16-04-24

PL/SQL → Report generation
Advantage: Readability in a formatted way.

→ Structured Language should satisfy Sequential, Conditional & Iterative.

→ Sections in PLSQL: 1) Declare - Optional
2) Begin -- end; - Mandatory
3) Exception - Optional

→ Every executable statement should be terminated by ;

→ Character Set:
- Alphabets A-Z
- Number 0-9
- Special char:
  - Arithmetic + - * / mod
  - Logical AND OR NOT NULL
  - Relational < > =
- -- & hyphens for comment lines.
- Supports OOPS, modularization

→ Block in PLSQL:

```
Begin
    Statements
end;
```

→ Procedure: 1) Write the program in any text editor.
2) Save the file with filename.sql
3) Set the servers circuit output on
SQL> set serveroutput on (once)
4) Compile and run
@filename. → total path

→ Example program: add.sql

```
--Addition of 2 number
declare
    a number(3) := 10;
    b number(3) := 20;
    c number(3);
begin
    c := a+b;
    dbms_output.put_line(c);      (c is a string here)
end;
```

{ For dynamic input
:= &a; }

Conditional controls in PLSQL:

1) If — (4) types
2) case - (2) types
3) goto (Unconditional branching or jump) - Minimize it's use

- General Syntax: (Simple if)

```
If booleancondition then
   -- Sequence of statements
end if;
```

→ If multiple statements then use begin and end

- General syntax: (if else)

```
If booleancondition then
   --true sequence of executable statements
Else
   --false sequence of executable statements
end if;
```

- General syntax: (if elsif)

```
if cond1 then
    result1
elsif cond2 then
    result2
- - - -
else
    default_statement
end if;
```

1) Write a PLSQL pgm to find given num is +ve, -ve or 0.

```
declare
   n number(3) := &n;
begin
   if n>0 then
      dbms_output.put_line('Positive number');
   Elsif n<0 then
      dbms_output.put_line ('Neg Num');
   Else
      dbms_output.put_line ('Zero');
end if;
end;
```

② Case statements in PLSQL:

1) Simple case - Evaluates for a value.

2) Search case - Truly just like if else

→ Syntax for case Simple case

```
Case condition/expression
when val1 then
        -- block of statements
when val2 then
        -- block of statements
Else
        -- default statement
End case;
```

→ Syntax for Searched case:

```
case
when expr1 then        check
    --statements1      ) if wrong
when expr2 then        check
    statements2        )
Else
    defaultstatement
End case;
```

③ GoTo:

→ Syntax:

```
GoTo labelname;
<<labelname>>
```

→ There will be forward goto and backward goto
↳ chance of infinite goto (be careful).

* Iterative Processing

1) Simple loop
2) while loop
3) Numeric loop

① Syntax for Simple loop:

```
loop
    executable_statements
end loop;
```

Terminations of loop

```
Exit;
Exit when condition;
```

② Syntax for while loop:

```
while condition
loop
    executable_statements
end loop;
```

③ Syntax for numeric / for loop:

```
for control_variable in initial..finalvalue    // forward loop
loop
    Executable_statements
end loop;
```

initial ≤ finalv (always)

```
for control_variable In Reverse initialval..finalvalue  // backward loop
loop
    Executable_statements
end loop;
```

Q)

1. Find given number is 5 multiplier or not
2. Find x power n
3. Gcd of a, b
4. Display multiplication table for given n.
5. Given num is prime / perfect / armstrong
6. Display emp table
7. Display emp table whose sal is b/w given a and b.
8. Display boats reserved by given sid.
10. Display sailors who are having the given rating
11. " " " " reserved by the given color boats.

1) To find the sum of digits for a given number.

```
declare
        n number(10) := &n;
        rem number;
        res number(15) := 0;
begin
        while n>0
        loop
            rem := n mod 10;
            res := res + rem
            n := trunc(n/10);
        end loop;
        dbms_output.put_line(res);
end;
/
```

2) Find whether the given number is palindrome or not.

```
declare
        n number(15) := &n;
        rem number(10);
        res number(15) := 0;
        temp number(15) := n;
begin
        while n>0 loop
            rem := n mod 10;
            res := (res * 10) + rem;
            n := float(n/10);
        end loop;
        if res = temp then
            dbms_output.put_line('Palindrome');
        else
            dbms_output.put_line('Not palindrome');
        end if;
end;
```

* Modular programming in PLSQL Modularization: completing by dividing & performing as tasks.

1) Procedure - Single executable statements.

→ A procedure is a module that perform one / more actions.

