-30 [Aug] jul

PL/SQL

PL/SQL is a procedural language extension for SQL.
It is the combination of procedural, data manupulation language.
Oracle 6.0 introduced PL/SQL.

To view version of pl/sql :

```
sql> select * from v$version;
```

Basically pl/sql is a block structured :

```
Declare [optional]
        variable declarations,
        cursors,
        undefined exceptions.
Begin [ mandatory ]
        DML, TCL
        select ---.--- into ---.---
        Conditional,
        Control statements ;
Exception [optional]
        Handling
        exceptions
End ; [ mandatory ]
```

TUESDAY

JAN 12

3

003-363 WK-01

There are two types of blocks supported by pl/sql :

1. Annoymous blocks
2. Named blocks

1. Annoymous Blocks

These blocks does not have a name & also not stored in database & also we are allowed to call these blocks in another blocks or in client application.

ex. Declare

—.—

—..—

Begin

—..——

—..—

end;

2. Named Blocks

These blocks having a name & also automatically stored in database.
These blocks used by all types of programmers in all applications, these are procedures,

WEDNESDAY
JAN 4

functions, triggers, packages, etc.

4

004-362 WK-01

PL/SQL Datatypes, Variables

I. It supports all sql datatypes (scalar datatypes)

+

boolean datatypes

II. lobs (clob, blob, bfile)

III. Composite datatypes.

IV. ref objects

V. non-pl/sql variables (or) bind variables (or) host variables.

Variable

variable is used to store a single value into memory location.

Syntax for declaring a variable :

variablename datatype (size);

February	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	♦	♦
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	♦	♦

04-01

THURSDAY

JAN 12

5

005-361 WK-01

Generally we are declaring variables in declare section of the pl/sql block.

ex.
```
declare
a number (10);
b varchar2 (10);
```

Storing a value into variable :-

Using assignment operator (:=) we are storing a value into variable.

Syntax : variablename := value;

ex. : a := 60;

Display message (or) variable value

```
dbms_output.put_line ('message');
```

packagename — procedurename

```
dbms_output.put_line (variablename);
```

ex. sql> set serveroutput on;

```
sql> begin
     dbms_output.put_line ('Kadam');
     end;
     /
```

(output) Kadam

ex.
```
declare
a number (10);
begin
a := 60;
dbms_output.put_line (a);
end;
/
```

(output) 60

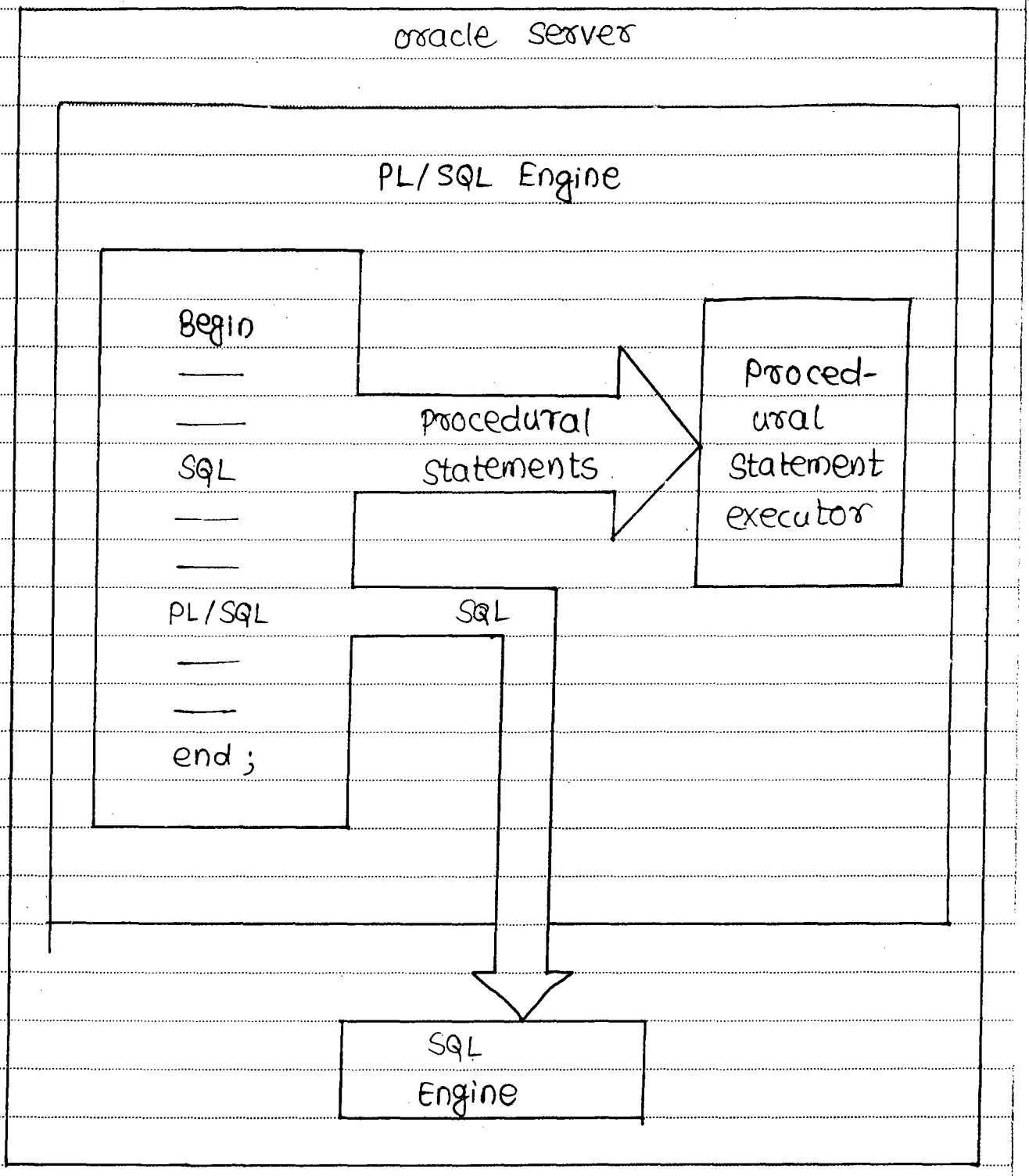
SATURDAY

JAN 12

7

007-359 WK-01

Fig. PL/SQL Architecture



January	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

07-01

SUNDAY

JAN

8

008-358 WK-01

Select —··— into —··— clause

This clause is used to data from table & storing into pl/sql variables.
select —··— into —··— clause always returns single record or value at a time.

Syntax: select col1, col2, ...- into var1, var2,

--- from tablename where condition;

This clause is used in executable section of pl/sql program.

? Write a pl/sql program for user entered empno ; display name of the employee & salary from emp table.

Sol^n

```
declare
v_ename  varchar2(10);
V_sal    number(10);
begin
select ename, sal into v_ename, v_sal
 from emp where empno = &no;
dbms_output.put_line(v_ename||' '||v_sal);
end;
/
```

MONDAY

JAN 12

9

009-357 WK-02

(output) Enter value for no : 7902

FORD 3600

— 31 [Aug] Jul

Note

Whenever we are using not null, constant clauses in variable declaration we must assign the value at the time of declaration only.

ex.
```
declare
a number(10) not null := 60;
b constant number(10) := 5;
begin
dbms_output.put_line(a);
dbms_output.put_line(b);
end;
/
```

(output)
```
60
5
```

Note

We can also use default keyword in place of assignment operator in declare section of the pl/sql block.

January	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

ex.
```
declare
a number (10) default 30;
begin
dbms_output.put_line (a);
end;
/
```

(output) 30

? Write a pl/sql program to display maximum salary from emp table.

Soln
```
declare
v_sal number (10);
begin
select max (sal) into v_sal from
emp;
dbms_output.put_line (v_sal);
end;
/
```

(output) 6100

Note

We are not allow to use group functions, decode conversion functions in pl/sql expressions.

WEDNESDAY
JAN 12

11

011-355 WK-02

But we are using number, character, date functions in pl/sql expressions.

ex.
```
declare
a   number (10);
b   number (10);
c   number (10);
begin
a := 40;
b := 30;
c := greatest (a,b);
dbms_output.put_line (c);
end;
/
```

(output) 40

Variable Attributes

Variable attributes are used in place of datatypes in variable declaration or procedure parameter declaration in pl/sql block.

There are two types of variable attributes supported by pl/sql.

1. Column level attributes
2. Row level attributes

THURSDAY
JAN 12
12
012-354 WK-02

1. Column Level Attributes

In this method we are defining attributes for individual columns.
This attribute is represented using %type.

syntax: variablename tablename.columnname%type;

Whenever we are using this attribute pl/sql runtime engine allocates memory for the variables corresponding to the columns in database tables.

ex.
```
declare
v_ename     emp.ename%type;
v_sal       emp.sal%type;
begin
select ename, sal into v_ename, v_sal
from emp
where empno = &no;
dbms_output.put_line (v_ename || ' ' ||
                      v_sal);
end;
/
```

FRIDAY

JAN '12

13

013-353 WK-02

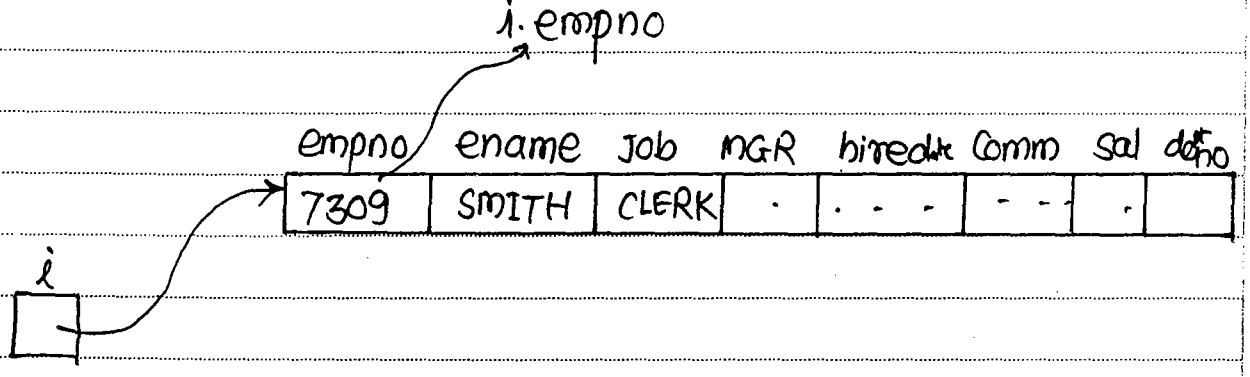
2. Row Level Attribute

In this method a single variable represents all different datatypes in entire row in a table thats why this variable is also called as record type variable. This variable is represented using %rowtype. It is also same as structure in C language.

Syntax : Variablename tablename %rowtype;

ex. Declare

i emp%rowtype;



ex. declare

i emp%rowtype;

begin

select ename, sal, hiredate, deptno

```
into  i.ename, i.sal, i.hiredate,
i.deptno  from  emp
where  empno = &no;
dbms_output.put_line ( i.ename || ' ' || i.sal ||
 ' ' || i.hiredate || ' ' || i.deptno );
end ;
/
```

ex.

```
declare
i  emp%rowtype;
begin
select * into i from emp where
empno = &no;
dbms_output.put_line ( i.ename || ' ' || i.sal
  || ' ' || i.hiredate || ' ' || i.deptno);
end;
/
```

Conditional Statements (if)

I. if
II. if else
III. elsif

SUNDAY

JAN 12

15

015-351 WK-02

I. if Syntax:

```
if condition then
    Statements;
end if;
```

II. if else syntax:

```
if condition then
    Statements;
else
    Statements;
end if;
```

III. elsif To check more number of conditions we are using elsif.

Syntax :

```
if condition 1 then
    Statements;
elsif condition 2 then
    Statements;
elsif condition 3 then
    Statements;
— " — .. —
else
    Statements;
end if;
```

January 2012	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

— 01 Aug —

ex.
```
declare
v_deptno number (10);
begin
select deptno into v_deptno from emp
where deptno = &no;
if v_deptno = 10 then
dbms_output.put_line ('ten');
elsif v_deptno =20 then
dbms_output.put_line ('twenty');
elsif v_deptno =30 then
dbms_output.put_line ('thirty');
else v_deptno =40
dbms_output.put_line ('others');
end if;
end;
/
```

(output)
```
enter value for no : 40
others
enter value for no: 90
error
```

Note

If a pl/sql block contains select into clause & also if requested data not avaliable in a table oracle server returns an error ora-1403 : no data found

Note

In the above program if we are using emp table in place of dept table & also if requesting data oracle server returns an error because whenever select ... into ... clause try to return more than one value oracle server returns an error

ora-1422 : exact fetch returns more than requested number of rows.

Note

when a pl/sql block contains dml statements & also if requested data not avaliable in a table oracle server does not return any error.

ex. sql > begin
delete from emp where ename='java';
end;
/

(output) PL/SQL procedure successfully completed.

To handle these types of blocks we are using implicit cursor attributes.

Control Statements (loops)

There are three types of loops supported by pl/sql.

1. Simple loop
2. While loop
3. For loop

1. Simple loop

Here body of the statement is executed repeatedly.

