**PL/SQL**

**PL/SQL stands for procedural structured query Language and It is an extension to SQL**

**Difference between SQL and PL/SQL**

|  |  |
| --- | --- |
| SQL | PL/SQL |
| SQL Stands for Structured Query Language | PL/SQL stands for procedural structured query Language and It is an extension to SQL |
| In SQL we can execute only one statement at a time | In PL/SQL we can execute more than one statement at a time |
| In SQL Network Traffic is high when compare to PL/SQL  e.g.: If execute SQL statements. It will execute each and every statement individually. It makes server busy | In PL/SQL Network Traffic is low when compare to SQL  e.g. :I t will execute block of statements at a time |
| we can't handle Exceptions in SQL | we can handle Exceptions by using Exception handlers in PL/SQL |
| Here we get different output messages for execution of a different statement | Here we get same output message for execution of any program |
| we can write SQL in PL/SQL | we can't write PL/SQL in SQL |

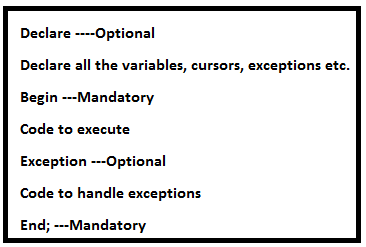
* **PL SQL is Blocked Structure language**

Here we have two types of Blocks

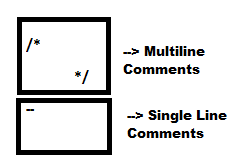
**1. Unnamed Block:** A block which doesn’t contain any name is called as “unnamed block”. These are also called as **“Anonymous Blocks”**

These programs won’t be stored in the Database. So, If we want to execute these programs again then we have to copy entire code and need to run the program from any environment (system).

**Syntax:**

****

**Comments**

****

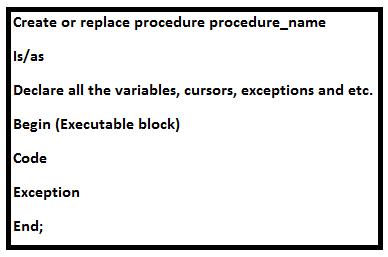
**2. Named Block:** A block which contains name is called as “Named block”. These blocks are also called as **“Sub programs”**

* These programs will be stored in the Database. So, that we can call these programs with that program name from any environment.

Types of sub programs or named blocks

1. Procedures
2. Functions
3. Packages
4. Triggers

**Procedure Syntax:**

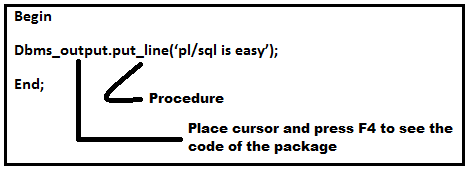
****

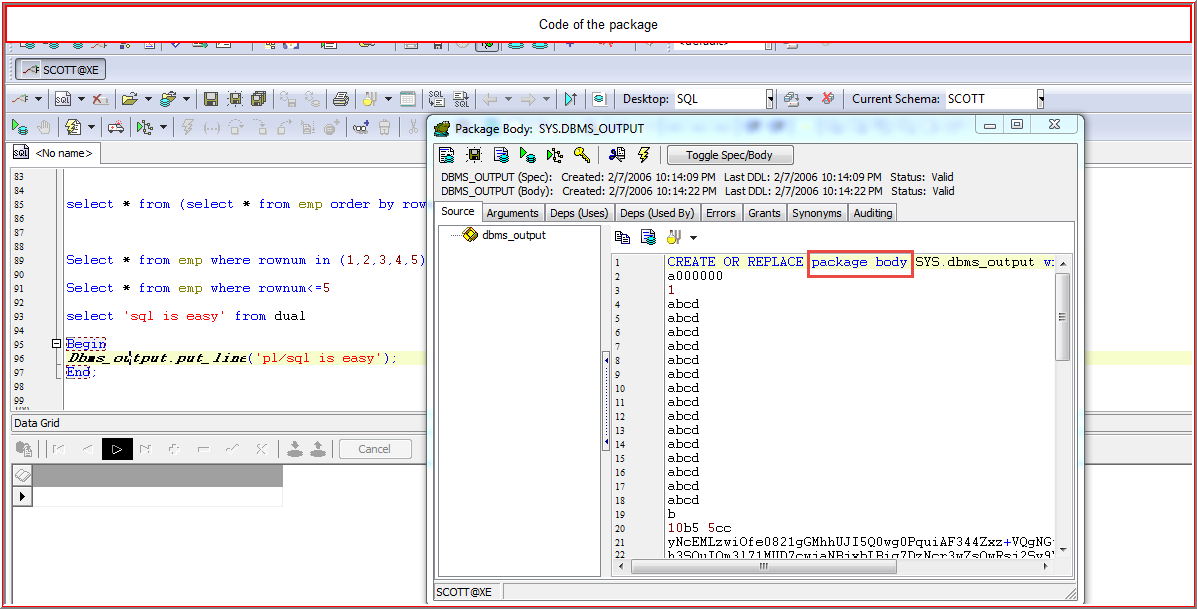
**Sample program**

**Begin**

**Dbms\_output.put\_line(‘pl/sql is easy’);**

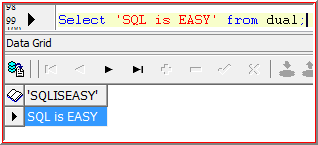
**End;**

****

****

**To print message in SQL**

**Select ‘SQL is EASY’ from dual;**

****

**Dbms\_output.put\_line**

Basically PL/SQL doesn’t have any capability to print the messages of its own. So, we are using **dbms\_output.put\_line**. Here dbms\_output.put\_line is a **package** and **put\_line** is a **procedure** used to pick the data from the DB and will keep it in the buffer

**Set serveroutput on:**

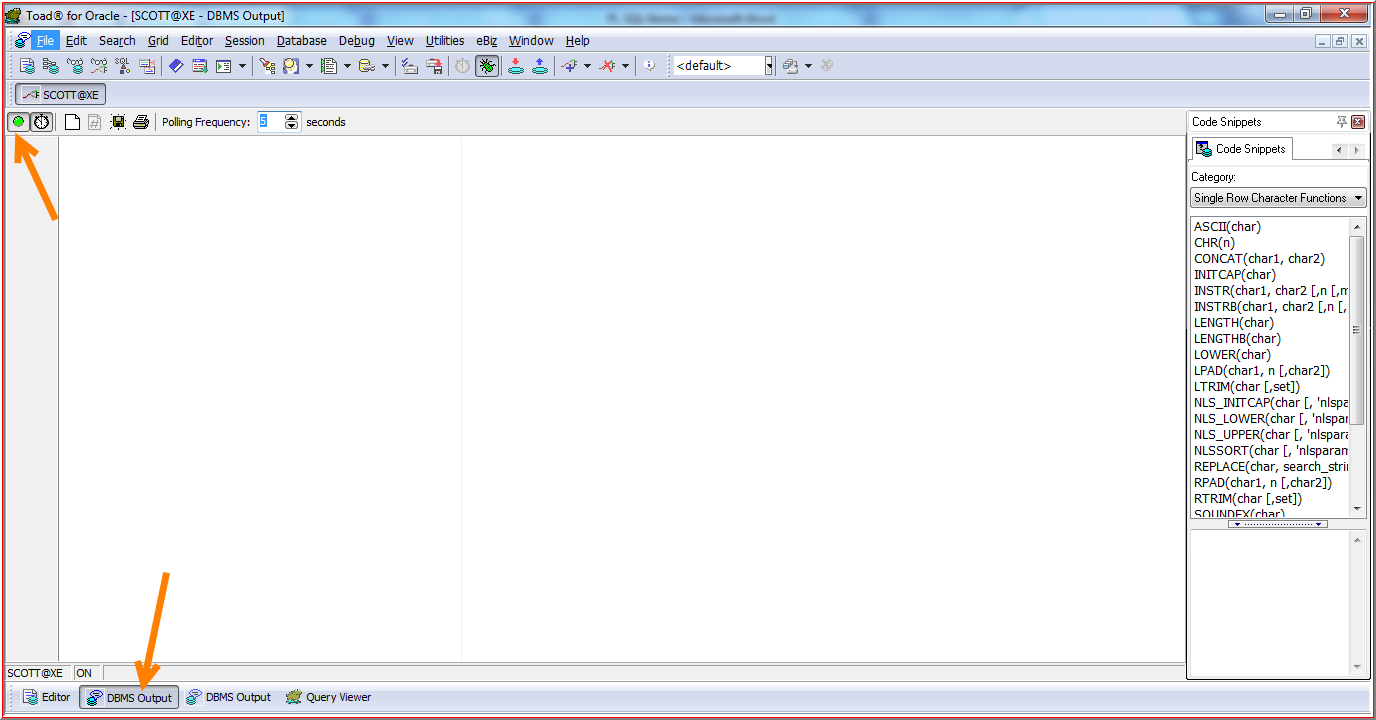
It is a sql\*plus environment command used to pick the data from the buffer and prints on the screen

**In toad**

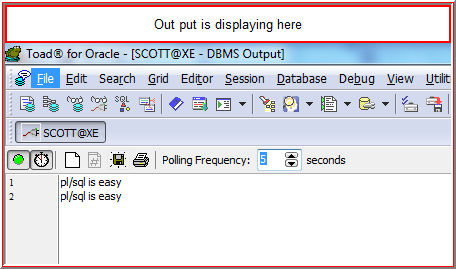
Select DBMS\_OUTPUT and ‘on’

Or press F2

Or go to view menu and select DBMS\_OUTPUT



Output is displaying in DBMS\_OUTPUT



**Declaring Variables:**

**Variable:**

* It will hold a value temporarily.
* We can overwrite variable value.

**Types of Variables**

**1. Normal Variable**

**2. Not Null:** we have to initialize the value while declaring variable there itself, but we can reinitialize the variable.

**3. Default:** we have to initialize the value while declaring the variable there itself, but we can reinitialize the value. If we are not passing any value it will take default value.

**4. Constant:** we have to initialize the value while declaring the variable there itself, but we cannot reinitialize the value.

