguments in the list from each other using comma (,).
- You can specify as many parameters as you need just make sure to include an argument in parameter list for each parameter when you open the cursor.
- While specifying a parameter during the declaration of the explicit cursor only specify the data type not the data width.

Some Wonderful Advantages of Parameterized Cursors

Makes the cursor more reusable

You can use a parameter and then pass different values to the WHERE clause each time a cursor is opened instead of hardcoding a value into the WHERE clause of a query to select particular information.

Avoids scoping problems

When you pass parameters instead of hardcoding the values, the result set for that cursor is not tied to a specific variable in a program or block. Therefore in case your program has nested blocks, you can define the cursor at a higher-level (enclosing) block and use it in any of the sub-blocks with variables defined in those local blocks.

When do we need a parameterized cursor?

You must be wondering when we need a cursor with parameters in our PL/SQL.

The simplest answer is whenever you need to use your cursor in more than one place with different values for the same WHERE clause of your SELECT statement.

If you can add something to this and have another idea for using a parameterized cursor. Then I am always open to listening to your thoughts do make sure to share it with me on my Facebook or twitter.

Example of Parameterized cursor.

```
SET SERVEROUTPUT ON;
DECLARE
```

```
    v_name VARCHAR2 (30);
    --Declare Cursor
    CURSOR p_cur_RebellionRider (var_e_id VARCHAR2) IS
    SELECT first_name FROM EMPLOYEES
    WHERE employee_id < var_e_id;
BEGIN
    OPEN p_cur_RebellionRider (105);
LOOP
    FETCH p_cur_RebellionRider INTO v_name;
    EXIT WHEN p_cur_RebellionRider%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_name );
END LOOP;
CLOSE p_cur_RebellionRider;
END;
/
```

You can watch the video tutorial on my YouTube channel for line by line explanation of the above code.

Parameterized Cursor With Default Value In Oracle Database

Previously we learnt that in PL/SQL we are allowed to create a cursor with parameter known as Cursor Parameter (a.k.a. Parameterized Cursor). Every cursor parameter has some restrictions which we have to comply with for example we have to specify the argument for each parameter when we open the cursor otherwise we get a PLS 00306 error.

To overcome this restriction, we have the option called default value. Default value is the value which you assign to the parameter of your cursor during the declaration of the cursor.

Example 1 of Cursor Parameter with Default Value

```
SET SERVEROUTPUT ON;
DECLARE
    v_name VARCHAR2(30);
    v_eid NUMBER(10);
    CURSOR cur_RebellionRider(var_e_id NUMBER := 190 )
    IS
    SELECT first_name, employee_id FROM employees
    WHERE employee_id > var_e_id;
BEGIN
    OPEN cur_rebellionrider;
    LOOP
        FETCH cur_rebellionrider INTO v_name, v_eid;
        EXIT WHEN cur_rebellionrider%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_name ||' '||v_eid);
    END LOOP;
    CLOSE cur_rebellionrider;
END;
```

As you can see in the above code I assigned a value (which is 190) using assignment operator to the parameter var_e_id in the cursor declaration and nothing while opening the cursor. Upon the execution this PL/SQL block will show you the first name and employee id of all the employees who have employee id greater than 190. For line by line explanation of the above code please watch the Video Tutorial on my YouTube.

Default Value Is Not a Substitute Value

Default value is not a substitute value for the parameter of your cursor. It comes into action only when the user does not specify the argument for the parameter while opening the cursor. In case user has supplied the argument for the parameter in OPEN CURSOR statement then the compiler will show the result according to that parameter and not according to the **Default value**.

Example 2 of Cursor Parameter with Default Value

```
DECLARE
   v_name VARCHAR2(30);
   v_eid NUMBER(10);
   CURSOR cur_RebellionRider(var_e_id NUMBER := 190 )
   IS
   SELECT first_name, employee_id FROM employees
   WHERE employee_id > var_e_id;
BEGIN
   OPEN cur_rebellionrider (200);
   LOOP
      FETCH cur_rebellionrider INTO v_name, v_eid;
      EXIT WHEN cur_rebellionrider%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_name ||' '||v_eid);
   END LOOP;
   CLOSE cur_rebellionrider;
END;
```

If you compare the result of both the codes demonstrated in the above examples you will see the difference because the result returned by Example 1 is according to the default value (which is 190) which you specified during the declaration while the result returned by Example 2 will be according to the argument (which is 200) you have specified in the OPEN cursor statement.

Use of Default Value in Cursor Parameter

Specifying the default value for the parameter of your cursor can increase the efficiency of your app or code by minimizing the chances of PLS 00306 error.4

Cursor For Loop In Oracle Database

As the name suggests Cursor For Loop is a type of For loop provided by oracle PL/SQL which makes working with cursors in oracle database a lot easier by executing OPEN, FETCH & CLOSE Cursor statements implicitly in the background for you.

Definition:

Cursor for Loop Is an Extension of the Numeric For Loop provided by Oracle PL/SQL which works on specified cursors and performs OPEN, FETCH & CLOSE cursor statements implicitly in the background.

Suggested Reading: Numeric For Loop In Oracle PL/SQL

Syntax of Cursor For Loop.

```
FOR loop_index IN cursor_name
LOOP
   Statements…
END LOOP;
```

Example 1: Cursor For Loop With Simple Explicit Cursor

```
SET SERVEROUTPUT ON;
DECLARE
   CURSOR cur_RebellionRider IS
   SELECT first_name, last_name FROM employees
   WHERE employee_id >200;
BEGIN
   FOR L_IDX IN cur_RebellionRider
   LOOP
      DBMS_OUTPUT.PUT_LINE(L_IDX.first_name||' '||L_IDX.last_name);
   END LOOP;
END;
/
```

Please watch the Video Tutorial on YouTube channel for detailed explanation of the above code.

Example 2: Cursor For Loop With Inline Cursor.

You can pass the cursor definition directly into the Cursor For Loop. The code for that is:

```
SET SERVEROUTPUT ON;
BEGIN
   FOR L_IDX IN (SELECT first_name, last_name FROM employees
   WHERE employee_id >200)
   LOOP
      DBMS_OUTPUT.PUT_LINE (L_IDX.first_name||' '||L_IDX.last_name);
   END LOOP;
END;
/
```

As you can see in the above code, instead of declaring a cursor into a separate declaration section of PL/SQL blockwe can write the Cursor's SELECT DML statement right inside the loop statement after IN keyword.

Just remember:

- Directly write the SELECT statement without specifying the cursor name into the loop statement.
- Enclose the SELECT statement into parenthesis.
- Do not terminate the SELECT statement with a semicolon (;)

How many times will Cursor For Loop execute?

Unlike Numeric For Loop with Cursor For Loop we don't have minimum or maximum range which will decide the number of execution. So how many times will this loop execute?

This loop will execute for each row returned by the specified cursor and it will terminate either when there is no row to return or there is an occurrence of an exception.

Cursor For Loop In Oracle Database vol.2

The following tutorial requires the knowledge of "Parameterized Cursor" and "Cursor For Loop" beforehand. So for better understanding I highly suggest you to check out the respective blogs to learn the concepts.

The question which was left unattended in the previous tutorial was how to create a cursor for loop with a parameterized cursor in Oracle database? Hence today in this PL/SQL tutorial we will learn how to create cursor for loop with parameterized cursor.

Using cursor for loop, working with parameterized cursor is as easy as working with simple explicit cursor. Checkout the example given below.

Example1. Cursor For Loop With Parameterized Cursor

```
SET SERVEROUTPUT ON;
DECLARE
   CURSOR cur_RebellionRider( var_e_id NUMBER) IS
   SELECT first_name, employee_id FROM employees
   WHERE employee_id > var_e_id;
BEGIN
   FOR l_idx IN cur_RebellionRider(200)
   LOOP
      DBMS_OUTPUT.PUT_LINE(l_idx.employee_id||' '||l_idx.first_name);
   END LOOP;
END;
```

You can pass the values for the parameters of your cursor just by simply writing the argument right after the name of your cursor in loop statement as shown in the above example (Statement In **bold**). Always remember to enclose the arguments inside the parenthesis.

Records In Oracle Database

So now that we are done with database cursors, it is time to take a look at something which is long due. Also since I have already promised to do this tutorial while doing PL/SQL tutorial 30 on Cursor For Loop. So here we are, this tutorial will be an introduction to Records in Oracle Database.

This PL/SQL tutorial is created through the perspective of ***Oracle Database certification*** and ***Job Interview***. Thus in this tutorial you will find the answers of some most commonly asked questions in an easy language. So let's start this tutorial with the most obvious question –

That is a Record in Oracle Database?

Records are composite data structures made up of different components called fields. These fields can have different data types. This means that you can store data of different data types in a single record variable. Similar to the way we define columns in a row of a table.

Definition

A record is a group of related data items stored in fields, each with its own name and datatype. ~Oracle Docs.

Types of Record datatype in Oracle database

In Oracle PL/SQL we have three types of Record datatype.

75. Table Based Record
76. Cursor Based Record, and
77. User Defined Record.

How to declare a Record Datatype in Oracle Database. (%ROWTYPE attribute.)

Similar to ***%TYPE*** which is for declaring an anchored datatype variable, Oracle PL/SQL provides us an attribute ***% ROWTYPE*** for declaring a variable with record datatype.

Syntax of Record Datatype

Table Based Record

```
Variable_ name     table_name%ROWTYPE;
```

Variable Name: A user defined name given to the variable. You have to specify the name of the variable which could be anything. But you have to follow the oracle guidelines for variable name.

Table Name: Table name will be the name of the table over which the record datatype is created. This table will serve as the base table for the record. Also the Oracle Server will then create corresponding fields in the record that will have the same name as that of a column in the table.

%ROWTYPE: Attribute which indicates the compiler that the variable which is declared is of Record Type.

Cursor based Record

```
Variable_ name     cursor_name%ROWTYPE;
```

The syntax of cursor based record is pretty similar to the one which we just saw. You just have to specify the name of the cursor in place of table name, rest of the things remain same.

How to access data stored is Record Type Variable.

Whenever you create a record based on a table, oracle server will create fields corresponding to each column of the table and all those fields have the same name as a column in the table. That makes referencing or accessing data from the record much easier. Here is how you access data from record variable.

```
Record_variable_name.column_name_of_the_table;
```

Using dot (.) notation you can access the data stored into the field of the record.

Table Based Record Type Variables

In the previous tutorial we learnt that the record datatype variables are composite data structures made up of different components called fields that have the same name and data types as that of the columns of a table using which you created that record. There we saw that there are three types of record variables available in Oracle Database. In this tutorial we will concentrate on the first type of record variable which is **"Table Based Record Datatypes"** Variables.

What are Table Based Record Datatype Variables?

As the name suggests Table Based Record Datatype Variables are variable of record datatype created over a table of your database.

Whenever you declare a record type variable oracle engine will declare a composite data structure with fields corresponding to each column of the specified table. For example if we have a record variable created using Employees table then that variable has 11 fields corresponding to 11 columns of the employees table. Also all these fields have the same name, data type and data width as that of the columns of the table.

How to Declare a Table Based Record Datatype Variables in Oracle Database?

The declaration of a table based record datatype variables is very similar to the declaration of simple user variables.
For example

```
SET SERVEROUTPUT ON;
DECLARE
    v_emp employees%ROWTYPE;
```

That is the declaration section of an anonymous PL/SQL block where we declared a record type variable with the name v_emp. V_emp record variable is based on *employees* table of the HR sample user. As this record variable is based on "*employee*" table of HR sample user thus it will have 11 fields corresponding to the 11 columns of the employees table.

How to Initialize Record Datatype Variables in Oracle Database?

In Oracle Database a record data type variable can be initialized by multiple ways similar to the simple user variables which we have seen in PL/SQL tutorial 2. However in this tutorial we will focus on the three most common ways of initializing a Record type variable. These are:

78. By fetching data from all the columns of a row of a table into the record variable using SELECT-INTO statement.
79. By fetching data from selected columns of the table into the record variable using SELECT-INTO statement.
80. By directly assigning the value into the record variable using assignment operator.

We will discuss each of the above ways of initializing a record data type variable one tutorial at a time. In this tutorial we will focus on the first way of initializing database records which is by ***Fetching data from all the columns of a row of a table***.

Inialize Record Variable by Fetching Data from All the Columns (SELECT-INTO Statement)

We already know that whenever you declare a record type variable oracle engine will declare a composite data structure with fields corresponding to each column of the specified table. We can initialize all these fields by fetching the data from all the columns of a row of the table using SELECT-INTO statement.

```
SELECT * INTO v_emp FROM employees
WHERE employee_id = 100;
```

As we are talking about fetching data from all the columns thus we are using the SELECT asterisk (*) statement and storing the data into our record variable V_EMP.

Take a note here that a table based record variable can hold data of a single row at a time, thus you have to make sure that the SELECT-INTO statement returns the data from one row only and for that you can filter the data using WHERE clause, as we did in the above statement.

Till so far we have learnt how to Declare and Initialize a database record type variable, next we will see how to access data from the record data type variable in Oracle Database.

How to access data from the record datatype variables in Oracle Database?

In order to access the data stored into a record variable we use the **dot (.) notation**. In which we first write the name of our record variable followed by a dot (.) and then the name of the field (or the name of the column) as both fields of a Table based Record Type variable and the column of the table over which the record variable is declared share the same name whose data we want to retrieve.

```
DBMS_OUTPUT.PUT_LINE (v_emp.first_name ||' '||v_emp.salary);
```

Now that we have learnt how to Declare and Initialize table based record type variables in Oracle Database. Now let's put together all these pieces into a single PL/SQL block

Example: Table based record type variable initialization with SELECT asterisk (*) statement.

```
SET SERVEROUTPUT ON;
DECLARE
   v_emp employees%ROWTYPE;
BEGIN
   SELECT * INTO v_emp FROM employees
   WHERE employee_id = 100;
   DBMS_OUTPUT.PUT_LINE (v_emp.first_name ||' '||v_emp.salary);
   DBMS_OUTPUT.PUT_LINE(v_emp.hire_date);
END;
/
```

That is a quick tutorial on How to Declare, Initialize and access data of a record data type variable in Oracle Database. Stay tuned as in the next tutorial we will learn another way of initializing a record variable.

Table Based Record Type Variables

In the previous tutorial we learnt that the record datatype variables are composite data structures made up of different components called fields that have the same name and data types as that of the columns of a table using which you created that record. There we saw that there are three types of record variables available in Oracle Database. In this tutorial we will concentrate on the first type of record variable which is **"Table Based Record Datatypes"** Variables.

What are Table Based Record Datatype Variables?

As the name suggests Table Based Record Datatype Variables are variable of record datatype created over a table of your database.

Whenever you declare a record type variable oracle engine will declare a composite data structure with fields corresponding to each column of the specified table. For example if we have a record variable created using Employees table then that variable has 11 fields corresponding to 11 columns of the employees table. Also all these fields have the same name, data type and data width as that of the columns of the table.

How to Declare a Table Based Record Datatype Variables in Oracle Database?

The declaration of a table based record datatype variables is very similar to the declaration of simple user variables.

For example

```
SET SERVEROUTPUT ON;
DECLARE
   v_emp employees%ROWTYPE;
```

That is the declaration section of an anonymous PL/SQL block where we declared a record type variable with the name v_emp. V_emp record variable is based on *employees* table of the HR sample user. As this record variable is based on “*employee*” table of HR sample user thus it will have 11 fields corresponding to the 11 columns of the employees table.

How to Initialize Record Datatype Variables in Oracle Database?

In Oracle Database a record data type variable can be initialized by multiple ways similar to the simple user variables which we have seen in PL/SQL tutorial 2. However in this tutorial we will focus on the three most common ways of initializing a Record type variable. These are:

81. By fetching data from all the columns of a row of a table into the record variable using SELECT-INTO statement.
82. By fetching data from selected columns of the table into the record variable using SELECT-INTO statement.
83. By directly assigning the value into the record variable using assignment operator.

We will discuss each of the above ways of initializing a record data type variable one tutorial at a time. In this tutorial we will focus on the first way of initializing database records which is by ***Fetching data from all the columns of a row of a table***.