Syntax :
```
loop
    statements;
end loop;
```

This loop is also called as infinite loop.

ex.
```
begin
loop
dbms_output.put_line ('Welcome');
end loop;
end;
/
```

THURSDAY

JAN 12

19

019-347 WK-03

To exit from infinite loop we are using following two methods

(i) Method 1

Syntax: exit when trueCondition;

ex.
```
declare
n number (10) := 1 ;
begin
loop
dbms_output.put_line (n);
exit when n>=10;
n := n+1 ;
end loop ;
end ;
/
```

(ii) Method 2 (using if)

Syntax:
```
if condition then
    exit ;
end if ;
```

ex.
```
declare
n number (10) := 1 ;
begin
loop
dbms_output.put_line (n);
if n>=10 then
```

January 2012	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
exit;
end if;
n := n + 1 ;
end loop;
end;
/
```

2. While loop

Here body of the statement executed repeatedly until condition is false.

Syntax:
```
While (condition)
    loop
        statements;
    end loop;
```

ex.
```
declare
n number (10) := 1 ;
begin
while (n>=10)
loop
dbms_output.put_line (n);
n:= n+1 ;
end loop;
end;
/
```

SATURDAY
JAN 12
21
021-345 WK-03

3. For loop

Syntax: for indexvariablename in
lowerbound .. upperbound
loop
statements;
end loop;

ex.
```
declare
n number(10);
begin
for n in 1..10
loop
dbms_output.put_line(n);
end loop;
end;
/
```

ex.
```
begin
for n in 1..10
loop
dbms_output.put_line(n);
end loop;
end;
/
```

ex.
```
begin
for n in reverse 1..10
loop
dbms_output.put_line (n);
end loop;
end;
/
```

Bind Variable

These variables are Session variables, these variables are created at host environment thats why these variables are also called as host variables.

These variables are non-pl/sql, but these variables are used in pl/sql to execute procedures having out parameters.

Step 1 Creating a bind variable.

Syntax: sql> variable~~name~~ variablename datatype;

Step 2 Using bind variable.

Syntax: :variablename;

Step 3 Display value from bind variable.

Syntax: sql> print variablename;

MONDAY
JAN 12
23
023-343 WK-04

ex. sql> variable g number;

```
sql> declare
     a number (10) := 500;
     begin
     :g := a/2;
     end;
     /

sql> print g;
```

(output)

G
250

— 02 Aug —

Cursor

Cursor is a private sql memory area which is used to process multiple records & also this is a record by record process.
There are two types of static cursors supported by oracle.

1. Implicit Cursor
2. Explicit cursor

TUESDAY
JAN 12
24
024-342 WK-04

1. Implicit Cursors

SQL statements returns a single record is called implicit cursors.
Implicit cursors are simple pl/sql programs which contains select into clause or dml statements.
When pl/sql block contains select ... into clause Oracle server creates a memory area and returns a single record.
This memory area is also called as sql area.
When a pl/sql block contains dml statements & also sql area returns multiple records.
In this case these all records are processed at a time this is also called as implicit cursor memory area or context area.

ex.
```
declare
v_ename  varchar2(10);
v_sal    number(10);
begin
select ename, sal into v_ename, v_sal
 from emp where empno = &no;
dbms_output.put_line (v_ename||' '||v_sal);
end;
/
```

(output) Enter value for no : 7902

FORD 3000

2. Explicit Cursors

for sql statement returns a multiple records is called explicit cursors.
Explicit cursor memory area is also called as active set area.
In this explicit cursors we are storing multiple records & also these records are controlled by database developers explicitly but database developers can not control implicit cursor memory area record.

Explicit Cursor Lifecycle

i. Declare
ii. Open
iii. Fetch
iv. close

i. Declare In declare section of the pl/sql block we are defining cursors using following Syntax:

cursor cursorname is select * from tablename where condition;

ii. Open whenever we are opening the cursor then only oracle server fetch data from table into cursor memory area, because whenever we are opening the cursor then only cursor select statements are executed.

Syntax: open cursorname;

This statement is used in executable section of the pl/sql block.

Note

When we are opening the cursor implicitly cursor pointer points to first record in the cursor.

iii. Fetch (Fetching from Cursor) Using fetch statement we are fetching data from cursor into pl/sql variables.

Syntax: Fetch cursorname into variable1, variable2, ... ;

iv. Close when we are closing the cursor all resources allocated from cursor memory area automatically released.

syntax: close cursorname ;

ex.
```
declare
cursor c1 is select ename, sal from emp;
v_ename varchar2 (10);
V_Sal number (10);
begin
open c1 ;
fetch c1 into v_ename, v_sal ;
dbms_output.put_line (v_ename||' '||v_sal);
fetch c1 into v_ename, v_sal ;
dbms_output.put_line (v_ename||' '||v_sal);
close c1 ;
end ;
/
```

(output) SMITH 2000
ALLEN 900

SATURDAY
JAN 12
28
028-338 WK-04

Explicit Cursor Attributes

Every explicit cursor having following four attributes :

1) %.notfound
2) %. found
3) %. isopen
4) %. rowcount

All these cursor attributes using alongwith cursorname only.

Syntax: cursorname %. attributename

Except %.rowcount all other cursor attribute records boolean value returns either true or false whereas %.rowcount returns number datatype.

1) %.notfound This attribute returns true when cursor does not have data after fetching records from cursor.

? Display all the records from emp table using explicit cursor lifecycle ?

SUNDAY

JAN 12

29

029-337 WK-04

Soln

```
declare
cursor C1 is select ename, sal
from emp;
v_ename varchar2 (10);
v_sal number (10);
begin
open C1;
loop
fetch C1 into v_ename, v_sal;
exit when C1 % notfound;
dbms_output.put_line (v_ename
                      ||' '|| v_sal);
end loop;
close C1;
end;
/
```

—03 Aug—

? Write a pl/sql cursor program to display first five highest salary employees from emp table using %rowcount attribute.

Soln

```
declare
cursor C1 is select sal from emp order
  by sal desc;
v_sal number(10);
 begin
```

January	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
open c1;
loop
fetch c1 into v_sal;
dbms_output.put_line (v_sal);
exit when c1%rowcount=5;
end loop;
close c1;
end;
/
```

? write a pl/sql cursor program to display even number of rows from emp table using %rowcount attribute.

Soln

```
declare
cursor c1 is select * from emp;
i emp%rowtype;
begin
open c1;
loop
fetch c1 into i;
exit when c1%notfound;
if mod (c1%rowcount,2)=0 then
dbms_output.put_line (i.ename||' '||i.sal);
end if;
end loop;
close c1;
end;
/
```

TUESDAY

JAN 12

31

031-335 WK-05

Whenever we are using cursor automatically pl/sql runtime engine creates four variables.

These variables stores a single value at a time these variables are identified through cursor attribute.

c%notfound	c1%found	c1%isopen	c1%rowcount
true	false	true	2

%rowcount

This cursor attribute always returns number datatype i.e. it stores number of record's number fetched from the cursor.

```
ex:  declare
     Cursor c1 is select * from emp;
     i emp%rowtype;
     begin
     open c1;
     fetch c1 into i;
     dbms_output.put_line (i.ename||' '|| i.sal);
     fetch c1 into i;
     dbms_output.put_line (i.ename||' '|| i.sal);
     dbms_output.put_line('no. of records
          fetched from cursor is||' '||c%rowcount);
```

January	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

ACTION PLAN

FEBRUARY '12

WK	Mon	Tue	Wed	Thu	Fri	Sat	Sun
5			1	2	3	4	5
6	6	7	8	9	10	11	12
7	13	14	15	16	17	18	19
8	20	21	22	23	24	25	26
9	27	28	29				

FEB

MAR

APR

OBJECTIVES THIS MONTH

DATE	DESCRIPTION	REMARKS

JANUARY '12

M	T	W	T	F	S	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

2012

MARCH '12

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

```
Close C1;
end;
/
```

(output)

```
SMITH   1800
ALLEN   700
no. of records fetched from cursor
is 2
```

ex.

```
declare
cursor C1 is select * from emp;
i emp%rowtype;
begin
open C1;
loop
fetch C1 into i;
exit when C1%notfound;
if i.sal>2000 then
dbms_output.put_line (i.sal||' '||'highsal');
end if;
end loop;
close C1;
end;
/
```

(output) SCOTT highsal
KING highsal
ADAM highsal
FORD highsal
MILLER highsal

Note

These explicit cursors are also used to transfer data from one oracle table to another oracle table.

```
sql> Create table target (name varchar2(10)
                          sal number(10));

sql> declare
     cursor c1 is select * from emp
     where sal > 2000;
     i emp%rowtype;
     begin
     open c1;
     loop
     fetch c1 into i;
     exit when c1%notfound;
     insert into target values (i.ename, i.sal);
     end loop;
     close c1;
     end;
     /
```

FRIDAY

FEB '12

3

034-332 WK-05

sql> select * from target;

Eliminating Explicit Cursor Lifecycle

Using cursor for loops we are eliminating explicit cursor lifecycle i.e when we are using cursor for loops internally pl/sql runtime engine uses open, fetch, close statements.

Cursor for loop

Syntax: for indexvariable in cursorname
loop
statements;
end loop;

This is also called as shortcut method of the cursor. This loop is used in executable section of the pl/sql block.

Note

Here cursor for loop index variable internally behaves like a "record type" variable" (i.e %rowtype)

ex.
```
declare
cursor c1 is select * from
emp;
begin
for i in c1
loop
dbms_output.put_line (i.ename||' '||i.sal);
end loop;
end;
/
```

Note

We can also eliminate declare section of the cursor using for loop in this case we are using select statements in place of cursorname in cursor for loop.

ex:
```
begin
for i in (select * from emp)
loop
dbms_output.put_line (i.ename||' '||i.sal);
end loop;
end;
/
```

SUNDAY

FEB 12

5

036-330 WK-05

-16 Aug-

Parametrized Cursors

we can also pass parameters to the cursors same like a subprograms in parameters.

These type of cursors are also called as parametrized cursors.

Syntax: Cursor cursorname (parametername datatype) is select * from tablename where columnname = parametername ;

Syntax: Open cursorname (actual parameters);

ex.
```
declare
Cursor C1 ( p-deptno number ) is
select * from emp where
deptno = p_deptno ;
i emp%rowtype ;
begin
open C1 (10) ;
loop
fetch C1 into i ;
exit when C1%notfound ;
dbms_output.put_line (i.ename||' '||i.deptno);
end loop ;
close C1 ;
```

```
end;
/
```

Note

Whenever we are passing formal parameter to the cursor, procedure, function then we can not specify datatype size in formal parameter declaration.

? Write a pl/sql parametrized cursor program to display employees working as managers or analyst from emp table & also produce this report statically.

```
sql> declare
cursor c1 (p_job varchar2) is select
* from emp where job = p_job;
i emp%rowtype;
begin
Open c1 ('MANAGER');
dbms_Output.put_line ('Employee working as
managers');
loop
fetch c1 into i;
exit when c1%notfound;
dbms_output.put_line (i.ename);
end loop;
Close c1;
```

```
open C1 ('ANALYST');
dbms_output.put_line ('Employee
    working as analyst');
loop
fetch C1 into i;
exit when C1%.notfound;
dbms_output.put_line ( i.ename);
end loop;
close C1;
end;
/
```

Note

Before we are reopening the cursor we must close the cursor properly otherwise oracle server returns an error

Cursor already open

Note

If we are try to close the cursor without opening the cursor oracle server returns an error

ora-1001 : invalid cursor

ex.

```
declare
cursor C1 ( p_deptno number) is
select * from emp where
deptno= p_deptno;
```

```
begin
for i in c1(10)
loop
dbms_output.put_line (i.ename||' '|| i.deptno);
end loop;
end;
/
```

Note

we can also pass default values using either default or ":=" operator.

Syntax: parametername datatype default actualvalue

Syntax: parametername datatype := actualvalue

ex.

```
declare
cursor c1 (p_deptno number default 20)
is select * from emp where
 deptno = p_deptno;
begin
for i in c1()
loop
dbms_output.put_line (i.ename||' '|| i.deptno);
end loop;
end;
/
```

THURSDAY

FEB 12

9

040-326 WK-06.

? write a pl/sql program to retrive different deptno from dept table using a cursor & also pass these deptno.'s into another parameterized cursor to retrive employee details for the corresponding deptno.

Soln

```
declare
cursor c1 is select deptno from dept;
Cursor c2 (p_deptno number) is
select * from emp where deptno=
 p_deptno ;
begin
 for i in c1
loop
dbms_output.put_line ('deptno is ' ||' '|| 
    j.deptno);
for j in c2 (i.deptno)
loop
dbms_output.put_line ( j.ename ||' '|| j.sal
          ||' '|| j.deptno);
end loop;
end loop;
end;
/
```

—17 Aug—

? Write a pl/sql cursor program to modify salaries of employees from emp table based

on following conditions :

i) If job = 'CLERK' then increment Sal 100

ii) If job = 'SALESMAN' then decrement sal 200

iii) If job = 'ANALYST' then increment sal 100

Soln

```
declare
cursor c1 is select * from emp;
i emp%rowtype;
begin
open c1;
fetch c1 into i;
exit when c1%notfound;
if i.job = 'CLERK' then
update emp set sal = i.sal + 100 where
empno = i.empno;
elsif i.job = 'SALESMAN' then
update emp set sal = i.sal - 200 where
empno = i.empno;
elsif i.job = 'ANALYST' then
update emp set sal = i.sal + 100 where
empno = i.empno;
end if;
end loop;
close c1;
end;
/
```

SATURDAY
FEB 12

11

042-324 WK-06

Where current of, for update clauses. used in cursors

(or)

Update, Delete statements used in cursors

Generally when we are using update, delete statements automatically locks are established. If you want to establish locks before update, delete statements then we are using cursor locking mechanism in all database systems.

In this case we must specify for update clause in cursor definition.

Syntax: Cursor Cursorname is Select * from tablename where condition for update [of columnname] [Nowait];

If you are specifying for update clause also oracle server does not establish the lock i.e whenever we are opening the cursor then only oracle server internally uses exclusive locks.

After processing we must release the locks using commit.

where current of this clause is used in update, delete statements used in cursor.

Syntax: update tablename set columnname =

February	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	♦	♦
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	♦	♦

newvalue where current of cursorname ;

Syntax: Delete from tablename where current of cursorname;

where current of clause uniquely identifying a record in each process because where current of clause internally uses rowid.

Note

Whenever we are using where current of clause we must use for update clause.

Note

where current of clause used to update or delete latestly fetched row from the cursor.

ex.
```
declare
cursor C1 is select * from emp
for update;
i emp%.rowtype ;
begin
open C1;
loop
fetch C1 into i;
exit when C1%.notfound;
```

```
if i.job = 'CLERK' then
update emp set sal = i.sal + 100
where current of C1 ;
end if ;
end loop ;
commit ;
close c1 ;
end;
/
```

? Write a pl/sql cursor program using cursor locking mechanism raise 5% salary who are working under king as manager from emp table.

sol^n

```
declare
cursor c1 ( p_mgr number ) is select
sal from emp where mgr = p_mgr for
update ;
v_mgr number (10) ;
begin
select empno into v_mgr from emp
where ename = 'KING' ;
for i in C1 ( v_mgr)
loop
update emp set sal = i.sal * 1.05
where current of C1 ;
end loop;
```

```
Commit ;
end ;
/
```

Implicit Cursor Attributes

When a pl/sql block contain pure DML statements & also contains select into clause oracle server internally creates an memory area, this memory area is also called as context area or sql area or implicit cursor. This sql area allocates four variables. These variables are identified using attributes i.e. implicit cursor having following four attributes

1. sql%notfound
2. sql%found
3. sql%isopen
4. sql%rowcount

Here sql%isopen always returns false whereas sql%notfound, sql%found returns boolean value either true or false & also sql%rowcount returns no. datatype.

ex.
```
begin
delete from emp where ename='abcd';
if sql%found then
dbms_output.put_line ('your record deleted');
```

WEDNESDAY
FEB 12
15
046-320 WK-07

```
end if;
if sql%notfound then
dbms_output.put_line ('your record
  does not exist');
end if;
end;
/
```

ex.
```
begin
update emp set sal = sal + 100 where
job = 'CLERK';
dbms_output.put_line (' affected number
 of clerks are:' || ' ' || sql%rowcount);
end;
/
```

(Output) ⇒ affected number of clerks are: 4

— 18 Aug —

Exception

Exception is an error occured during runtime. Whenever runtime error occure use an appropriate exception name in exception handler under exception section.

