Chapter-7

Exception Handling:

- <u>No_data_found:</u>
  This exception can be raised is when a SELECT INTO statement does not return any rows. Or a select statement returns 0 rows.
- <u>Too_many_rows:</u>
  Raised when a select statement returns more than one row.
- Login_denied
  Raised when an invalid username\password is logged on to the oracle.
- Program_error
  Raised when PL\SQL has an internal problem.
- Timeout_on_resource
  Raised when oracle has been waiting to access a resource beyond the user defined
  Time out limit.
- Value_error
  Raised when the data type or data size is invalid.
- Others
  Stands for all other exceptions not explicitly named.

Syntax:

```
        Exception
        When <ExceptionName> then
        <User define action to be carried out>
```

Ex:

```
Declare
        Temp varchar2(5);
Begin
        Temp = '&Temp';
        Select pro_no from product_master where pro_no  = Temp;
Exception
        When no_data_found then
                Dbms_output.put_line('Record not found');
End;
```

Procedure / Function

A Procedure or Function is a logically grouped set of SQL and PL/SQL statements that perform a specific task. A **stored Procedure** is a named PL/SQL code block that been compiled and stored in one of the Oracle engine’s system tables.

To make a Procedure or Function either of them can be passed parameters before execution. A Procedure or Function can then change the way it works depending upon the parameters passed prior to is execution.

<u>Procedure or Functions are made up of:</u>

- A declarative part

- An E\executable part
- An optional exception handling part

Declarative Part:

The declarative part may contain the declarations of course, constants, variables exception and subprograms. These object are local to the procedure. The objects become invalid once the procedure exists.

Executable Part:

The executable part is a PL/SQL block consisting of SQL and PL/SQL statements that assign values, control execution and manipulate data. The action that the procedure is expected to perform is coded here. The data that is to be returned back to the calling environment is also return from here. The variables declared are put to use within this block.

Exception Handling:

This part constrains code that deals with exception that may be raised during the execution of code in the executable part. And Oracle exception handler can be redirected to the exception handling section of the procedure or function where the procedure or function determines that actual action that must be carried out by oracle's exception handler.

- **Where do stored Procedures and Functions Reside?**

  Procedure or Function**s** are stored in the Oracle database. They can be invoked or called by any PL/SQL block that appears within an application. Before a Procedure or Function **is** stored, the Oracle engine parses and compiles the Procedure or Function**.**
- **How does the Oracle engine create a stored** Procedure **or** Function**?**
  - Compiles the Procedure or Function**.**
  - Stores the Procedure or Function in the database.

The oracle engine compiles the PL/SQL code block. If an error occurs during the compilation of the Procedure or Function**,** an invalid Procedure or Function gets created. The oracle engine displays a message after creation that the Procedure or Function was created with compilations errors.

How does the oracle engine execute Procedure or Function?

- Verifies user access
- Verifies Procedure or Function validity
- Execute the Procedure or Function**.**

Advantages of using Procedure or Function

<u>Security:</u>

Stored Procedure or Function can help enforce data security. For e.g. by givein permission to a Procedure or Function that can query a table and granting the granting Procedure or Function to users, permissions to manipulate the table itself need not be granted to users.

Performance:

= An amount of information sent over a network is less
= Not compilation step is required to execute the code
= Once the Procedure or Function is present in the shared pool of the SGA retrieval from disk is not required every time different users call the Procedure or Function i.e. reduction In disk I/O.

Memory allocation:

The amount of memory used reduces as stored Procedure or Function has shared memory capabilities. Only one copy of procedure needs to be loaded for execution by multiple users. Once a copy of a Procedure or Function is opened in the oracle engine's memory, other users who have appropriate permission may access it when required.

Productivity:

By writing Procedure or Function redundant coding van be avoided, increasing productivity.

Integrity:

A Procedure or Function needs to be tested only once to guarantee that it returns an accurate result. Since Procedure or Function**s** are stored in the oracle engine they become a part of the engine's resource. Hence the responsibility of maintaining there integrity rests with the oracle engine. The oracle engine has high level of in built security and hence integrity of Procedure or Function can be safely left to the oracle engine.