**Example program:**

declare

v\_empno number not null:=2089;

v\_ename varchar2(20):='Varun';

v\_sal number default 5000;

v\_deptno constant number:=30;

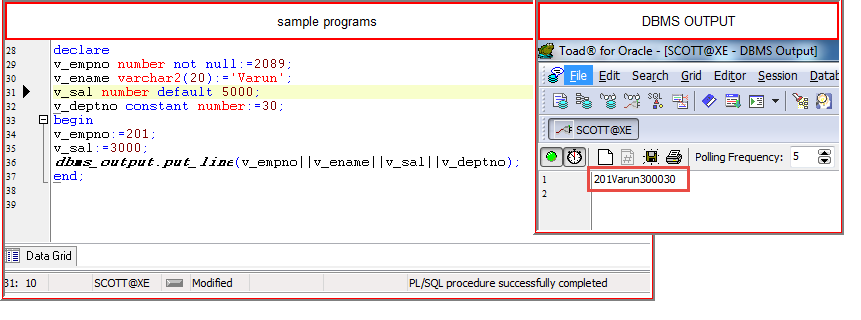
begin

v\_empno:=201;

v\_sal:=3000;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal||v\_deptno);

end;



**Using DML Statements in PL SQL:**

**Insert**

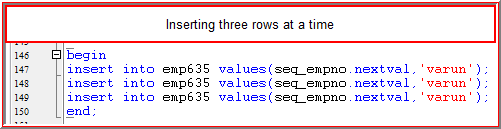
begin

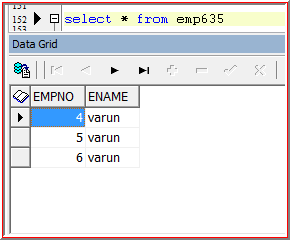
insert into emp635 values(seq\_empno.nextval,'varun');

insert into emp635 values(seq\_empno.nextval,'varun');

insert into emp635 values(seq\_empno.nextval,'varun');

end;



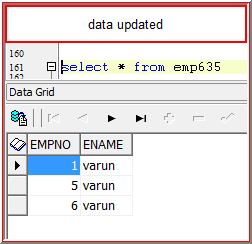


**Update**

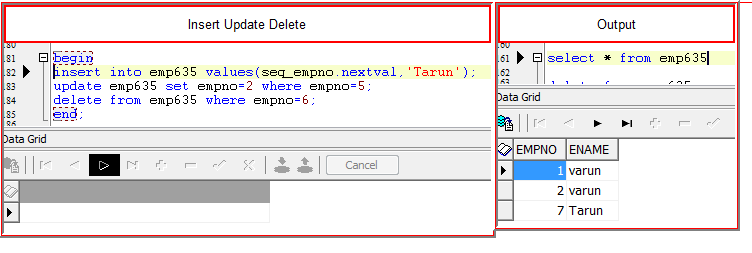
begin

update emp635 set empno=1 where empno=4;

end;



**Insert, Update, Delete**



**Writing DQL statements in PL SQL**

In PL SQL “Select” statement has to return a value (record) that to only “one” value or record. If the select statement returns more than one, it will through an **error**

**Example**

Declare

V\_empno number(20);

V\_ename varchar2(20);

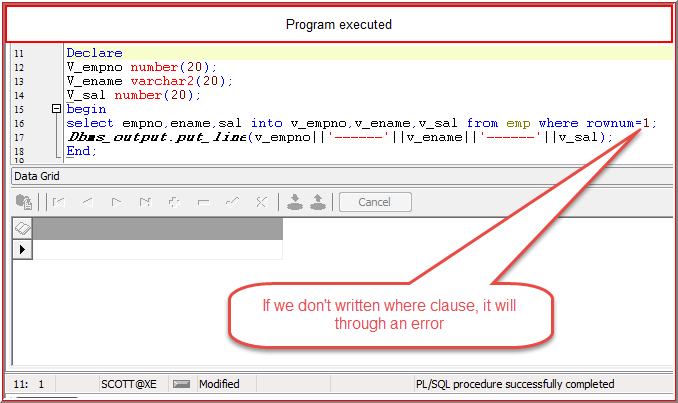
V\_sal number(20);

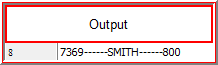
begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where rownum=1;

***Dbms\_output.put\_line***(v\_empno||'------'||v\_ename||'------'||v\_sal);

End;

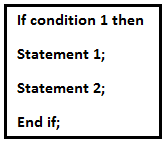




**Conditional Statements**

1. **IF**
2. **If Else**
3. **Elsif**

**1. If:**

****

**Example Program:**

Declare

V\_ename varchar2(20);

V\_sal number(20);

Begin

Select ename,sal into v\_ename,v\_sal from emp where empno=&empno;

If v\_sal between 0 and 2000 then

***Dbms\_output.put\_line***(v\_ename||v\_sal||'is a low salary');

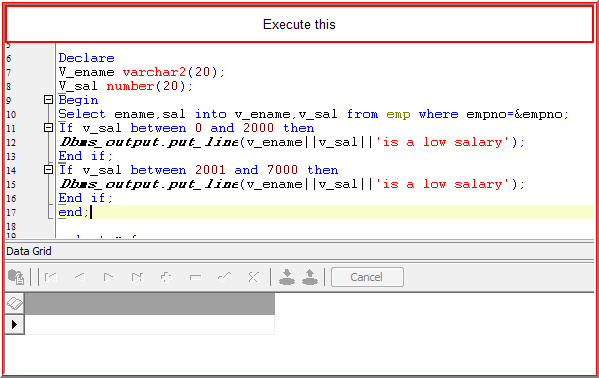
End if;

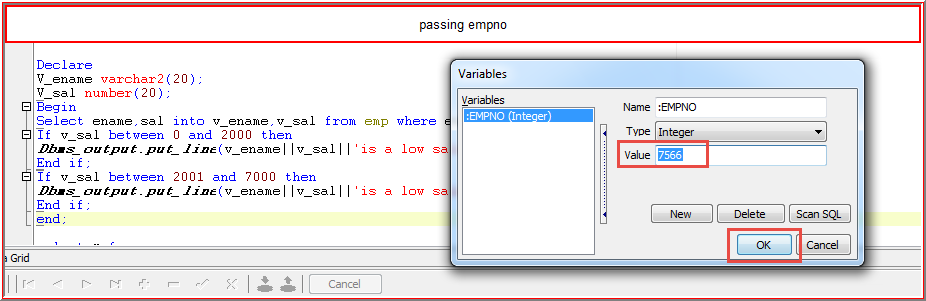
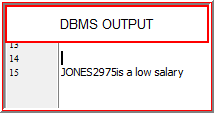
If v\_sal between 2001 and 7000 then

***Dbms\_output.put\_line***(v\_ename||v\_sal||'is a low salary');

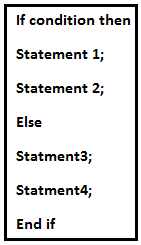
End if;

end;

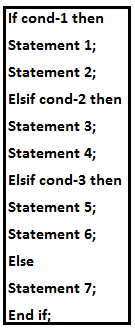


**2. If Else**

****

**3. Elsif**

****

**Example Program:**

Declare

V\_ename varchar2(20);

V\_sal number(20);

Begin

Select ename,sal into v\_ename,v\_sal from emp where empno=&empno;

If v\_sal between 0 and 2000 then

***Dbms\_output.put\_line***(v\_ename||v\_sal||'is a low salary');

elsif v\_sal between 2001 and 7000 then

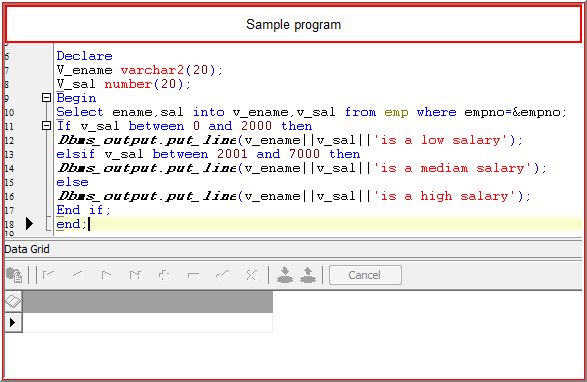
***Dbms\_output.put\_line***(v\_ename||v\_sal||'is a medium salary');

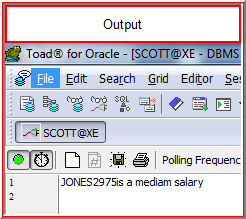
else

***Dbms\_output.put\_line***(v\_ename||v\_sal||'is a high salary');

End if;

end;





**Non PL SQL variables**

**1. Bind Variables :** It is a non PL SQL variable which will declare at calling environment(Runtime)

We can use this variable like a normal PL SQL variable through the session for any program

Open SQL \* PLUS

Ed p1 (type this and click on “Enter”)

Note pad opens

Declare

V\_empno number(20);

V\_ename varchar2(20);

begin

select empno,ename,sal into v\_empno,v\_ename**,:v\_sal** from emp where empno=7369;