There are three types of exceptions supported by oracle

1. Predefined exceptions

THURSDAY

16

047-319 WK-07

2. Userdefined exceptions

3. Unnamed exceptions

1. Predefined Exceptions

Oracle defined 20 predefined exceptions names for regularly occured errors.
Whenever these errors occure use an appropriate exceptionname in exception handler.

Syntax : when predefinedexceptionname1 then
statements;
when predefinedexceptionname2 then
statements;
.....
.....
when others then
statements;

Predefined exceptions

i) no_data_found
ii) too_many_rows
iii) zero_divide
iv) invalid_cursor
v) Cursor_already_open
vi) invalid_number
vii) value_error

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FRIDAY

FEB 12

17

048-318 WK-07

i) no_data_found

When a pl/sql block contains select ---- into clause & also if requested data not avaliable in a table oracle server returns an error ora-1403: no data found. To handle this error we are using no_data_found exceptionname.

ex.
```
declare
    v_ename   varchar2(10);
    v_sal     number(10);
    begin
    select ename, sal into v_ename, v_sal
    from emp where empno = &no;
    dbms_output.put_line ( v_ename || ' ' || v_sal );
    exception
    when no_data_found then
    dbms_output.put_line ('employee does not exist');
    end;
    /
```

(Output) ⇒ Enter value for no: 7902
FORD 4000

⇒ /
Enter value for no: 9999
employee does not exist.

ii) too_many_rows

when a select ---- into clause try to return more than one record or more than one value then oracle server returns an error ora-1422 : exact fetch returns more than requested number of rows.
To handle this error we are using too_many_rows exceptionname.

```
ex:  declare
     v_sal number (10);
     begin
     select sal into v_sal from emp;
    dbms_output.put_line (v_sal);
     exception
     when too_many_rows then
    dbms_output.put_line ('not to return more
     rows');
     end;
     /
```

iii) zero_divide

ora-1476 : divisor is equal to zero

```
ex:  declare
     a number (10);
```

```
b   number(10);
c   number(10);
begin
a:=5;
b:=0;
c:= a/b;
dbms_output.put_line(c);
exception
when  zero_divide then
dbms_output.put_line('b cannot be zero');
end;
/
```

iv) invalid_cursor

Whenever we are performing invalid operations on the cursor oracle server returns an error i.e. if you are try to close the cursor without opening the cursor oracle server returns an error ora-1001 : invalid cursor.

To handle this cursor we are using invalid_cursor exceptionname.

ex:

```
declare
cursor c1 is select * from emp;
i  emp%rowtype;
begin
loop
```

```
fetch c1 in i ;
exit when c1 %.notfound ;
dbms_output.put_line (i.ename||' '||
                      i.sal );
end loop;
close c1;
exception
when invalid_cursor then
dbms_output.put_line ('First we must open
                       the cursor');
end;
/
```

v) cursor_already_open

when we are try to reopen the cursor without closing the cursor oracle server returns an error ora-6511: cursor already open To handle this error we are using cursor_already_open exceptionname.

ex:

```
declare
cursor c1 is select * from emp;
i emp %.rowtype;
begin
open c1;
loop
fetch c1 into i;
```

```
exit when c1%notfound;
dbms_output.put_line(i.ename||' '||i.sal);
end loop;
open c1
exception
when cursor_already_open then
dbms_output.put_line('before reopen
 we must close the cursor');
end;
/
```

invalid_number, value_error

Whenever we are try to convert string type to number type oracle server return errors invalid_number, value error.

vi) invalid_number

When a pl/sql block contains sql statements & also if you are try to convert string type to number type oracle server returns an error, ora-1722: invalid number.
To handle this error we are using invalid_error exceptionname.

ex:
```
begin
insert into emp (empno, sal) values (1, 'abc');
```

```
exception
when invalid_number then
dbms_output.put_line (' insert proper
 data only' );
end;
/
```

vii) value_error

When a pl/sql block contains pl/sql statements & also if you are try to convert string type to number type oracle server returns an error,
ora-6502 : numberic or value error : Character to number conversion error.
To handle this error we are using value_error exceptionname.

ex:
```
declare
z number(10);
begin
z = '&x' + '&y';
dbms_output.put_line (z);
exception
when value_error then
dbms_output.put_line ('enter proper data
only' );
end;
/
```

THURSDAY

FEB 12

23

054-312 WK-08

—19 Aug—

Note

Whenever we are try to store large amount of data than the specified datatype size in variable declaration oracle server returns an error ora-6502 : numeric or value error : character string buffer too small. To handle this error we are using value_error exceptionname.

ex:

```
declare
z varchar2(3);
begin
z := 'abcd';
dbms_output.put_line (z);
exception
when value_error then
dbms_output.put_line ('invalid string length');
end;
/
```

Exception propagation

Exceptions also raised in either declare section or executable section or in exception section.

IF exceptions are raised in executable section those exceptions are handled using either inner blocks or an outer block.

February 2012	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	♦	♦
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	♦	♦

Whereas if exceptions are raised in declare section or in exception section those exceptions are handled using outer blocks only.

```
ex:  begin
     declare
     z  varchar2(3) = 'abcd';
     begin
     dbms_output.put_line (z);
     exception
     when value_error then
     dbms_output. put_line ('invalid string length');
     end;
     exception
     when value_error then
     dbms_output.put_line ('invalid string length
     handled using outer block');
     end;
     /
```

2. Userdefined Exceptions

We can also create our own exception names & also raise whenever necessary.

These types of exceptions are called user defined exception.

SATURDAY

FEB 12

25

056-310 WK-08

Step I. declare

Step II. raise

Step III. handling exception

Step I. declare

In declare section of the pl/sql program we are defining our own exceptionname using exception type.

Syntax: userdefinedexceptionname exception ;

ex: declare
a exception ;

Step II. raise

whenever necessary raise user defined exception either in executable section or in exception section, in this case we are using raise keyword.

Syntax: raise userdefinedexceptionname ;

In this case oracle server only raise that exception.

February	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	♦	♦
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	♦	♦

Step III. handling exception

we can also handle userdefined exceptions as same as predefined exception using predefined handler.

Syntax: when userdefinedexceptionname1 then
statements;
when userdefinedexceptionname2 then
statements;
——.——.——
——.——.——
when others then
statements;

ex:
```
declare
z exception;
begin
if to_char (sysdate, 'DY') = 'SUN' then
raise z;
end if;
exception
when z then
dbms_output.put_line ('my exception raised
today');
end;
/
```

MONDAY

FEB '12

27

058-308 WK-09

ex:

```
declare
v_sal number (10);
z exception;
begin
Select sal into v_sal from emp
where empno = 7902;
if v_sal > 2000 then
raise z;
else
update emp set sal = sal + 100
where empno = 7902;
end if;
exception
when z then
dbms_output.put_line ('salary
already high');
end;
/
```

Note

Generally user defined exceptions are used to transfer control to the number of block without checking each individual condition.

Raising Predefined Exceptions

We can also raise predefined exceptions

February	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	♦	♦
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	♦	♦

using raise statement.

syntax: raise predefinedexceptionname;

ex: declare
cursor C1 is select * from emp where job = 'SW' ;
i emp%rowtype;
begin
open C1;
fetch C1 into i;
if C1%rowcount =0 then
raise no_data_found;
end if;
close C1;
exception
when no_data_found then
dbms_output.put_line ('your job not avaliable');
end;
/

-21 Aug-

Note

Generally user defined exceptions are also used to test exception propagation.

WEDNESDAY

FEB 12

29

060-306 WK-09

Exceptions raised in executable Section

Exceptions are raised in executable section handled using innerblocks or outerblocks.

Using innerblock

```
ex: declare
    z exception;
    begin
    raise z;
    exception
    when z then
    dbms_output.put_line ('handled using
    inner block');
    end;
    /
```

(output) handled using inner block.

Using outerblock

```
ex: declare
    z exception;
    begin
    begin
```

```
raise z;
end;
exception
when z then
dbms_output.put_line ('handled using outer
 block');
end;
/
```

Exceptions raised in declare section

Exceptions raised in declare section must be handled using outerblocks only.

ex:
```
begin
declare
z varchar2 (3)= 'abcd';
begin
dbms_output.put_line ('handled using outer z
 block');
end;
exception
when value_error then
dbms_output.put_line ('handled using
 outer blocks');
end;
/
```

Notes

Exception raised in exception Section

When exceptions are raised in exception section must be handled using outerblock only.

```
ex.  declare
     z1  exception;
     z2  exceptions;
     begin
     begin
     raise  z1;
     exception
     when  z1  then
     dbms_output.put_line ('z1 handled');
     raise  z2;
     end;
     exception
     when  z2  then
     dbms_output.put_line ('z2 handleded using
     outer block only');
     end;
     /
```

ACTION PLAN

MARCH '12

WK	Mon	Tue	Wed	Thu	Fri	Sat	Sun
9				1	2	3	4
10	5	6	7	8	9	10	11
11	12	13	14	15	16	17	18
12	19	20	21	22	23	24	25
13	26	27	28	29	30	31	

MAR

APR

OBJECTIVES THIS MONTH

DATE	DESCRIPTION	REMARKS

FEBRUARY '12

M	T	W	T	F	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29				

APRIL '12

M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

THURSDAY

MAR 12

1

061-305 WK-09

Error Trapping Functions

There are two error trapping functions supported by oracle.

i) sqlcode

ii) sqlerrm

sqlcode returns numbers whereas sqlerrm returns error number with error message.

Sqlcode return values	meaning
0	no errors
negative (-)	oracle errors
100	no data found
1	user defined exception

ex:
```
declare
v_sal number(10);
begin
select sal into v_sal from emp;
dbms_output.put_line (sqlcode);
dbms_output.put_line (sqlerrm);
```

```
end;
/
```

(output) ⇒ 1
user-Defined exception

raise_application_error( )

If you want to display user defined exception messages in more descriptive from then we are using raise_application_error( ) i.e. this function is used to display user defined exception message as same as oracle error displayed format.
This function is used in either executable section or in exception section.

Syntax: raise_application_error (error number, message);
error number: -20000 to -20999
message: upto 512 characters

ex:
```
declare
v_sal number(10);
z exception;
begin
select sal into v_sal from emp where
empno = 7902;
if v_sal > 2000 then
```

```
raise z;
else
update emp set sal=sal+100
where emp=7902;
end if;
exception
when z then
raise_application_error (-20752, 'salary
already high');
end;
/
```

(output) ORA-20752: salary already high

Note

Generally this function is used in triggers because whenever condition is true it raise a message & also it stops invalid data entry according to condition whereas dbms_output.put_line displays plain text message but it does not stops invalid data entry.

-22 Aug-

3. Unnamed Exceptions

If you want to handle other than oracle 20 predefined errors we are using unnamed method.

because oracle define exception names for regularly occured errors.
other than 20 they are not defining exception names.
In this case we are providing exception names & also associate this exceptionname with appropriate error no. using exception_init function.

Syntax: pragma Exception_init (userdefined exceptionname, errornumber);

Here pragma is a compiler directive i.e. at the time of compilation only pl/sql runtime engine associate error number with exceptionname.
This fun is used in declare section of the pl/sql block.

ex:
```
declare
z exception;
pragma exception_init (z, -1400);
begin
insert into emp (ename) values ('Murali');
exception
when z then
dbms_output.put_line ('not to insert nulls');
end;
/
```

MONDAY
MAR 12
5
065-301 WK-10

ex: declare
z exception;
pragma exception_init (z, -2292);
begin
delete from dept where deptno = 10;
exception
when z then
dbms_output.put_line ('not to delete master record');
end;
/

? write a pl/sql program handle -2291 error using unnamed method from emp, dept tables.

Soln declare
z exception;
pragma exception_init (z, -2291);
begin
insert into emp (empno, deptno) values (1, 60);
exception
when z then
dbms_output.put_line ('not to insert other than primary values');
end;
/

TUESDAY
MAR 12

Subprograms

6

066-300 WK-10

Subprograms are named pl/sql blocks which is used to solve some perticular task. There are two types of subprograms supported by oracle :

i. Procedures
ii. Functions

Procedures may or may not return a value. Functions must return a single value.

i. Procedures

Procedures are named pl/sql blocks which is used to solve some perticular task & also procedures may or may not return values. Whenever we are using create or replace keyword in front of procedure those procedures automatically permantally stored in database. These type of procedures are also called as stored procedures.

Generally procedures are used to improve performance of the application because procedures internally having one time compilation i.e one time compilation program units automatically improves performance i.e whenever we are loading a procedure into database automati-

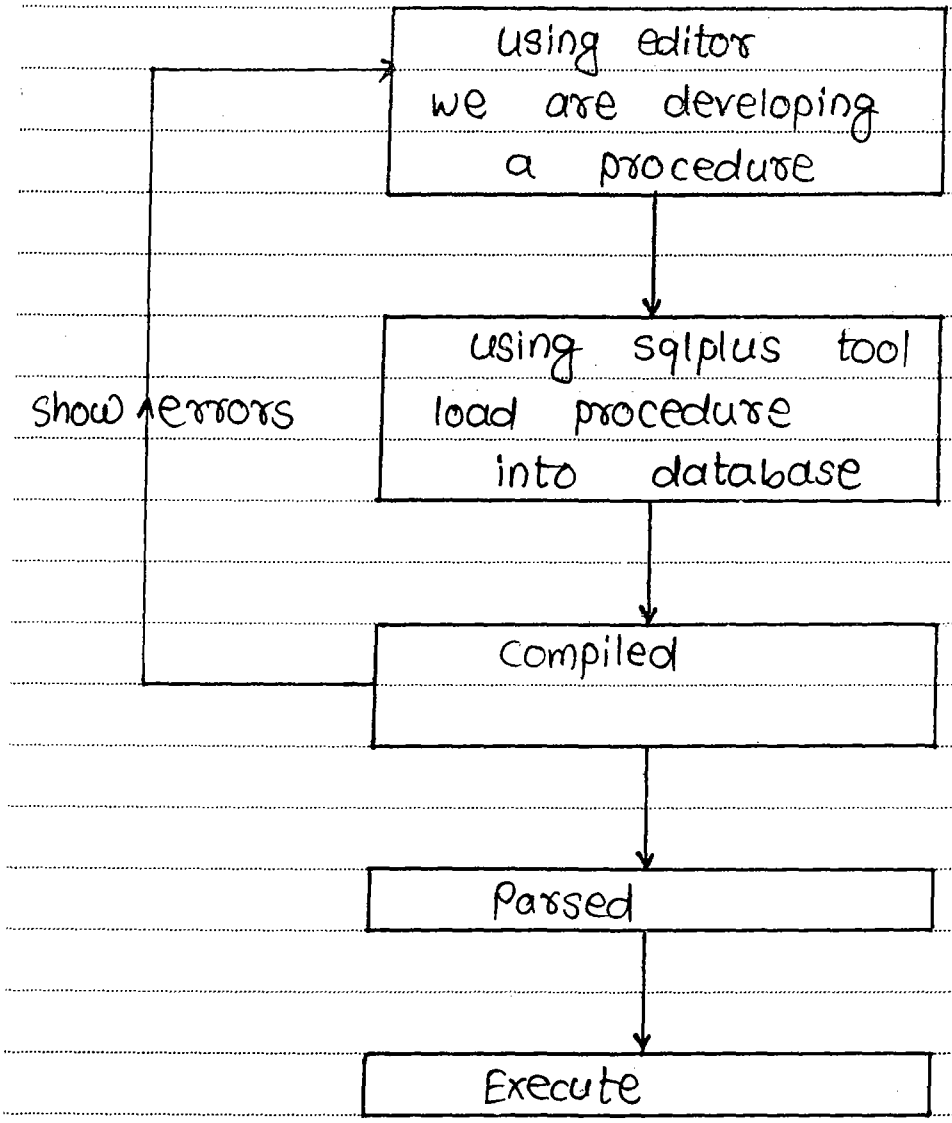
WEDNESDAY

MAR 12

7

Cally that procedure compiled.