Procedures V/s Functions

- A function must return a value back to the caller. A function can return only one value to the calling PL/SQL code block.
- By defining multiple OUT parameters in procedure, multiple values can be passed to the caller. The OUT variable being global by nature, its value is accessible by any PL/SQL code block including the PL/SQL block via which it was called.

Syntax:

```
CREATE OR REPLACE PROCEDURE [Schema.] <procedure name>
(<Argument> {IN, OUT, IN OUT} <data type, …>) {IS, AS}
        <Variable declaration>;
        <Constant declaration>;
BEGIN
        <PL/SQL subprogram body>;
EXCEPTION
        <Exception PL/SQL block>;
END;
```

REPLACE	Recreate the procedure if it already exists.
Schema	The schema to contain the procedure. The oracle engine takes the default schema to be the current schema, if it is omitted.
Procedure	The name of the procedure to be created.
Argument	The name of an argument to the procedure. Parentheses can be omitted if no arguments are present.
IN	Indicates that the parameter will accept a value from the user.
OUT	Indicates that the parameter will return a value to the user.
IN OUT	Indicates that the parameter will either accept a value from the user or return a value to the user.
Data type	The data type of an argument. It supports any data type supported by Pl/SQL
PL/SQL subprogram body	The definition of procedure consisting of PL/SQL statement.

DELETING A STORED PROCEDURE OR FUNCTION:

Syntax:

```
        DROP PROCEDURE <Procedure name>;
```

Example:

```
Create or Replace Procedure proc_check (v_emp_code IN varchar2, val OUT
number) AS
        Dummy varchar2 (10);
Begin
        Select emp_code INTO dummy from emp
        Where emp_code=v_emp_code;
        Val: =1;
Exception
        When no_data_found then
        Val: =0;
End;
```

Syntax for creating Function:

```
        CREATE OR REPLACE FUNCITON [Schema.] <function name>
        (<Argument> IN <data type, …>) RETURN <data type> {IS, AS}
        <Variable declaration>;
        <Constant declaration>;
        BEGIN
                <PL/SQL subprogram body>;
        EXCEPTION
                <Exception          PL/SQL   block>;
```

```
	END;
```

Keywords:

REPLACE	Recreate the function if it already exists.
Schema	The schema to contain the function. The oracle engine takes the default schema to be the current schema, if it is omitted.
Function	The name of the function to be created.
Argument	The name of an argument to the function. Parentheses can be omitted if no arguments are present.
IN	Indicates that the parameter will accept a value from the user.
Return Data type	The data type of the function's return value. Because every function must return a value, this clause is required. It supports any data type supported by PL/SQL
PL/SQL subprogram body	The definition of function consisting of PL/SQL statement.

DELETING Function:

Syntax:

```
	DROP FUNCTION <Function name>;
```

Example:

```
	Create or Replace Function checkmf( v_emp_code IN varchar2)
	Return number IS
		Dummy varchar2 (10);
	Begin
		Select emp_code INTO dummy from emp
		Where emp_code=v_emp_code;
		Return 1;
	Exception
		When no_data_found then
		Return 0;
	End;
```

PL/SQL code block

```
 Declare
	E_no employee_master%type;
	Found_val number (1);
 BEGIN
	Found_val := checkmf(e_no);
	If found_val = 1 then
		Dbms_output.put_line('Record found');
```

```
        End if;
        If found_val = 0 then
                Dbms_output.put_line('Record not found');
        End if;
End;
```

PACKAGE:

A package is an oracle object, which holds other objects within it. Objects commonly held within a package are procedure, functions, variables, constants, cursors and exceptions. The tool used to create a package is SQL*PLUS, it is a way of creating generic, encapsulated, re-useable code.

A package once written and debugged is compiled and stored in Oracle's system tables held in an oracle database. All users who have execute permission on the oracle database can then use the package.