Reference Book for SQL Expert 1z0-047 *Affiliate link*

OCA Oracle Database SQL Certified Expert Exam Guide (Exam 1Z0-047)

Initialize Record Variable by Fetching Data from All the Columns (SELECT-INTO Statement)

We already know that whenever you declare a record type variable oracle engine will declare a composite data structure with fields corresponding to each column of the specified table. We can initialize all these fields by fetching the data from all the columns of a row of the table using SELECT-INTO statement.

```
SELECT * INTO v_emp FROM employees
WHERE employee_id = 100;
```

As we are talking about fetching data from all the columns thus we are using the SELECT asterisk (*) statement and storing the data into our record variable V_EMP.

Take a note here that a table based record variable can hold data of a single row at a time, thus you have to make sure that the SELECT-INTO statement returns the data from one row only and for that you can filter the data using WHERE clause, as we did in the above statement.

Till so far we have learnt how to Declare and Initialize a database record type variable, next we will see how to access data from the record data type variable in Oracle Database.

How to access data from the record datatype variables in Oracle Database?

In order to access the data stored into a record variable we use the **dot (.) notation**. In which we first write the name of our record variable followed by a dot (.) and then the name of the field (or the name of the column) as both fields of a Table based Record Type variable and the column of the table over which the record variable is declared share the same name whose data we want to retrieve.

```
DBMS_OUTPUT.PUT_LINE (v_emp.first_name ||' '||v_emp.salary);
```

Now that we have learnt how to Declare and Initialize table based record type variables in Oracle Database. Now let's put together all these pieces into a single PL/SQL block

Example: Table based record type variable initialization with SELECT asterisk (*) statement.

```
SET SERVEROUTPUT ON;
DECLARE
  v_emp employees%ROWTYPE;
BEGIN
  SELECT * INTO v_emp FROM employees
  WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE (v_emp.first_name ||' '||v_emp.salary);
  DBMS_OUTPUT.PUT_LINE(v_emp.hire_date);
END;
/
```

That is a quick tutorial on How to Declare, Initialize and access data of a record data type variable in Oracle Database. Stay tuned as in the next tutorial we will learn another way of initializing a record variable.

How to Initialize Table Based Record Type Variables

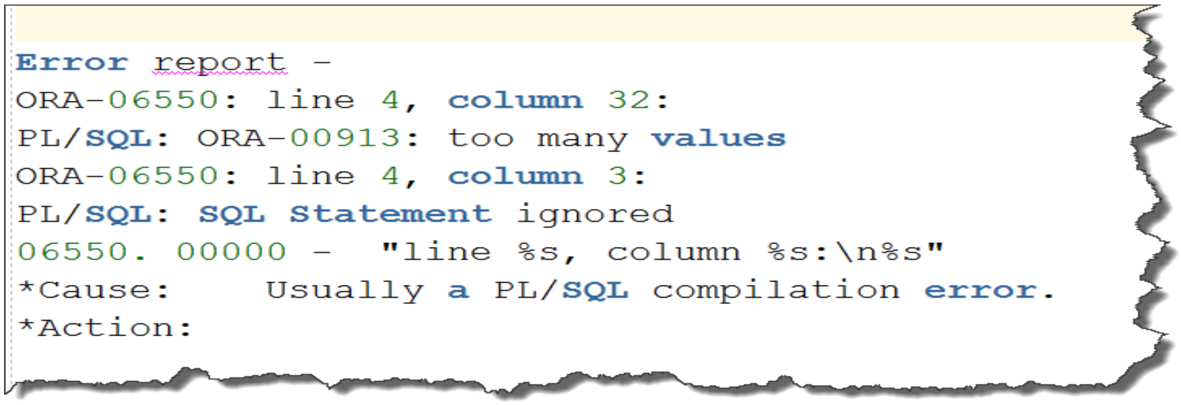
In the previous tutorial we saw, how to initialize a record variable by fetching data from all the columns of a table using SELECT-INTO statement. But what if we want to fetch the data from selected columns of a table? Can we still use the same process for initializing the record variable?

The answer is No, we cannot use the same process shown in the previous tutorial for initializing a record datatype variable by fetching data from select columns of a table. The execution of the following example where we are using the data from one column "*First Name*" of the "*Employees*" table for initializing the record datatype variable "*v_emp*" will return an error.

```
SET SERVEROUTPUT ON;
DECLARE
  v_emp employees%ROWTYPE;
BEGIN
  SELECT first_name INTO v_emp FROM employees
  WHERE employee_id = 100;
```

```
    DBMS_OUTPUT.PUT_LINE(v_emp.first_name);
END;
/
```

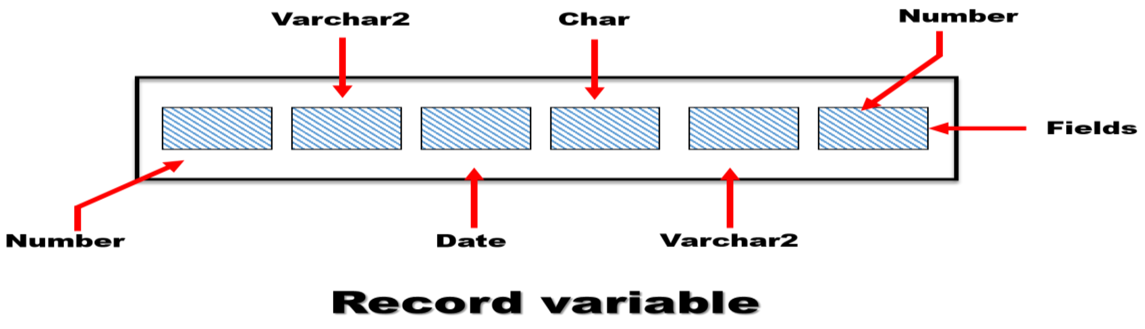
That error is "**PL/SQL: ORA-00913: too many values**" So what does this error means? Why is it saying "too many values" even when we are fetching only one data. I know sometimes PL/SQL error can be deceptive and confusing.



```
Error report -
ORA-06550: line 4, column 32:
PL/SQL: ORA-00913: too many values
ORA-06550: line 4, column 3:
PL/SQL: SQL Statement ignored
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```

In order to find out the solution of this problem and understand this error we have to recall what we learnt in the previous tutorial. There we learnt that Database Records are composite data structures made up of multiple fields which share same name, datatype and data width as that of the columns of the table over which it is created. And the data which was fetched gets stored into these fields.

In case of initializing a record variable by fetching data from all the columns of a table, we don't have to do anything, oracle engine does all the dirty work in the background for you. It first fetches the data from the table and then stores that data into the corresponding field into the record which has the same data type and data width, and for that matter also shares the same name.



But when it comes to initializing a record datatype variable by fetching the data from selected columns we have to do all this work manually. We have to specify the name of the column from which we want to fetch the data and then we have to specify the name of the field where we want to store that data.

In the above example we did specify the column name of the table whose data we want to fetch but we didn't specify the name of the record's field where we want to store the fetched data. That confuses the compiler on where to store the data which ultimately causes the error.

How to Solve "PL/SQL ORA-00913: Too Many Values" Error?

The *PL/SQL ORA-00913 error* can easily be solved by specifying the record's field name where you want the fetched data to be stored. But here you have to be a bit careful. You have to make sure that the field's datatype and data width must match with the datatype and data width of the column whose data you have fetched.
Now the question here is that how to specify the record data type field's name? We can specify the name of the field of the record variable using **Dot (.) notation** where we first write the *name of the already declared record variable*followed by *a dot* (.) and the n*ame of the field.* And luckily both the columns of the table and fields of the record share the same name.

So now that we have learnt why we are having PL/SQL ORA-00913 error on initializing the record data type variable using data from select columns. Let's modify the above example accordingly and try to resolve it.

Example 1: Initialize the Record Datatype variable using data from One Column.

```
SET SERVEROUTPUT ON;
DECLARE
   v_emp employees%ROWTYPE;
BEGIN
   SELECT first_name INTO v_emp.first_name FROM employees
   WHERE employee_id = 100;
   DBMS_OUTPUT.PUT_LINE (v_emp.first_name);
END;
/
```

Example 2: Initialize the Record Datatype variable using data from Multiple Columns.

```
SET SERVEROUTPUT ON;
DECLARE
   v_emp employees%ROWTYPE;
BEGIN
   SELECT first_name,
      hire_date
   INTO v_emp.first_name,
      v_emp.hire_date
   FROM employees
   WHERE employee_id = 100;
   DBMS_OUTPUT.PUT_LINE (v_emp.first_name);
   DBMS_OUTPUT.PUT_LINE (v_emp.hire_date);
END;
/
```

That is how we can initialize a record datatype variable using data from selected columns of a table.

How to Create Cursor Based Record Type Variables

After table based record variable, the next tutorial in Record datatype variables series is Cursor Based Record thus here we will learn what Cursor Based Record Datatype Variables are and how to create them in Oracle Database.

What are Cursor Based Records in Oracle Database?

Cursor based records are those variables whose structure is derived from the SELECT list of an already created cursor. As we know that record datatypes are composite data structures made up of multiple fields and in the case of Cursor based record these fields are initialized by fetching data from the SELECT-LIST of the cursor that was used to create this variable

How To Create Cursor Based Record In Oracle Database?

The creation of cursor based record variable involves:

- Declaration of Cursor based Record.
- Initialization of Cursor Based Record and
- Accessing the data stored into the cursor based record variable.

Let's discuss each of these above steps one by one.

Declare Cursor Based Record In Oracle Database

As it is obvious that in order to declare a Cursor Based Record we need to have a Cursor over which we can design our Record variable. For the demonstration let's have the same cursor which we have discussed in PL/SQL tutorial 27 on "How to Create Simple Explicit Cursor."

```
SET SERVEROUTPUT ON;
DECLARE
CURSOR    cur_RebellionRider    IS
SELECT first_name, salary FROM employees
WHERE employee_id = 100;
```

Once you have created your cursor then you are all set to declare your Cursor based Record variable.

```
var_emp    cur_RebellionRider%ROWTYPE;
```

Declaration of Cursor Based Record is pretty much similar to the declaration of table based record. The declaration starts with the name of the record variable, which is completely user defined followed by the name of an already created cursor. And at the end "%ROWTYPE" attribute which indicates that we are declaring a Record Datatype variable to the compiler.

For more detail on how to declare a cursor based record in Oracle Database please watch my Video Tutorial on the same topic.

Initialize Cursor Based Record Datatype

Once you have successfully declared your record datatype variable next you have to initialize it and by that I mean putting some value into the record variable. Though the initializing process of cursor based record is very simple and pretty much similar to the one we have seen in the previous tutorial on "Table Based Record" but unlike table records here we are initializing a record variable which is based on a cursor thus we have to make sure that we should follow the cursor cycle properly.

```
BEGIN
   OPEN cur_RebellionRider;
   FETCH cur_RebellionRider INTO var_emp;
```

Every cursor needs to be opened before being used. Thus here we first opened the cursor and then using FETCH-INTO statement we fetched the data into the record variable "var_emp".

Access data from the cursor based record

After Declaration and Initialization of cursor based record the thing which we have to learn is How to access data from the cursor based record?

```
DBMS_OUTPUT.PUT_LINE (var_emp.first_name);
DBMS_OUTPUT.PUT_LINE (var_emp.salary);
```

You can see in the above two statements that once again we used the Dot (.) notation for fetching the data from the record variable. Though these two statements look very similar to one which we have seen in table based record variable tutorial, but they are not. In case of table based record we have to write the name of the column of the table but in case of cursor based record we can only write the name of those columns of the table which we specified in the SELECT-List of the cursor.

Once you have declared, initialize and access the data then make sure to close the cursor.

Now let's compile all the above line of codes (LOCs) together and make them into a single PL/SQL block.

```
SET SERVEROUTPUT ON;
DECLARE
   CURSOR cur_RebellionRider
   IS
   SELECT first_name, salary FROM employees
   WHERE employee_id = 100;
   --Cursor Based Record Variable Declare
   var_emp cur_RebellionRider%ROWTYPE;
BEGIN
   OPEN cur_RebellionRider;
   FETCH cur_RebellionRider INTO var_emp;
   DBMS_OUTPUT.PUT_LINE (var_emp.first_name);
   DBMS_OUTPUT.PUT_LINE (var_emp.salary);
   CLOSE cur_RebellionRider;
END;
```

So that is how we create a cursor based record. Hope you enjoyed reading. Do make sure to share this blog on your social media with your friends. Also since I am giving away RebellionRider's merchandise to randomly selected winners so make sure to tag me.

Cursor Based Record Type With The Cursor Returning Multiple Values

We have already seen in the previous tutorial how to create Cursor based Record Type Variable based on Simple Cursor which returns a single row of data. Now the question arises here is that can we use the same single record datatype variable with the cursor which returns multiple rows of data? To know the answer, read along & learn how to handle multiple values returned by a cursor using single cursor based record datatype variable.

As we are dealing with Cursor-Based records thus a little bit knowledge of cursor is required. For the sake of simplicity and to make this concept easy to understand I will use a Simple Explicit-Cursor for the demonstration.

Step 1: Declare a Simple Explicit Cursor

A Cursor-based record datatype requires an already created cursor. This cursor will become an underlying base for our record type variable. All the fields of the record type variable which is created using this cursor will have the same name and datatype as that of the columns used in the SELECT-List of the cursor.

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR cur_RebellionRider IS
  SELECT first_name, salary FROM employees
  WHERE employee_id > 200;
```

Unlike the cursor from the previous tutorial which is returning a single row of data, this cursor will return multiple rows. All these rows will consist of the first name and salary of all the employees with employee id greater than 200.

Step2: Declare the Cursor Based Record Datatype Variable

As we have created the cursor, now we are all set to declare our record variable using this cursor.

```
var_emp   cur_RebellionRider%ROWTYPE;
```

Var_emp is the record type variable and as it is based on the cursor cur_RebellionRider thus we can proudly call it a Cursor-Based record type variable. Now we have to see whether this single record variable is capable of holding all the data returned by the underlying cursor cur_RebellionRider.

Step 3: Initialize the Cursor-Record Variable

As we discussed in the PL/SQL tutorial 34 that initialization of a record variable is the process of assigning some values to it. In case of Cursors, FETCH-INTO statement does this work. But we have to make sure that we have followed the Cursor Life Cycle properly.

If you don't know what is this Cursor Life Cycle and the steps involved in creating a Cursor then checkout this blog on "Introduction to Database Cursor".

```
BEGIN OPEN cur_RebellionRider; LOOP FETCH cur_RebellionRider INTO var_emp; EXIT WHEN cur_RebellionRider%NOTFOUND; DBMS_OUTPUT.PUT_LINE (var_emp.first_name||' '||var_emp.salary ); END LOOP;--Simple Loop End CLOSE cur_RebellionRider; END;
```

The above execution section of PL/SQL block which we are creating here has been explained line-by-line in the video tutorial on my YouTube channel. Do check that out.

So that is it. That's all we have to do. Now let's combine all these chunks of code which we saw in different steps above into a single anonymous PL/SQL block.

Cursor-Based Record Datatype Variable in Oracle Database

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR cur_RebellionRider IS
  SELECT first_name, salary FROM employees
  WHERE employee_id > 200;
  var_emp cur_RebellionRider%ROWTYPE;
BEGIN
  OPEN cur_RebellionRider;
  LOOP
    FETCH cur_RebellionRider INTO var_emp;
    EXIT WHEN cur_RebellionRider%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (var_emp.first_name||' '||var_emp.salary);
  END LOOP;
  CLOSE cur_RebellionRider;
END;
```

When you compile and run the above code you will get all the data which FETCH-INTO statement fetched from the cursor cur_RebellionRider and stored into the Cursor-based Record variable var_emp. This implies that we can indeed handle multiple rows of data using a single Cursor Based Record.

In my PL/SQL video tutorial I asked that can we simplify this code or is there any other way of doing the same task. The answer is **yes**, there are multiple ways of achieving the same result and one of them is by using "Cursor For-Loop". This is a special kind of loop which declares the record variable as well as Opens, Fetches and Closes the underlying cursor implicitly in the background for you. You can read more Cursor For-Loop here.

Here is the code done using Cursor For-Loop which is equivalent to the above code. As you can see it is much less complex with few Line-of-Codes (LOCs).