067-299 WK-10



Every procedure having two parts :

1) procedure specification

2) procedure body

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

07-03

THURSDAY
MAR 12
8
068-298 WK-10

In procedure specification we are specifying name of the procedure, type of the parameters.
Whereas in procedure body we are solving actual task.

Syntax:
- procedure specification:
  - create [or replace] procedure procedurename (formal parameters)
    - formal parameters: parametername [mode] datatype
      - mode: in / out / in out
  - IS / AS
  - variable declaration, cursors, userdefined exceptions ;
- procedure body:
  - begin
  - ---------- [Exception]
  - ----------
  - end; [procedurename] ;

executing a procedure

Method 1 :
sql> exec procedurename (actual parameters);

Method 2 : (using annonymous blocks)
```
sql> begin
       procedurename (actual parameters);
     end;
     /
```

FRIDAY

MAR 12

9

069-297 WK-10

Method 3:

sql> Call proceedurename (actual parameters);

? write a pl/sql stored procedure for passing empno as a parameter display name of employee & his salary.

-23 Aug-

Sol^n

```
create or replace procedure p1 (p_empno number)
is
v_ename varchar2(10);
v_sal number(10);
begin
select ename, sal into v_ename, v_sal from
emp where empno = p_empno;
dbms_output.put_line ( v_ename || ' ' || v_sal );
end;
/
```

Execution –

method 1: sql> exec p1 (7902);

FORD 4600

method 2: using annonymous block

```
sql> begin
       p1 (7566);
     end;
     /
```

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

09-03

JONES 1064

method 3: sql> call p1(7902);
FORD 4600

All stored procedures information stored under user_source data dictonary.

ex: sql> desc user_source ;

sql> select text from user_source where NAME = 'P1' ;

? write a pl/sql stored procedure for passing deptno as a parameter to display employee details corresponding to that deptno.

```
Soln  create or replace procedure p1 (p_deptno number)
      is
      cursor c1 is select * from emp where
        deptno = p_deptno;
      i  emp%.rowtype;
      begin
      open c1;
      loop
      fetch c1 into i;
      exit when c1%.notfound;
      dbms_output.put_line (i.ename||' '||i.sal||' '||i.deptno);
```

SUNDAY
MAR 12
11
071-295 WK-10

```
end loop;
close c1;
end;
/
```

method 1:

```
sql> exec p1(10);
```

method 2:

```
begin
  p1(20);
end;
/
```

Parameters used in procedures

parameters are used to pass the values into procedure & also return values from the procedure.
In this case we must use two types of parameters :

1) Formal parameters
2) Actual parameters

1) Formal parameters

Formal parameters are defined in procedure specification.

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

MONDAY
MAR 12
12
072-294 WK-11

In formal parameter we are defining parameter name & mode of the parameter.
There are three types of modes supported by oracle.

i) in mode
ii) out mode
iii) In out mode

Syntax: parametername [mode] datatype

- in
- out
- in out

Note

Whenever we are using procedures, fun^ns, cursors we are not allow to use datatype size in formal parameter declaration.

i) in mode

By default procedure parameters having in mode.
in mode is used to pass the values into procedure body.
This mode behaves like a "constant" in procedure body. Through the in mode we can also pass default values using default or := operator.

TUESDAY
MAR 12
13
073-293 WK-11

? write a pl/sql stored procedure to insert a record into dept table using in parameter.

Soln
```
create or replace procedure p1 (p_deptno
in number, p_dname in varchar2,
p_loc in varchar2)
is
begin
insert into dept
values (p_deptno, p_dname, p_loc);
dbms_output.put_line ('record inserted
through procedure');
end;
/
```

There are three types of execution methods supported by in parameter.

(I) Positional notations
(II) Named notations
(III) mixed notations

(I) Positional ~~para~~ notations

ex:
```
Sql> exec p1 (1, 'a', 'b');
```

(II) Named notations (⇒)

ex:
```
Sql> exec p1 (p_dname ⇒ 'x', p_loc ⇒ 'y',
                p_deptno ⇒ 2);
```

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

(iii) Mixed notation

It is the combination of positional, named notation.

After positional their can be all named notations but after named their can not be positional.

ex: Sql> exec p1 (3, p_loc => 'y', p_dname => 'x');
record inserted through procedure

ii) Out mode

This mode is used to return values from procedure body.
Out mode internally behaves like a uninitialized variable in procedure body.
Here we must specify out keyword explicitly

Syntax: parametername out datatype

```
ex: create or replace procedure p1 (a in
    number, b out number)
    is
    begin
    b = a*a;
    end;
    /
```

THURSDAY

MAR 12

15

075-291 WK-11

-24 Aug-

In oracle if a subprogram contains out, in out parameters those subprograms are executed using following two methods

Method 1 : using bind variable

Method 2 : using annonymous blocks

Bind variable

These variables are session variables.

These variables are created at host environment thats why these variables are also called as host variables.

Generally these variables are used to pass the values bet^n database server & client app^n.

Basically these variables are not a pl/sql variables, but we can also use these variables in pl/sql to execute subprograms having out parameter.

Step 1 : Creating a bind variable

Syntax : sql> variable variablename datatype;

Step 2: Using a bind variable

Syntax: :variablename

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

15-03

Step 3: display value from bind variable

Syntax: sql> print bindvariablename;

ex: sql>variable g number;

```
sql> declare
     a number(10) :=500;
     begin
     :g := a/2
     end;
     /
```

```
sql> print g;
```

G
250

Method 1: using bind variable

```
sql> variable z number;
sql> exec p1 (4, :z);
```

```
sql> print z;
```

Z
16

SATURDAY

MAR 12

17

077-289 WK-11

method 2: Using annonymous block

```
ex:   declare
      x number(10);
      begin
      p1(6, x)
      dbms_output.put_line(x);
      end;
      /
```

36

? Write a pl/sql stored procedure for passing employee name as in parameter return salary of that employee using out parameter from emp table.

Soln

```
create or replace procedure p1 ( p_ename in
varchar2, p_sal out number)
is
begin
select sal into p_sal from emp where
name = p_name ;
end;
```

Execution:

method 1: using bind variable

```
Sql> variable z number;
```

March 2012

Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
Sql> exec p1 ('KING', :z);

Sql> print z;

         Z
----------
      6800
```

Execution :

method 2: using annonymous block

```
declare
x number (10);
begin
p1 ('SMITH', x);
dbms_output.put_line (x);
end;
/

2800
```

? Write a pl/sql stored procedure for passing deptno as a parameter return number of deptno's as out parameter from emp table.

Sol^{n}

```
Create or replace procedure p1 (p_deptno
in number, p_a out number)
is
begin
```

```
select count(*) into p_a from emp
where deptno = p_deptno;
end;
/
```

Exection:

Method 1: using bind variable

```
sql > variable r number;
sql > exec p1(10, :r);

sql> print r;

   R
---
   3
```

iii) in out mode

This mode is used to pass the values into subprogram & return values from subprogram. i.e this mode internally behaves like a constant, initialized variable.

Here also explicitly we must specify in out keyword.

Syntax: parametername in out datatype

ex:

```
create or replace procedure p1 (a in out
 number)
is
begin
a := a*a;
```

```
end;
/
```

Execution:

method 1: using bind variable

```
sql> variable z number;
sql> exec :z := 8;

sql> exec p1 (:z);

sql> print z;

        z
---------
       64
```

Execution:

method 1: using annonymous block

```
declare
x number(10) := &x;
begin
p1 (x);
dbms_output.put_line (x);
end;
/
```

(output) enter value for x = 7

49

WEDNESDAY

MAR 12

21

081-285 WK-12

-25 Aug-

```
create or replace procedure p1 ( p_z
 in out number)
 is
begin
Select sal into p_z from emp where
 sal = p_z ;
end;
/
```

Execution :

```
sql> variable z number;
sql> exec :z := 7902;

sql> exec p1 (:z);

sql> print z;
```

z
4600

Pass by value, Pass by reference

Whenever we are using modular programming two types of passing parameter mechanism supported by all languages

1. Pass by value
2. Pass by Reference

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

These two parameter mechanism specifies when we are modifying formal parameters, actual parameters are affected or not.

In pass by value method actual value does not change in [calling] main program, Because copy of the values are passed into called program.

If we want to change actual parameters according to formal parameter modification we are using pass by reference method.

Oracle also support these two passing parameter mechanisms when we are using sub-program parameters & also by default all in parameters are passed by reference whereas all out parameters are passed by value.

Whenever we are returning large amount of data using out parameter again copy of the value is created because in databases out parameters are Pass by value. To overcome this problem oracle introduce ~~no~~ "NOCOPY" hint in the out parameter.

Syntax: parametername out nocopy datatype

ex: Create or replace procedure p1 (p_ename in varchar2, p_sal out, number)
nocopy
is

FRIDAY

MAR 12

23

083-283 WK-12

```
begin
select sal into p_sal from emp
where ename=p_ename;
end;
/
```

Autonomous Transactions

Autonomous transactions are independent transactions used in either procedures or in triggers. Generally autonomous transactions are used in child procedures, these procedures are not affected from the main transactions when we are using commit or rollback.

If we want to convent a procedure into autonomous we must use "autonomous_transaction pragma, commit".

Syntax: pragma autonomous_transaction ;

This pragma used in declare section of the procedure.

using autonomous transaction

ex:

```
Sql> create table test ( name varchar2 (10) );
```

March 2012

Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
sql> Create or replace procedure p1
     is
     pragma autonomous_transaction;
     begin
     insert into test values ('india');
     commit;
     end;
     /

sql> begin
     insert into test values ('hyd');
     insert into test values ('mumbai');
     p1;
     rollback;
     end;
     /

sql> Select * from test;

     Name
     india
```

without autonomous transaction

ex:

```
sql> delete from test;

sql> Create or replace procedure p1
     is
```

```
begin
insert into test values ('india');
commit;
end;
/
```

```
sql> begin
insert into test values ('hyd');
insert into test values ('mumbai');
p1;
rollback;
end;
/
```

```
sql> select * from test;
```

```
NAME
hyd
mumbai
india
```

Note

When a procedure contains commit pl/sql runtime engin not only commit procedure transaction but also commit above of the procedure transaction. To overcome this problem oracle 8i introduce autonomous transaction.

MONDAY
MAR 12

26

086-280 WK-13

-26 Aug-

AUTHID CURRENT_USER

When we are reading data from table & also provides security for the data, database developer using authid current_user clasue in procedure.

When we are using this clause another user can not execute procedure if database administrator giving privileges also.

giving procedure privilization

Syntax: grant execute on procedurename to
user1, user2, ... ;

Syntax: create or replace procedure procedurename
(formal parameters)
authid current_user
is/as
--> variable declarations, cursors,
userdefined exceptions;
begin

[exception]
---
end;

TUESDAY

MAR 12

27

087-279 WK-13

ex: sql> conn scott/tiger;

```
sql> create or replace procedure p1 (
       p_empno number)
     authid current_user
      is
     v_ename varchar2(10);
     V_sal number(10);
     begin
     select ename, sal into v_ename, v_sal
      from emp where empno=p_empno;
     dbms_output.put_line (v_ename||' '||V_sal);
     end;
     /
```

sql> grant execute on p1 to murali;

sql> conn murali/murali;
sql> set serveroutput on;

sql> exec scott.p1(7902);

error:- table or view does not exist.

Handled or unhandled exception in procedure

whenever we are calling inner procedures into

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Outer procedures ~~mwe~~ must implement inner procedures exeception handler otherwise pl/sql runtime engine execute outer procedure default procedure handler.

Inner procedure

```
ex: create or replace procedure p1 ( x in number,
    y in number)
    is
    begin
    dbms_output.put_line (x/y );
    exception
    when zero_divide then
    dbms_output.put_line ('y can not be zero');
    end;
    /
```

Outer procedure

```
ex: sql> create or replace procedure p2
        is
        begin
        p1 (7,0)
        exception
        when others then
       dbms_output.put_line ('any error');
        end;
        /

    sql> exec p2;
```

(output) ⇒ y can not be zero

THURSDAY

MAR 12

29

089-277 WK-13

All stored procedure information stored under user_procedure, user_source.

we can also drop procedure using -

drop procedure procedurname only.

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FRIDAY
MAR 12
30

090-276 WK-13

ii. Functions

Function is a named pl/sql block which is used to solve some perticular task. And also bydefault functions returns a single value.

Functions also having two parts :

1. Function specification
2. Function body

In function specification we are specfying name of the function & type of the parameters whereas in function body we are solving the actual task.