Package can contain PL/SQL blocks of code, which have been written to perform some process entirely on their own. These PL/SQL blocks of code do not require any kind of input from other PL/SQL blocks of code. These are the package's standalone subprogram.

Alternatively, a package can contain subprograms that require input from another PL/SQL block to perform its programmed process successfully. These are also subprograms of the package but these subprograms are not standalone. Subprogram held within a package can be called from other stored program, like triggers, recompiles or nay other interactive oracle program like SQL*Plus.

Component of an oracle package:

A package has usually two components, a specification and a body. A package's specification declares the types (variables of the record type), memory variables constants exceptions, cursors, and subprograms that are available for use.

Why use packages?

- Packages enable the organization of commercial applications into efficient modules. Each package is easily understood and the interfaces between packages are simple, clear and well defined.
- Packages allow granting of privileges efficiently.
- A package's public variables and cursors persist for the duration of the session. Therefore all cursors and procedures that execute in this environment can share them.
- Packages enable the overloading of procedures and functions when required.
- Packages improve performance by loading multiple objects into memory at once. Therefore subsequent calls to related subprograms in the package require no I/O.
- Package promotes code reuse through the use of libraries that contain stored procedures and functions, thereby reducing redundant coding.

Package specification:

- ❖ Name of the package
- ❖ Names of the data types of any argument
- ❖ This declaration is local to the database and global to the package.

Invoke a package via the oracle engine:

- ❑ Verify user access:
  Confirms that the user has execute system privilege granted for the subprogram
- ❑ Verify procedure validity:
  Checks with the data dictionary to determine whether the subprogram is valid or not, if the subprogram is invalid, it is automatically recompiled before being executed.

- ❑ Execute:
  The package subprogram is executed.

The syntax for Dot notation:

```
PackageName.Type_Name
PackageName.Object_Name
PackageName.Subprogram_Name
```

In this syntax:

- ➢ PackageName is the name of the declared package.
- ➢ Type_Name is the name of the type that is user defined such as record
- ➢ Object_Name is the name of the constant or variable that is declare by the user
- ➢ Sub_Program is the name of the procedure or function contained in the package body

Example:

Specification part

```
Create or Replace Package emp_add AS
        Procedure addemployee(p_deptno IN department.deptno%Type,
        P_emp_code IN employee.emp_code%Type);
End emp_add;
```

Body part

```
Create or Replace Package Body emp_add as
        Procedure addemployee(p_deptno IN department.deptno%Type,
                P_emp_code IN employee.emp_code%Type)IS
        BEGIN
                Insert into emp_temp(deptno, emp_code)values(P_deptno,
        p_emp_code);
        commit;
        end addemployee;
        end emp_add;
```

pl\sql code block:
```
        BEGIN
                Emp_add.addemployee('D1','E101');
        END;
```

Trigger:

Database triggers are database objects created via the SQL*Plus tools on the client and strode on the server in the oracle engine's system table. These database objects consist of the following distinct section:

- A named database event
- A PL/SQL block that will execute when the event occurs

Introduction:

The Oracle engine allows the definition of procedures that are implicitly executed, when an insert, delete or update is issued against a table from SQL*Plus or through an application. These procedures are called database triggers. The major issues that make these triggers standalone are those, they are fired implicitly by the oracle engine itself and not explicitly i.e. called by the user.

Used of database Trigger:

Since the oracle engine supports database triggers it provides a highly customizable database management system. Some of the uses to which the database triggers be put to customize management information by the oracle engine are as follows:

- A trigger can permit DML statements against a table only if they are issued, during regular business hours or on predetermined weekdays
- A trigger can also be used to keep an audit trail of a table along with the operation performed and the time on which the operation was performed.
- It can be used to prevent invalid transactions.
- Enforce complex security authorization

Triggers V/s Procedure

There are very few differences between database triggers and procedures. Triggers do no accept parameters where procedure can. The oracle engine itself upon modification of an associated table or its data executes a trigger implicitly. To execute a procedure it has to be explicitly called by a user.