***Dbms\_output.put\_line***(v\_empno||'------'||v\_ename||'------'||:v\_sal);

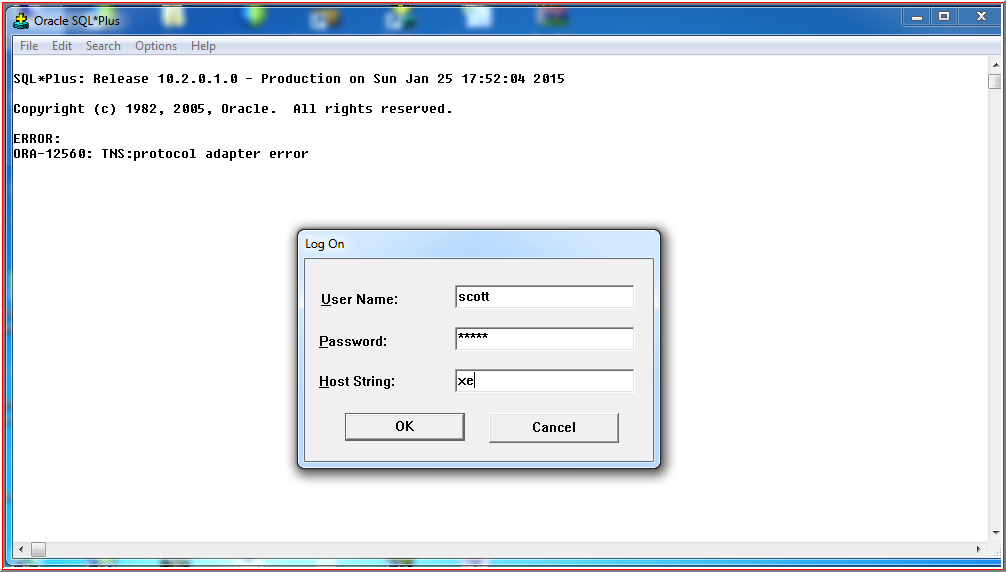
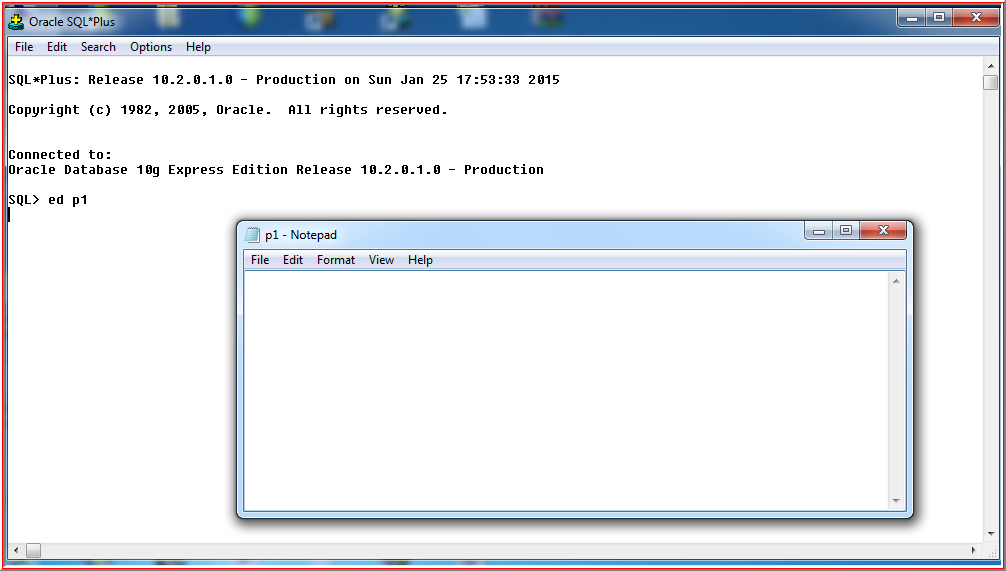
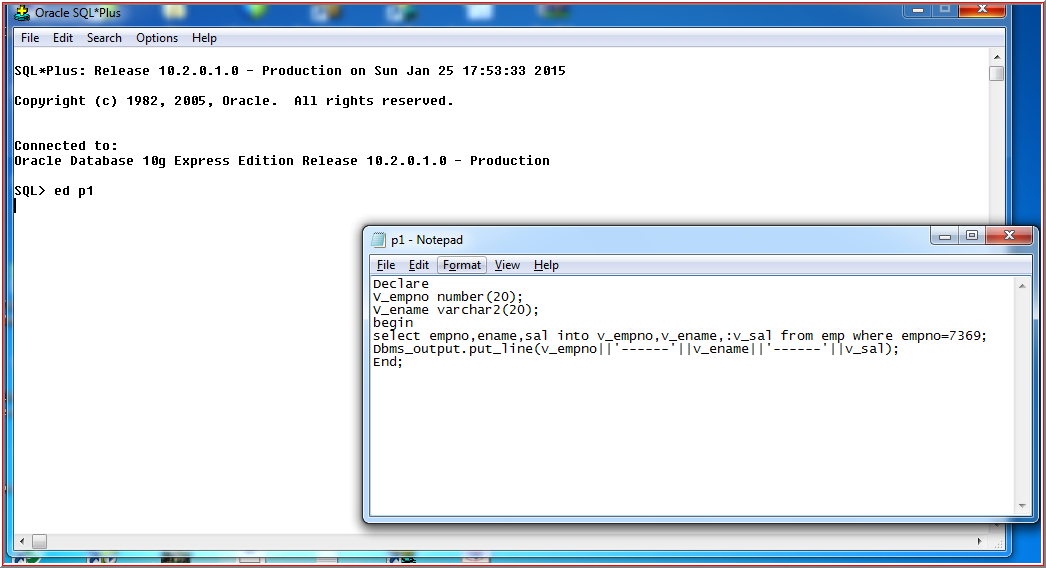
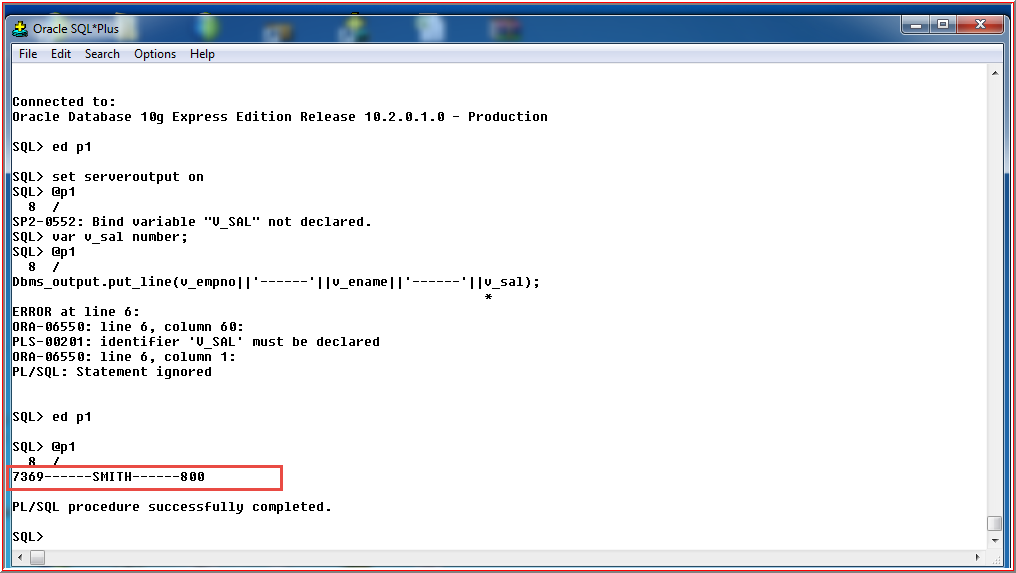
End;

Set serveroutput on

Var v\_sal number;

@p1(press “Enter”)

/

The bind variable is used only for this session for any number of programs, once we close this session again we need to declare

**Host Variable**

By using this variable we can pass the “values at Runtime”

**Example program**

Declare

V\_empno number(20);

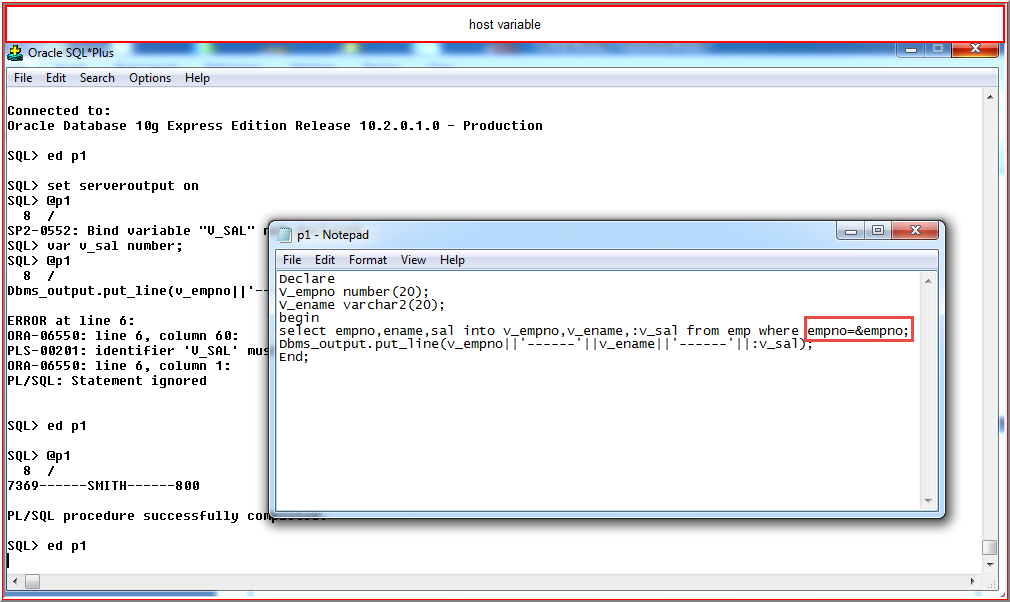
V\_ename varchar2(20);

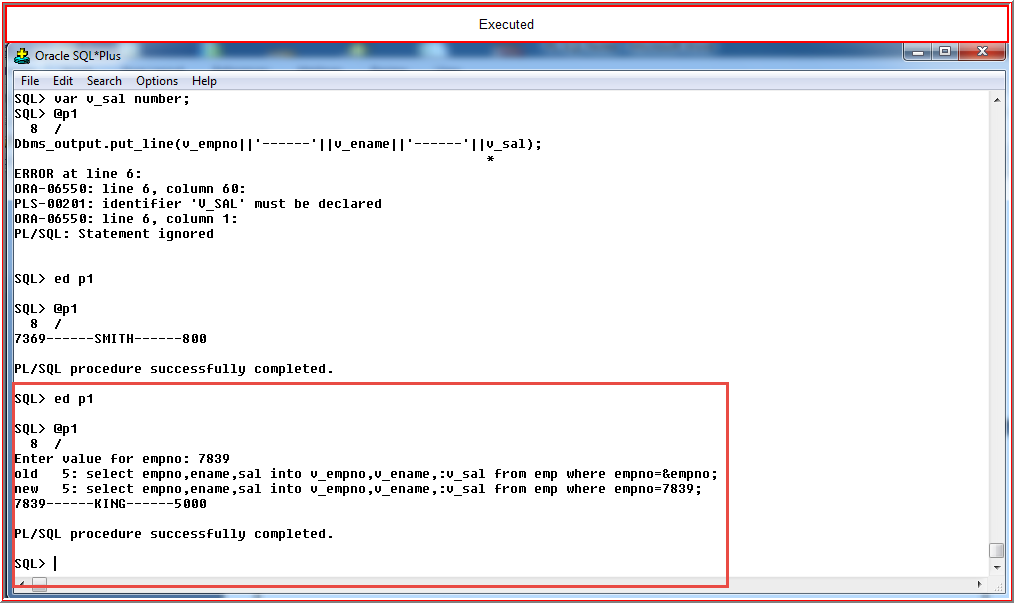
begin

select empno,ename,sal into v\_empno,v\_ename,:v\_sal from emp where empno=&empno;