Syntax: create or replace function functioname(
formal parameters)
return datatype ← parametername [mode]
Is / AS datatype
→ variable declarations, cursors,
Begin

Return expression
end [function];

SATURDAY
MAR 12
31
091-275 WK-13

Executing a function

Method 1 :-

```
select functionname (actualparameters) from dual;
```

Method 2:- (using annonymous block)

```
begin
variablename := functionname (actualparameters);
end;
```

ex:
```
sql> create or replace function f1(a varchar2)
     return varchar2
     is
     begin
     return a;
     end;
     /
```

execution : method 1

```
sql> select f1 ('hi') from dual;
```

(output) ⇒ hi

Method 2 (using annonymous block)

```
declare
z varchar2(10);
```

March	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

ACTION PLAN

APRIL '12

WK	Mon	Tue	Wed	Thu	Fri	Sat	Sun
	30						1
14	2	3	4	5	6	7	8
15	9	10	11	12	13	14	15
16	16	17	18	19	20	21	22
17	23	24	25	26	27	28	29

OBJECTIVES THIS MONTH

DATE	DESCRIPTION	REMARKS

MARCH '12

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY '12

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

SUNDAY

APR 12

1

092-274 WK-13

```
begin
z := f1 ('hi');
dbms_output.put_line (z);
end;
/
```

? Write a pl/sql stored function for passing number return a message either even or odd based on that number.

Soln

```
create or replace function f1 (a number)
return varchar2
is
begin
if mod (a,2) =0 then
return 'even';
else
return 'odd';
end if;
end;
/
```

execution : method 1

```
sql> select f1(5) from dual;
/
```

(output) ⇒ odd

Method 2 : using annonymous block

```
declare
x varchar2(10);
begin
x :=f1(6);
dbms_output.put_line (x);
end;
/
```

Method 3: using bind variable

```
sql> variable z varchar2(10);

sql> begin
     :z := f1(7);
     end;
     /

sql> print z;
```

(output)⇒

z
odd

Method 4 :

```
sql> exec dbms_output.put_line (f1(3));
```

(output) ⇒ odd

TUESDAY

APR 12

3

094-272 WK-14

Method 5:

```
begin
dbms_output.put_line ( f1(4));
end;
/
```

(output) ⇒ even

27 Aug

Note

we can also use user defined ~~statement~~ fun ign insert statement.

ex: sql> create table y1 (sno number(10),
msg varchar2 (10));

sql> insert into y1 values (1, f1(8));

sql> select * from y1;

SNO	MSG
1	even

? Write a pl/sql stored function for passing empno as parameter return gross salary from emp table based on following condition

gross := basic + hra + da - pf;

hra ---→ 10% of sal
da ---→ 20% of sal
pf ---→ 10% of sal

Soln
```
create or replace function f1 (p_empno
    number)
return number
is
 v_sal number (10);
 gross number (10);
  hra number (10);
  da number (10);
  pf number (10);
begin
 select sal into v_sal from emp where
 empno = p_empno;
 hra := v_sal * 0.1;
  da := v_sal * 0.2;
  pf := v_sal * 0.1;
gross := v_sal + hra + da + pf;
  return gross;
 end;
```

execution: method 1

```
Sql> Select f1 (7566) from dual;
```

(output)→ 60987

Method 2 : using annonymous block

```
declare
```

```
z number (10);
begin
  z := f1(7566);
dbms_output.put_line (z);
end;
/
```

Note

We can also use predefined aggrigate fu^n in user defined functions and also ## use this user defined functions in same table or different table.

ex: Create or replace function maximum
return number
is
v_sal number (10);
begin
Select max(sal) into V_sal from emp;
return vSal;
end;
/

execution

sql> select ename, sal, maximum-sal from emp;

Note

We are not allow to use DML statements in functions.

Out

If we want to return more no. of values from fun we are using out parameter

```
ex: create or replace function f1 (p_deptno
    in number, p_dname out varchar2,
    p_loc out varchar2)
    return varchar2
    is
    begin
    select dname, loc into p_dname, p_loc
    from dept where deptno = p_deptno;
    return p_dname;
    end;
    /
```

execution : using bind variable

```
Sql> variable a varchar2(10);

Sql> variable b varchar2 (10);

Sql> variable c varchar2 (10);

Sql> begin
     :a := f1 (10, :b, :c );
     end;
     /
```

SATURDAY
APR 12
7
098-268 WK-14

Sql> print b c ;

(Output) ⇒ B

ACCOUNTING

C

NEW YORK

Pl/Sql

? Write a stored function for passing empno, date as parameter return number of years that employee is working based on date from emp table.

Soln Create or replace function f1 (p_empno number, p_date date)
return number
is
z number (10);
begin
Select months_between (p_date, hiredate)/ 12 into z from emp where empno=p_empno;
return (round (z));
end;
/

execution

Sql> select empno, ename, hiredate, f1 (empno,

sysdate) || 'years' "exper" from emp
where empno = 7902
/

EMPNO	ENAME	HIREDATE	EXPER
7902	FORD	03-DEC-81	31 years

Note

we are not allow to use named, mixed notations in fun. To over come this problem Oracle 11g introduce named, mixed notation.

-28 Aug-

Note

we can also develop our own aggrigatable funs & also use these funs in group by clause and also if these funs returns multiple values, we must use cursors in these user defined funs.

ex: create or replace function f1 (p_deptno number)
return varchar2
is
a varchar2 (200);
cursor c1 is select ename from emp
where deptno = p_deptno;
begin
for i in c1

MONDAY

APR 12

9

100-266 WK-15

```
loop
a := a || ' ' || i.ename ;
end loop;
return a;
end;
/
```

execution:

```
sql> select deptno, f1 (deptno) from emp
     group by deptno
```

Note

Oracle 10g introduce wm_concat ( ) predefined aggrigate fun which returns number of rows group wise. This fun accept all datatype columns

```
ex: Select deptno, wm_concat (ename) From emp
    group by deptno
```

```
ex: select job, wm_concat (ename) from emp
    group by job
```

? write a pl/sql stored function for passing empno as parameter, calculate tax based on following condition using emp table

i) If annual sal > 10000 then tax --> 10%.

ii) If annual sal > 15000 then tax --> 20%.

iii) If annual sal > 20000 then tax --> 30%.

Soln create or replace function f1
(p_empno number)
return number
is
v_sal number (10);
annsal number (10);
it number (10);
begin
select sal into v_sal from emp where
empno = p_empno;
annsal := v_sal * 12;
if annsal > 10000 ~~then~~ and annsal <=15000
then
it := annsal * 0.1;
elsif annsal > 15000 and annsal <= 20000
then
it := annsal * 0.2;
elsif annsal > 20000 then
it := annsal * 0.3;
else
it :=0;
end if;
return it;
end;
/

SQL> select f1 (7902) from dual;

WEDNESDAY
APR 12
11
102-264 WK-15

Note

we can also drop function using
drop function functionname ;

triggers

trigger is also same as stored procedure & also it will automatically invoked whenever DML operation performed against table or view. There are two types of triggers supported by pl/sql.

1. Statement level triggers
2. Row level triggers

In statement level trigger, trigger body is executed only once for DML statement. Whereas in row level trigger, trigger body is executed ~~one~~ for each row for DML statements.

Syntax:

trigger Specification

create [or replace] trigger triggername
before / after → Trigger timing → Trigger event
insert / update / delete / on tablename
[for each row] --> row level
[where condition] --> trigger
[declare]

THURSDAY
APR 12
12
103-263 WK-15

→ variable declarations, cursors;

trigger body { begin / —— / —— / end;

Row Level Triggers

In row level trigger, trigger body is executed for each row for DML statement, thats why we are using for each row clause in trigger specification and also data internally stored in 2 rollback segment qualifiers these are old, new.
These qualifiers are used in either trigger specification or in trigger body.
When we are using these modifiers in trigger body we must use colon prefix in the qualifiers.

Syntax: :old.columnname
(or)
:new.columnname

But when we are using these qualifiers in when clause we are not allow to use colon infront of the qualifiers.

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FRIDAY

APR 12

13

104-262 WK-15

Syntax: Old.columnname

(or)

new.Columnname

	insert	update	delete
:new	✓	✓	x
:old	x	✓	✓

? Write a pl/sql row level trigger on emp table whenever user inserting data into a emp table sal should be (above) more than 5000.

Sol^n

```
create or replace trigger tz1
before insert on emp
for each row
 begin
if :new.sal < 5000 then
raise_application_error (-20123, 'Salary should
be more than 5000');
end if;
end;
/
```

```
sql> insert into emp (empno, sal) values
     (1, 2000);
```

⇒ ORA-20123 : Salary should be more than 5000

SATURDAY
APR 12
14
105-261 WK-15

? write a pl/sql trigger ~~using~~ using emp, dept table implement on delete cascade concept, without using on delete clause.

Soln

```
create or replace trigger tr1
after delete on dept
for each row
begin
delete from emp where deptno = :old.deptno;
end;
/
```

```
sql> delete from dept where deptno = 20;
```

? write a pl/sql row level trigger on dept table whenever updating deptno's in dept table automatically those deptno's modified into emp table.

Soln

```
create or replace trigger tr1
after update on dept
for each row
begin
update emp set deptno = :new.deptno where
deptno = :old.deptno;
end;
/
```

SUNDAY

APR 12

15

106-260 WK-15

```
sql> update dept set deptno=1
     where deptno = 10 ;

sql> select * from emp;
```

Auto increment

If we want to generate primary key values automatically all database system uses auto increment concept.

In oracle if we want to implement auto increment concept we are using triggers, ~~or~~ sequences i.e we are creating sequences in ~~pl~~ sql and use that sequences in triggers.

ex: sql>Create sequence s1 start with 1;

```
sql> create table test (sno number(10) primary
       key, name varchar2(10) );

sql> create ~~table~~ or replace trigger tw2
     before insert on test
     for each row
     begin
     select s1.nextval into :new.sno from dual;
     end;
     /
```

April 2012

Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
sql> insert into test (name) values
     ('murali');

sql> insert into test (name) values
     ('subhas');

sql> insert into test (name) values ('satish');

sql> select * from test;
```

(output) ⇒

sno	name
1	murali
2	Subhas
3	Satish

Generally if we want to generate sequence values we are using dual table in pl/sql blocks, but Oracle 11g introduce without dual table also we are generating sequences in pl/sql block.

```
syntax: begin
        variablename := sequencename.nextval;
        end;
```

11g : create or replace trigger tw2
before insert on test

TUESDAY

APR 12

17

108-258 WK-16

```
for each row
begin
:new.sno := s1.nextval;
end;
/
```

ex:

```
create or replace trigger tw2
after insert on test
for each row
begin
select s1.nextval into :new.sno from dual;
end;
/
```

error :- ~~eh~~ cannot change NEW values for this trigger type

before, after triggers

In before triggers, trigger body is executed before DML statements are affected into database, whereas in after triggers, trigger body is executed after DML statements are affected into database. Generally if we want to restrict invalid data entry always we are using before triggers, whereas if we are performing operation on the one table those operations are affected into another table

then we are using after trigger.
Whenever we are ~~as~~inserting values into new qualifier we must use before triggers otherwise oracle server returns an error.

? write a pl/sql row level trigger whenever user inserting data into ename column after inserting data must be converted into upper case.

soln create or replace trigger tw3
before insert on emp
for each row
begin
:new.ename := upper (:new.ename);
end;
/

Sql> insert into emp (empno, ename) values (1,'abc');

— 30 Aug —

In statement level trigger, trigger body is executed only once for each DML statement thats why generally statement level triggers used to define type based condition and also used to implement auditing reports.

THURSDAY

APR '12

19

110-256 WK-16

These triggers does not contain new, old qualifiers.

? write a pl/sql statement level trigger on emp table not to perform DML operations in saturday, sunday.

Soln
```
create or replace trigger te1
before insert or update or delete on emp
begin
if to_char (sysdate, 'DY') in ('SAT', 'SUN')
then
raise_application_error (-20123, 'we can not
perform DMLs in sat, sun day');
end if;
end;
/
```

```
Sql> delete from emp where empno = 7902;
```

⇒ ora-20123 : we can not perform DMLs in sat, sun day.

? write a pl/sql statement level trigger on emp table not to ~~perm~~form DML operation on last day of the month

Soln
```
Create or replace $ trigger te2
before insert or update or delete on emp
```

April	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
begin
if sysdate = last_day (sysdate) then
raise_application_error (-20123, 'we can
not perform DMLs on lastday');
end if;
end;
/
```

```
sql> delete from emp where empno = 7902;
```

⇒ ora-20123 : we cannot perform DMLs on lastday.

Note 1

If to_char (sysdate) = to_char (last_day (sysdate)) then

Note 2

If to_char (sysdate, 'DY') = TO_CHAR (last_day (sysdate), 'DY')) then

Triggering Events [Or]
Trigger Predicate Clauses

If we want to define multiple conditions on multiple tables then all database systems uses triggering events.
These are inserting, updating, deleting clauses

SATURDAY

APR 12

21

112-254 WK-16

syntax: if inserting then
statements;
elsif updating then
statements;
elsif deleting then
statements;
end if;

These clauses are used in either row level or statement level triggers.

? Write a pl/sql Statement level trigger on emp table not to perform any DML operation in any days using triggering event

Soln
```
create or replace trigger te3
before insert or update or delete on emp
begin
if inserting then
raise_application_error(-20123,'we can not perform inserting operation');
elsif updating then
raise_application_error(-20124,'we can not perform update operation');
elsif deleting then
raise_application_error(-20125,'we can not perform deleting operation');
```

April	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
end if;
end;
/
```

ex: sql> create table test (msg varchar2(10));

```
sql> create or replace trigger te4
     after insert or update or delete on emp
     declare
      z varchar2(20);
      begin
     if inserting then
     z:='rowsinserted';
     elsif updating then
    z := 'rowsupdated';
     elsif deleting then
      z := 'rows deleted';
     end if;
     insert into test values (z);
     end;
     /
```

```
sql> insert into emp (empno) values (1);
sql> insert into emp (empno) values (2);
sql> insert into emp (empno) values (3);
```

```
sql> update emp set sal = sal + 100 where
       deptno = 30;
```

MONDAY

APR 12

23

114-252 WK-17

⇒ 6 rows updated

sql> select * from test;

⇒

msg
rowsinserted
rowsinserted
rowsinserted
rowsupdated

? Write a pl/sql row level trigger on emp table whenever user inserting data those values stored in another table, whenever user updating data those values stored in another table, whenever user deleting data those values are stored in another table.