```
SET SERVEROUTPUT ON;
BEGIN
  FOR var_emp IN (SELECT first_name, salary FROM employees
  WHERE employee_id >200)
  LOOP
```

```
    DBMS_OUTPUT.PUT_LINE(var_emp.first_name||'                    '||var_emp.salary);
  END                                                                          LOOP;
END;
```

How To Create User defined Record Datatype Variable

So far we have seen how to create [Table based]() and [cursor based]() record datatype variables. The one that is left is the user define record datatype which we are going to cover in today's tutorial.

As the name suggests, user define records are the record variables whose structure is defined by the user, which is unlike the table based or cursor based records whose structures are derived from their respective tables or cursor. This means that with user define records you can have complete control over the structure of your record variable.

The creation process of user define record variable is divided into two parts. Before defining the record we first need to define the TYPE for the record variable. This TYPE will become the base of the User Define Record variable and will help in driving its structure. Once the TYPE is successfully declared then we can use it for creating our user define record variable.

Syntax of User Define Records in Oracle Database

Below is the syntax for creating the TYPE for User Defined Record Datatype Variable.

```
TYPE            type_name            IS            RECORD            (
  field_name1                        datatype                        1,
  field_name2                        datatype                        2,
  ...
  field_nameN                        datatype                        N
);
```

Once we have our TYPE declared we are all set to create our Record Variable. This variable will then acquire all the properties of the type using which it is created. And here is the syntax for creating the user define record datatype variable.

```
record_name TYPE_NAME;
```

Did you notice that unlike the Table based or Cursor Based Record Variable we do not have to use %ROWTYPE attribute here for declaring the record variable?

You can watch the [Video Tutorial]() on my [YouTube channel]() for a detailed explanation of the above syntax.

Example: How To Create User Defined Record Datatype Variable.

Step 1: Declare Type for the User Defined Record Variable

```
SET                    SERVEROUTPUT                    ON;
DECLARE
```

```
    TYPE rv_dept IS RECORD (
      f_name VARCHAR2(20),
      d_name DEPARTMENTS.department_name%TYPE
  );
```

Step 2: Declare User Define Record Variable

After creating the TYPE you are all set to create your User Defined Record Variable.

```
  var1 rv_dept;
```

This above PL/SQL statement will create a record variable with the name VAR1.

Step 3: Initialize the User Defined Record Variable.

User defined record variable can be initialized in multiple ways. For instance you can initialize the record variable directly by assigning value to it using assignment operator or you can fetch the values stored into the column of a table using SELECT-INTO statement. So let's move ahead with our example and learn how to initialize a user defined record variable using SELECT-INTO statement.

Next I will write the execution section. In the execution section we will have a SELECT statement which will be joining employees table and departments table and returning the first name and department name of the specific employee.

```
  BEGIN
    SELECT first_name , department_name
    INTO var1.f_name, var1.d_name
    FROM employees join departments
    Using (department_id) WHERE employee_id = 100;

    DBMS_OUTPUT.PUT_LINE(var1.f_name||' '||var1.d_name);
  END;
```

The select statement which we have here will return the first name and the department name of the employee whose employee id is 100. The data for both the columns are coming from different tables thus we used a JOINhere. As there are two different tables involved in the query thus in such a situation use of Table Based Record Variable is not possible therefore the viable solution is the user define record variable.

Let's join all the above chunks of codes together into a single anonymous PL/SQL block.

```
  SET SERVEROUTPUT ON;
  DECLARE
    TYPE rv_dept IS RECORD(
      f_name VARCHAR2(20),
      d_name departments.department_name%type
    );
    var1 rv_dept;
  BEGIN
                SELECT first_name , department_name
                   INTO var1.f_name, var1.d_name
                FROM employees join departments
    Using (department_id) WHERE employee_id = 100;
```

```
                DBMS_OUTPUT.PUT_LINE(var1.f_name||'        '||var1.d_name);
END;
```

PL/SQL Functions In Oracle Database

To anyone, who has ever studied programming languages like C, C++ or Java, the concept of functions isn't new. Functions are nothing but a group of executable statements. Using Functions you can save yourself from re-writing the same programming logic again and again. So how can we define a function in Oracle PL/SQL?

What are PL/SQL functions in Oracle Database?

In Oracle Database we can define a PL/SQL function as a self-contained sub-program that is meant to do some specific well defined task. Functions are named PL/SQL block which means they can be stored into the database as a database object and can be reused. That is also the reason why some books refer to PL/SQL functions as stored functions.

Type of PL/SQL functions in Oracle Database

There are two types of PL/SQL functions in Oracle Database, these are

84. Pass-by-Value Functions and
85. Pass-by-Reference functions

In Oracle Database both types of functions should have to return some values and these values should be a valid SQL or PL/SQL datatype.

Syntax of PL/SQL Functions in Oracle Database

```
CREATE [OR REPLACE] FUNCTION function_name
(Parameter 1, Parameter 2…)
RETURN datatype
IS
   Declare variable, constant etc. here.
BEGIN
   Executable Statements
   Return (Return Value);
END;
```

I have discussed the PL/SQL function's syntax line by line in the video tutorial on my YouTube channel on the same topic. I suggest you to go and check out that tutorial once.

Function Execution Method

Depending on your creativity and programming skills, a PL/SQL function can be called by multiple ways. Here are some general ways of calling a PL/SQL function in Oracle Database

86. You can use SQL*Plus utility of the Oracle Database to invoke a PL/SQL function that can be called from PL/SQL as procedural statement.
87. An anonymous PL/SQL block can also be used to call a function.
88. You can even call a function directly into a SELECT or DML statement.

Stay tuned we will discuss each of these execution methods of PL/SQL functions in the next tutorial.

Restrictions on calling a function

89. A function that returns SQL datatype can be used inside SQL statement and a PL/SQL function that returns PL/SQL datatype only works inside PL/SQL blocks. An exception to this rule is that, you cannot call a function that contains a DML operation inside a SQL query. However you can call a function that performs a DML operation inside INSERT, UPDATE and DELETE.
90. A function called from an UPDATE or DELETE statement on a table cannot query (SELECT) or perform transaction (DMLs) on the same table.
91. A function called from SQL expressions cannot contain the TCL (COMMIT or ROLLBACK) command or the DDL (CREATE or ALTER) command

That's it for this tutorial on Introduction to PL/SQL Functions in Oracle Database. Do make sure to check out the next tutorial where I demonstrate the creation of PL/SQL function using a very simple example.

PL/SQL Functions in Oracle Database

So now that we have already learnt in the previous PL/SQL tutorial about what are PL/SQL Functions in Oracle Database let's take a leap ahead and learn how to create a PL/SQL function using a very simple example.

This tutorial will require a proper knowledge of the syntax of PL/SQL functions thus I will suggest you to check out the previous tutorial first. That being said let's start this tutorial.

In order to keep this tutorial simple and easy to understand, we will create a very easy function which will calculate the area of a circle. I guess that will serve the purpose and help you in learning how to create PL/SQL functions in Oracle Database.

As discussed in the previous tutorial that the function body is divided into two parts

- First is the header of the PL/SQL function and
- Second is the execution part of the PL/SQL function

So let's start with header of our function.

Step 1. Create the Header of a PL/SQL Function.

The header consists of the signature of the function or the declaration of the PL/SQL function.

```
--Function Header
CREATE OR REPLACE FUNCTION circle_area (radius NUMBER)
RETURN NUMBER IS
```

Step 2. Declare Variables or the Constant.

If your program requires you to declare any variable or constant or anything then you can do it right after creating the header, that too without using the DECLARE keyword.

```
--Declare a constant and a variable
pi CONSTANT NUMBER(7,2) := 3.141;
area NUMBER(7,2);
```

Step 3. Create the Execution Part of the PL/SQL function.

Once you have created the header of your function and declared all your necessary variables as well as constants then you are all set to create the execution part of your PL/SQL function. Here in the execution section of a PL/SQL function, you write all your execution statements. This part also defines the working of your function.

```
BEGIN
  --Area of Circle pi*r*r;
  area := pi * (radius * radius);
  RETURN area;
END;
```

Quick Info:
To calculate the square of the circle's radius in the area of circle, you can also use the inbuilt function of POWER (p, q). This function takes two numeric input and returns one numeric value which will be the answer to the arithmetic expression of p raise to q.

Now let's join all the above chunks of codes together into a single named unit.

PL/SQL function for calculating "Area of the Circle".

```
--Function Header
CREATE OR REPLACE FUNCTION circle_area (radius NUMBER)
RETURN NUMBER IS
--Declare a constant and a variable
pi CONSTANT NUMBER(7,2) := 3.141;
area NUMBER(7,2);
BEGIN
  --Area of Circle pi*r*r;
  area := pi * (radius * radius);
  RETURN area;
END;
```

A successful compilation will create a named PL/SQL block which is your PL/SQL function with the name circle_area.

As PL/SQL functions are named PL/SQL block thus they are permanently saved in your database which you can use anytime.

In order to see your PL/SQL Function in action you have to call it through your program. Your program can be an anonymous PL/SQL block, or a named PL/SQL Block or even using a SELECT statement. Few of these various ways of calling a function have been demonstrated in my video tutorial on the same topic on my YouTube channel. I highly encourage you to watch that video.

What are stored procedures in Oracle Database?

Similar to PL/SQL Functions a stored Procedure is a self-contained subprogram that is meant to do some specific tasks. Also similar to functions, procedures are named PL/SQL blocks thus they can be reused because they are stored into the database as a database object. But unlike PL/SQL functions a stored procedure does not return any value.

Syntax of PL/SQL Stored Procedures

```
CREATE [OR REPLACE] PROCEDURE pro_name (Parameter – List)
IS [AUTHID DEFINER | CURRENT_USER]
  Declare statements
BEGIN
  Executable statements
END procedure name;
```

The above Syntax of PL/SQL stored procedure is pretty much similar to the syntax of PL/SQL Functions that we saw in the last PL/SQL tutorial. Except for two things:

There is no Return clause.

A core difference between a PL/SQL Function and a stored procedure is that unlike Functions a stored procedure does not return any value.

AUTHID Clause.

The AUTHID clause is used for setting the authority model for the PL/SQL Procedures. This clause has two flags.

a. DEFINER and
b. CURRENT_USER

92.
As this clause is optional thus in case if you do not use AUTHID clause then Oracle Engine will set the authority (AUTHID) to the DEFINER by default for you. Now, you must be wondering what these DEFINER and CURRENT_USER rights are?

DEFINER right:

Definer right is the default right assigned to the procedure by oracle engine. This right means anyone with Execution Privilege on the procedure acts as if they are the owner of the schema in which the privilege is created.

CURRENT_USER right:

Setting the authority level of a stored procedure to the current_user right overrides the default right which is definer and change it to the invoker rights.
Invoker right authority means that you call the procedure to act on your local data and it requires that you replicate data objects in any participating schema.

Some Extra Points About Stored Procedure

- You can define a procedure with or without formal parameters.
- A parameter can be either pass-by-value or pass-by-reference.
- A procedure will be a pass-by-value procedure when you don't specify the parameter mode because it uses the default IN mode.

That's it on Introduction to PL/SQL Stored Procedures. I guess you will also enjoy reading these below mentioned blogs. All the blogs are written by taking Job Interview and Oracle Database Certification in Mind, do make sure to check them out.

- Differences between PL/SQL Function and PL/SQL Stored Procedures?
- What are Parameter Modes in PL/SQL Functions and Procedures?
- What are Formal and Actual Parameters?

PL/SQL Stored Procedures without Parameters

While discussing the syntax in the Introduction to PL/SQL stored procedures we learnt that a stored procedure can have zero, one or many parameters. Today in this tutorial we will learn how to create a PL/SQL stored procedure with zero parameters or say without any parameters. Apart from creating a stored procedure in oracle database, we will also learn in this tutorial the multiple ways of calling a stored procedure in a PL/SQL program.

How To Create PL/SQL Stored Procedure without Parameters In Oracle Database

In the following example we will create a very simple procedure. I will try to keep the example as easy as possible so that all the PL/SQL learning enthusiasts out there can understand the process of creating a stored procedure easily.

```
CREATE OR REPLACE PROCEDURE pr_RebellionRider IS
  var_name VARCHAR2 (30):= 'Manish';
  var_web VARCHAR2 (30) := 'RebellionRider.com';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Whats Up Internet? I am '||var_name||' from '||var_web);
END Pr_RebellionRider;
```

In the above example I have created a PL/SQL Stored procedure with the name pr_RebellionRider which has two variables capable of holding strings of VARCHAR2 datatype. In the execution section this PL/SQL procedure has only one DBMS OUTPUT statement which is displaying the strings stored into those variable back to the user in a formatted manner.

For the detailed explanation of the above code please watch **video tutorial on my YouTube channelon the same topic.**

How to Call PL/SQL Stored Procedures in Oracle Database

After successfully creating and compiling the stored procedure, next you have to call this subroutine. You can do so in multiple ways such as:

- Call a PL/SQL stored procedure using EXECUTE statement.
- Call a PL/SQL stored procedure using an Anonymous PL/SQL block.
- Call a PL/SQL stored procedure using a Named PL/SQL block.

If in case your subroutine such as stored procedure consists of server side PL/SQL statement then do make sure to set the "Server Output On" to see the result.

Call a PL/SQL stored procedure using EXECUTE statement

The best way to quickly check the output of your stored procedure or test the working of your PL/SQL procedure is to call it using EXECUTE keyword. In order to call a stored procedure using EXECUTE keyword you simply have to write the same keyword followed by the name of the procedure.

```
EXECUTE PR_RebellionRider;
```

Or you can also write the first 4 letters of the EXECUTE keyword followed by the procedure name.

```
EXEC PR_RebellionRider;
```

Both the statements are the same and will do the same work.

Call a PL/SQL stored procedure using an Anonymous PL/SQL block

The second way of calling a procedure is to place a procedure call statement inside the execution section of an anonymous PL/SQL block.

```
BEGIN
   PR_RebellionRider;
END;
```

You simply have to write the name of your stored procedure inside the execution section of an anonymous and named PL/SQL block. The compiler will automatically interpret that as a procedure call statement. If your procedure accepts any parameters then you can supply values for parameters here. We will talk in detail about stored procedures with parameters in our next tutorial.

PL/SQL Stored Procedures *with Parameters*

In the previous tutorial we have discussed how to create stored procedure without parameters. But sometimes it may happen that you will need to create a stored procedure which accepts parameters. After all, these subroutines are there to help you in getting the solution of your problem in the easiest way possible. Thus today in this blog we will learn how to create ***stored procedures with parameters*** in Oracle Database

For those who are new to PL/SQL Programming and wondering what is stored procedure? Don't worry I have done a separate blog for you explaining the fundamental theory of the stored procedure. You can check that blog here.

So let's see the demonstration of how to create PL/SQL stored procedure with parameters!

Step 1: Create the header of the stored procedure

In the header of the procedure we define its signature.

```
CREATE OR REPLACE PROCEDURE emp_sal
(dep_id NUMBER, sal_raise NUMBER)
IS
```

The header is pretty similar to the one which we saw in the last tutorial except that this time our procedure is accepting parameters which are dep_id and sal_raise of NUMBER datatype.

Step 2: Create the execution section of the stored procedure

In the execution section we write all the executable statements which define the working of the stored procedure.