Dbms\_output.put\_line(v\_empno||'------'||v\_ename||'------'||:v\_sal);

End;

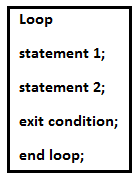




**Iterative Statements**

**1. Loop**

**Syntax:**



**Example program:**

Declare

v\_num number:=1;

Begin

Loop

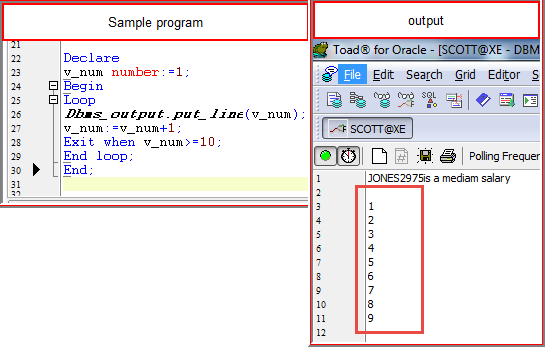
Dbms\_output.put\_line(v\_num);

v\_num:=v\_num+1;

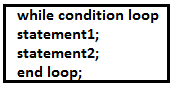
Exit when v\_num>=10;

End loop;

End;



**2. While**



Declare

v\_num number:=1;

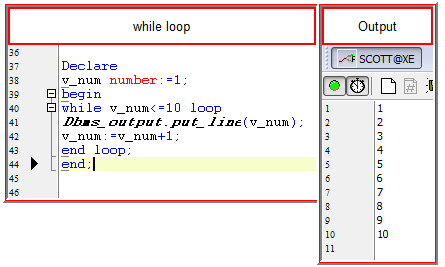
begin

while v\_num>=10 loop

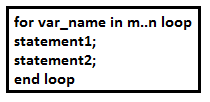
v\_num:=v\_num+1;

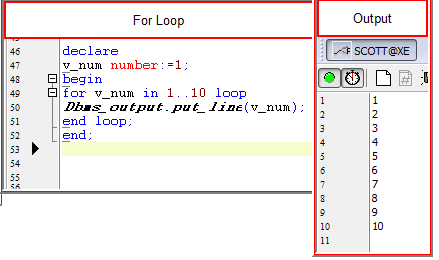
end loop;

end;



**3. for Loop**





Whatever the SQL or PL SQL statements we are writing all those statements will follow the following steps:

1. It will focus on memory area
2. That memory area will be populated with actual data
3. That populated data will be processed as per our requirement
4. After fulfilling the requirement that memory area will be closed

**Cursors**

Cursor is a private SQL working area provided by Oracle SQL Engine in order to process SQL statements

Two Types of Cursors

1. Implicit Cursor
2. Explicit Cursor

**1. Implicit Cursor:**

It will open by the “Oracle Engine” and used for the DML statements and for the “Select” statement which is returning only one row

**Implicit cursor attributes:**

1. %ISOPEN
2. %FOUND
3. %NOT FOUND
4. %ROWCOUNT

By using these attributes, we can know the status of the cursor and we can control the cursor.

**1. %ISOPEN:** It will return true if the cursor is opened else false.

In case of implicit cursor, **it will return always false.**

**2. %FOUND:** It will return true, if the DML or select statement affect the row else false

**3. %NOTFOUND:** It will return true, if the DML or select statement does not affect the row else false

**4. %ROWCOUNT:** It will return number rows affected by the DML or Select Statements.

**E.g.:** For the select statement always the rowcount will be ‘1’. Because select statement has to return only one value

begin

update emp set sal=4500 where empno=7902;

if sql%isopen then

dbms\_output.put\_line('Cursor is opened');

end if;

if sql%FOUND then

dbms\_output.put\_line('Number of rows Effected'||'-----'||sql%rowcount);

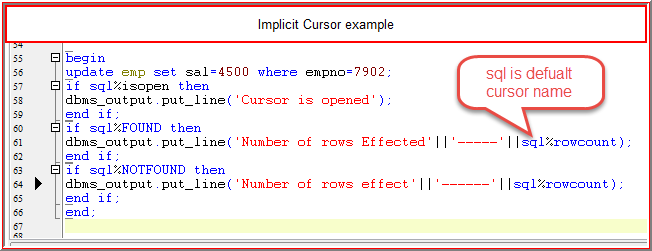
end if;

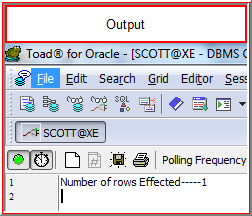
if sql%NOTFOUND then

dbms\_output.put\_line('Number of rows effect'||'------'||sql%rowcount);

end if;

end;



****

If we want to make select statement return more than ‘1’ row we use explicit cursor.

**2. Explicit Cursor:** It will be open by the user and used for the select statement which is returning more than one row.

**Explicit cursor attributes**

By using these attributes, we can know the status of the cursor and we can control the cursor.

**1. %ISOPEN:** It will return true if the cursor is opened else false.

In case of implicit cursor, **it will return always false.**

**2. %FOUND:** It will return true, if the last record fetch is successful else false

**3. %NOTFOUND:** It will return true, if the last record fetch is unsuccessful else false

**4. %ROWCOUNT:** It will return number rows fetched by the select statement

**Cursor life cycle**

1. Declare cursor 🡪 cursor cursor\_name is (sub query)

2. Open cursor 🡪 open cursor\_name

3. Fetch cursor 🡪fetch cursor\_name into variables |records

4. Close cursor 🡪close cursor\_name

Example program

Declare

Cursor emp\_cur is select empno,ename,sal from emp;

V\_empno number(20);

V\_ename varchar2(20);

V\_sal number(20);

Begin

Open emp\_cur;

Fetch emp\_cur into v\_empno,v\_ename,v\_sal;

***dbms\_output.put\_line***(v\_empno||' '||v\_ename||' '||v\_sal);

Fetch emp\_cur into v\_empno,v\_ename,v\_sal;

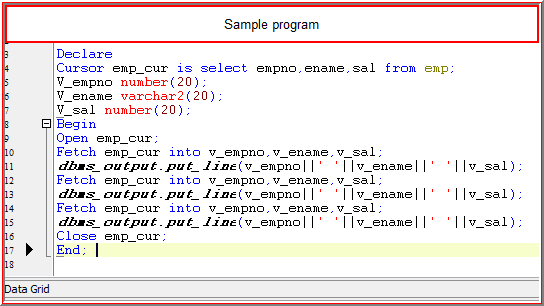
***dbms\_output.put\_line***(v\_empno||' '||v\_ename||' '||v\_sal);

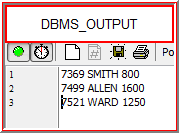
Fetch emp\_cur into v\_empno,v\_ename,v\_sal;

***dbms\_output.put\_line***(v\_empno||' '||v\_ename||' '||v\_sal);

Close emp\_cur;

End;

****

****

**Disadvantages:**

If we write fetch statement in less than emp table records, It will print that many records, If we write more than emp records**, It will print the last fetch record repetitively.**

In this scenario fetch returns only one record

To overcome the above problem we use loop

**Cursor for loop**

If we write cursor by using for loop then it is called as Cursor **For Loop**. At that time we no need to open, fetch and close the cursor all these operations will be done automatically.

**Example program**

declare

cursor emp\_cur is select \* from emp;

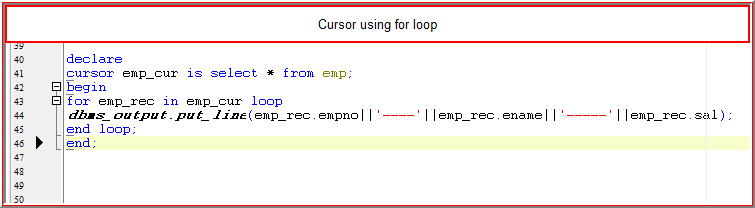
begin

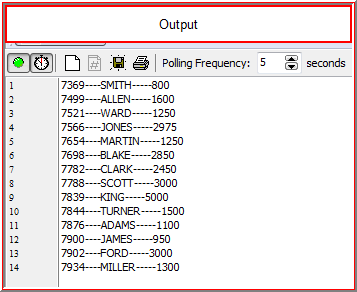
for emp\_rec in emp\_cur loop

***dbms\_output.put\_line***(emp\_rec.empno||'----'||emp\_rec.ename||'-----'||emp\_rec.sal);

end loop;

end;

****

****

**Parameterized cursor**

By using this we can pass the values to the cursor dynamically at runtime

declare

cursor emp\_cur(p\_deptno number) is select \* from emp where deptno=p\_deptno;

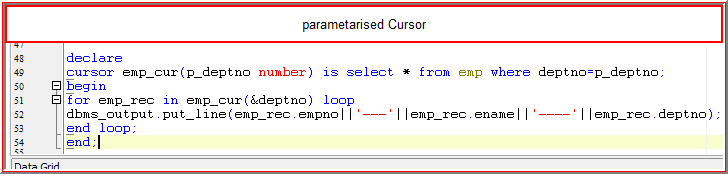
begin

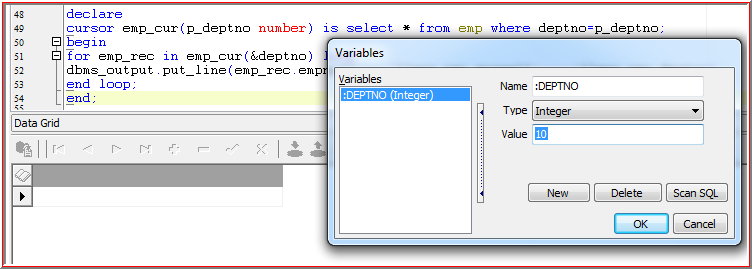
for emp\_rec in emp\_cur(&deptno) loop

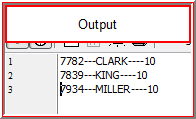
dbms\_output.put\_line(emp\_rec.empno||'---'||emp\_rec.ename||'----'||emp\_rec.deptno);

end loop;

end;







**Exception:**

Exception is nothing but an error

We will get error at two timings

* Compilation Time Error
* Runtime Error

**1. Compilation Time Error:**

Here we will get all the syntactical errors and database objects does not existed kind of errors.