→ A procedure call is a standalone executable statement.

→ General syntax:

```
Create/Replace                                  ← formal parameters
Create or Replace Procedure Procname (parameter_list)
IS
    local variables declaration.
begin
    Executable statements
end;
```

→ Calling a procedure: Procedure_name(param_list);

2) Function - dependent subprogram

→ A function is a module that returns data through it's return clause.

→ A function can exist only as a part of an executable statement.

→ General syntax:

```
Create or Replace Function functionname(parameters_list)
RETURN datatype
IS
    -local variable declaration
begin
    -executable statements
end;
```

→ Calling a function by assigning a variable/expression/statement.

```
var := function_name (parameter_list);
```

→ Can be used as expression.

3) Parameter Modes

→ Parameter Modes defines the way in which they can be used.

1) IN

Description: Read-only

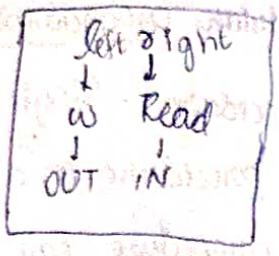
- default mode is IN
- Simply its a call by value.

we can use, we cannot modify.
Simply RHS side.

2) OUT

Description: Write-only mode

- Able to modify but we cannot use
- We can use it for assignment of operation.



3) IN OUT

Description: Read/Write (both)

- The module can both reference (read) & modify (write) the parameters.

Example:

```
Create or replace Procedure add (a Integer, b Integer, c OUT Integer)
IS                                          ↓
begin                                   by default IN
    c:=a+b;
end;
--main
declare
      a Integer:=&a;
      b Integer:=&b;
      c Integer;
begin add(a,b,c);
      dbms.... (a||b||c);
/end
```

2) Write a procedure to swap 2 numbers.

```
Create or Replace Procedure Swap (X In out Integer, Y In out Integer)
Is
    Temp Integer;
begin
    Temp:=X;
    X:=Y;
    Y:= Temp;
End;
/
--main.
declare
    x Integer := &x;
    y Integer := &y;
```

```
begin
    dbms_output.put_line(x || y);
    exchange(x, y);
    dbms_output.put_line(x || y);
end;
/
```

Q) Write a procedure 2 numbers.

```
Create or Replace Procedure addnum(a IN Integer, b Integer,
                                   c OUT Integer)
IS
begin
    c:=a+b;
```

Q) Factorial of a given number using procedure.

```
Create or Replace Procedure factorial(n IN Integer, a OUT Integer)
IS
begin
    for i in 1..n
    loop
    a:=a*i;
    end loop;
end;
--main
declare
    n Integer:=8;
    a Integer:=1;
begin
    factorial(n, a);
    dbms_output.put_line('Factorial =' || a);
end;
```

* Assigning SQL (DML) Query results into PL/SQL variables

→ For 'Select' we need to use 'INTO'. For that we need to know.

1. The table must exist in the database.
2. We need to know the column names (Col names must be known)
3. Need to know the usage of PL/SQL variables i.e how to declared write.

Q) Disphy deptno for given empno.

PLSQL ← report generation
must be in readable form

```
declare
    empid emp.empno%type;        (column)
    dno   emp.deptno%type;
begin
    Select deptno into dno from emp where empno = &empid;
    dbms_output.put_line('emp' || empid || ' is in department=' || dno);
```

x) Select deptno from emp where empno = 7934.

DML → directly

Deptno
10

} No scadability.

PLSQL → O/p : Emp is in Department = 10 (✓)

* Procedures and Functions:

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called as 'Modular design'. A subprgm can be invoked by another subprgm or prgm which is called as the calling program.

→ A subprogram can be created at:

1) • at schema level
2) • Inside a package
3) • Inside a PLSQL block.

1) A schema level subprogram is a standalone subprogram.
Created with the CREATE PROCEDURE or CREATE FUNCTION statement.
It is stored in the database and can be deleted with DROP PROCEDURE or DROP FUNCTION statement.

2) A subprogram inside a package is a packaged subprogram.
It is stored in the database and can be deleted only when package is deleted with DROP package statement.

→ PLSQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.

PL/SQL provides 2 kinds of subprograms

- Functions: Return a single value, mainly used to compute and return a value.
- Procedure: Do not return a value directly, mainly used to perform an action.