```
BEGIN
  UPDATE employees SET salary = salary * sal_raise WHERE department_id = dep_id;
END;
```

For a better understanding I have tried to make this code as simple as possible. In the execution section we only have one DML statement which is UPDATE. Using this we update the salary column of employee table.

You can write the business logic like this then wrap them up into a procedure and call them in your app when needed. This will give you more control on your app. It will also save you from writing the same code again and again.

This procedure will accept two parameters which is the department id and the numeric value for salary raise. First parameter which is the dep_id, is used to determine the ID of the department. The second parameter which is sal _ raise will become the multiplication factor in the salary raise.

For the more in-depth understanding of the same, please watch the video tutorial on my YouTube channel. There I have explained every single line and keyword of the above stored procedure in detail.

Let's combine all the above chunks of code into a single one named PL/SQL unit.

Stored Procedure for Department Wide Salary Raise

```
CREATE OR REPLACE PROCEDURE emp_sal( dep_id NUMBER, sal_raise NUMBER)
IS
BEGIN
  UPDATE emp SET salary = salary * sal_raise WHERE department_id = dep_id;
END;
```

Calling Notation for PL/SQL Subroutines

Since previous few tutorials were about PL/SQL Subroutines such as PL/SQL Functions and Stored Proceduresthus it becomes mandatory to talk about their calling notations. Learning the concepts of PL/SQL Subroutines will not be considered as complete until we learn their calling notations as well.

What is Calling Notation for PL/SQL Subroutines?

Calling notation is a way of providing values to the parameters of a subroutine such as PL/SQL function or a stored procedure.

Types of Calling Notations for Subroutines

In Oracle PL/SQL there are 3 types of calling notations. These are:

93. Positional Notation
94. Named Notation and
95. Mixed calling notation

Positional Calling Notations

Positional notation is the most common calling notation which you can see in almost every computer programming language. In positional notation we have to specify the value for each formal parameter in a sequential manner. This means that you have to provide the values for the formal parameters in the same order as they were declared in the procedure or in the function.

In positional notation the datatype and the position of the actual parameter must match with the formal parameter.

Actual Parameters vs. Formal Parameters

Example: Positional Notation for calling PL/SQL Subroutines.

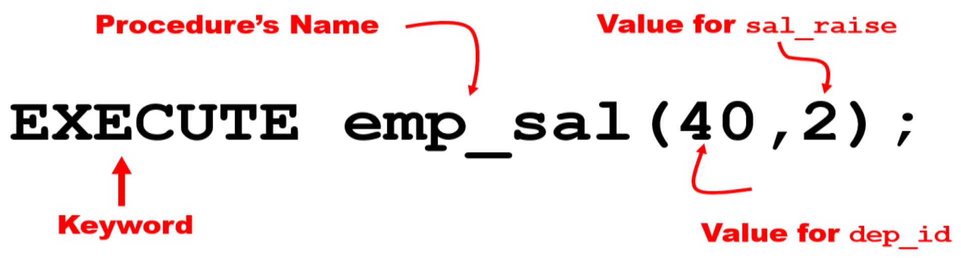
```
CREATE OR REPLACE PROCEDURE emp_sal (dep_id NUMBER, sal_raise NUMBER)
IS
BEGIN
  UPDATE employees
  SET salary = salary * sal_raise
  WHERE department_id = dep_id;
  DBMS_OUTPUT.PUT_LINE ('salary updated successfully');
END;
/
```

This is the same example which we did in PL/SQL Tutorial 42 on how to create stored procedure with parameters albeit some minor changes. Now if we use positional calling notation then we have to supply the values to both the parameters of the above procedure in the same manner in which they are declared.

Stored Procedure call using positional notation in Oracle Database

```
EXECUTE emp_sal (40,2);
```

In this simple procedure call, the value 40 is corresponding to the formal parameter dep_id and value 2 is corresponding to the parameter sal_raise.



Named Calling Notations

Named calling notation lets you pass values to the formal parameters using their names. This will in turn let you assign values to only required or say mandatory parameters.

This calling notation is useful when you have a subroutine with parameters where some of those parameters are mandatory and some are optional and you want to pass the values to only the mandatory ones.

Association Operator

In order to assign values to the formal parameters using their names we use association operator. It is a combination of equal to (=) sign and greater than (>) sign. We write the name of the formal parameter to the left hand side of the operator and the value which you want to assign to the right hand side of the operator.

Example of Named Calling Notation for calling a PL/SQL Subroutines

```
CREATE OR REPLACE FUNCTION add_num
(var_1 NUMBER, var_2 NUMBER DEFAULT 0, var_3 NUMBER ) RETURN NUMBER
IS
BEGIN
  RETURN var_1 + var_2 + var_3;
END;
/
```

The above function has 3 parameters. Among these 3 parameters 2 are mandatory and 1 is optional with a default value 0.

You can call this function using positional notation. But it has a restriction which you have to fulfill and that is that you have to supply values to all the formal parameters in the same order in which they are declared and the datatype of formal and actual parameters must match.

So if you want to omit the optional parameter and want to use their default value or you just forgot the order of the parameter in which they were declared! Then it will be slightly difficult for you to call the above subroutine using positional notation. In such a scenario you can take advantage of Named Calling Notation. This calling notation will provide you the desired flexibility in calling your subroutines.

PL/SQL Function call using Named Calling Notation in Oracle Database

```
DECLARE
  var_result NUMBER;
BEGIN
  var_result := add_num(var_3 => 5, var_1 =>2);
  DBMS_OUTPUT.put_line('Result ->' || var_result);
END;
/
```

I have explained the PL/SQL Function call in detail in the Video Tutorial on this same topic on my YouTube channel.

Mixed Calling Notation for calling PL/SQL Subroutines

As the name suggests in mixed calling notation you can call subroutines using the combination of named as well as positional calling notations. Mixed calling notation is very helpful where the parameter list is defined with all mandatory parameters first and optional parameters next.

Example of Mixed calling notation for calling PL/SQL subroutines

Here is the anonymous block in which we are calling the same function add_num ( ) which we coded when doing named calling notation.

```
DECLARE
  var_result                                                           NUMBER;
BEGIN
  var_result    :=    add_num(var_1    =>    10,    30    ,var_3    =>19);
  DBMS_OUTPUT.put_line('Result          ->'          ||          var_result);
END;
/
```

That's how we use mixed calling notation for calling PL/SQL Subroutines.

Try it yourself

Using the knowledge from the above concepts try solving the following question:

Write a PL/SQL Function with parameters for swapping two numbers and call that function using mixed calling notation.

PL/SQL Packages

Till so far we have seen most of the named PL/SQL blocks such as Database Cursors, PL/SQL Functions, Stored Procedure and Triggers now it's time to move on to another most demanded PL/SQL tutorial on my YouTube channel which is PL/SQL Packages.

What are PL/SQL Packages in Oracle Database?

Packages are stored libraries in the database which allow us to group related PL/SQL objects under one name. Or in simple words, Packages are logical groups of related PL/SQL objects. Packages are named PL/SQL Blocks which mean they are permanently stored into the database schema and can be referenced or reused by your program.

Definition of PL/SQL Packages

Packages are stored libraries in the database which allow us to group related PL/SQL objects under one name.

What are the contents included in a package?

A package can hold multiple database objects such as

- Stored Procedures
- PL/SQL Functions
- Database Cursors
- Type declarations as well as
- Variables

Package Architecture

PL/SQL package is divided into two parts:

96. The Package Specification, also known as the Header and
97. The Package Body

Both these parts are stored separately in the data dictionary. The package specification is the required part whereas the package body is optional, but it is a good practice to provide the body to the package.

Package Specification

Package specification is also known as the package header. It is the section where we put the declaration of all the package elements. Whatever elements we declare here in this section are publically available and can be referenced outside of the package.

In *package specification* we only declare package elements but we don't define them. Also this is the mandatory section of the package.

Syntax of Package specification

```
CREATE OR REPALCE PACKAGE pkg_name IS
  Declaration of all the package element…;
END [pkg_name];
```

Package Body

In package body we provide the actual structure to all the package elements which we have already declared in the specification by programing them. Or we can say that a package body contains the implementation of the elements listed in the package specification. Unlike package specification a package body can contain both declaration of the variable as well as the definition of all the package elements. Any package elements such as PL/SQL Function, a cursor or a stored procedure which is not in the package specification but coded in the package body is called Private Package Elements and thus they cannot be referenced outside the package.

Syntax of the package body

```
CREATE OR REPALCE PACKAGE BODY pkg_name IS
  Variable declaration;
  Type Declaration;
BEGIN
  Implementation of the package elements…
END [pkg_name];
```

PL/SQL Packages

In the previous tutorial we covered all the necessary theories about PL/SQL packages in Oracle Database. There we learnt what are packages, their architecture and the syntax of creating the same. Now that we have seen all the theories required to answer the questions in your certification exam and interview, it's time to take a step ahead and do an example demonstrating the process of creating a PL/SQL package in Oracle Database.

How to Create Package?

For the demonstration I am going to create a very simple package which will consist of two elements – a function and a stored procedure. I have tried to keep this example as simple as possible in order to keep the concept easy to understand.

As we talked while discussing the architecture of the package in the previous tutorial that the package is divided into two parts, which are

- Package header and
- The package body

So we will start with creating the package header first

Package Header

```
CREATE OR REPLACE PACKAGE pkg_RebellionRider IS
  FUNCTION prnt_strng RETURN VARCHAR2;
  PROCEDURE proc_superhero(f_name VARCHAR2, l_name VARCHAR2);
END pkg_RebellionRider;
```

By taking a look at the above code we can clearly say that the package is going to hold two package elements which are – a PL/SQL function prnt_strng and a stored procedure proc_superhero.

Package Body

```
--Package Body
CREATE OR REPLACE PACKAGE BODY pkg_RebellionRider IS
                  --Function Implimentation
  FUNCTION prnt_strng RETURN VARCHAR2 IS
  BEGIN
    RETURN 'RebellionRider.com';
  END prnt_strng;

  --Procedure Implimentation
  PROCEDURE proc_superhero(f_name VARCHAR2, l_name VARCHAR2) IS
  BEGIN
    INSERT INTO new_superheroes (f_name, l_name) VALUES(f_name, l_name);
  END;

END pkg_rrdr;
```

In the above code for the package body we implemented both the package elements which we defined into the package header.

How to access the package elements?

We have our package header and body created and compiled successfully, what's next? Like every other database object, package serves a unique purpose. Using a package you can group different database objects under one name and as PL/SQL packages are named database blocks thus they can be stored into the database and can be used later when needed.

So the question arises here is how to access the package elements? To access the elements defined and implemented into a package we use ***dot (.) notation***. According to which in order to access the package element you have to first write the name of your package followed by dot (.) operator and then the name of the package element.

Example:

For example say you want to call the PL/SQL function prnt_strng of our package pkg_RebellionRider.

```
--Package Calling Function
BEGIN
  DBMS_OUTPUT.PUT_LINE (PKG_RebellionRider.PRNT_STRNG);
END;
```

In the above code using anonymous block we call the function prnt_strng of the package pkg_RebellionRider.

PL/SQL Exception Handling

We cannot say that the code is robust until it can handle all the exceptions. Bugs and abrupt termination of a program are the nightmares of a programmer's life. No programmer wants to develop a code which will crash in a midway or behave unexpectedly. Thus for the smooth execution of a software it is necessary to handle all kinds of exceptions.

Knowing your problem is the first step towards finding its solution. So let's learn more about exception handling in Oracle database.

What is an Exception?

Any abnormal condition or say event that interrupts the normal flow of your program's instructions at run time is an exception. Or in simple words you can say an exception is a run time error.

Info Byte

Exceptions are designed for run time error handling rather than compile time error handling. Errors that occur during compilation phase are detected by the PL/SQL compiler and reported back to the user.

Types of exceptions

There are two types of PL/SQL exceptions in Oracle database.

98. System-defined exceptions and
99. User-defined exceptions

System-Defined Exceptions

System-defined exceptions are defined and maintained implicitly by the Oracle server. These exceptions are mainly defined in the***Oracle STANDARD package***. Whenever an exception occurs inside the program. The Oracle server matches and identifies the appropriate exception from the available set of exceptions.

System defined exceptions majorly have a negative error code and error message. These errors have a short name which is used with the exception handler.

User-Define Exceptions

Unlike System-Define Exception, User-Define Exceptions are raised explicitly in the body of the PL/SQL block (more specifically inside the BEGIN-END section) using the RAISE statement.

How to Declare a User-Define Exception in Oracle Database.

There are three ways of declaring user-define exceptions in Oracle Database.

By declaring a variable of EXCEPTION type in declaration section. You can declare a user defined exception by declaring a variable of EXCEPTION datatype in your code and raise it explicitly in your program using RAISE statement and handle them in the Exception Section.

Declare user-defined exception using PRAGMA EXCEPTION_INIT function.

Using PRAGMA EXCEPTION_INIT function you can map a non-predefined error number with the variable of EXCEPTION datatype. Means using the same function you can associate a variable of EXCEPTION datatype with a standard error.

RAISE_APPLICATION_ERROR method.

Using this method you can declare a user defined exception with your own customized error number and message.

That's it for this section. Hope you enjoyed reading this brief introduction to exception handling in Oracle Database. In this blog I tried to answer a few questions which you can expect in your Oracle Database Certification as well as ***in the Interview***. Stay tuned as in the next tutorial we will do some cool examples explaining the above mentioned concepts which will help you in enhancing your knowledge and give you a strong hold on the concepts of PL/SQL exception handling.

PL/SQL Exception Handling

In the Introduction to PL/SQL Exceptions we learnt that there are three ways of declaring user-define exceptions in Oracle Database. In this tutorial we are going to explore the first way and learn how to declare user-define exception using a variable of Exception datatype.

Declaring a user-define exception using Exception variable is a three step process. These three steps are –

100. **Declare a variable of exception datatype** – This variable is going to take the entire burden on its shoulders.
101. **Raise the Exception** – This is the part where you tell the compiler about the condition which will trigger the exception.
102. **Handle the exception** – This is the last section where you specify what will happen when the error which you raised will trigger.

In this PL/SQL tutorial I am going to explain to you each of these three steps with the help of a PL/SQL code.

For the demonstration purpose I will write a code which will check whether the divisor is zero or not in the division operation. If it is zero then an error will occur and will be displayed to the user otherwise an actual value which is the result of the division arithmetic will be returned on the output screen.

Step 1 Declare a variable of Exception datatype

By Exception variable I mean a variable with Exception datatype. Like any other PL/SQL variable you can declare an Exception variable in declaration section of the anonymous as well as named PL/SQL block. This exception variable will then work as user-define exception for your code.

```
DECLARE
  var_dividend    NUMBER    :=    24;
  var_divisor     NUMBER    :=    0;
  var_result      NUMBER;
  /*Declare Exception variable*/
  ex_DivZero      EXCEPTION;
```

In this declaration section we have 4 variables. Among these 4 variables first 3 are normal Number datatype variables and the 4th one which is ex_DivZero is the special EXCEPTION datatype variable. This variable will become our User-Define Exception for this program.

Step 2 Raise the Exception

The next step after declaring an Exception variable is to raise the exception. To raise the exception in PL/SQL we use ***Raise statement***.

Raise statement is a special kind of PL/SQL statement which changes the normal flow of execution of the code. As soon as compiler comes across a raise condition it transfers the control over to the exception handler.

```
BEGIN
  IF var_divisor = 0 THEN
    RAISE ex_DivZero;
  END IF;
```

Here raise condition is accompanied with the IF-THEN condition . With the help of this we can avoid unwanted switches during the control flow of the program. Using If Condition we are making sure that this error will come into action only when the divisor is equal to 0.

```
var_result := var_dividend/var_divisor;
  DBMS_OUTPUT.PUT_LINE('Result = ' ||var_result);
```

After writing the logic for raising the error you can write your other executable statements of the code just like we did here. After the Raise statement we are performing the arithmetic of division operation and storing the result into the variable var_result, as well as displaying it back as the output using the DBMS OUTPUT statement.

"Raise statement changes the normal flow of execution of the code."

Click Here To Tweet

Step 3 Handle the exception

That is the main section of the code. Here we write the logic for our user-define exception and tell the compiler what it should do if and when that error occurs.