**Example:**

select \* from **emp43** ------emp43 not existed in Database

select \* **fro** emp -------from keyword is error(Syntactical error)

**We can’t handle these kind of errors programmatically i.e. we need to handle manually**

**2. Runtime Error:**

Here we get all logical errors

Select \* from emp43 ------there is no data in this table, but table exist

We can handle these kinds of errors programmatically

**Types of Exceptions**

**1. Predefined Exceptions:** No need to raise these exceptions and allow the oracle server to raise then implicitly.

**Example:**

No Data found, too many rows, invalid cursor etc.

**2. Predefined Exceptions:** These exceptions are only for readable purpose.

🡪we have to declare these exceptions and allow the oracle server to raise them implicitly.

**3. User Defined Exceptions**: Oracle won’t raise the exceptions for all the situations. So, we have to declare these exceptions and allow the oracle server to raise them explicitly.

**Other Handlers:** By using this handler we can handle any kind of exceptions

* Other handler should be the last handler among all the handlers

**SQL Code:** It will return error number

**SAL Errm:** It will return error message

**Example for Predefined:**

declare

v\_empno number(20);

v\_ename varchar2(20);

v\_sal number(20);

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where empno=7222;

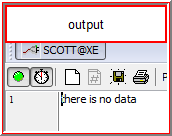
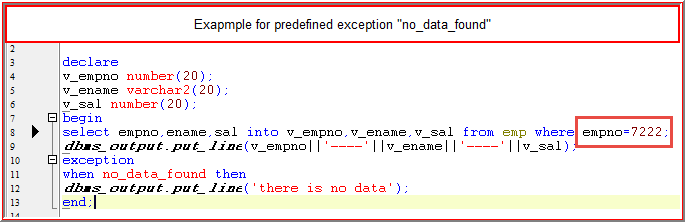
***dbms\_output.put\_line***(v\_empno||'----'||v\_ename||'----'||v\_sal);

exception

when no\_data\_found then

***dbms\_output.put\_line***('there is no data');

end;



**Other Handlers example**

Declare

v\_empno number(20);

v\_ename varchar2(20);

v\_sal number(20);

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where deptno=0;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal);

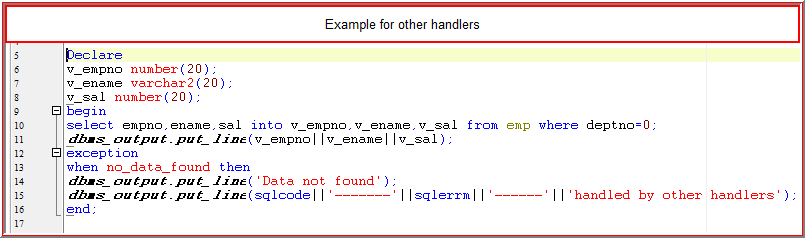
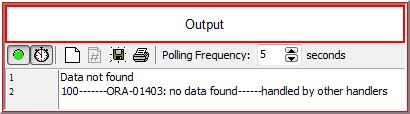
exception

when no\_data\_found then

***dbms\_output.put\_line***('Data not found');

***dbms\_output.put\_line***(sqlcode||'-------'||sqlerrm||'------'||'handled by other handlers');

end;

**Example for passing the values at Runtime when no data found:**

Declare

v\_empno number(20);

v\_ename varchar2(20);

v\_sal number(20);

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp3 where empno=&empno;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal);

exception

when no\_data\_found then

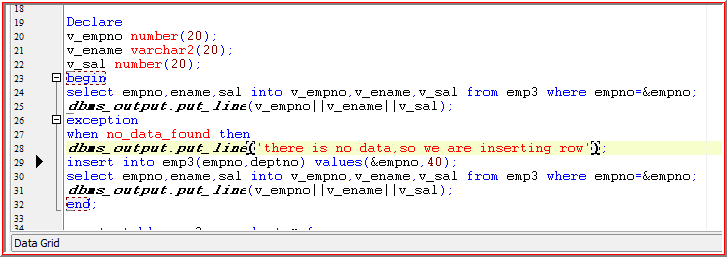
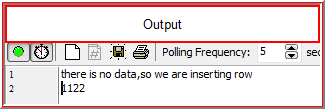
***dbms\_output.put\_line***('there is no data,so we are inserting row');

insert into emp3(empno,deptno) values(&empno,40);

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp3 where empno=&empno;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal);

end;

**Non Predefined Exceptions**

These exceptions are useful only for readable purpose

**1. Pragma:** Pragma is compiler directory which serves the instructions to the compiler whenever compiler is compiling the code.

**Types of Pragma’s:**

1. Pragma autonomous transaction

2. Pragma exception\_init

3. Pragma restrict\_reference

4. Pragma serially\_reusable

**Pragma exception\_int:**

declare

v\_empno number(20);

v\_ename number(20);

v\_sal number(20);

datatype\_missmatch exception;

pragma exception\_init(datatype\_missmatch,-6502);

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where empno=7902;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal);

exception

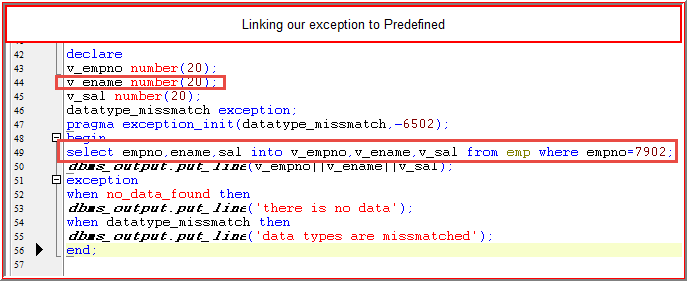
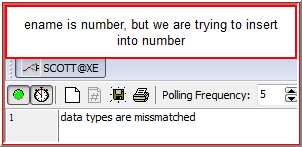
when no\_data\_found then

***dbms\_output.put\_line***('there is no data');

when datatype\_missmatch then

***dbms\_output.put\_line***('data types are mismatched');

end;

**User Defined Exceptions**

Oracle doesn’t raise the exceptions for all the situations

**Example:** If update statement is failed to affect the rows, then oracle won’t raise any exceptions. So we have to raise our own exceptions.

We can raise user defined exceptions by using the keyword raise.

**Difference between DBMS\_OUTPUT.PUT\_LINE and RAISE\_APPLICTION\_ERROR**

**DBMS\_OUTPUT.PUT\_LINE** will print the error message like a normal text message and it will execute all the statements which we are writing after that DBMS statement i.e. it won’t stop the program

**RAISE\_APPLICATION\_ERROR** will print the error message like predefined error message and It won’t execute all the statements which we are writing after that statement i.e. it will stop the program

**1.Example program:**

Declare

Update\_failed exception;

Begin

Update emp set sal=87000 where empno=79009;

If sql%notfound then

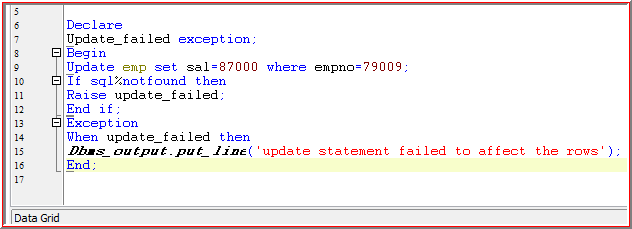
Raise update\_failed;

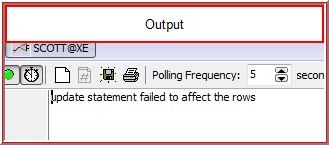
End if;

Exception

When update\_failed then

***Dbms\_output.put\_line***('update statement failed to affect the rows');

End;



**2. Example program:**

Declare

Update\_failed exception;

Begin

Update emp set sal=87000 where empno=79009;

If sql%notfound then

Raise update\_failed;

End if;

Exception

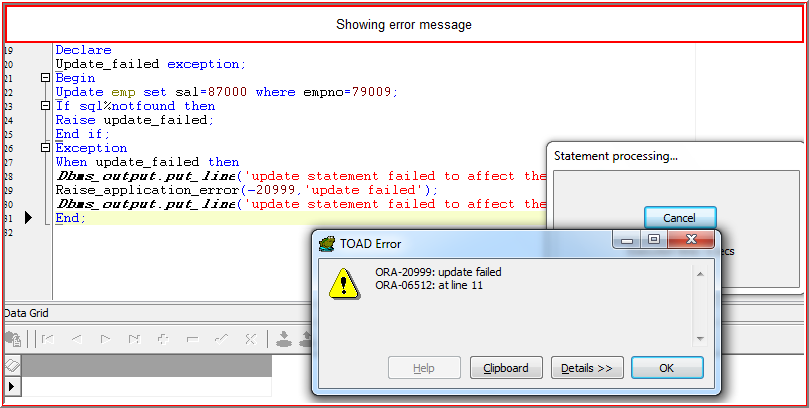
When update\_failed then

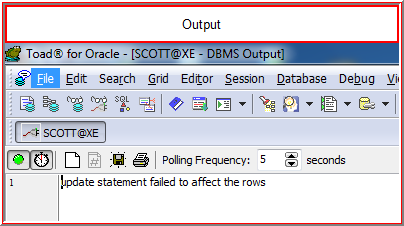
***Dbms\_output.put\_line***('update statement failed to affect the rows');

Raise\_application\_error(-20999,'update failed');

***Dbms\_output.put\_line***('update statement failed to affect the rows');

End;





**3.Example program:**

Declare

Update\_failed exception;

Begin

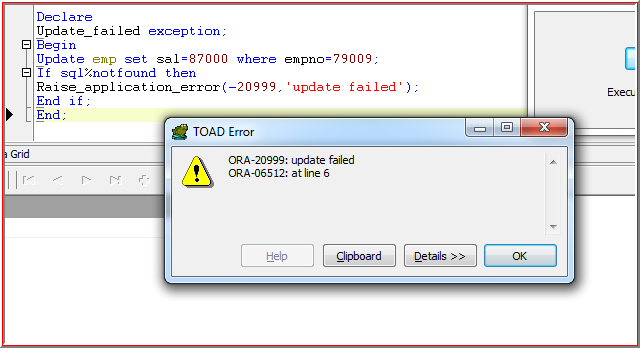
Update emp set sal=87000 where empno=79009;