```
sql> create table t1 as select * from emp
     where 1=2;

sql> create table t2 as select * from emp
     where 1=2;

sql> create table t3 as select * from emp
     where 1=2;

sql> create or replace trigger te5
     after insert or update or delete on emp
```

```
for each row
begin
if inserting then
insert into t1 values (:new.empno, :new.ena
me,...);
elsif updating then
update insert into t2 values (:old.empno,
:old.ename,...);
elsif deleting then
insert into t3 values (:old.empno, :old.ename
...);
end if;
end;
/
```

Execution order in Triggers

I. before statement level

II. before row level

III. after row level

IV. after statement level

Whenever we are creating same level of triggers on same table we can not control execution order of the triggers.

WEDNESDAY

APR 12

25

116-250 WK-17

To overcome this problem oracle 11g introduces follows clause in triggers. follows clause provides "gurantee" execution order.

This clause used in trigger specification only.

-31 Aug-

Syntax: create or replace trigger triggername
before / after
insert / update / delete on tablename
[for each row]
follows
another trigger1, trigger2, ...
declare
begin

---
end;

ex: create or replace trigger tj2
before insert on test
for each row
declare } // This is wrong one

ex: create table test (col1 number(10), col2 number(10), col3 date);

sql> create sequence s1 start with 5878;

April 2012

Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
sql> create or replace trigger tj1
     before insert on test
     for each row
     begin
     select s1.nextval into :new.col1 from dual;
     dbms_output.put_line ('trigger1 fired');
     end;
     /

sql> create or replace trigger tj2
     before insert on test
     for each row
     declare
     v_col2 varchar2(10);
     begin
     select to_char(reverse (:new.col1)) into
      v_col2 from dual ;
       :new.col2 := v_col2 ;
     dbms_output.put_line ('trigger 2 fired');
     end;
     /

sql> insert into test (col3) values (sysdate);
     trigger2 fired
     trigger1 fired

     1 row created
```

FRIDAY
APR 12
27
118-248 WK-17

sql> select * from test;

col1	col2	col3
5878		31-AUG-12

Solution (oracle 11g)

```
create or replace trigger tj2
before insert on test
for each row
follows tj1
declare
— .. —
```

Calling a procedure into trigger

Using call statement we are calling procedure into trigger.

Syntax:
```
create or replace trigger triggername
before / after
insert / update / delete on tablename
call procedurename
/
```

ex: sql> create table test (totsal number(10));

```
sql> create or replace procedure p1
```

```
is
z number (10);
begin
delete from test;
select sum (sal) into z from emp;
insert into test values (z);
end;
/
```

```
Sql> Create or replace trigger tj3
     after insert or update or delete on emp
     call p1
     /
```

```
Sql> update emp set sal = sal + 100 ;
```

```
Sql> select * from test;
```

(output)

TOTSAL
44456

? write a pl/sql trigger on emp table whenever user deleting records from emp table automatically display remaining number of existing record number in bottom of the delete statement.

SUNDAY
APR 12
29
120-246 WK-17

Soln

```
sql> create or replace trigger tj4
     after delete on emp
     declare
     z number(10);
     begin
     select count(*) into z from
     emp;
     dbms_output.put_line(z);
     end;
     /
```

```
sql> delete from emp where empno=7902;
```

(output) ⇒ 13

ex:

```
create or replace trigger tj4
after delete on emp
for each row
declare
z number(10);
begin
select count(*) into z from emp;
dbms_output.put_line(z);
end;
/
```

```
sql> delete from emp where empno=1;
```

error:- ora-4091 : table scott.emp is mutating

MONDAY
APR 12
30
121-245 WK-18

-01 Sep-

Mutating

Into a row level trigger based on a table trigger body can not read data from same table and also we can not perform DML operations on same table.

If we are trying this oracle server returns an error ora-4091 : table is mutating

This error is called mutating error.

This trigger is called mutating trigger.

This table is called mutating table.

Mutating errors are not accured in statement level trigger because through these statement level trigger when we are performing DML operations automatically data commited into the database, whereas in row level triggers when we are performing transaction data is not commited & also again we are reading this data from the same table then only mutating error is occured.

To avoid mutating errors we are using autonomous transaction in triggers.

But autonomous transaction gives previous result.

Sol^{n} - (using autonomous transaction)

```
Sql> create or replace trigger tj1
     after delete on emp
```

```
for each row
declare
z number(10);
pragma autonomous_transaction;
begin
select count(*) into z from emp;
commit;
dbms_output.put_line (z);
end;
/
```

```
sql> delete from emp where empno=2;
```

(output) ⇒ 16

DDL Triggers

We can also create triggers on schema level, database level. These type of triggers are called DDL triggers or system triggers.
These type of triggers are created by database administrator.
These triggers are created on DDL events.

Syntax:

```
Create or replace trigger triggername
before / after
create / alter / drop / truncate / rename
on username.schema
```

ACTION PLAN

MAY '12

WK	Mon	Tue	Wed	Thu	Fri	Sat	Sun
18		1	2	3	4	5	6
19	7	8	9	10	11	12	13
20	14	15	16	17	18	19	20
21	21	22	23	24	25	26	27
22	28	29	30	31			

OBJECTIVES THIS MONTH

DATE	DESCRIPTION	REMARKS

APRIL '12

M	T	W	T	F	S	S
					1	2
9	10	11	12	13	14	15
23	24	25	26	27	28	29

M	T	W	T	F	S	S
3	4	5	6	7	8	
16	17	18	19	20	21	22
30						

2012

JUNE '12

M	T	W	T	F	S	S
				1	2	3
11	12	13	14	15	16	17
25	26	27	28	29	30	

M	T	W	T	F	S	S
4	5	6	7	8	9	10
18	19	20	21	22	23	24

TUESDAY
MAY 12

1

122-244 WK-18

```
declare
— .. —
begin
— .. —
— .. —
end;
/
```

? Write a pl/sql DDL trigger on scott schema not to drop emp table.

Sol^n

```
Create or replace trigger tj2
before drop on scott.schema
begin
if ora_dict_obj_name = 'EMP' and
 ora_dict_obj_type = 'TABLE' then
raise_application_error (-20123, 'we can not
 drop emp table');
end if;
end;
/
```

```
sql> drop table emp;
ora-20123: we can not drop emp table.
```

Note

There are 12 types of triggers supported by oracle based on statement, row level

before, after, insert, update, delete and also oracle support instead of triggers on views & DDL triggers are databases.

enable / disable triggers

1) enable / disable a single trigger

Syntax: alter trigger triggername enable/disable;

2) enable / disable all triggers in a table

Syntax: alter table tablename enable / disables all triggers;

ex: alter table emp disable all triggers;

All triggers information stored under user_ triggers; datadictonary.

ex: desc user_ triggers;

we can also drop trigger.

THURSDAY

MAY 12

3

124-242 WK-18

Packages

Package is a database object which is used encapsulate variables, constants, procedures, cursors, functions, types into single unit.

Packages does not accepts parameters, can not be nested, can not be invoked.

Generally packages are used to impove performance of the application because when we are calling packaged subprogram first time total package automatically loaded into memory area. Whenever we are calling subsequent subprogram calls pl/sql run time engine calling those subprogram from memory area.

This process automatically reduces disk I/O thats why packages improves performance of the application.

packages having two parts

1. Package Specification
2. Package body

In package specification we are defining global data and also declare objects, Subprograms whereas in package body we are implementing Subprograms & also package body subprogram

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

internally behaves like a private subprogram.

Package Specification

Syntax:
```
create or replace package packagename
IS / AS
global variable declaration;
constant declaration;
cursor declaration;
types declaration;
procedure declaration;
function declaration;
end;
/
```

Package Body

Syntax:
```
Create or replace package body
packagename
IS / AS
procedure implementations;
function implementations;
end;
/
```

Invoking packaged subprograms

1) sql > exec packagename.procedurename (actural parameters);

SATURDAY
MAY 12

5

126-240 WK-18

2) select packagename.functionname (actual parameters) from dual;

ex: sql> create or replace package ph1
is
procedure p1;
procedure p2;
end;
/

ex: sql> create or replace Package body ph1
is
procedure p1
is
begin
dbms_output.put_line ('first proc');
end p1;
procedure p2
is
begin
dbms_output.put_line ('second proc');
end p2;
end;
/

execution

sql> exec ph1.p2;

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

global variables used in package

In pl/sql we are defining global variables in package specification only.

ex:
```
sql> create or replace package ph2
     is
     g number(10) := 500;
     procedure p1
     function f1 (a number) return number;
     end;
     /

sql> create or replace package body ph2
     is
     procedure p1
     is
     z number (10);
     begin
     z := g/2;
     dbms_output.put_line (z);
     end p1;
     function f1 (a number) return number
     is
     begin
     return a*g;
     end f1;
     end;
     /
```

MONDAY
MAY 12
7
128-238 WK-19

sql> exec ph2.p1;

(output) 250

sql> select ph2.f1(3) from dual;

(output) 1500

-03 Sep-

State of the global variable

If we want to maintain state of the global variable we are using serially_reusable pragma in packages.

Syntax: pragma serially_reusable;

ex: sql> create or replace package pj1
is g number(10) := 5;
pragma serially_reusable;
end;
/

sql> begin
pj1.g := 70
end;
/

```
sql> begin
     dbms_output.put_line (pj1.g);
     end;
     /
```

(output) 5

State of the globalized Cursor

ex:
```
sql> create or replace package pj2
     is
     cursor c1 is select * from emp;
     pragma serially_reusable;
     end;
     /

sql> begin
 open pj2.C1;
     end;
     /

sql> begin
     open pj2.C1;
     end;
     /
```

WEDNESDAY

MAY 12

9

130-236 WK-19

Overloading Procedures

overloading refers to same name can be used for different purposes i.e we are implementing overloading procedures through packages only, these procedures having same name and also different type and number of arguments.

ex: sql> create or replace package pj1
is
procedure p1(a number, b number);
procedure p1(x number, y number);
end;
/

sql> Create or replace package body pj1
is
procedure p1 (a number, b number).
is
c number (10);
begin
c:= a+b;
dbms_output.put_line (c);
end p1;
~~end;~~
/
procedure p1 (x ~~m~~ number, y number)

```
is
z number (10);
begin
z := x-y;
dbms_output.put_line (z);
end p1;
end;
/
```

execution

```
sql> exec pj1.p1 (a=>9, b=>5);
```

(Output) 14

```
sql> exec pj1.p1 (x=>8, y=>5);
```

(output) 3

Forward declaration

Whenever we are calling procedures into another procedure then only we are using forward declaration i.e. whenever we are calling local procedure into global procedure first we must implement local procedures before calling otherwise use a forward declaration in package body.

FRIDAY

MAY 12

11

132-234 WK-19

ex: sql> create or replace package pj2;
is
procedure p1;
end;
/

sql> create or replace package body pj2
is
procedure p2;
procedure p1
is
begin
p2;
end;
procedure p2
is
begin
dbms_output.put_line ('local proc');
end p2;
end;

execution

sql> @ exec pj2.p1;

(output) local proc

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

11-05

SATURDAY

12

133-233 WK-19

Types used in Packages

In Oracle we can also create our own user defined types using type keyword.

When we are creating types we are using two step process –

We are creating type using appropriate syntax then only we are creating a variable of that type.

Oracle server supports following types

1. PL/SQL Record
2. Index by table (or)
   PL/SQL table (or)
   Associative arrays
3. Nested tables
4. Varrays
5. Ref cursors

(2, 3, 4 – Collections)

1. PL/SQL Record

This is an user defined type which is used to represent different datatypes into single unit. It is also same as structures in C language. This is an user defined type so we are creating two step process.

June	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

12-05

SUNDAY

MAY 12

13

134-232 WK-19

Syntax: (i) Type Typename is Record (
attribute1 datatype (size),
attribute2 datatype (size), ...);

(ii) variablename Typename;

ex: Sql> create or replace package pj3
is
type t1 is record (a1 number (10),
a2 number (10), a3 number (10));
procedure p1;
end;
/

sql> create or replace package body pj3
is
procedure p1
is
v_t t1;
begin
select empno, ename, sal into v_t from
emp where ename = 'KING';
dbms_output.put_line (v_t.a1 ||' '|| v_t.a2
||' '|| v_t.a3);
end p1;
end;
/

execution

sql> exec pj3.p1;

(output) 7839 KING 7700

2. Index by Table

This is an user defined type which is used to store multiple data items in a single unit
Basically this is an unconstraint table.
Generally these tables are used to improve performance of applications because these tables are stored in memory area thats why these tables are also called as memory tables.
Basically these table contains key value pairs i.e. actual data is stored in value field & indexes are stored in
These indexes range from negative to positive numbers.
These indexes are not consecutive.
These indexes key behaves like a primary key i.e it does not accept duplicate, null values.
Bosically this key datatype is binary_integer.

-04 Sep-

Syntax: ① Type Typename is Table of datatype (size)
index by binary_integer;

TUESDAY

MAY 12

15

136-230 WK-20

(11) variablename Typename;

If we want to manipulate data in collection we are using collection methods.

Index by table having following collection methods exists, first, last, prior, next, count, delete (range of indexes), delete.

ex.
```
declare
type t1 is table of number (10)
index by binary_integer;
v_t t1;
begin
v_t(1) := 10;
v_t(2) := 20;
v_t(3) := 30;
v_t(4) := 40;
v_t(5) := 50;
dbms_output.put_line (v_t(3));
dbms_output.put_line (v_t.first);
dbms_output.put_line (v_t.last);
dbms_output.put_line (v_t.prior(3));
dbms_output.put_line (v_t.next(4));
dbms_output.put_line (v_t.count);
v_t.delete (1,3);
dbms_output.put_line (v_t.count);
v_t.delete;
dbms_output.out_line (v_t.count);
end;
```

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

WEDNESDAY
MAY 12

16

137-229 WK-20

? write a pl/sql prog to transfer all employee names from emp table & store it into index by table & also display data from index by table.

```
Sol^n  declare
       type t1 is table of varchar2 (10)
       index by binary_integer;
       v_t  t1;
       Cursor  c1 is select ename from emp;
       n  number (10) := 1;
        begin
        open  c1;
        loop
        fetch  c1  into  v_t (n);
        exit  when  c1 % not found;
        n := n+1;
        end  loop;
        close  c1;
        for  i  in  v_t.frist .. v_t. last
         loop
        dbms_output. put_line ( v_t(i));
        end  loop;
        end;
```

THURSDAY
MAY 12
17
138-228 WK-20

ex: declare
type t1 is table of varchar2(10)
index by binary_integer;
v_t t1;
begin
select ename bulk collect into v_t
from emp;
for i in v_t.frist .. v_t.last
loop
dbms_output.put_line (v_t(i));
end loop;
end;
/

ex: declare
type t1 is table of date
index by binary_integer;
v_t t1;
begin
for i in 1..10
loop
v_t (i) := sysdate + i;
end loop;
for i in v_t.frist .. v_t.last
loop
dbms_output.put_line (v_t(i));
end loop;
end;
/

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

FRIDAY

MAY 12

18

139-227 WK-20

? write a pl/sql prog to retrive all joining dates from emp table & store it into index by table & also display content from index by table.