How to apply database trigger:

- A triggering event or statement
- A trigger restriction
- A trigger action

Triggering Event or Statement:

It is a SQL statement that causes a trigger to be fired. It can be Insert, Delete, and Update statement for a specific table.

Trigger Restriction:

A trigger restriction specifies a Boolean expression that must be true for the trigger to fire. It is an option available for triggers that are fired for each row. Its function is to conditionally control the execution of a trigger. A trigger restriction is specified using a WHEN clause.

Trigger Action:

A trigger action is the PL/SQL code to be executed when a triggering statement is encountered and any trigger restriction evaluates to true. The PL/SQL block can contain SQL and PL/SQL statements, can define PL/SQL language constructs and can call stored procedures. Additionally for row triggers, the statement the PL/SQL block has access to column values (: new and :old) of the current row being processed.

Types of Trigger:

While defining a trigger the number of times the trigger action is to be executed van be specified. This can be once for every row affected by the triggering statement, or once for the triggering statement no matter how many rows it affects.

Row Triggers:

A row triggers is fired each time a row in the table is affected by the triggering statement. For example if an UPDATE statement updates multiple rows of a table a row trigger is fired once for each row affected by UPDATE statement.. if the triggering statement affects no rows, the trigger is not executed at all. Tow triggers should be used when some processing is required whenever a triggering statement affects a single tow in table.

Statement Trigger:

A statement trigger is fired once on behalf of the triggering statement independent of the number of rows the triggering statement affects. Statement triggers should be used when a triggering statement affects rows in a table but the processing required is completely independent of the number of rows affected.

Before V/s After Triggers:

When defining a trigger it is necessary to specify the trigger timing i.e. specifying when the triggering action is to be executed in relation to the triggering statement. BEFORE and AFTER apply to both row and the statement triggers.

Before triggers:

BEFORE triggers execute the trigger action before the triggering statement.

- BEFORE: triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using a BEFORE trigger user can eliminate unnecessary processing if the triggering statement.
- If a BEFORE trigger are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

AFTER triggers:

- AFTER trigger executes the trigger action after the triggering statement is executed. These types of triggers are commonly used in the following saturations:
- AFTER triggers are used when the triggering statement should complete before executing the trigger action
- If a BEFORE trigger is already present an AFTER trigger can perform different actions on the same triggering statement.

Combinations triggers:

Using the option explained above four types of triggers could be created.

BEFORE statement trigger:

Before executing the triggering statement the trigger action is executed.

BEFORE row trigger:

Before modifying each row affected by the triggering statement and before applying appropriate integrity constraints the trigger is executed.

AFTER Statement Trigger:

After executing the triggering statement and applying and deferred integrity constraints the trigger action is executed.

AFTER Row Trigger:

After modifying each row affected by the triggering statement and applying appropriate integrity constraints the trigger action is executed for the current row. Unlike BEFORE row triggers. AFTER row trigger have rows locked.

Syntax:

```
Create or Replace Trigger [Schema.] <TriggerName>
{Before, After}
{Delete, Insert, Update [OF column, …]}
ON [Schema.] <Table Name>
[REFERENCES {OLD as old, NEW as new}]
[FOR EACH ROW [WHEN condition]]
```

Example:

```
DECLARE
        <Variable declaration>;
        <Constant declaration>;
BEGIN
        <PL/SQL subprogram body>;
EXCEPTION
        <Exception PL/SQL block>;
END;
```