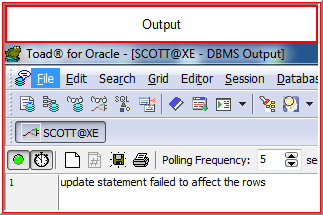
If sql%notfound then

Raise\_application\_error(-20999,'update failed');

End if;

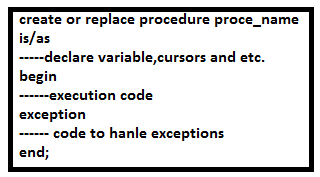
End;





**PROCEDURES**

* Procedure is a subprogram use to perform any kind of action.
* Whenever we create a procedure then it will be stored in the Database with compilation Form.
* Now if we want to run that program we can run by using that program name by using **exec** command or Anonymous block **Begin End.**



**Sample Program**

create or replace procedure emp\_details

is

v\_empno number(20);

v\_ename varchar2(20);

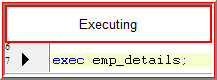
v\_sal number(20);

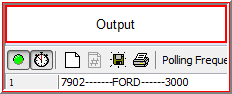
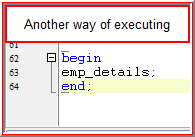
begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where empno=7902;

***dbms\_output.put\_line***(v\_empno||'-------'||v\_ename||'------'||v\_sal);

end;





**Types of parameters:**

**In:**  By using this we can pass the values at runtime. It is a Default parameter.

**Sample program**

create or replace procedure get\_details(p\_empno in number)

is

v\_empno number(20);

v\_ename varchar2(20);

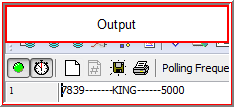
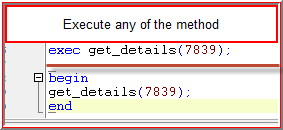
v\_sal number(20);

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where empno=p\_empno;

***dbms\_output.put\_line***(v\_empno||'-------'||v\_ename||'------'||v\_sal);

end;



**OUT:**  By using this we can declare the variables at calling environment

**Sample program:**

create or replace procedure get\_details1(p\_empno in number,p\_sal out number)

is

v\_empno number(20);

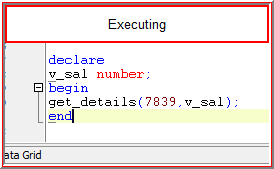
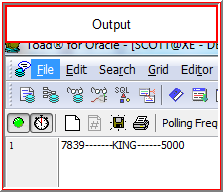
v\_ename varchar2(20);

begin

select empno,ename,sal into v\_empno,v\_ename,p\_sal from emp where empno=p\_empno;

***dbms\_output.put\_line***(v\_empno||'-------'||v\_ename||'------'||p\_sal);

end;

**INOUT:** It is a combination of IN and OUT parameters

create or replace procedure get\_emps1(p\_var in out varchar2)

is

v\_empno number(20);

v\_ename varchar2(20);

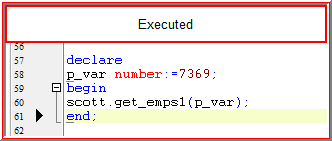
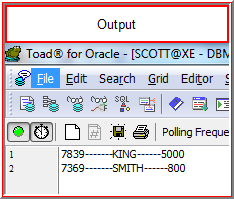
begin

select empno,ename,sal into v\_empno,v\_ename,p\_var from emp where empno=p\_var;

***dbms\_output.put\_line***(v\_empno||'-------'||v\_ename||'------'||p\_var);

end;

p\_var is passed it to empno=7369 and then salary is passed to p\_var and it returns

INOUT and OUT Combination:

Create or replace procedure get\_emps(p\_var in out varchar2,p\_sal out number)

is

v\_empno number(20);

begin

select empno,ename,sal into v\_empno,p\_var,p\_sal from emp where empno=p\_var;

dbms\_output.put\_line(v\_empno||p\_var||p\_sal);

end;

declare

v\_var varchar2(20):=101;

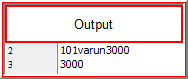
v\_sal number;

begin

get\_emps(v\_var,v\_sal);

***dbms\_output.put\_line***(v\_sal);

end;



**Functions**

It is a subprogram used to perform an action.

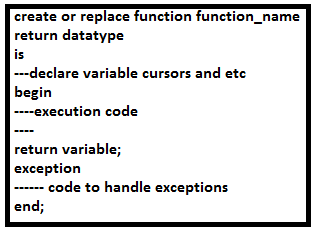
**Difference between functions and procedures**

Procedure may or may not return any value

Function has to return a value

Functions and procedure doesn’t require any begin keyword while declaring variables

**Syntax of function**



**Sample program**

create or replace function get\_annualsal return number

is

v\_annualsal number;

begin

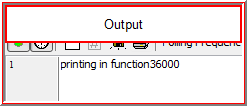
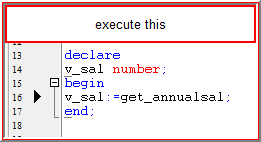
select sal\*12 into v\_annualsal from emp where empno=101;

***dbms\_output.put\_line***(v\_annualsal);

return v\_annualsal;

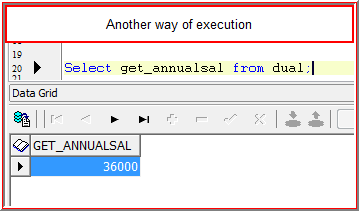
end;

If we execute this, it won’t display any v\_annualsal, we need to execute this by calling the function name like below



We can also execute this using dual table

Select get\_annualsal from dual;



**Get the annual salary using parameterized**

create or replace function get\_annualsal(p\_empno in number) return number

is

v\_annualsal number;

begin

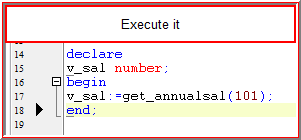
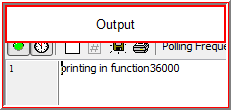
select sal\*12 into v\_annualsal from emp where empno=p\_empno;

***dbms\_output.put\_line***('printing in function'||v\_annualsal);

return v\_annualsal;

dbms\_output.put\_line('program completed');

end;

* We can write ‘n’ number of return statements in a function but it will **execute only 1st return statement** i.e. it won’t execute all the statements which we have written after that **1st return statement**
* Observe the above program execution, it doesn’t execute the statement present after the return statement.
* If we have a **DML (insert, update, delete)** statements present in the function, then **we can’t call that function from a select statement**. So, we have to call that function from anonymous block

**Package**

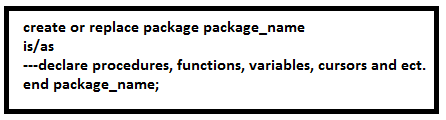
It is a collection of functions, procedures etc.

Package contains two parts

**1. Specification:**

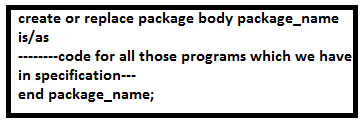
Here we declare all the procedures, functions, variables etc.

**Syntax**

****

**2. Body:**

Here we write the code for all the programs which we have declared in package specification.



* We can create a specification without body but not vice versa (body without specification)
* Whatever the programs we are declaring in package specification all those programs need to be appearing in package body, but not vice versa.

**Private functions:**

The function declaration is not in specification but present in body we can use these type of functions within the package only but not in outside the package

**Example program:**

**Procedure:**

create or replace procedure get\_empdetails(p\_empno in number)

is

v\_empno number(20);

v\_ename varchar2(20);

v\_sal number(20);

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where empno=p\_empno;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal);

end;

**Example for package:**

**Creating package specification:**

create or replace package emp\_pkg

is

procedure get\_empdetails(p\_empno in number);

function get\_annualsal(p\_empno in number) return number;

end emp\_pkg;

**Creating package Body**

create or replace package body emp\_pkg

is

procedure get\_empdetails(p\_empno in number)

is

v\_empno number;

v\_ename varchar2(20);

v\_sal number;

begin

select empno,ename,sal into v\_empno,v\_ename,v\_sal from emp where empno=p\_empno;

***dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal);

end get\_empdetails;*---optional*

function get\_annualsal(p\_empno in number)

return number

is

v\_annsal number;

begin

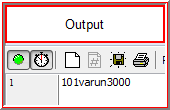
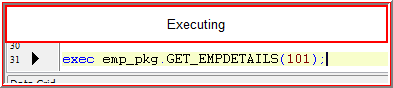
select sal\*12 into v\_annsal from emp where empno=p\_empno;

***dbms\_output.put\_line***(v\_annsal);

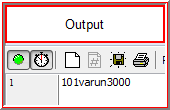
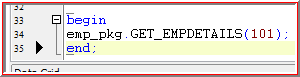
return v\_annsal;

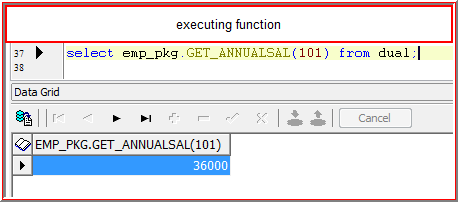
end get\_annualsal;

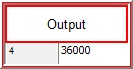
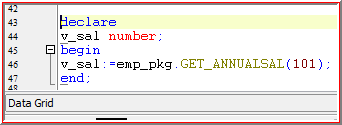
end emp\_pkg;

****

**Another way of executing**

****

****

****

**Triggers**

It is a subprogram and it will fire whenever a particular event takes place.