Soln

```
declare
type t1 is table of date
index by binary_integer;
v_t t1;
begin
select hiredate bulk collect into v_t
 from emp;
for i in v_t.first .. v_t.last
 loop
dbms_output.put_line (v_t(i));
end loop;
end;
/
```

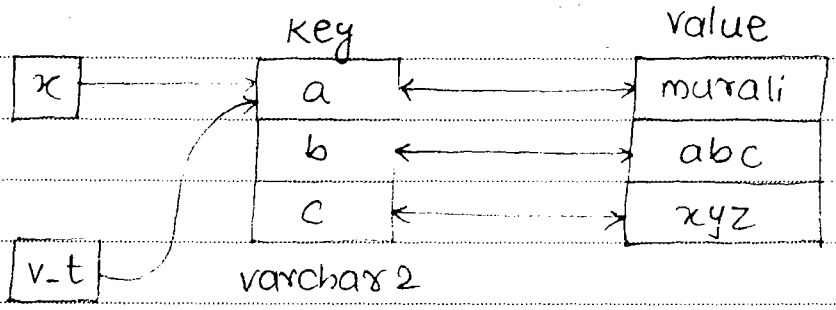
ex:

```
declare
type t1 is table of varchar2(10)
index by varchar2 (10);
 v_t t1;
 x varchar2 (10);
 begin
 v_t('a') := 'murali';
 v_t('b') := 'abc';
 v_t('c') := 'xyz';
```

June	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

SATURDAY
MAY 12
19
140-226 WK-20

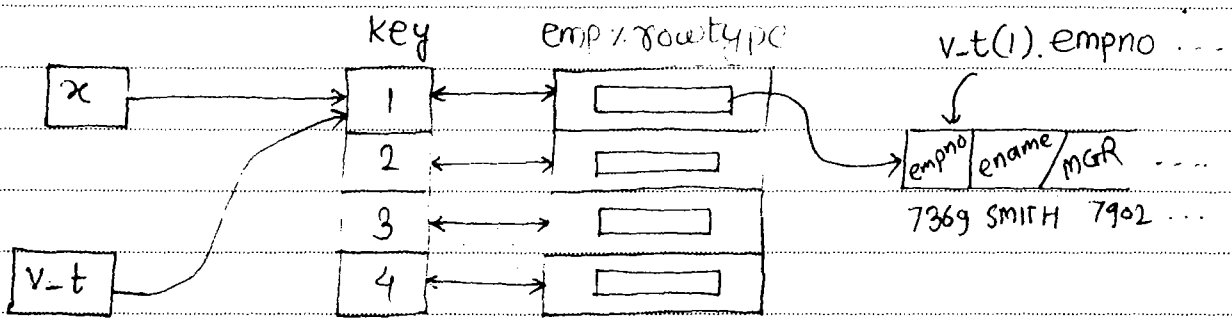
```
x := 'a'
loop
dbms_output.put_line (v_t(x));
x := v_t.next (x);
exit when x is null ;
end loop;
end;
/
```



-05 Sep-

ex:
```
declare
type t1 is table of emp%rowtype
index by binary_integer;
v_t t1 ;
x number(10);
begin
select * bulk collect into v_t from emp;
x := 1 ;
loop
dbms_output.put_line ( v_t(x).empno || ' ' ||
v_t(x).ename || ' ' || v_t(x).sal );
```

```
x := v_t.next(x);
exit when x is null;
end loop;
end;
/
```



14

(or)

ex:
```
declare
type t1 is table of emp%rowtype
index by binary_integer;
v_t t1;
begin
Select * bulk collect into v_t from emp;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line ( v_t(i).ename || ' ' ||
 v_t(i).sal );
end loop;
end;
/
```

MONDAY

MAY 12

21

142-224 WK-21

Return Result Set

If we want to return large amount of data from oracle database into client application we are using following two methods

a) method 1 :- Using index by tables
b) method 2 :- Using ref cursors

Method 1 :- Using index by tables

If we want to implement these type of appn we must implement one server appn using functions. Here we are specifying fun return type as user-defined type & also to execute the server appn we must implement client appn.

Server Application

```
ex: create or replace package pr1
    is
    type t1 is table of emp%rowtype
    index by binary_integer;
    function f1 return t1;
    end;
    /

sql> create or replace package body pr1
    is
    function f1 return t1
    is
```

```
is
v_t t1;
begin
 select * bulk collect into v_t from emp;
return v_t;
end f1;
end;
/
```

To execute this appn we use pl/sql Client (execution)

```
declare
x pr1.t1;
begin
x := pr1.f1;
for i in x.first.. x.last
loop
dbms_output.put_line (x(i).ename || ' ' || x(i).sal);
end loop;
end;
/
```

Method 2:- Using ref cursors

Oracle 7.2 introduce ref cursors.
Ref cursors are user-define types which is used to process multiple records and also this is a record by record process.

WEDNESDAY

MAY 12

23

144-222 WK-21

Generally through the static cursors we are using only one select statement at a time for single active set area whereas in ref cursors we are executing no. of select statements dynamically for a single active set area.

Thats why these type of cursors are also called as dynamic cursors.

This is an user-defined type so we are creating in two step process i.e first we are ~~cr~~reating type & then only we are creating a variable of ~~data~~that type. thats why these type of ~~ref~~ cursors are also called as cursor variables.

Generally we are not allow to pass Static cursors as parameters to the sub-programs. To overcome this problem ANSI / ISO SQL introduce ref cursors. i.e ref cursors are used to pass parameters to the subprograms because these cursors basically user-defined types.

There are two types of ref cursors supported by oracle

(i) Strong ref cursor

(ii) weak ref cursor

Strong ref cursor is a ref cursors

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

which having a return type whereas weak ref cursor is ref cursor which does not have a return type.

Syntax: Type Typename is ref cursor return recordtypedatatype ;

variablename Typename ;
↳ strong ref cursor variable

Syntax: Type Typename is ref cursor;

variablename Typename;
↳ weak ref cursor variable

Note

In ref cursors we are executing select statements using open...for statement.
These statements are used in executable section of the pl/sql block.

Syntax: open refcursorvar for select statement;

06-Sep

ex: declare
type t1 is ref cursor;
v_t t1;
i emp%rowtype;

FRIDAY

MAY '12

25

146-220 WK-21

```
begin
open v_t for select * from emp
where sal>2000;
loop
fetch v_t into i;
exit when v_t%.notFound;
dbms_output.put_line (i.ename||' '||i.sal);
end loop;
close v_t;
end;
/
```

? write a pl/sql program using ref cursor whenever user get the deptno display 10th deptartment details from emp table & also whenever user enter deptno 20 then display 20th department details from dept table.

Soln

```
declare
type t1 is ref cursor;
v_t t1;
i emp%.rowtype;
j dept%.rowtype;
v_deptno number(10) := &deptno;
begin
if v_deptno=10 then
open v_t for select * from emp where
```

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
deptno= 10 ;
loop
fetch v_t into i ;
exit when v_t%.notfound ;
dbms_output.put_line (i.ename ||' ' || i.deptno );
end loop;
close v_t ;
elsif v_deptno=20 then
open v_t for select * from dept where
deptno=20;
loop
fetch v_t into j ;
exit when v_t%. not found;
dbms_output.put_line (j.deptno ||' '|| j.dname );
end loop ;
close v_t;
end;
/
```

Passing ref cursor as parameter to the Subprogram

```
sql> create or replace package pg1
     is
     type t1 is ref cursor return emp%.rowtype;
     type t2 is ref cursor return dept%.rowtype;
     procedure p1 ( p_t1 out t1 );
     procedure p2 ( p_t2 out t2);
```

```
end;
/

sql> create or replace package body pg1
is
procedure p1 (p_t1 out t1)
is
begin
open p_t1 for select * from emp;
end p1;
procedure p2 (p_t2 out t2)
is
begin
open p-t2 for select * from dept;
end p2;
end;
/
```

execution

```
sql> variable a refcursor;
sql> variable b refcursor;

sql> exec pg1.p1 (:a);
sql> exec pg1.p2 (:b);

sql> print a b;
```

Note

We are not allow to use ref cursors directly in packages.

Note

In place of weak ref cursor oracle 9i introduce sys_ref cursor predefined type.

Syntax: variable sys_refcursor;

```
ex:  declare
     v_t  sys_refcursor;
     i  emp%.rowtype;
     begin
     open  v_t  for  select  *  from emp where
     deptno=10;
     loop
     fetch  v_t  into  i ;
     exit  when  v_t%notfound;
     dbms_output.put_line (i.ename||' '||i.deptno);
     end loop;
     close  v_t ;
     end;
     /
```

```
ex:  create or replace function f1 (a varchar2)
     return  sys_refcursor
     is
```

```
v_t   sys_refcursor;
begin
open  v_t for a;
return  v_t;
end;
```

execution

```
sql> select f1 ('select * from emp where
     sal >2000') from dual;
```

3. Nested tables, varray

Oracle 8.0 introduce nested tables, varrays. These are user defined types which is used to store multiple data items in a single unit but before we are storing actual data we must initialize using constructor.

Here constructor name is same as typename. Generally we are not allow to store index by tables permanantly into database, to overcome this problem Oracle 8.0 introduce extension of the index by tables & these user defined types stored permantly into database using Sql.

In index by tables we cannot add a remote indexes whereas in nested tables, varray we can add a remote indexes using extend, trim collection methods.

WEDNESDAY
MAY 12
30
151-215 WK-22

Nested Table

Nested table basically is an unconstrain table and also this is an user defined type which stores number of data items.

Syntax: Type Typename is Table of datatype(size);
variablename Typename := Typename ();
↳ constructorname

In nested tables, varrays always indexes start with 1 onward, & also these indexes are consecutive.
Nested tables are dense, whereas index by tables are basically no need to allocate the memory explicitly.

ex:
```
declare
type t1 is table of number(10);
v_t t1 := t1();
begin
v_t.extend(500);
v_t(500) := 10;
dbms_output.put_line(v_t(500));
end;
/
```

THURSDAY
MAY 12
31
152-214 WK-22

```
ex: declare
    type t1 is table of number(10);
    v_t  t1 := t1 ( 10, 20, 30, 40, 50 );
    begin
    dbms_output.put_line ( v_t. first);
    dbms_output.put_ line (v_t. last);
    dbms_output.put_line (v_t. prior(3));
    dbms_output.put_line (v_t. next (3));
    dbms_output.put_ line (v_t. count );
    for i in v_t.first .. v_t.last
    loop
    dbms_output.put_ line ( v_t(i));
    end loop;
    end;
    /
```

07-Sep-12

```
ex: declare
    type t1 is table of number(10);
    v_t1  t1;
    v_t2  t1 := t1();
    begin
    if v_t1 is null then
    dbms_output.put_line ('v_t1 is null');
    else
    dbms_output.put_line ('v_t1 is not null');
    end if;
    if v_t2 is null then
```

May	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

ACTION PLAN

JUNE '12

WK	Mon	Tue	Wed	Thu	Fri	Sat	Sun
22					1	2	3
23	4	5	6	7	8	9	10
24	11	12	13	14	15	16	17
25	18	19	20	21	22	23	24
26	25	26	27	28	29	30	

OBJECTIVES THIS MONTH

DATE	DESCRIPTION	REMARKS

MAY '12

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

JULY '12

M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

FRIDAY

JUN 12

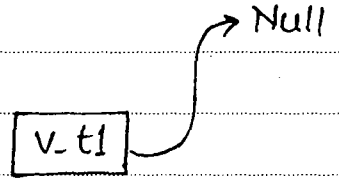
1

153-213 WK-22

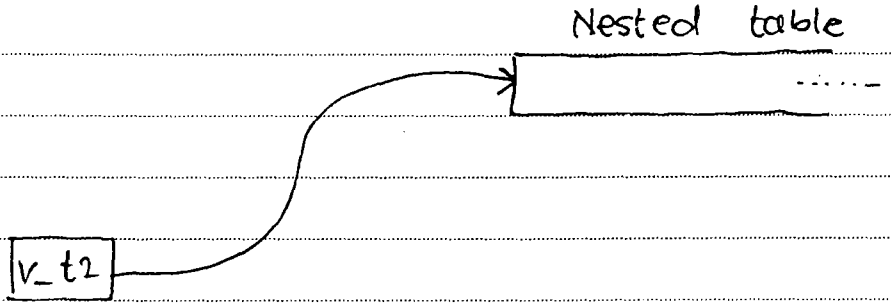
```
dbms_output.put_line ('v_t2 is null');
else
dbms_output.put_line ('v_t2 is not null');
end if;
end;
```

(output) ⇒ V_t1 is null
V_t2 is not null

i] Type t1 is table of number(10);
V_t1 t1;

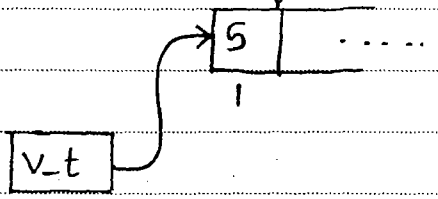


ii] Type t1 is table of number(10);
V_t2



iii] declare
Type t1 is table of number(10);
V_t t1 := t1();

```
begin
v_t  extend;
v_t(1) := 5;
dbms_output.put_line (v_t(1));
end;
```



? Write a pl/sql program to transfer all employee names from emp table & store it into nested table & also display content from nested table.

Soln
```
sql> declare
        type t1 is table of varchar2 (10);
        v_t  t1 := t1();
        cursor c1 is select ename from emp;
        n  number (10) := 1 ;
        begin
        for i in c1
        loop
        v_t.extend ();
        v_t (n) := i.ename;
        n := n+1 ;
        end loop ;
```

```
for i in v_t.first .. v_t.last
loop
dbms_output.put_line (v_t(i));
end loop;
end;
/
```

—(or)—

```
sql> declare
type t1 is table of varchar2(10);
v_t t1 := t1();
begin
Select ename bulk collect into v_t
from emp;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line (v_t(i));
end loop;
end;
/
```

Varray

Oracle 8.0 introduced varrays.
It is an user-defined datatype which is used to store number of dataitems in a single unit.

It is also same as arrays in normal language.

Here always index start with 1 and also indexes are consecutive.

We can also store varrays permantly into database using sql, this is also same as nested table i.e before we are storing actual data we must use constructor.

Syntax : ① Type typename is varray (maxsize) of datatype (size);

② variablename typename := typename ();

ex:
```
declare
type t1 is varray(10) of number(10);
v_t t1 := t1('a','b','c','d');
z boolean;
begin
dbms_output.put_line (v_t.first);
dbms_output.put_line (v_t.last);
dbms_output.put_line (v_t.prior(3));
dbms_output.put_line (v_t.next(2));
dbms_output.put_line (v_t.count);
for i in v_t.first.. v_t.last
loop
dbms_output.put_line (v_t(i));
end loop;
z := v_t.exists(3);
```