```
EXCEPTION WHEN ex_DivZero THEN
  DBMS_OUTPUT.PUT_LINE('Error Error - Your Divisor is Zero');
END;
```

```
/
```

Here we have the exception handler for the variable ex_DivZero. In the exception handling section we have a DBMS OUTPUT statement which will get displayed when our user define error which is ex_DivZero occurs.

Now let's group all these chunks of codes together.

Divide by zero error using PL/SQL User-define Exception in Oracle Database

```
SET SERVEROUTPUT ON;
DECLARE
        var_dividend NUMBER := 24;
  var_divisor NUMBER := 0;
  var_result NUMBER;
  ex_DivZero EXCEPTION;
BEGIN
  IF var_divisor = 0 THEN
    RAISE ex_DivZero;
  END IF;
  var_result := var_dividend/var_divisor;
  DBMS_OUTPUT.PUT_LINE ( 'Result = ' ||var_result );
  EXCEPTION WHEN ex_DivZero THEN
    DBMS_OUTPUT.PUT_LINE ( 'Error Error - Your Divisor is Zero' );
END;
/
```

Before running this program do make sure that you have set the SERVEROUTPUT on otherwise you will not be able to see the result.

As in Step-1 we set the divisor's value on zero that will in turn raise the user define error ex_DivZero because of this on compiling the above code you will see the string "Error Error - Your Divisor is Zero" the same one which we specified in our exception handler (step 3).

PL/SQL Exception Handling

As discussed in the introduction to PL/SQL exception handling there are three ways of declaring user-define exceptions in Oracle PL/SQL. Among those three we have already discussed and learnt the first way in the previous tutorial. Today in this blog we will take a step ahead and see the second way of declaring user define exception and learn how to declare user-define exception using RAISE_APPLICATION_ERROR method.

What is RAISE_APPLICATION_ERROR method?

RAISE APPLICATION ERROR is a stored procedure which comes in-built with Oracle software. Using this procedure you can associate an error number with the custom error message. Combining both the error number as well as the custom error message you can compose an error string which looks similar to those default error strings which are displayed by Oracle engine when an error occurs.

How many errors can we generate using RAISE_APPLICATION_ERROR procedure?

RAISE_APPLICATION_ERROR procedure allows us to number our errors from -20,000 to -20,999 thus we can say that using RAISE_APPLICATION_ERROR procedure we can generate 1000 errors.

Raise_application_error is part of which package?

You can find RAISE_APPLICATION_ERROR procedure inside *DBMS_STANDARD* package.

Syntax of Raise_Application_Error

```
raise_application_error (error_number, message [, {TRUE | FALSE}]);
```

Here the error_number is a negative integer in the range of -20000.. -20999 and the message is a character string up to 2048 bytes long. In case of the optional third parameter being TRUE, the error is placed on the pile of all the previous errors. However in case of FALSE (the default) parameter, the error replaces all previous errors. RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD, and you do not need to qualify references to package STANDARD.

Example of RAISE_APPLICATION_ERROR procedure

In the following example we will take an input of numeric datatype from the user and check if it is 18 or above. If it is not then the user-define error, which we will declare, will be raised otherwise there will be the normal flow of execution of the program.

Step1: Set the server output on

If we want to see the result returned by the PL/SQL program on the default output screen then we will have to set the server output 'on' which is by default set to 'off' for the session.

```
SET SERVEROUTPUT ON;
```

Step 2: Take User Input

Though we can hardwire the values in our code as we did in the last tutorial but it is not that fun. In order to make the code more dynamic I decided to accept the input by user this time by letting them to enter the value into a pop-up box with a customized message printed on it.

We can take the input using pop-up box with a customized message printed on it using ACCEPT command in Oracle PL/SQL.

```
ACCEPT var_age NUMBER PROMPT 'What is your age?';
```

Command starts with the keyword ***Accept*** followed by the ***name of the variable*** and its ***datatype***. In our case the name is ***var_age*** and datatype is ***NUMBER***. That is the first half of the statement. This part will help us in storing the input value. The other half of the statement will be responsible for printing the customized message on the pop-up box. Using the ***keyword PROMPT*** which is right after the datatype of the variable, you can specify any desired string which you want printed on your pop-up box. In our case this customized message will be '***What is your age?***'

If you want me to do an extensive tutorial explaining the ACCEPT command in detail then share this blog using the hashtag ***#RebellionRider***. You can also write to me on my twitter @RebellionRider.

Step 3: Declare Variable

```
DECLARE
   age NUMBER := &var_age;
```

Due to the scope restriction we cannot use the value stored into the variable which we used in the ***Accept command***. This means we cannot directly use that value into our PL/SQL program. We can solve this problem by assigning the value that was stored into the variable ***var_age*** to a variable which is local to the PL/SQL block. That is exactly what we did in the above segment of the code.

In the above code segment we declared a local variable with name '***age***' and assigned the value stored into the variable '***var_age***' using the assignment operator.

Step 4: Declare the user-define exception by using RAISE_APPLICATION_ERROR procedure

```
BEGIN
  IF age < 18 THEN
    RAISE_APPLICATION_ERROR (-20008, 'you should be 18 or above for the DRINK!');
  END IF;
```

Here in this code segment we declared the user-define exception using RAISE_APPLICATION_ERROR procedure. This procedure is called using two parameters. In which first parameter is the negative number which, in my case, is -20008 and the second number is a string which gets displayed if the same error occurs.

Now the question is when will this error occur?

As you can see that the code is enclosed inside an IF-THEN conditional control block which is making sure that the error will be raised only if the age of the user is less than 18 years.

Step 5: Executable statement

```
DBMS_OUTPUT.PUT_LINE('Sure, What would you like to have?');
```

Executable statements are those that get compiled and run when there is no error and the program has a normal flow of execution. In order to make the code simple and easy to understand I just wrote a single statement which is the DBMS OUTPUT statement.

Step 6: Write the Exception handler

Now that we have declared as well as raised the user-define exception next we need to write the exception handler for it. As said in the previous PL/SQL tutorial that in the Exception Handler section we specify what will happen when the error which you raised will trigger.

```
EXCEPTION WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

In this exception handling section I have called a SQLERRM function using DBMS OUTPUT statement. This is a utility function provided by Oracle which retrieves the error message for the last occurred exception.

Let’s compile all these small chunks of code into one big program.

User-Define Exception Using Raise_Application_Error Procedure

```
SET SERVEROUTPUT ON;
ACCEPT var_age NUMBER PROMPT 'What is yor age';
DECLARE
  age NUMBER := &var_age;
BEGIN
  IF age < 18 THEN
    RAISE_APPLICATION_ERROR (-20008, 'you should be 18 or above for the DRINK!');
  END IF;
```

```
DBMS_OUTPUT.PUT_LINE ('Sure, What would you like to have?');
EXCEPTION WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE (SQLERRM);
END;
/
```

This is a short yet descriptive tutorial on how to declare user-define exception using Raise_Application_Error procedure in Oracle Database. Hope you learnt something new and enjoyed reading. You can help others in learning as well as help me and my channel in growing by sharing this blog on your Social Media. Thanks & have a great day!

PL/SQL Exception Handling

As discussed in the introduction of PL/SQL Exception Handling, there is three ways of declaring user-define exceptions. Among those three ways we have already learnt the first two ways which are declaring user-define exception using

103. Raise Statement and
104. Raise_Application_Error procedure

The only way which is left to be discussed now is declaring user define exceptions using PRAGMA EXCEPTION_INIT function in Oracle Database.

Thus in this PL/SQL tutorial we will learn how to declare PL/SQL user-define exception in Oracle Database by using PRAGMA EXCEPTION_INIT function.

What is PRAGMA EXCEPTION_INIT?

Pragma Exception_Init is a two part statement where first part is made up of ***keyword PRAGMA*** and second part is the ***Exception_Init*** call.

PRAGMA Keyword

A pragma is a compiler directive which indicates that the Statements followed by keyword PRAGMA is a compiler directive statement this means that the statement will be processed at the compile time & not at the runtime.

Exception_Init

Exception_init helps you in associating an exception name with an Oracle error number. In other words we can say that using Exception_Init you can name the exception.

Why name the exception?

Yes, there is a way of declaring user-define exception without the name and that is by using

Raise_Exception_Error procedure. This indeed is a simple and easy way but as we learnt in the last tutorial that to handle exceptions without name we use OTHERS exception handler.

Now think that in your project you have multiple exceptions and that too without name. In order to handle all those exceptions you have a single exception handler with name OTHERS. In this case on the occurrence of an exception condition the compiler will display the error stack produced by the OTHER handler.

Can you imagine how difficult it will be to trace that part of your project which is causing the error. In order to trace that part you need to go through each & every line of your code. This will be mere waste of time.

You can save all those time wasting efforts just by naming the exception, that way you can design an exception handler specific to the name of your exception which will be easily traceable. This is the advantage of naming the exception.

Syntax of Pragma Exception_Init.

```
PRAGMA EXCEPTION_INIT (exception_name, error_number);
```

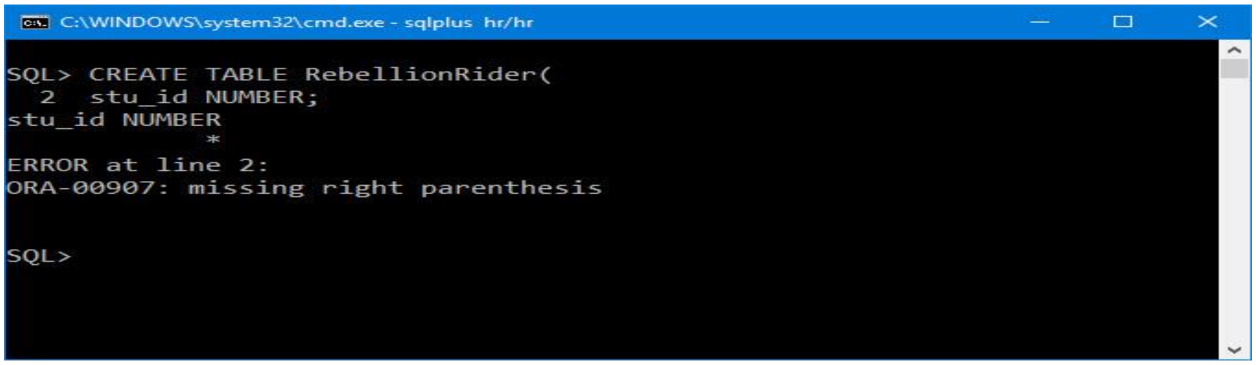
Example: Declare User-define exception using Pragma Exception_Init

```
DECLARE
  ex_age                                                       EXCEPTION;
  age                  NUMBER                     :=                17;
  PRAGMA                        EXCEPTION_INIT(ex_age,           -20008);
BEGIN
  IF                 age                <                 18          THEN
    RAISE_APPLICATION_ERROR(-20008, 'You should be 18 or above for the drinks!');
  END                                                                  IF;

  DBMS_OUTPUT.PUT_LINE('Sure!   What   would   you   like   to   have?');

  EXCEPTION                     WHEN                 ex_age            THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

I have explained this example in my Video Tutorial on my YouTube channel in detail I would request you to check out the tutorial there.

Why use PRAGMA EXCEPTION_INIT with RAISE_APPLICATION_ERROR?

Though it is not mandatory to use PRAGMA EXCEPTION_INIT with RAISE_APPLICATION_ERROR procedure however it is more of a personal preference than a programming rule. If you want to print an error message with an error number like the Oracle's standard way of showing an error then it is the best practice to use PRAGMA EXCEPTION_INIT with RAISE_APPLICATION_ERROR procedure.



```
SQL> CREATE TABLE RebellionRider(
  2  stu_id NUMBER;
stu_id NUMBER
             *
ERROR at line 2:
ORA-00907: missing right parenthesis


SQL>
```

But if you just want to print the error message & not the error number then you can use PRAGMAEXCEPTION_INIT with RAISE statement.

PL/SQL Collections

After Exception handling, the topic which we finished off with the last PL/SQL tutorial, collection is the most demanded topic on my social media. That is why our next series of tutorials will be based on PL/SQL collections. Starting with today.

What are PL/SQL Collections in Oracle Database?

A homogeneous single dimension data structure which is made up of elements of same datatype is called collection in Oracle Database. In simple language we can say that, an array in Oracle Database is called Collection.

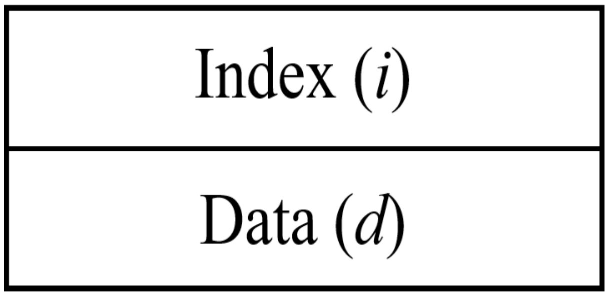
Definition

A homogeneous single dimension data structure which is made up of elements of same datatype is called collection in Oracle Database.

Why we call collection a homogeneous data structure?

As we know that array consists data of same datatype and so does the PL/SQL collection which is why we call them homogenous data structure.

The structure of PL/SQL collections consist of a cell with subscript called index. Data is stored into these cells and can be identified and accessed using the index number. This is again very similar to the structure of arrays, but unlike array PL/SQL Collections are strictly one-dimensional.



info byte

The collection in Oracle Database are strictly One-Dimensional. It is not possible to realize them on 2D co-ordinates. However when the collection has an attribute of object type or collection type then it is possible to realize a multi-dimensional array.

Types of PL/SQL Collections in Oracle Database

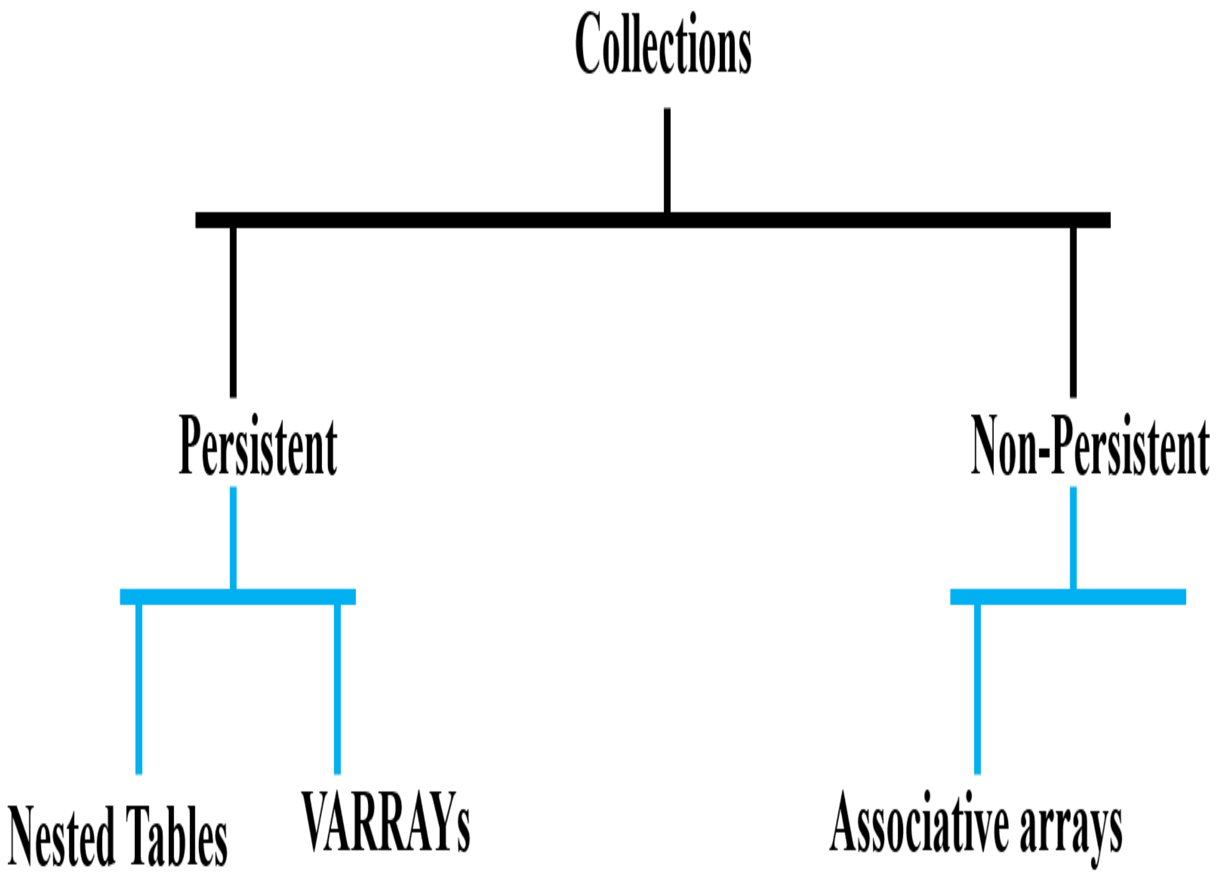
PL/SQL collections can be divided into two categories:

105. Persistent and
106. Non-persistent.

Persistent collection, as the name suggests, are those which physically store the collection structure with the data into the database and can be accessed again if needed. Whereas **non-persistent collection** only stores data and structure for one session.

On the basis of above categories collections are further divided into three types:

107. Nested Tables
108. Variable Sized Arrays or VARRAYs and
109. Associative arrays.



Nested Table – Nested tables are persistent collection which means they can be stored into the database and can be reused. Nested tables has no upper limits on rows thus they are unbounded collections. Nested tables are initially dense but can become sparse through deletion.

VARRAYs – Similar to Nested tables Variable-Sized Arrays are also persistent collections thus they can be created in database as well as PL/SQL block and can be reused. But unlike nested tables VARRAYs are bounded in nature which means that they can hold only a fixed amount of elements.

info byte

The size and storage schema of VARRAYs makes them different from nested tables.

Associative Array – Unlike nested table and VARRAYs, associative arrays are non-persistent collections thus they cannot be stored into the database. Since they cannot be store hence they cannot be reused but they are available in PL/SQL block for the session. But similar to nested tables associative arrays are unbounded which means they also don't have lower and upper limits on rows.

Commonly used terms in PL/SQL Collection.

Bounded & Unbounded Collection – A collection which has lower or upper limits on values of row number or say a collection which can hold only limited number of elements are called bounded collections. A collection which has no lower or upper limits on row numbers are called unbounded collections.

Dense & Sparse Collection. – Collections is said to be dense if all the rows between the first and the last are defined and given a value. And a collection in which rows are not defined and populated sequentially are called sparse collection.

That's it for this tutorial. You can help others in learning as well as help me and my channel in growing by sharing this blog with your friends or on your social media.

Nested Table in PL/SQL Block

Welcome to the second tutorial of the PL/SQL Collection series. In this tutorial we will learn the first type of Collection that is "Nested Table". A table inside a table is the simplest definition one can come up with and it is correct in every way because a table which is embedded inside another table is exactly what the name nested table suggests.

But, if we have to define the collection 'Nested table' in a more fancy and technical way then we can say Nested tables are one-dimensional structures that are persistent and unbounded in nature. They are accessible in SQL as well as PL/SQL and can be used in tables, records and object definitions. Since it is an unbounded PL/SQL collection hence it can hold any number of elements in an unordered set.

Definition

Nested tables are one-dimensional structures that are persistent and unbounded in nature. They are accessible in SQL as well as PL/SQL and can be used in tables, records and object definitions. Since it is an unbounded PL/SQL collection hence it can hold any number of elements in an unordered set.

A nested table can be created inside the PL/SQL block or in database as a collection type object (Schema Object). In case of the former nested table behaves as a one-dimensional array without any index type or any upper limit.

So for the time being let's concentrate on how to create Nested Table inside PL/SQL block and leave the rest for the next tutorial.

Syntax for Creating Nested Table