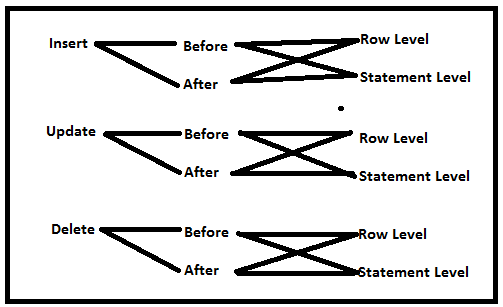
**Types of Triggers:**

**1. DML Triggers:**

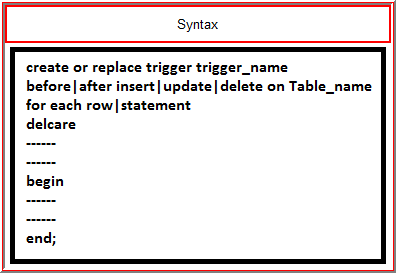
These triggers will fire on all the DML events

These DML Triggers will fire at two timings (Before and after) and two levels (Row level and Statement Level)

Total we have 12 types of DML statements



Default Trigger firing level is “statement Level”



**Example:**

We have two tables’ emp\_details1 and emp\_summary

Whenever we perform any DML operations on emp\_details1 the total\_records count and total\_salaries and modified date will be inserted into emp\_summary

**Tables:**

create table emp\_details as select \* from emp

create table emp\_summary(total\_records number(20),total\_salaries number(20),modified\_date date)

**Trigger**

create or replace trigger emp\_dml\_trigger after insert or delete or update on emp\_details

declare

v\_tr number;

v\_ts number;

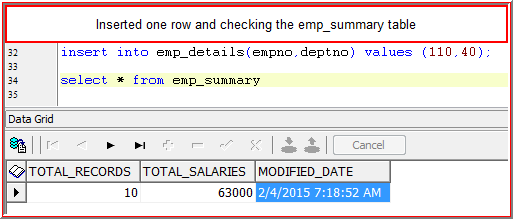
begin

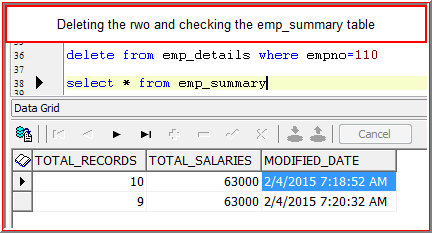
select count(\*),sum(sal) into v\_tr,v\_ts from emp\_details;

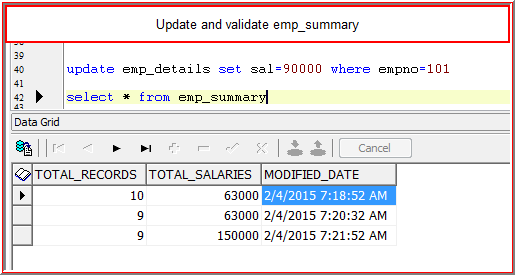
insert into emp\_summary values(v\_tr,v\_ts,sysdate);

end;

**Performing operations and check the emp\_summary**







**2. System Triggers** (e.g.: swipe in and swipe out time tracking)

**A.DDL:** These Triggers will fires on all the DDL events like create, alter.

**B.DB Triggers:** These Triggers will fire on all the DB events like shutdown, login, logoff etc.

**3. Instead of Triggers** These Triggers will be fired on all complex views

**Pragma Autonomous Transaction**

Autonomous Transaction is Transaction within the context of another Transaction

If we use Autonomous Transaction then child transaction won’t affect parent Transaction

**Case1: Description**

This program we are using without autonomous transaction. In this if we execute this it will affect the parent Transaction

E.g. when we call the child procedure the commit present in child is also impacting the parent also. So we can avoid this in **case 2** using **Pragma Autonomous Transaction**

**Case1:**

create or replace procedure emp\_ins(p\_empno number,p\_deptno number)

is

begin

insert into emp\_details(empno,deptno) values (p\_empno,p\_deptno);

commit;

end;

The commit statement in the above will be committing the Transactions whatever we have done before, once we commit we can’t rollback.

**Another program:**

create or replace procedure emp\_del(p\_empno number)

is

begin

delete from emp\_details where empno=p\_empno;

emp\_ins(112,50);

end;

------The commit statement present in emp\_ins is also effecting emp\_del transactions also

Exec emp\_del(110)

Begin

Emp\_ins(110,40)

End;

Rollback;

Here we can’t roll back, because the delete statement already commited

**Case2 :**

**Declaring Pragma in parent**

create or replace procedure emp\_ins(p\_empno number,p\_sal number,p\_deptno number)

is

pragma autonomous\_transaction; *---It will prevent from child table transactions to parent table*

begin

insert into emp\_details(empno,sal,deptno) values (p\_empno,p\_sal,p\_deptno);

commit;

end;

**Child table**

create or replace procedure emp\_del(p\_empno number)

is

begin

delete from emp\_details where empno=p\_empno;

emp\_ins(113,20000,50);

end;

exec emp\_del(110);

begin

emp\_ins(110,30000,90);

end;

check the table before rollback and after rollback

**Locks**

* Oracle is a multiuser platform. So, that ‘n’ number of users can access the data at same time.
* Oracle Database may contain valuable information. So, oracle has to provide security for that data.
* Oracle will provide the security by using the method called ‘locking’.
* Oracle will apply the locks to the data without any user interaction.

But Oracle will apply the locks by considering the following concepts:

* **Types of locks to be applied**
* **Level of locks to be applied**

**Types of Locks to be applied:**

We have two types of locks based on the following two operations:

Read Operation (select) -----**Shared Lock**

Write Operation (Insert, update, delete) --------**Exclusive Lock**

**A. Shared Lock:**

* Read Operation won’t affect the data. So, Oracle will apply Shared Lock on the data which is currently being viewed.
* There may be ‘n’ number of shared locks on the Data.

**B.Exclusive Lock:**

* Write operation will affect the data. So, oracle will apply exclusive lock on the data which is currently being manipulated.
* The user 1st who occupied the lock is owner of that lock until and unless he releases the lock by using “Commit” or “Rollback” other users can’t manipulate the data but they can view the Data.
* There is only **“one exclusive lock”**

**\*But there is chance to view wrong data**

**Why wrong data?**

**Suppose when the user update the data but he didn’t commit the data. In that case the other user can view uncommitted data.**

**C. Explicit Lock**

If Oracle will apply the exclusive lock on the data then other users can’t manipulate the data, but they can view the data. So, there is a chance to view the wrong data, that’s why we are using the **Explicit Lock**

If we apply the explicit lock on the data, other users can’t manipulate the data and view the data

We can apply explicit lock by using the method

**SELECT…FOR UPDATE OF..**

**D. No wait**

It will avoid unnecessary waiting time

**Example:**

Declare

Cursor emp\_cur is select \* from emp where deptno=10 for update of sal nowait;

Begin

For emp\_rec in emp\_cur loop

Update emp set sal=13000 where empno=emp\_rec.empno;

End loop;

End;

It will display error message like “another user is doing some operations on this table”. When we manipulate the same data in another session

**Where Current of:**

By using this option we can access the rows which are being accessed by the cursor.

**E.g.:**

Declare

Cursor emp\_cur is select \* from emp where deptno=10 for update of sal;

Begin

For emp\_rec in emp\_cur loop

update emp set sal=3000 where current of emp\_cur;

End loop;

End;

**Implicit Cursor:**

Open and type the program in one session:

Declare

Cursor emp\_cur is select \* from emp where deptno=10;

Begin

For emp\_rec in emp\_cur loop

Update emp set sal=13000 where empno=emp\_rec.empno;

End loop;

End;

Open the same program in another session and run it, System will hang out

To avoid this use **Explicit Lock** and do the same thing

**Explicit Cursor:**

Declare

Cursor emp\_cur is select \* from emp where deptno=10 for update of sal;

Begin

For emp\_rec in emp\_cur loop

Update emp set sal=13000 where empno=emp\_rec.empno;

End loop;

End;

**It will avoid manipulating the data and accessing the data**

**Composite Data types**

**1. Records**

**2. Collections**

**Records:**

It will allow you to group the variables as a unit, but all the variables may be same data type or different data type.

**Example program:**

declare

Type emp\_type is record(v\_empno number,v\_ename varchar2(20),v\_sal number(20));

V\_emp emp\_type;

Begin

Select empno,ename,sal into v\_emp from emp where empno=101;

***Dbms\_output.put\_line***(v\_emp.v\_empno||v\_emp.v\_ename||v\_emp.v\_sal);

End;

**Record Attributes:**

**%ROWTYPE:**

It is a record type and will contain all the columns data types and sizes of a row dynamically

**Example 1:**

declare

v\_emp emp%rowtype;

begin

select empno,ename,sal into v\_emp.empno,v\_emp.ename,v\_emp.sal from emp where empno=101;

***dbms\_output.put\_line***(v\_emp.empno||v\_emp.ename||v\_emp.sal);

end;

**Example 2:**

declare

v\_emp emp%rowtype;

begin

select \* into v\_emp from emp where empno=101;

***dbms\_output.put\_line***(v\_emp.empno||v\_emp.ename||v\_emp.sal||v\_emp.deptno);

end;

**%TYPE:**

By using this we can assign columns or variable’s datatype or size dynamically to another variable

**Example program**

Declare

V\_empno number;

V\_ename emp.ename%type;

V\_sal number;

V\_job v\_ename%type;

Begin

Select empno,ename,sal,job into v\_empno,v\_ename,v\_sal,v\_job from emp where empno=101;

***Dbms\_output.put\_line***(v\_empno||v\_ename||v\_sal||v\_job);

End;

**2. Collections**

It allows you to group the variables as a unit but all the variables should have same Data type.

Types of Collections:

* INDEX BY TABLE
* NESTED TABLE
* VARRAY

**Index by table:**

Eg.1

declare

type emp\_table\_type is table of emp%rowtype;

indexby binary\_integer;

v\_emp\_type emp\_table\_type;

*---v\_count number:=7369;*

begin

for i in 7369..7369 loop

select \* into v\_emp\_type(i) from emp where empno=i;

***dbms\_output.put\_line***(v\_emp\_type(i).empno||v\_emp\_type(i).ename);

end loop;

end;

Eg.2

declare

type emp\_table\_type is table of emp%rowtype;

indexby binary\_integer;

v\_emp\_type emp\_table\_type;

begin

for i in 1..7 loop

select \* into v\_emp\_type(i) from emp where empno=i;

end loop;

for i in v\_emp\_type.first..v\_emp\_type.last loop

***dbms\_output.put\_line***(v\_emp\_type(i).empno||v\_emp\_type(i).ename);

end loop;

end;

**Nested**

declare

type job\_type is table of emp.job%type;

v\_offices job\_type;

begin

v\_offices:=job\_type('clerk');

for i in 1..v\_offices.count loop

***dbms\_output.put\_line***(v\_offices(i));

end loop;

end;

**VARRAY:**

declare

type job\_type is varray(20) of emp.job%type;

v\_offices job\_type;

begin

v\_offices:=job\_type('SALESMAN','CLERK','MANAGER');

for i in 1..v\_offices.count loop

***dbms\_output.put\_line***(v\_offices(i));

end loop;

end;

**Dynamic SQL**

* PL/SQL doesn’t support DDL statements directly. So, we can execute DDL statements in PL/SQL program by using Dynamic SQL**(execute immediate)**
* We can use **execute immediate** to execute DDL statements as well as DML, DQL statements also.

e.g.:

begin

execute immediate 'create table xyz(a number)';

end;

**SQL Concept:**

**Analytic Functions:**

**1. Cube**: It is an extension to group by clause.

It will return all the possible different combinations along with grand table.