→ Parts and Description:

1. Declarative part - optional part
2. Executable " - mandatory part
3. Exception handling - optional.

→ Difference:

Function is a subpart of executable statement

Procedure is a standalone " "

[(Next Cursors) P.T.O]

Questions on Functions

1) Create a function to add 2 numbers.

```
Create Or Replace Function Add(a In Integer, b In Integer) Return Integer
Is
    c Integer;
begin
    c:=a+b;
    return c;
end;
Declare
    a Integer := &a;
    b Integer := &b;
    c Integer;
begin
    c:= Add(a,b);
    dbms_output.put_line(c);
End;
/
```

② Factorial.

* Cursors

24-04-24

→ Cursor is a location in the memory where the SQL statement is processed. ie whenever we send an SQL statements the system allocates some memory by giving a name. After processesing the statement it automatically cleans the memo

→ There are 4 steps, which are processed by system whenever we send an SQL statement they are:

1. Creating cursor : Allocates memory
2. Opening cursor: Process the sql statement
3. Fetching cursor: Values satisfied by sql statement are fetched from table into cursor row by row
4. Closing cursor: This statement close the memory allocated for cursor.

⇒ Types of cursors : 2 types

1) Implicit cursor or Simple cursors

→ Oracle implicity opens a cursor to process each sql statement not associated with explicity declared cursor. PL/sql lets us refer to the most recent implicit cursors as the sql cursor.

→ Implicit cursors have 4 attributes:

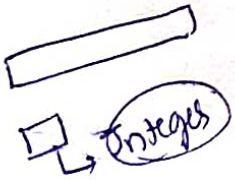
- sql%rowcount - Returns the no:- of rows affected by an insert, update or delete.
- sql%found - This attributes evaluates to true, if an insert or update or delete affected to one or more rows. It evaluates to false if no row is affected.
- sql%notfound - This attribute is logical opposite to sql%found. It returns true if an insert or update or delete does not affect to any row. If returns false if any one row is affected.
- sql%isopen - Checks whether the cursor has opened or not.

Ex1: Write a pgm to check whether there is atleast one row satisfying the given select statement or not using implicit cursors.

Declare

e emp%rowtype;

eno emp.empno%type;



```
Begin
    select * into c from Emp where Empno = &eno;
    If sql%found then
        dbms_output.( 'Record found');
    end if;
End;
/
```

Ex2: To display 1 to 5 records from Emp table.

2) Explicit cursors

→ The user creates these cursors for applying 2 or more DML operations on a table. Using a cursor we can fetch all the row values from the table into a variable for printing them or performing some action with them.

→ Uses 4 steps (Creating to closing)

→ Declaring a Cursor: A cursor is defined in declarative part by naming it and associating it with a query.

```
Declare
    Cursor cursor_name is <select statement>
```

→ Opening a cursor: The cursor must be initialized or opened with open statement
  └ Executes query and identifies the result set which consists of all rows that meets the query search criteria.

```
Begin
    Open cursor_name
End;
```

→ Fetching with a cursor: Used to retrieve the cursors current row. Fetching can be

executed repeatedly until all rows have been retrieved.

```
Begin
    Open <cursor_name>
    Fetch <cursor_name> into <var1>, <var2>, ...
    ---
End;
```

→ Closing a cursor: When the last row has been processed, the cursor is closed with the close statement which disables the cursor.

```
Begin
    Open --
    Fetch--
    Close <cursor-name>
End;
```

Ex1) Using an explicit cursor, Display all rows from Emp table.

```
Declare
    Cursor c1 is Select * from Emp; -- creating
        e Emp%rowtype;
Begin
    Open c1;
    Loop      --Simple loop
        Fetch c1 into e;
        Dbms_output.put_line (e.Empno || 'is' || e.Ename _ ||e.Sal);
        Exit when c1%notfound;
    End loop;
    Close c1;
End;
```

Ex2) Display Emp table whose salary is b/w 1000 and 5000 using explicit cursor.

```
Declare
    Cursor c1 is Select * from Emp where Sal between 1000 and 5000;
    e Emp%rowtype;
Begin
    open c1;
    loop
        Fetch c1 into e;
        dbms_output.put_line (e.Ename || 'has a salary of' || e.Sal);
        Exit when c1%notfound;
    End loop;
End; Close c1;
```

* Exception Handling

Types of Exception - 2 types 1) Pre-Defined
2) User Defined.

1) Pre Defined Exceptions -

→ Used to handle some logical errors known to the system are pre-defined

→ Important pre-defined exception:

① no_data_found

This exception raises when there is no rows to be retrieved from a table according to given condition.

② dup_val_on_index

Raises when ever you try to store duplicate values into a table, which has been indexed (unique indexed)

③ cur_already_open

Raises whenever you program attempts to open an already opened cursor. A cursor must be closed before its can be reopened. A cursor 'for' loop automatically opens the cursor to which it refers. So your pgm can't open that cursor inside the loop.