```
DECLRE
TYPE nested_table_name IS TABLE OF element_type [NOT NULL];
```

I’ve explained this very syntax in detail in my video tutorial on my YouTube channel. I highly recommend you to refer to that video.

Example: How to Create Nested Table inside a PL/SQL Block?

The following example is only for demonstrating how to create nested table, there is nothing fancy about it.

```
1 SET SERVEROUTPUT ON;
2 DECLARE
3 TYPE my_nested_table IS TABLE OF number;
4 var_nt my_nested_table := my_nested_table (9,18,27,36,45,54,63,72,81,90);
5 BEGIN
6 DBMS_OUTPUT.PUT_LINE ('Value Stored at index 1 in NT is ' ||var_nt (1));
7 DBMS_OUTPUT.PUT_LINE ('Value Stored at index 2 in NT is ' ||var_nt (2));
8 DBMS_OUTPUT.PUT_LINE ('Value Stored at index 3 in NT is ' ||var_nt (3));
9 END;
```

Above example is a very simple one in which we created a nested table and named it ‘my_nested_table’ (line number 3). In the next line (line number 4) we created an instance of the same collection and used it to initialize the nested table and store some data into it. In the execution section we access the stored data individually using the index number, the same way we used to do in arrays.

Instead of accessing data one by one manually using index we can use loops and cycle through each element of the collection nested table.

```
SET SERVEROUTPUT ON;
DECLARE
TYPE my_nested_table IS TABLE OF number;
var_nt my_nested_table := my_nested_table (9,18,27,36,45,54,63,72,81,90);
BEGIN
FOR i IN 1..var_nt.COUNT
LOOP
DBMS_OUTPUT.PUT_LINE ('Value stored at index '||i||'is '||var_nt(i));
END LOOP;
END;
/
```

That is another example of how to create nested table in which we cycle through the data and display it back to the user using For Loop.

Nested Table as Database Object

If you plan to reuse the nested table that you want to create then doing so as a database object is the best choice for you. You can store them in your database permanently and use them whenever you want.

Apart from creating Nested Table type PL/SQL Collection inside a PL/SQL block you can also create them as database object and store permanently. Also you can reuse them whenever you want. Nested table created as database object can be based on either Primitive Datatype or User-Define Datatype. In this tutorial we will concentrate on former and leave the latter for the next tutorial.

How to Create Nested table type collection based on primitive datatype

By primitive datatype we mean the datatypes which are predefined by the language and are named by a reserved keyword. You can refer to this Oracle Document to read more about PL/SQL Datatypes. *eated purely for demonstrating how to create nested table as database object.*

Step 1: Set Server output on

```
SET SERVEROUTPUT ON;
```

Step 2: Create Nested Table type collection

```
CREATE OR REPLACE TYPE my_nested_table IS TABLE OF VARCHAR2 (10);
/
```

The above statement on successful execution will create a nested table with name 'my_nested_table' which will be based on primitive datatype VARCHAR2.

Step 3: How to use nested table?

The collection type which we created above can be used to specify the type of a column of a table.

```
1 CREATE TABLE my_subject (
2 sub_id NUMBER,
3 sub_name VARCHAR2 (20),
4 sub_schedule_day my_nested_table
5 ) NESTED TABLE sub_schedule_day STORE AS nested_tab_space;
6 /
```

The above table is a normal table except that its 3rd column is of nested table type which can hold multiple values. In order to define a column of a table as nested table type you have to tell the compiler the name of the column and a storage table. You can do so by using NESTED ABLE and STORE AS clause, as we did here in line number 5. Using clause NESTED TABLE we specify the name of the column and using STORE AS clause we specify the storage table for the nested table.

Insert rows into the table

```
INSERT INTO my_subject (sub_id, sub_name, sub_schedule_day)
VALUES (101, 'Maths', my_nested_table('mon', 'Fri'));
```

You insert rows into the nested table same as you insert into the normal table. However in order to insert data into the column of nested table type you first have to write the name of nested table which in our case is 'my_nested_table' (refer step 2) and then write the data according to the datatype of your nested table and enclose it inside the parenthesis.

Retrieve data from the table

A simple SELECT DML statement can be used to retrieve the data from the table.

```
SELECT * FROM my_subject;
```

This simple DML statement will show you all the data stored into the table that we created above. To see the data from a specific row you can use WHERE clause with SELECT DML

```
SELECT * FROM my_subject WHERE sub_id = 101;
```

If you want then you can take help of sub-query to just check the data from the column which you defined as nested table type.

```
SELECT * FROM TABLE (
  SELECT sub_schedule_day FROM my_subject WHERE sub_id = 101
);
```

The above query will show you the data of subject which has subject-id 101 only from sub_schedule_day column. In this query we used TABLE expression to open the instance and display the data in relational format.

Update data of the table

You can either update all the values of the column which you define as nested table or you can update a single instance of the same.

Update all the values of the nested table type column.

```
UPDATE my_subject SET sub_schedule_day = my_nested_table('Tue', 'Sat')
WHERE sub_id = 101;
/
```

The above query will update all the values of sub_schedule_day from 'Mon', 'Fri' to 'Tue' and 'Sat'. Now suppose you want to update only a single instance of this column by replacing 'Sat' with 'Thu'. How will you do that?

Nested Table Using User-Define Datatype

Hey guys! Today we will learn how to create nested table type collection using user-define datatype. Hope you had great time with the last tutorial where we learnt the creation process of nested table with primitive datatype. I highly suggest you to take a look at that tutorial as we are going use the concepts from there.

Similar to primitive datatype a nested table can be created using user define datatypes also. For the demonstration we will use user Oracle Object. Objects require no introduction, if you have ever studied OOP Concepts. In Oracle, like other programming languages, object type is a kind of a datatype which works in the same way as other datatypes such as Char, Varchar2, Number etc. but with more flexibility.

To create an Oracle Object we use our old and trusty 'Create Type' statement.

```
CREATE OR REPLACE TYPE object_type AS OBJECT (
  obj_id NUMBER,
  obj_name VARCHAR2(10)
);
/
```

The above statement will create an oracle object with the name 'object type' with two attributes obj_id and obj_name on successful execution. This datatype then can be used to create a nested table.

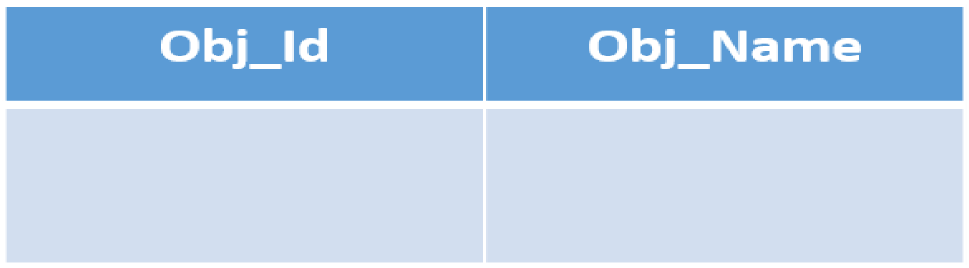
```
CREATE OR REPLACE TYPE My_NT IS TABLE OF object_type;
/
```

I think if you checked the last tutorial then you'll find this above statement very familiar, except the element type which is a primitive datatype there. Here we use a user-define datatype which is an Oracle Object.

The above statement has been explained in detail in the last tutorial which you can check here.

What does creating a nested table using Oracle Object means?

Whenever you create a nested table using an Oracle Object then the attributes of the Object become the columns of that table. For example, in our case we created a nested table '*My_NT*' using the user-define datatype which is an Oracle Object '*Object_Type*' which has two attributes *obj_id* and *obj_name*. These two attributes of the object will act as the columns of the table. The following pic will help you in understanding this more clearly.



Obj_Id	Obj_Name

Table: My_NT

Now that we have created the nested table using user-define datatype it's time for putting it to some work.

```
CREATE TABLE Base_Table(
   tab_id NUMBER,
   tab_ele My_NT
)NESTED TABLE tab_ele STORE AS stor_tab_1;
/
```

The above table named 'Base_Table' is a simple one which has 2 columns 'tab_id' and 'tab_ele'. The first column is of Number Datatype while second column is of Nested Table type. This means that the second column contains a table into it and that table is our Nested table 'My_Nt'

Though this 'Base_Table' is a simple table but one of its column contains a nested table into it which arises some questions such as:

How to insert data into the table? How to update the data of the table? Or how to retrieve the data from the table? Let's try to find out the answers to all these questions one at a time.

How to insert data into the nested table?

Yes, I agree that inserting data into a table which has a column of nested table type can be tricky but somehow we have to find the way of doing it. As a table without data is of no use to us. Right? Let's see how we can do that.

```
1 INSERT INTO base_table (tab_id, tab_ele) VALUES
2 ( 801, -- value for 1st colum
3 My_NT (object_type (1,'Superman') -- values for 2nd column )
4 );
```

As you can see in this INSERT statement everything is same as a normal Insert DML except the line number 3 where we are inserting data into the second column of the table. In order to insert data into the column which is of Nested Table type you first have to write the name of your nested table which in this case is 'My_NT' then you have to write the name of your Oracle Object which here is 'Object_Type' followed by the values you want to insert into your table. Don't forget to match the parenthesis for table name and object name, otherwise you will get an error.

How to update values of the nested table?

```
UPDATE base_table SET tab_ele = My_NT(object_type(1,'SpiderMan')) WHERE tab_id = 801;
```

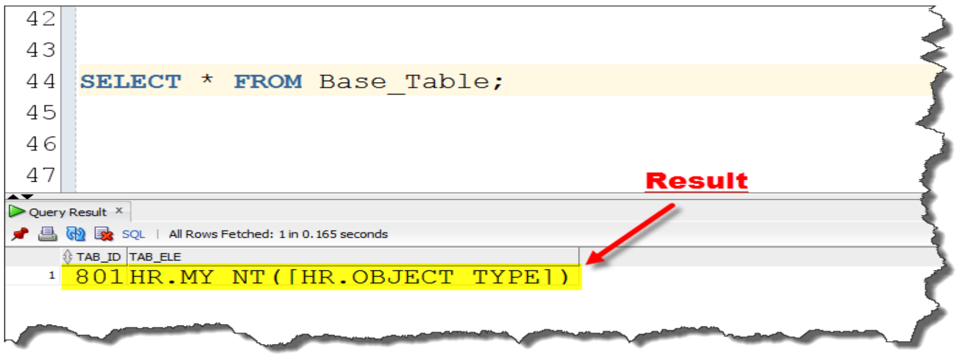
The above DML statement will update the values accordingly on successful execution.

How to retrieve data from the nested table?

You can simply execute the Select statement on your table to get the data.

```
Select tab_id, tab_ele FROM base_table;
```

The following picture will show you the result returned from this table.

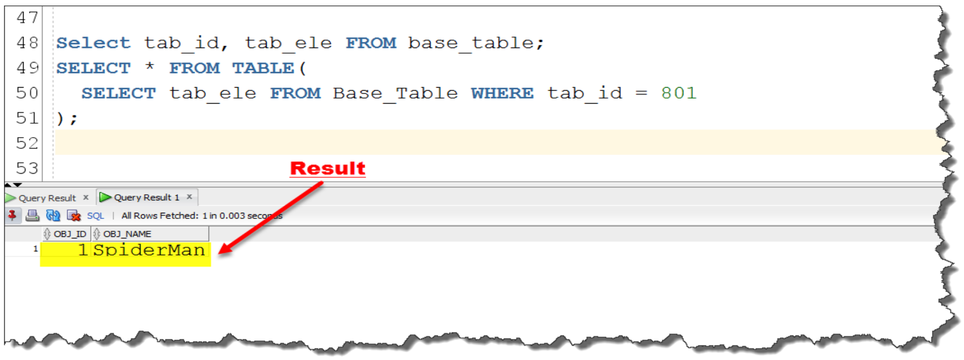

```
SELECT * FROM Base_Table;
```

```
  TAB_ID TAB_ELE
1    801 HR.MY_NT([HR.OBJECT_TYPE])
```

As you can see this SELECT statement will show you the data from the columns which are of primary datatype but only the name of your nested table along with the Oracle Object from the column which you define as a Nested Table type. You can easily overcome this problem by using TABLE expression like this.

```
SELECT * FROM TABLE(
    SELECT tab_ele FROM Base_Table WHERE tab_id = 801
)
```

Successful execution of the above query will show you the data from second column of your table 'Base_Table' in a relational format.