OR REPLACE	Returns the trigger if it already exists.
Schema	The schema, which contains the trigger. If the schema is omitted, the oracle engine creates the trigger in the users own schema.
Trigger Name	The name of the trigger to be created.
BEFORE	Indicates that the oracle engine fires the trigger before executing the triggering statement.
AFTER	Indicates that the oracle engine fires the trigger after executing the triggering statement.
DELETE	Indicates that the oracle engine fires the trigger whenever a DELETE statement removes a row from the table.
INSERT	Indicates that the oracle engine fires the trigger whenever an INSERT statement add row from the table.
UPDATE	Indicates that the oracle engine fires the trigger whenever an INSERT statement change a value in one of the columns specified in the OF clause. If the OF clause is omitted, the oracle engine fires the trigger whenever an UPDATE statement changes a value in any column of the table.
ON	Specifies that schema and name of the table, which the trigger is to be created.
REFERENCING	Specifies correlation names. Correlation names can be used in the PL/SQL block and WHEN clause of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If the row trigger is associated with a table named OLD or NEW, its can be used to specify different correlation names to avoid confusion between table name and the correlation name.
FOR EACH ROW	Designates the trigger to be a row trigger. The oracle engine fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the WHEN clause. If this clause is omitted the trigger is a statement trigger.
WHEN	Specifies the trigger restriction. The trigger restriction constraint a SQL condition that must be satisfied for the oracle engine to fire the trigger. This condition must contain correlation names and cannot contain a query.
PL/SQL block	The PL/SQL block that the oracle engine executes when the trigger is fired.

Deleting A Trigger:

Syntax: DROP Trigger <Trigger Name>;

Example [1]:

```
Create Or Replace Trigger tri_after
AFTER Insert ON Employee
BEGIN
        Insert into status Values (:new.empno, :new.ename, ’After Trigger Fire’);
End;
```

Example [2]:

```
Create Or Replace Trigger protrg
BEFORE Insert ON Employee
BEGIN
        Insert into status Values (:new.empno, :new.ename, ’Before Trigger Fire’);
End;
```

USING TRIGGER PRADICATES:

There are three Boolean functions that you can user to determine wheat the operation is. These predicates are INSERTING, UPDATING and DELETING. Their behavior is described in the following table.

INSERTING	TRUE if the triggering statement is an INSERT; FALSE otherwise
UPDATINNG	TRUE if the triggering statement is an UPDATE; FALSE otherwise
DELETING	TRUE if the triggering statement is DELETE; FALSE otherwise

Using :old and :new in row-level Trigger:

A row level trigger fires once per row processed by the triggering statement. Inside the trigger, you can access the row that is currently being processed. This is accomplished through two pseudo-records :old and: new. :OLD and: NEW are not true records. Although syntactically they are treated as records. Thus they are known as pseudo-records.

Example [3]:

```
Create Or Replace Trigger tri_operation
BEFORE Insert or Delete or Update ON Employee
DECLARE
        Change varchar2 (1);
BEGIN
        IF inserting THEN
                Change: =’I’;
        ELSE IF Deleting THEN
```

```
                Change: ='D'
        ELSE
                Change: ='U'
        End if;
        Insert into status (emp_no, emp_name, sal, newsal, operation,
modify_date)
        Values (:old.empno, :old.ename, :old.salary,
:new.salary,change,sysdate);
End;
```

<u>Show trigger list:</u>

```
Select trigger_type, Table_name, triggering_event FROM user_triggers;
```

<u>**Composite data type:**</u>

NESTED TABLE:

A nested table is table within a table. A nested table is a collection of rows, represented as a column within the main table. For each record within the main table the nested table may contain multiple rows. In one sense, it's a way of storing a one-to-many relationship within one table.

Consider a table that contained information about department in which each department may have many employees. In a strictly relational model, two separate tables would be created: [1] Department [2] Employee.

Nested tables allow storing the information about employee within the department table. The employee table record can be accessed directly via the department table. Without the need to perform a join, the ability to select data without traversing joins makes data access easier. Even if methods for accessing nested data are not defined. Department and employee data have clearly been associated.