④ invalid_cursor

Raises whenever your pgm attempts an illegal cursor operation, such as closing an unopened cursor.

⑤ zero_divide

Raises whenever your pgm attempts to divide a numba by zero.

⑥ program_error

Raises whenever PL/SQL internal problem

⑦ storage_error

Raises whenever PL/SQL runs out of memory.

⑧ too_many_rows

Raises whenever a select statement returns more than one row.

⑨ login_denied

Raises when ever your prgm attempts to login to oracle with an invalid username and/or password.

⑩ value_error

Raises whenever an arithmetic conversion or size constraint error occurs. For ex: when we selects a column value into a variable. If the value is longer than declared length of variable pl/sql aborts the assignment and raise the exception value_error.

→ Syntax:

```
begin
---
Exception
    When pre_defined_exception_name then
        Handling_code
    when predefined_exception_name then
        Handling_code
---
end.
```

→ Programs

1) Write a prgm for handling an error when we try to open an already opened cursor.

```
Declare
    Cursor c1 is Select * from Emp;
    x Emp%rowtype;
Begin
    open c1;
    fetch c1 into x;
    dbms_output---(x.Empno || '_' || x.Ename);
    open c1;
    close c1;
    Exception
        when cursor_already_open then
```

dbms_output.put_line ('Sorry friend, cursor already opened');

end;
/

2). WAP for handling datatype of a variable by exceeding to store a value declare.

```
Declare
    name varchar(3);
Begin
    name := 'Anitha';
    Exception
    when value_error then
        dbms_output.put_line ('Storage exceeded');
End;
/
```

3) WAP to display a no data found in the table.

```
Declare
    e Emp%rowtype;
Begin
    Select * into e from Emp where Empno=&empno;
    dbms_out.. ('Empno Ename Job');
    dbms_out--(e.Empno||' '||e.Ename||' '||e.Job);
    Exception
    when no_data_found then
        dbms_output.put_line('Sorry! No data found);
End;
/
```

2) User-Defined Exceptions:

A user can define / create exceptions, which must be raised automatically, because the system does not know

1) Creating Exception
2) Raising "
3) Handling "

→ Syntax:

```
Declare
    <ExceptionName> Exception;   --creating
begin
    Raise Exception;   --Raising   (name)
    Exception                      -- handling.
        when <exception_name> then  --handling
        Message
End;
/
```

→ To generate an error message in a pre-defined error format, we use **

raise_Application_Error

Syntax:

Raise_Application_Error (Error number, 'Message');

Example:

Raise_Application_Error (-20173, 'Some error');

#Note: Error number for (user defined Exception) must be in b/w -20001 to -20999.

→ Programs:

1) WAP for creating an exception & raising it when_ever you try to insert any -ve number into a table.

```
Declare
    Invalid_number exception;
    e' Employee.empno%type := &empno;
Begin
    If e<0 then
        raise invalid_number;
    else
        insert into Employee(empno) values(e);
        dbms_output.putline('Record Inserted..');
    end if;
    Exception
        when invalid_number then
```

dbms_output.put_line ('Cannot insert -ve number');

End;

/

* Triggers

→ Trigger is an event handling mechanism, a self contained block of statements, a firing program.

→ Triggers are stored programs, which are automatically executed or fired when some events occur.

→ Triggers are written to be executed in response to any of events such as DML, DDL, a database operation.

→ Triggers can be written for following purpose:

- Generating some derived column values automically.
- Enforcing referential integrity - Ex: deleting rec in master table
- Event logging and storing information on table access - Ex: Sharing tables
- Auditing ie maintaining data such as permissions, how many accessed etc
- Synchronous replication of tables
- Imposing security authorization
- ~~Security~~ Preventing invalid transactions

→ Triggers Syntax:

```
Create or Replace Trigger Trigger_name
{ Before | After | Instead of }
{ Insert [OR] UPDATE [OR] Delete }  --any combination
[OF col_name]
ON table_name
[Referencing Old as o New as n]
[For Each Row]
When (condition)
Declare
---
Begin --
Exception
End
```

Ex: To restrict employee only on working days.

```
Create or Replace Trigger secure_emp
Before Update or Insert or Delete on employee
Begin
    If (to_char(sysdate, 'Dy') in ('Sun')) THEN
        Raise_Application_Error(-20500, 'You may insert into
                                employee table only on working day');
    End if;
End;
/
```