```
Select tab_id, tab_ele FROM base_table;
SELECT * FROM TABLE(
  SELECT tab_ele FROM Base_Table WHERE tab_id = 801
);
```

```
  OBJ_ID OBJ_NAME
1      1 SpiderMan
```

That's it for this tutorial hope you enjoyed and learnt something new. Make sure to subscribe and signup. Have a great day!

Introduction to VARRAYs

VARRAYs were launched in Oracle 8i back in 1998 as a modified version of Nested table type collection which we have discussed in the previous blog.

Suggested Reading

How to Create Nested Table –

- As PL/SQL Block element
- As Database Object
- Using User-Define Datatype
    - Also, Now you can test your knowledge on the subject by taking a test here.

VARRAY is an important topic because it is seen that generally there is always a question on it in certification exam. In order to minimize any confusion we will first take a brief look at the intro of VARRAYs collection.

VARRAYs which is an acronym of Variable Sized Arrays were introduced in Oracle 8i back in 1998 as a modified format of nested tables. The major modifications can be seen in storage orientation. ***There are no noticeable changes in the implementation but their storage orientation is completely different compared to the nested tables***. Unlike nested table which requires an external table for its storage, ***VARRAYs are stored in-line with their parent record as raw value in the parent table***. It means **no more need for STORE AS clause**. Oh, what a relief, no unnecessary I/Os and on top of that increased performance.

Can we save and reuse VARRAYs?

Similar to Nested Tables VARRAYs are Persistent type of Collection which means they can be created as database object that can be saved for later use. VARRAYs can also be created as member of PL/SQL Blocks. The scope of the VARRAY which is declared inside a PL/SQL block is limited to the block in which it is created.

Are VARRAYs bounded or Unbounded?

Unlike Nested table VARRAYs are bounded form of collection. By bounded I mean, you have to decide how many elements you want to store in your collection while declaring it. Whereas in nested table which is unbounded type of collection there is no upper cap on number of elements.

VARRAYs Storage Mechanism

The storage mechanism of VARRAYs is the biggest difference which makes them a superior choice than Nested tables. Unlike nested tables which requires an external table for its storage, VARRAYs are stored in-line with their parent record as raw value in the parent table. This means there is no requirement of STORE AS clause or separate storage table.

The in-line storage of VARRAYs help in reducing disk Inputs/Outputs (I/O) which makes VARRAYs more performance efficient than nested table. But when VARRAYs exceed 4K data then Oracle follows out-of-line storage mechanism and stores VARRAYs as an LOB.

Syntax for creating PL/SQL VARRAYs

In this section we will see the syntax for creating VARRAYs as

- Database Object and
- Member of PL/SQL Block.

You can head over to the Video on the same topic on my YouTube channel where I explained both these syntax in detail.

VARRAY as Database Object

```
CREATE [OR REPLACE] TYPE type_name
IS {VARRAY | VARYING ARRAY} (size_limit) OF element_type;
```

VARRAY as a member of PL/SQL Block

```
DECLARE
  TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit) OF
  element_type;
```

Both the above syntaxes are same as of nested table except here we have an additional clause which is Size_Limit. Size limit is a numeric integer which will indicate the maximum number of elements your VARRAY can hold.

Always remember similar to nested table we can declare VARRAY only in the declaration section of PL/SQL block.

How to modify the size limit of VARRAYs type collection?

Size limit of a VARRAY can be altered using ALTER TYPE DDL statement.

```
ALTER TYPE type_name MODIFY LIMIT new-size-limit [INVALIDATE | CASCADE]
```

Where
ALTER TYPE is a reserved phrase that indicates to the compiler which DDL action you want to perform.
TYPE NAME is the name of type which you want to alter.
MODIFY LIMIT is a clause which informs the compiler that user wants to modify the size limit.
NEW-SIZE-LIMIT is an integer which will be the new size limit of your VARRAY.
INVALIDATE clause is an optional clause which will Invalidate all dependent objects without any checking mechanism.
CASCADE clause again is an optional clause which will propagate changes to dependent types and table.

How to drop a VARRAY type collection?

To drop a VARRAY type you can take help from DROP DDL statement.

```
DROP TYPE type_name [FORCE];
```

Where
Drop Type
Is a DDL statement using which you can drop any type created on your database.
Type name
Type name is the name of an already created type which you want to drop.
Force
Specify FORCE to drop the type even if it has dependent database objects. Oracle Database marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible. Remember this operation is not recoverable and could cause the data in the dependent tables or columns to become inaccessible.

How to create varray as PL/SQL block Member

Today we are going to do some practical demonstration to learn how to create a VARRAY in Oracle Database so feel free to visit the last tutorial for all the necessary theories on VARRAYs. Like always I will try to keep the example as simple as possible. So, if you are new to programming, don't worry, I got you!

Previously we discussed in the introduction to PL/SQL VARRAYs that like nested table VARRAYs can be created

- As a member of PL/SQL Block and
- As a database object.

Today we will learn how to create VARRAYs as a Member of PL/SQL block and leave the rest for the future tutorials.

Step 1: Define a Varray inside PL/SQL block

You can define a varray only inside the declaration section of a PL/SQL block.

```
SET SERVEROUTPUT ON;
DECLARE
```

```
TYPE inBlock_vry IS VARRAY (5) OF NUMBER;
```

In the above code we created a VARRAY and named it ***inBlock_vry***. This varray is capable of holding 5 elements of Number datatype.

Step 2: Initialize the Varray

Initialization of a varray can easily be done using the collection variable. To initialize the VARRAY we will first define a collection variable and then use it for initializing.

```
vry_obj inBlock_vry := inBlock_vry();
```

In the above code we created a collection variable with name vry_obj and used that to initialize the varray inBlock_vry.

Info:

Some books refer to collection variable as collection object, so please don't get confused as both are the same.

Step 3: How to insert data into the VARRAY

Inserting data into the varray is very similar to inserting data into the array of other programming language. You can insert data either directly into each cell of the varray using the index number or you can use LOOP for populating the varray.

Step 3.1: How to insert data into the VARRAY using index of the cell

As we know that the structure of a cell PL/SQL collection consists of a cell with a subscript called index. We can use this index for inserting the data into the varray.

```
5 BEGIN
6 vry_obj.EXTEND(5);
7 vry_obj(1):= 10*2;
8 DBMS_OUTPUT.PUT_LINE(vry_obj(1));
9 END;
10 /
```

Execution section -1

In the above code we wrote the execution section of the PL/SQL block. It consists of 3 executable statements. These three statements are –

Line 6: Statement 1

First statement is an EXTEND procedure call. In this statement we are allocating the memory to each cell of VARRAY using the EXTEND procedure.

Line 7: Statement 2

In the second statement we are assigning a numeric value (value derived from arithmetic multiplication expression) into the first cell of the varray (cell with index number 1).

Info:

In PL/SQL collection VARRAY index number of the cell starts with 1 whereas the index number of cells in array starts with 0.

Line 8: Statement 3

Third statement is an output statement where we are displaying the value which we stored into the 1st cell of the VARRY back to the user.

That is how you can store and display the value stored in individual cell of the varray. This process is good only when you have a short varray. Otherwise it is not an efficient way. Another way of inserting data into the Varray is by using Loop.

Step 3.2: How to insert data into a VARRAY using PL/SQL Loop

The most common way of dealing with data of a collection is by using Loops. Most programmers are used to using Loops to cycle through the data of any kind of array because this is easy, less time consuming and have less line of codes which keep your code cleaner and makes it easy to read. In short it is easy and efficient.

```
5 BEGIN
6 FOR i IN 1 .. vry_obj.LIMIT
7 LOOP
```

```
8     vry_obj.EXTEND;
9     vry_obj (i):= 10*i;
10    DBMS_OUTPUT.PUT_LINE (vry_obj (i));
11    END LOOP;
12    END;
13    /
```

Execution section -2

I have explained the above code in detail in the Video tutorial on my YouTube channel which you can watch here.

Still to minimize the confusion I will explain to you here the two main functions used in the above execution section (Execution Section – 2) that are – Limit & Extend.

To understand the working of above shown code requires the understanding of PL/SQL for loop. Gladly, I have done a detailed tutorial on For-Loop, which you can read here.

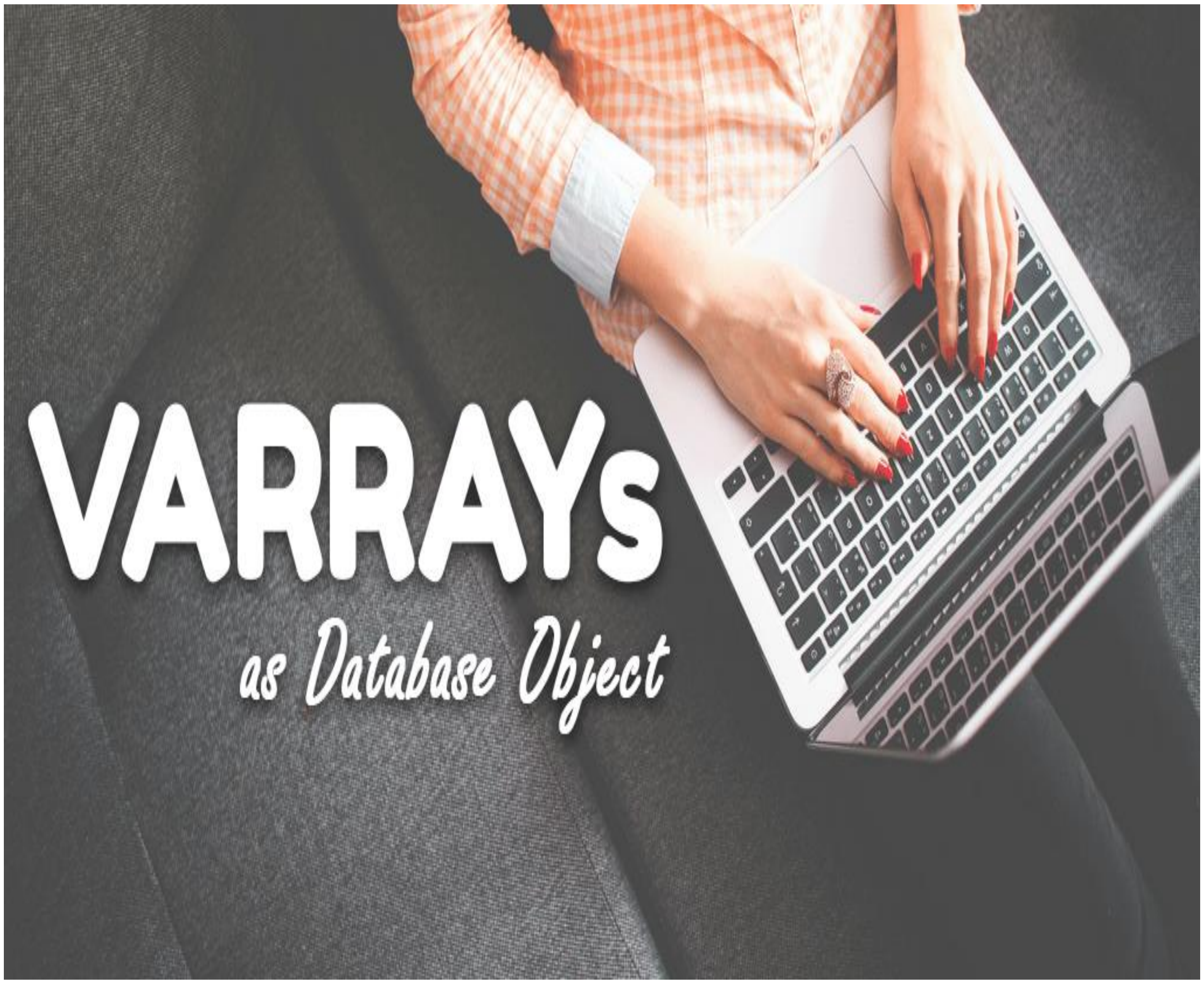
Limit (line 6): Limit is a collection method which returns the maximum number of elements which are allowed in the VARRAY. In our case the maximum number of elements which are allowed in the VARRAY is 5 (line 3) which in turn becomes the upper limit of the For-Loop here.

Extend (Line 8): Extend is a procedure which is used for allocating the memory and appends an element to the VARRAY. If used without argument (Execution Section-2 Line 8) it appends single null element and if used with an argument EXTEND (n) (execution section -1 Line 6) it then appends n numbers to the collection. Where n is the integer you supplied as an argument to the procedure EXTEND.

VARRAYs as Database Object

The scope of the VARRAY which is created as PL/SQL block member is limited to the block in which it is created, that means we cannot use this VARRAY outside its block or even reuse it and that is its biggest drawback. So go ahead & read on to find out how we can overcome this disadvantage of VARRAY.

This drawback can easily be overcome if we can find out a way to create the VARRAY outside the PL/SQL blockand store it permanently into the schema. Fortunately we can achieve both the goals by creating the VARRAY as a database object. That is exactly what we are going to learn in this tutorial.



In this tutorial we will learn –

- How to create VARRAY as database object.
- How to use that varray.
- How to insert data into the VARRAY.
- How to retrieve data from the VARRAY
- How to update the data of the VARRAY.

Let's start with the first step.

How to Create VARRAY as Database Object?

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE TYPE dbObj_vry IS VARRAY (5) OF NUMBER;
```

```
/
```

Above code on successful execution will create a VARRAY with name dbObj_vry which will have the size limit of 5 elements and their datatype will be NUMBER. This VARRAY has wider scope and can be used not only inside the PL/SQL block but also with other schema objects.

How to Use the VARRAY Created as Database Object?

The benefit of defining the VARRAY as database object is that it may be referenced from any program that has the permission to use it. You can use the VARRAY with tables, records or even with PL/SQL blocks.

Let’s do the example:

Example 1. How to define a column of a table using VARRAY?

```
CREATE TABLE calendar(
    day_name VARCHAR2(25),
    day_date dbObj_vry
);
/
```

In the above code we created a table with the name Calendar which has two columns day_name and day_date. The first column can hold data of VARCHAR2 datatype whereas the second column can hold data of a dbObj_vry type which is a VARRAY.

Info: What does defining a column of a table as VARRAY type means?

Defining a column of a table as VARRAY type means that it can hold 'n' number of values into it. Where 'n' is equal to the size limit of that varray. In our case the size limit of VARRAY is 5 that means the column 'Day Date' of table Calendar can hold 5 values.

How to insert data into the VARRAY?

```
9. INSERT INTO calendar ( day_name, day_date )
10.    VALUES ( 'Sunday', dbObj_vry (7, 14, 21, 28) );
```

This insert DML statement will insert a row into the Calendar table. Inserting data into the first column 'Day Name' which is of varchar2 datatype is easy. You just have to write the desired data and enclose it into single quotes. But same is not true with the second column 'Day Date' which is of VARRAY type. In order to insert data into the column which is of VARRAY type you first have to write the name of the varray and supply the data.

Also, you have to make sure four things

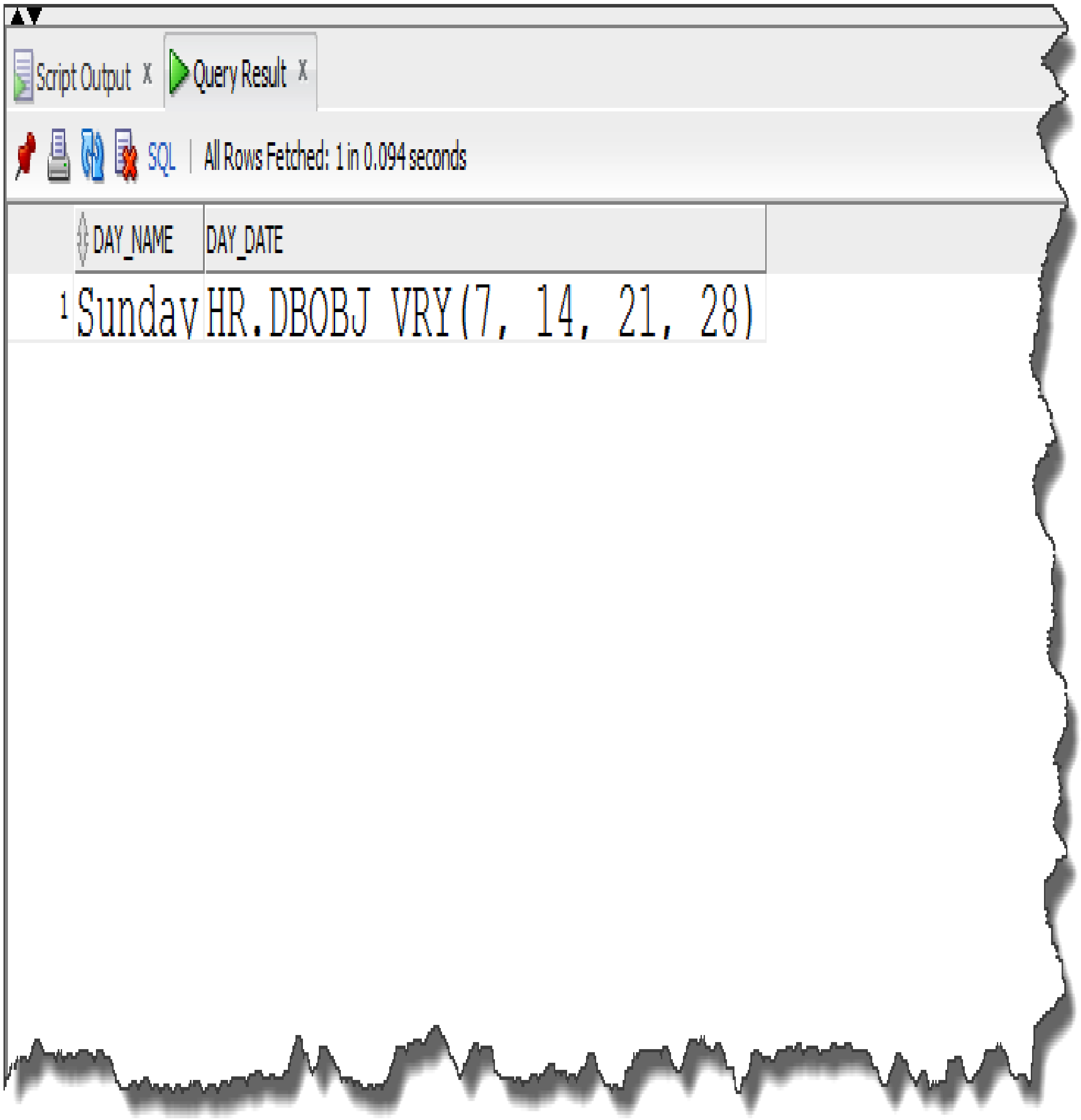
110. The data that you are supplying must be enclosed inside the parenthesis.
111. The Datatype of the data must match with the datatype of the elements of your VARRAY which in our case is NUMBER.
112. The number of elements you are inserting into the column must either be less than or equal to the size limit of the VARRAY. In our case it is 5 and we are inserting 4 elements into the column which is completely ok. But if suppose I insert 6 elements into the column then there will be an error.
113. If inserting multiple data into VARRAY column then make sure to separate the elements from each other using semi-colon.