<u>**Example 1:**</u>

```
[1] Step:
        Create or Replace type CourseList AS Table of Varchar2(64);
[2] Step:
        Create table Department (Name Varchar2(20), Director Varchar2(20),
                Office Varchar2(20), Course CourseList)
        Nested Table Course STORE AS course_tab;
[3] Step: Insert Record
        INSERT INTO Department
        VALUES('Compuer','DPC','GKCK',CourseList
('BCA','PGDCA'));
```

<u>**Example 2:**</u>

```
Declare
```

```
        Type nested_t1 is table of varchar2(5);
        V1 nested_t1 := nested_t1(1,2,3);
    Begin
        V1(1) := ‘a’;
        V1(2) := ‘b’;
        V1(3) := ‘c’;
        Dbms_output.put_line(v1.count);
        Dbms_output.put_line(v1.first);
        Dbms_output.put_line(v1.last);
        Dbms_output.put_line(v1.next(2));
        If (v1.exist (2)) then
            Dbms_output.put_line(‘exist’);
        else
            Dbms_output.put_line(‘not exist’);
        End if;
        V1.delete(3);
        Dbms_output.put_line(‘value 3 is deleted’);
        V1.extend(4,5);
        V1(4) := ‘e’;
        Dbms_output.put_line(v1(4));
        V1.trim(2);
```

VARYING ARRAYS:

A varray is a data type very similar to an array in C or Pascal. Syntactically, varray is accessed in much the same way as a nested or index0by table. However a varray is implemented differently. Rather then begin a sparse data structure with no upper bound, element are inserted into a varray starting at index 1 up to the maximum length declared n the varray type. Varying arrays, also known as VARRAYS, allow storing repeating attributes in tables.

Declaring a Varray:

```
        TYPE type_name IS {VARRAY|VARYING ARRAY}
    (maximum_size) OF element_type [NOT NULL];
```

Where type_name is the name of the new varray type, maximum_size is an integer specifying the maximum number of elements in the varray, and element_type is a PL/SQL scalar, record or object type. The element_type can be specified using %TYPE as will but cannot be BOOLIAN, NCHAR, NCLOB, NVARCHAR2, REF CURSOR, TABLE or other VARRAY type.

Example:

[1] Step:

```
        CREATE OR REPLACE TYPE Marks_va AS VARRAY(5) of
    number(5);
    [2] Step
        CREATE TABLE Student (Std_no number(5) PRIMARY KEY,
        Name Varchar2 (15), Marks Marks_va);
    [3] Describe Student;
        Show the structure of above table.
    [4] Step:
        INSERT INTO Student
        VALUES (1001,'Divyesh',Marks_va(78,75,77,86,85));
```

Query:

```
    Select type_code, attributes from user_types
    Where type_name ='Marks_va';
```

<u>Example 2:</u>

```
    Declare
        Type vr1 is varray(30) of number(3) not null;
        V1 vr1 := vr1(1,2,3);
    Begin
        V1(1) := 1;
        V1(2) := 2;
        V1(3) := 3;
        Dbms_output.put_line(v1.count);
        Dbms_output.put_line(v1.first);
        Dbms_output.put_line(v1.last);
        Dbms_output.put_line(v1.next(2));
        If (v1.exist (2)) then
            Dbms_output.put_line('exist');
        else
            Dbms_output.put_line('not exist');
        End if;
        V1.delete(3);
        Dbms_output.put_line('value 3 is deleted');
        V1.extend(4,5);
        V1(4) := 'e';
        Dbms_output.put_line(v1(4));
        V1.trim(2);
        Dbms_output.put_line(v1.limit);
```

Varray Vs. Nested Tables:

<u>Similarities</u>

- Both types allow access to individual elements using subscript notation.
- Both types can be stored in database tables.

Difference

- Varray have a maximum size, while nested tables do not.
- Varray are stored inline with the containing table while nested tables are stored in a separate table, which can have different storage characteristics.
- When sorted in the database varray retain the ordering and subscript values for the elements, while nested tables do not.
- Individual elements can be deleted from a nested table, which cause the size of the table to shrink. A varray is always a constant size however.

RECORDS:

A records declaration declares a new type that can then be used to declare variables of
that type. The individual components of the record are fields and each ahs its own data type. That data type can either be one of the standard PL/SQL data types(including another record but not a table) or it can be a reference to the type of a particular column in a specific table. Each field also may have a NOT NULL qualifier that specifies that the field must always have a not null value. You can refer to the individual fields in a record using dot notation.