E.g.: **CUBE (A,B,C)**

A

B

C

A B

B C

C A

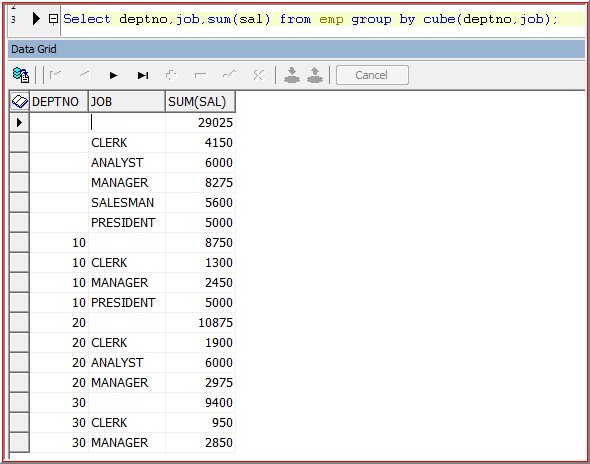
A B C

B C A

C A B

**EXAMPLE:**

**Select deptno,job,sum(sal) from emp group by cube(deptno,job);**

****

**2. Rollup:** It is an extension to group by clause

It will return the output in hierarchical way along with grand total

e.g:

**rollup(a,b,c)**

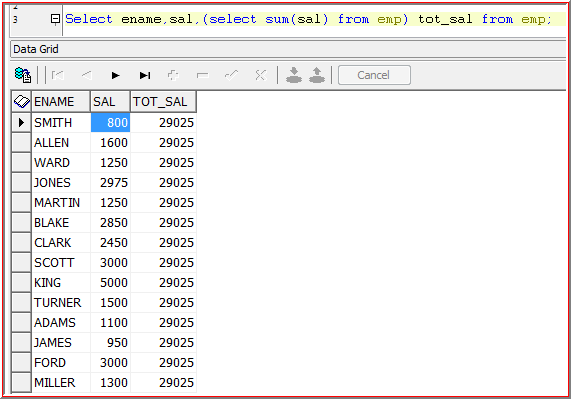
a

a b

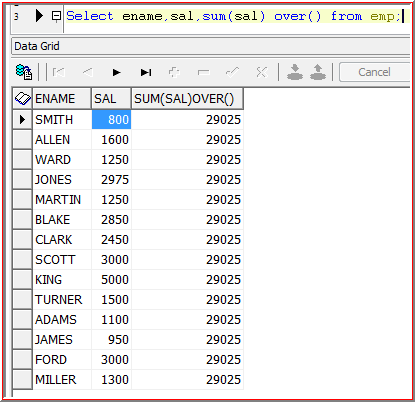
a b c

**3. Over:**

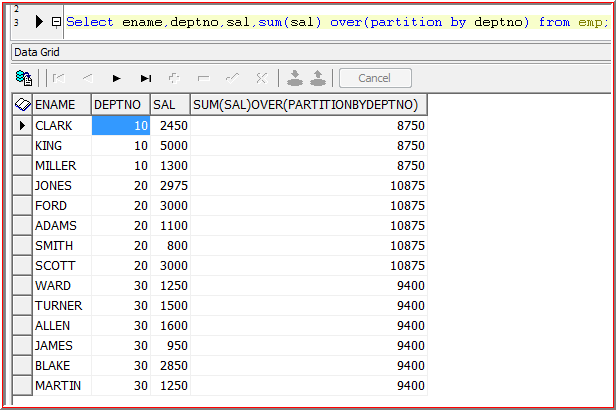
Select ename,sal,(select sum(sal) from emp) tot\_sal from emp;



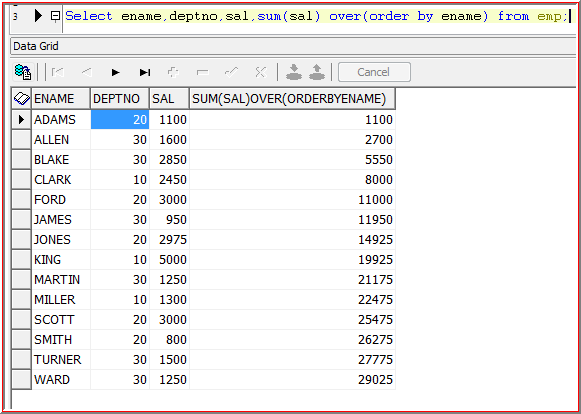
Select ename,sal,sum(sal) over() from emp;



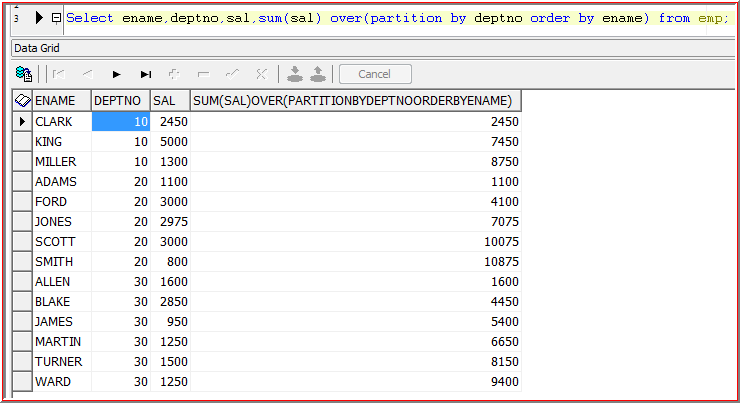
Select ename,deptno,sal,sum(sal) over(partition by deptno) from emp;



Select ename,deptno,sal,sum(sal) over(order by ename) from emp;



Select ename,deptno,sal,sum(sal) over(partition by deptno order by ename) from emp;



**Cursors**

**Ref Cursor:**

* In normal cursors once we define the select statement for a cursor, later we can’t change that select statement for the cursor. So, it is called as “**static cursor”**

But by using **“Ref Cursor”** we can change the select statement dynamically. So, it called as “**Dynamic Cursor”**

* Ref cursor is also called as cursor variable(cursor variable is variable with the data type of cursor)

**Types of Ref Cursors:**

**1. Strong Ref Cursor:** Here we know that the structures of a cursor i.e. we know what are columns we are retrieving from that cursor.

**Syntax:**

**Type ref\_cur\_type is ref cursor; -----EMP%rowtype**

**ref\_cur ref\_cur\_type;**

**2. Weak ref cursor:** Here we know the structure of cursor i.e. we don’t know what the columns are retrieving from that cursor.

**Syntax:**

**Type ref\_cur\_type is ref cursor;**

**ref\_cur ref\_cur\_type;**

**3. Static Cursor Example :**

declare

cursor emp\_cur is select \* from emp;

v\_emp emp%rowtype;

cursor dept\_cur is select \* from dept;

v\_dept dept%rowtype;

begin

for emp\_rec in emp\_cur loop

***dbms\_output.put\_line***(emp\_rec.empno||emp\_rec.ename);

end loop;

***dbms\_output.put\_line***('---------------------------');

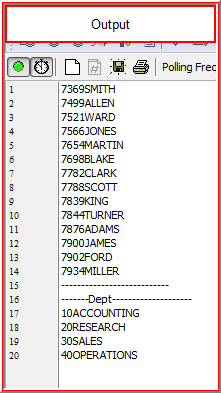
***dbms\_output.put\_line***('-------'||'Dept'||'--------------------');

for dept\_rec in dept\_cur loop

***dbms\_output.put\_line***(dept\_rec.deptno||dept\_rec.dname);

end loop;

end;



**4. Dynamic Cursor:**

**Example program**

declare

type ref\_cur\_type is ref cursor;

ref\_cur ref\_cur\_type;

v\_emp emp%rowtype;

v\_dept dept%rowtype;

begin

open ref\_cur for select \* from emp;

loop

fetch ref\_cur into v\_emp;

exit when ref\_cur%notfound;

dbms\_output.put\_line(v\_emp.empno||v\_emp.ename);

end loop;

close ref\_cur;

***dbms\_output.put\_line***('============'||'Dept'||'===============');

open ref\_cur for select \* from dept;

loop

fetch ref\_cur into v\_dept;

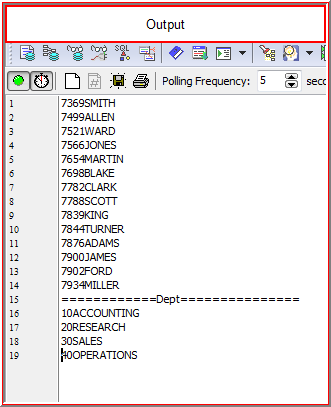
exit when ref\_cur%notfound;

***dbms\_output.put\_line***(v\_dept.deptno||v\_dept.dname);

end loop;

close ref\_cur;

end;



**Bulk Processing:**

* Bulk process used to fetch the batches of data at a time rather than a row by row
* It enables PL/SQL to retrieve a batch of data to process in one pass

Or

* It enables PL/SQL to send a batch of data to process in one pass

**Example:**

declare

type dba\_type is table of dba\_objects.object\_name%type index by binary\_integer;

v\_do\_type dba\_type;

v\_start\_time pls\_integer:=***dbms\_utility.get\_time***;

v\_elapsed pls\_integer;

begin

select object\_name bulk collect into v\_do\_type from dba\_objects where rownum<=50000;

for i in 1..v\_do\_type.count loop

***dbms\_output.put\_line***(v\_do\_type(i));

end loop;

v\_elapsed:=***dbms\_utility.get\_time***.v\_start\_time;

***dbms\_output.put\_line***(v\_elapsed/100);

end;