How to retrieve the data from the VARRAY?

Data can be retrieved using SELECT statement. Any correctly written SELECT statement will do the work. For example

11. SELECT * FROM calendar;

This will retrieve all the data from the table calendar.



All Rows Fetched: 1 in 0.094 seconds

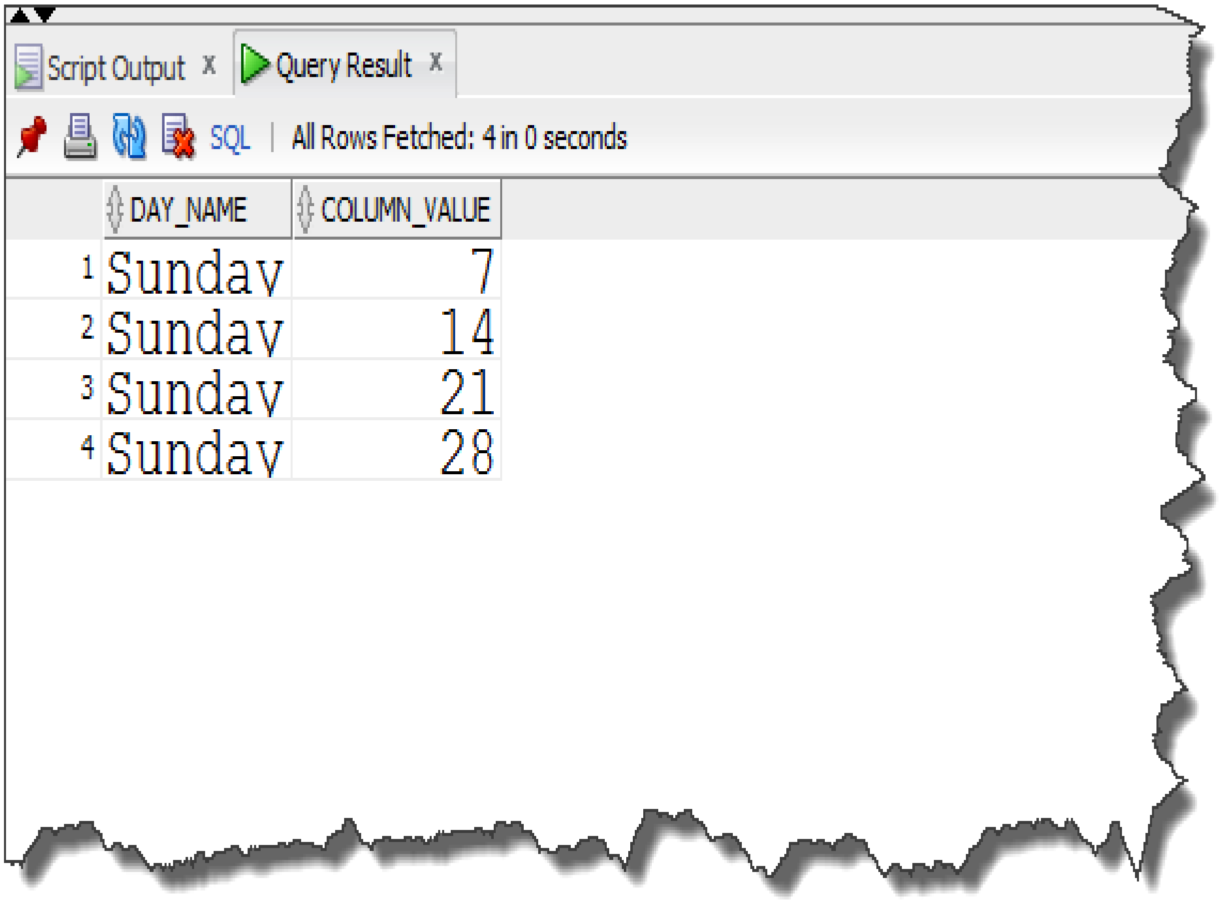
```
  DAY_NAME DAY_DATE
1 Sunday   HR.DBOBJ_VRY(7, 14, 21, 28)
```

Result 1: Simple SELECT Statement

In case you want to display the data stored into the column, which is holding data of VARRAY type, in a relational format then you can take the help of TABLE expression. For example

```
SELECT
tab1.day_name,
vry.column_value AS "Date"
FROM calendar tab1, TABLE (tab1.day_date) vry;
```

This SELECT statement will show you the data from both the columns in a relational format. The TABLE expression can open the collection instance and represent the object rows in relational format.



All Rows Fetched: 4 in 0 seconds

	DAY_NAME	COLUMN_VALUE
1	Sunday	7
2	Sunday	14
3	Sunday	21
4	Sunday	28

Result 2: TABLE Expression

How to update the data of VARRAY type column?

Updating the values of VARRAY type column is pretty simple. Below example will show you how to update the values of day_date columns.

```
UPDATE calendar
  SET day_date = dbObj_vry(10,14,21,28)
  WHERE day_name = 'Sunday';
```

Example 2. How to use VARRAY with PL/SQL block?

In the above example we learnt, how to use the VARRAY which is created as Database object to define the column of a table. Now we will see how to use the same varray inside a PL/SQL block.

```
DECLARE
vry_obj dbObj_vry := dbObj_vry();
BEGIN
FOR i IN 1..vry_obj.LIMIT
LOOP
vry_obj.EXTEND;
vry_obj(i):= 10*i;
DBMS_OUTPUT.PUT_LINE(vry_obj(i));
END LOOP;
END;
/
```

You have seen this example in the last tutorial. There are no such big changes here except that this time instead of defining the VARRAY inside the block we created it as a standalone database object. I suggest you to take a look at the last tutorial where I explained the above code in detail.

Associative Array

Associative array is formerly known as PL/SQL tables in PL/SQL 2 (PL/SQL version which came with Oracle 7) and Index-by-Table in Oracle 8 Database. After Nested Table and VARRAYs, Associative Array is the third type of collection which is widely used by developers.

Let’s find out the answers of a few questions about associative array which would help you in understanding them better. In this section you will also find out some of the core differences & similarities between Associative Array and other collections such as VARRAY & Nested Tables.

Are Associative arrays bounded or Unbounded?

Similar to Nested tables, ***Associative arrays are unbounded form of collection***. This means there is no upper bound on the number of elements that it can hold. Same is not true for VARRAYs as Variable arrays are bounded in nature.

Are Associative arrays persistent or non-persistent?

Unlike Nested Table & VARRAYs, ***Associative arrays are non-persistent form of collection***. This means neither the array nor the data can be stored in the database but they are available in PL/SQL blocks only.

Are Associative arrays sparse or dense?

Whereas VARRAYs are densely populated arrays, ***Nested tables and Associative Arrays are sparsely populated arrays*** which mean that subscript numbering must be unique but not necessarily sequential.

Can we create Associative array as database object?

Because of their non-persistent nature Associative arrays cannot be stored into the schema. They can only be created in PL/SQL blocks but not at schema level as database object.

Can we reuse associative array?

As mentioned above Associative array is a non-persistent collection which cannot be created at schema level thus it cannot be stored into the schema hence it cannot be reused.

Is index numbering/Subscript numbering in Associative array implicit or explicit?

Unlike Nested Tables and VARRAYs, indexing in Associative array is Explicit. Where Oracle Engine assigns subscript/Index number to the elements of the Nested table and VARRAY collections implicitly in the background, in associative array users have to specify the index number explicitly while populating the collection.

How does Data gets stored into the Associative Array?

Associative array stores data in Key-Value pairs where index number serves as the key and data stored into the cell serves as the value.

These are a few core questions which you can expect in your exam or interview. Read along to find out the technical differences between Associative arrays and other collections.

Define PL/SQL Collection - Associative Array?

Using the information derived from above questions we can define Associative Arrays as one-dimensional, homogenous collection which stores data into key-value pair. It is sparse, unbounded and non-persistent in nature.

What is the Syntax of PL/SQL Associative Array?

```
TYPE aArray_name IS TABLE OF element_datatype [Not Null]
    INDEX BY index_elements_datatype;
```

As said above Associative array is non-persistent type of collection thus it cannot be created as standalone database object hence cannot be reused like the rest of the other collections. It can only be available in PL/SQL block. Always make sure you create your associative array in DELCARATION section of your PL/SQL Block. [Read here to know how many sections are there in PL/SQL block?] Let's see the syntax in detail –

Type: Keyword marks the beginning of the statement.

aArray_name: Name of the associative array. It is completely user-defined and complies with Oracle Database naming norms.

IS TABLE OF: Oracle Database reserved phrase using which user tells the compiler what type of elements the array is going to hold?

Element_Datatype: Datatype of the elements the array is going to hold. In Oracle Database all the collections are homogenous in nature, which means every element of the collection must be of the same datatype.

Not_null: An optional clause, which if used makes sure that every index has a value corresponding to it rather than a NULL.

INDEX BY: Clause using which user specifies the datatype of array's subscript.

Index_elements_dataype: Datatype of the array's subscript elements.

Example: How to Create Associative Array in Oracle Database?

Associative array can only be created inside a PL/SQL block thus its scope is limited to the block in which it is created which means it cannot be used outside that block. Let’s see how to create an Associative Array in Oracle Database?

Step 1: Create Associative Array

```
SET SERVEROUTPUT ON;
DECLARE
TYPE books IS TABLE OF NUMBER
INDEX BY VARCHAR2 (20);
```

In the above code we created an Associative array with the name ‘***Books***’ which can hold elements of NUMBER datatypes and subscript of VARCHAR2 datatype.

Step 2: Create Associative Array Variable

```
Isbn Books;
```

You need an Associative array variable for referencing the array in the program. Array variable can be created very easily. You just have to write the name of the variable (which is ‘***isbn***’ in our case) which is user defined followed by the name of the associative array.

Step 3: Insert Data into the Associative Array

As mentioned above Associative array holds data into key-value pairs. Thus unlike rest of the other collections the users have to insert both the subscript of the array (the key) and the data.

```
BEGIN
-- How to insert data into the associative array
isbn('Oracle Database') := 1234;
isbn('MySQL') := 9876;
```

Like Nested table and VARRAYs we insert data into the Associative array in the execution section of PL/SQL block. If you noticed here unlike other collections we didn’t use the INSERT DML statement for inserting the data rather we inserted it using the Array variable ‘***isbn***’ . Below you can see the syntax of insert statement for associative array using array variable.

```
Array_variable (subscript/key) := data;
```

As you can see in order to insert the data into the associative array you first have to write the name of array variable followed by the array’s subscript and then the data for your array.

Step 4: How to update the data of collection - Associative array?

Updating values of Associative array is as easy as inserting them. If you want to change any value write the same statement which is used for insertion with the modified values. For example say you want to change the value against the key MySQL from 9876 to 1010 then you just write

```
-- How to update data of associative array.
isbn('MySQL') := 1010;
```

Again you don’t need to write the UPDATE DML for updating the values. You simply use the array variable.

Step 5: How to retrieve data from the Collection-Associative array?

Just like we don’t need Insert DML statement for inserting values or Update DML for updating values similarly we don’t need Select DML for retrieving values.

Suppose you want to see the value stored against the key ‘Oracle Database’. For that you just need to write…

```
-- how to retrieve data using key from associative array.
DBMS_OUTPUT.PUT_LINE ('Value '||isbn ('Oracle Database'));
```

Let’s combine all these chunks of code into a single program.

```
SET SERVEROUTPUT ON;
DECLARE
  TYPE books IS TABLE OF NUMBER
    INDEX BY VARCHAR2(20);
  isbn Books;
BEGIN
  -- How to insert data into the associative array
  isbn('Oracle Database') := 1234;
  isbn('MySQL') := 9876;
  DBMS_OUTPUT.PUT_LINE('Value Before Updation '||isbn('MySQL'));

  -- How to update data of associative array.
  isbn('MySQL') := 1010;

  -- how to retrieve data using key from associative array.
  DBMS_OUTPUT.PUT_LINE('Value After Updation '||isbn('MySQL'));
END;
/
```

Here is the program with some minute modifications. The above PL/SQL program shows how to retrieve one specific value using the key. You can watch the Video Tutorial to learn how to retrieve all the values from Associative Array using Loops. There I have explained it in great detail.

Before winding up this tutorial, there are few pointers which I think you should know. These pointers are –

- PL/SQL Associative Array support BINARY_INTEGER, PLS_INTEGER, POSITIVE, NATURAL, SIGNTYPE or VARCHAR2 as index datatype.
- RAW, NUMBER, LONG-ROW, ROWID and CHAR are unsupported index datatypes.

In case of Element Datatype, PL/SQL collection Associative Array Supports –

- **PL/SQL scalar data type**: DATE, BLOB, CLOB, BOOLEAN or NUMBER & VARCHAR2 with their subtypes.
- **Inferred data**: Term used for such data types that are inherited from a table column, cursor expression or predefined package variable
- **User-defined type**: An object type or collection type which is user defined.

That is a detailed tutorial on PL/SQL Collection – Associative Array. This tutorial covers all the topics which you can expect in Oracle Database Certification Exam as well as in Interview. Hope you enjoyed reading.

Collection Methods

We have used them time and again but never got the chance to talk about them. Thus today we will discuss collection methods.

What are Collection Methods?

Collection methods are PL/SQL's in-built functions and procedures which can be used in conjunction with collections.

So, what they can do for you?

Using PL/SQL collection method you can get the information as well as alter the content of the collection.

How many collection methods do we have?

In Oracle Database we have ***3 Collection Procedures*** and ***7 Collection functions***. In total we have ***10 collection methods***. Here are their names —

Collection Functions

114. Count
115. Exists
116. First
117. Last
118. Limit
119. Prior
120. Next

Collection Procedures

121. Delete

122. Extend
123. Trim

Since the syntax for using the collection built-ins is different from the normal syntax used to call procedures and functions therefore they are referred to as methods.

How do we use collection methods?

In Oracle PL/SQL, collection methods can be used using ***Dot (.) notation***. Let’s take a look at the syntax.

Collection. Method (parameters)

That is the quick introduction to collection methods in Oracle Database. In the upcoming videos we will explore in detail each of the above mentioned methods individually.