Syntax:

```
RECORD record_type IS RECORD(   Field1 type1 [NOT NULL] [:=expr1]
Field2 type2 [NOT NULL] [:=expr2],FieldN typeN [NOT NULL] [:=exprN]
```

Here record_type is the name of the new type. Filed1 through fieldN are type names of the fields within the record, and type1 through typeN are the types of the associated fileds. A record can have as many fields as desired. Each field declaration looks essentially the same as a variable declaration outside a record, including NOT NULL constraints and initial values. Expr1 declaration outside a record, the initial value and NOT NULL constraint are optional.

Example:

```
TYPE t_studentrecord IS RECORD(
ID student.ID%type,
Name student.Name%type
Subject        student.subject%type);
V_student t_studentrecord;
BEGIN
        SELECT id, name, subject INTO V_student FROM student
        WHERE id=1001;
        DBMS_OUTPUT.PUT_LINE      (V_student.id      ||      '      '||
        V_student.name || ' '|| V_student.subject );
End;
```

PL/SQL tables (Index by table)

Pl/sql tables works like an arrary in c.

Syntax:

```
Type type_name is table of data type
        Index by binary_integer;
```

Example:

```
Declare
        Type ind_t1 is table of varchar2(5) index by binary_integer;
        V1 ind_t1;
Begin
        V1(1) := ‘hello’;
        V1(2) := ‘how’;
        V1(3) := ‘are u’;

        Dbms_output.put_line(v1.count);
        Dbms_output.put_line(v1.first);
        Dbms_output.put_line(v1.last);
        Dbms_output.put_line(v1.next(2));
        If (v1.exist (2)) then
                Dbms_output.put_line(‘exist’);
        else
                Dbms_output.put_line(‘not exist’);
        End if;
        V1.delete(3);
        Dbms_output.put_line(‘value 3 is deleted’);
```

Difference between pl/sql table and nested table:

- Nested table are stored in database system table while index by table are not.
- Nested table supports trim and extend method while pl/sql table do not.
- It supports both negative and positive subscript value while nested table only positive subscript value.
- Through create type we can use nested table in sql prompt while pl/sql tables can only used in pl/sql.(syntax difference.)
- In nested table give sequential index value while in pl/sql table random value.

Objects:

```
Create [or replace] type [schema.]type_nm as object
(Attribute nm datatype…..);
```

Example:

```
Create or replace type ob1 as object (id number (5), nm varchar2 (10));
Create  table  temp  (no  varchar2   (5), v1 ob1);
```

Insert into temp values ('12', ob1 (11,'abc'));

- It's a DDL statement so cannot be used in pl/sql code block.
- Its field cannot be constrained with not null.
- Not initialized with default value.
- Like record one cannot refer to the attributes using dot notation.

Formatting Command of SQL PLUS:

The basic SQL PLUS commands are bellow:

REM (Remark):

Which stand for remark. SQL PLUS ignores anything's on a line that begins with their letters, thus allowing you to add comments, and explanations to any start file you create.

Example : Rem name : Sales Register

T TITLE:

Ttitle puts a title at the top of each page of a report. OFF and ON suppress and restore the display of the text without changing its contents. Title by itself displays the current titles options and test or variable. SQL PLUS uses ttitle in the new form if the first word after ttitle is valid option. The valid options are:
LE[FT], CE[NTER] and R[IGHT] left-justify, center and right justify data on the current line. Any text or variables follwong these commands are justified as a group up to the end of the command or a left center right or column.
Format specifies the format model that will control the format of subsequent text or variables, and follows the same syntax as format in a COLUMN command, such as FORMAT A12 or FORMAT $999,990.99.
Date value is printed according to the default format unless a variable has been loaded with a date reformatted by to_char.

Example:

Ttitle center 'This is SALES REPORT'

BTITLE:

Btitle puts a title at the bottom of each page of a report. OFF and ON suppress and restore the display of the text without changing its contents. Title by itself displays the current btitles options and test or variable. SQL PLUS uses btitle in the new form if the first word after btitle is valid option. The valid options are:
LE[FT], CE[NTER] and R[IGHT] left-justify, center and right justify data on the current line. Any text or variables follwong these commands are justified as a group up to the end of the command or a left center right or column.
FORMAT string specifies the format model that will control the format of subsequent text or variables, and follows the same syntax as format in a COLUMN command, such as FORMAT A12 or FORMAT $999,990.99.
Date value is printed according to the default format unless a variable has been loaded with a date reformatted by to_char.

Example:

btitle center 'SALES   REPORT.sql'

REPHEADER:

Repheader puts a header on the first page of a report. OFF and ON suppress and resoter the display of the test without changing its contents. Repheader by itself displays the current repheader option and text or variable.
LE[FT], CE[NTER] and R[IGHT] left-justify, center and right justify data on the current line. Any text or variables follwong these commands are justified as a group up to the end of the command or a left center right or column.
FORMAT string specifies the format model that will control the format of subsequent text or variables, and follows the same syntax as format in a COLUMN command, such as FORMAT A12 or FORMAT $999,990.99.
Example:

```
 repheader center 'This is SALES REPORT'
```

REPFOOTER:

Repheader puts footer on the last page of a report. OFF and ON suppress and resoter the display of the test without changing its contents. Repfooter by itself displays the current repfooter option and text or variable.
LE[FT], CE[NTER] and R[IGHT] left-justify, center and right justify data on the current line. Any text or variables follwong these commands are justified as a group up to the end of the command or a left center right or column.
FORMAT string specifies the format model that will control the format of subsequent text or variables, and follows the same syntax as format in a COLUMN command, such as FORMAT A12 or FORMAT $999,990.99.

Example:

```
 repfooter center 'This is SALES REPORT'
```

SET HEADING:

The set heading on/off command is used to turn off/on the column titles that normally would appear select command.

BREAK ON:

A break occurs when SQL PLUS detects a specified change such as the end of a page or a change in the value of an expression. A break will cause SQL PLUS to perform some action you have specified in the BREAK command, such as SKIP and to print some result from COMPUTE command, such as averages or totals for a column. Only one break command may be in effect at a time. We used to turn on/off the break command.
Example:

```
Break on JOB Skip 2
  Select *from Employee
  Order by JOB;
```

COLUMN:

Column allows you to change the heading and format of any column in a select statement.
Example:

Column Emp_nm Heading 'Employee'

Re labels the column and gives it a new heading.

COMPUTE SUM:

The totals calculated for each section on the report were produced by the compute sum command. This command always works in conjunction with the **break on** command.

Example:

Break on JOB skip 2

Compute sum of SAL on JOB

You can use a break on command without a compute sum command, such as for organizing your report into sections where no totals are needed.

SET LINESIZE:

The command set linesize governs the maximum number of characters that will appear on a single line. If you put more columns of information in SQL query than will fit into the linesize you have allotted. SQLPLUS also uses linesize to determine where to center the ttitle and where to place the date and page number.

Example:

Set Linesize 18

SET PAGESIZE:

The set pagesize command sets the total number of lines SQLPLUS will place on each page, including the ttietle, btitle, column headings and any blank line it prints. Set pagesize is coordinate with set newpage.

Example:

Set pagesize 66

SAVE:

If the changes you wish to make to your SQL statement are extensive or you simply wish to work in your own editor save the SQL you have created so far in interactive mode, by writing the SQL to a file SQL to a file, like this:

Example : SQL > save emp.sql

EDITING:

If the changes you wish to make in your sql file then give Ed (Edit) command. This command is use from SQL prompt. In SQL were editing with notepad from window operating system.

Example:

Ed emp.sql

START :

One backs in SQLPLUS test your editing work by executing the file you have just edited:

Example:

Start emp.sql

SHOW:

Looking at command settings that follow the set command requires using the word SHOW:

Example:
Show linesize
Linesize 79