```
if z = true then
dbms_output.put_line ('ur index exists
with an element' || ' ' || v_t(3));
else
dbms_output.put-line ('index 3 does
not exists');
end if;
v_t.extend;
v_t(5) := 'e';
v_t.extend(2);
v_t(6) := 'f';
v_t(7) := 'g';
v_t.extend(3,2);
for i in v_t.first.. v_t.last
 loop
dbms_output.put_line (v_t(i));
end loop;
 v_t.trim(5);
dbms_output.put_line (v_t.count);
 v_t.delete;
dbms_output.put_line (v_t.count);
end;
/
```

? write a pl/sql program to transfer first 10 ename from emp table & store it into varray & also display content from varray.

WEDNESDAY
JUNE-12
6
158-208 WK-23

Soln
```
declare
type t1 is varray(10) of varchar2(10);
v_t t1 := t1();
cursor c1 is select ename from emp where rownum <=10;
n number(10):=1;
begin
for i in c1
loop
v_t.extend();
v_t(n) := i.ename
n:=n+1;
end loop;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
/
```

— 10 Sep —

Difference betn index by table, nested table, varray

Index by table	Nested table	Varray
This is an unconstraint table.	This is an unconstraint table.	This is an unconstraint table.
It is not	It is stored	It is stored

THURSDAY

JUN 12

7

159-207 WK-23

Stored perminently in database	Stored perminently in database using SQL.	Stored perminently in database using SQL.
we can not add or remove indexes.	we can add or remove indexes using extends, trim collection method.	We can add or remove indexes using extends, trim collection method.
Indexes starting from negative to positive numbers and also having key value pairs.	Indexes starting from 1.	Indexes starting from 1.
It supports exists, First, last, count, prior, Next, delete, delete (range of indexes) collection methods.	It supports exists, extends, trim, first, last, prior, next, count, delete, delete (range of indexes) collection methods	It supports exists, limit, extends, trim, first, last, prior, next count, delete Collection methods.

FRIDAY

JUN 12

8

160-206 WK-23

Bulk bind

whenever we are submitting pl/sql block always sql statements are executed in sql engine and always procedure statemens are executed in procedure statement executor.

Whenever pl/sql block contain large amount of coding & also sql & procedure statements are used frequently then internally oracle server control transfer betⁿ sql engine, pl/sql engine no. of times. These type of execution methods are also called as context switching execution method.

Always context switching exectution methods degrads performance of the appⁿ. To improve the performance of the application oracle introduce bulk bind process using collection i.e. in this process we are putting all sql statement related values into collection & in this collection we are performing insert, update, delete at a time using for all statement

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SATURDAY

JUN 12

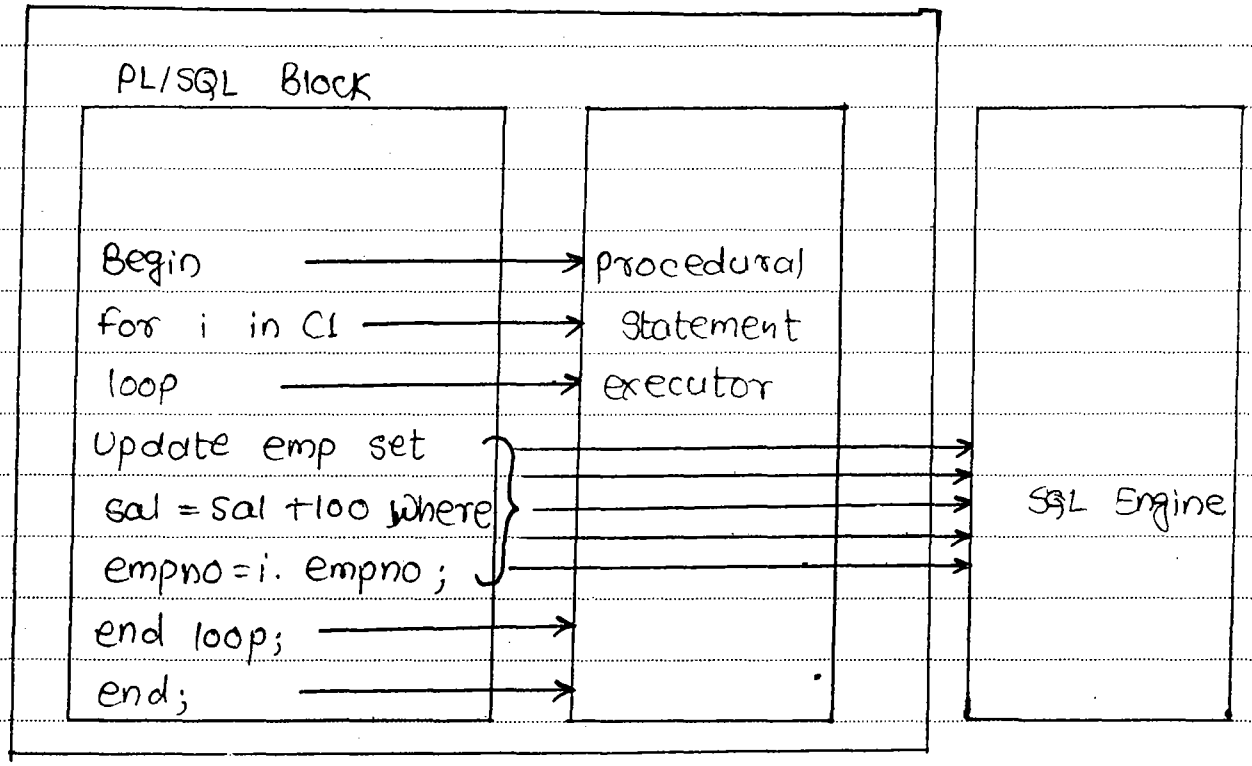
9

161-205 WK-23

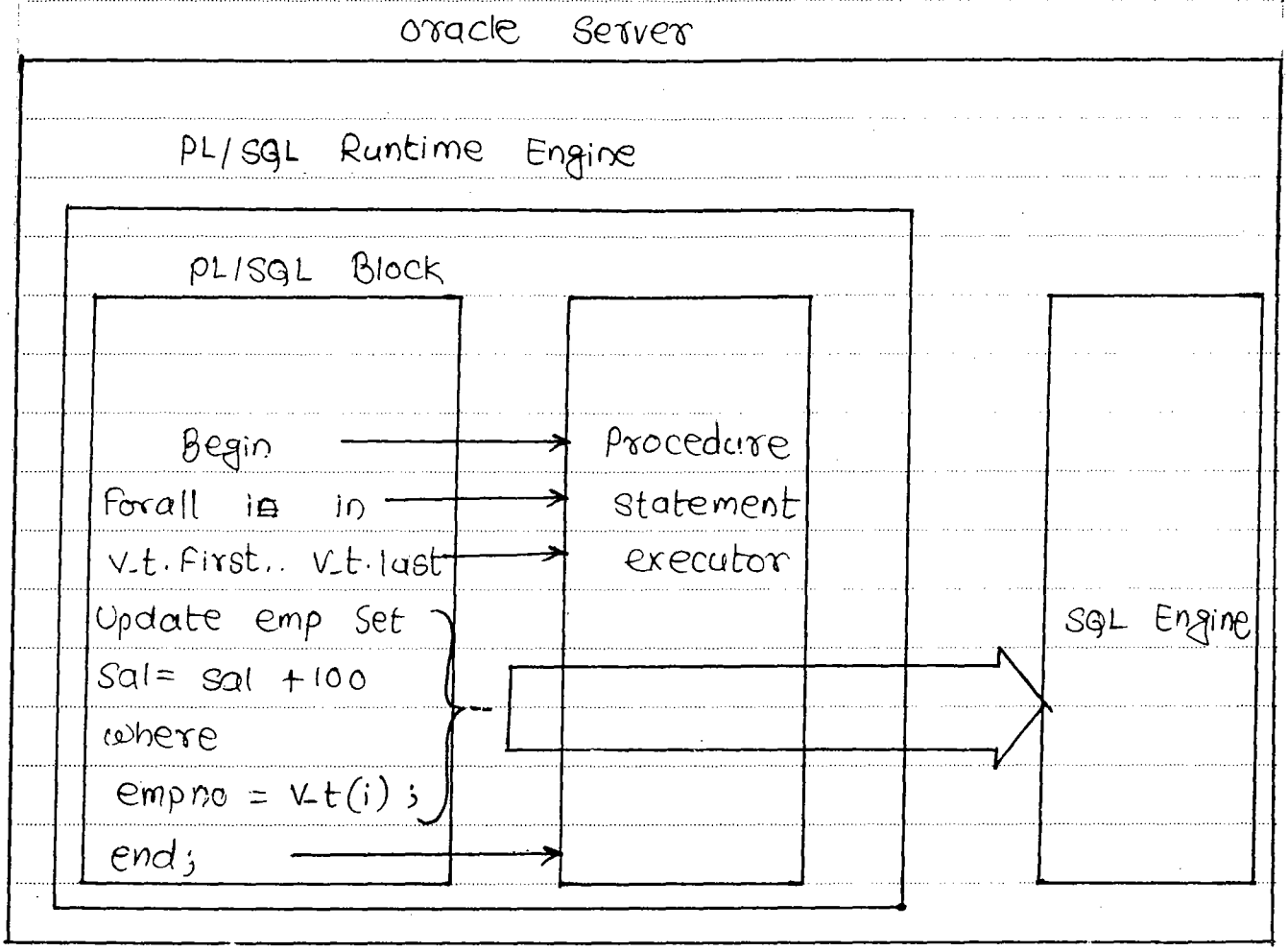
Without bulk bind

oracle server

PL/SQL Runtime Engine



With bulk bind



MONDAY

JUN 12

11

163-203 WK-24

Before we are perform bulk of dml operation using select clause we must fetch the data from the resource into collection we are using bulk collect clause i.e. before bulk bind process we must use bulk collect clause.

Bulk Collect

This clause is used to fetch data from resource into collection.

This clause used in

(i) select .... into .... clause

(ii) cursor .... fetch .... statement

(iii) dml .... returning .... clauses

(i) bulk collect used in select .... into .... clause

Syntax: select * bulk collect into collection var
from tablename where condition;

ex:
```
declare
type t1 is table of emp%rowtype
index by binary_integer;
v_t t1;
begin
select * bulk collect into v_t from emp;
for i in v_t.first.. v_t.last
loop
```

```
dbms_output.put_line (v_t(i).ename);
end loop;
end;
/
```

(ii) bulk collect used in cursor.... fetch.... statement

syntax: fetch cursorname bulk collect into
collectionvar [limit variablename]

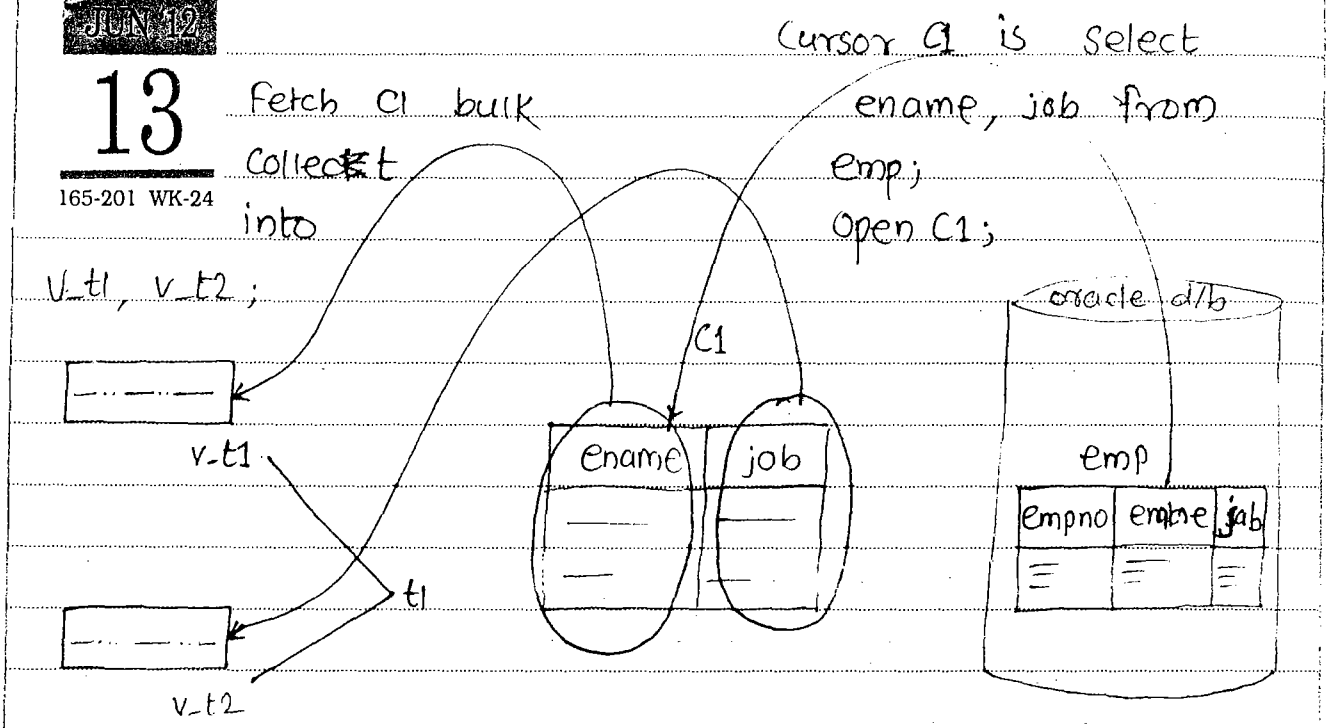
ex:
```
declare
type t1 is table of varchar2(10) index
by binary_integer;
v_t1 t1;
v_t2 t1;
cursor c1 is select ename, job from emp;
begin
open c1;
fetch c1 bulk collect into v_t1, v_t2;
close c1;
for i in v_t1.first .. v_t1.last
loop
dbms_output.put_line ( v_t1(i)||' '|| v_t2(i));
end loop;
end;
/
```

WEDNESDAY

JUN 12

13

165-201 WK-24



— 11 Sep —

ex:

```
declare
type t1 is table of all_objects.OBJECT_NAME%type
index by binary_integer;
v_t1  t1;
v_t2  t1;
cursor c1 is select OWNER, OBJECT_NAME from
all_objects;
z1  number(10);
z2  number(10);
begin
z1 := dbms_utility.get_time;
for i in c1
loop
null;
end loop;
```

June	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
z2 := dbms_utility.get_time ;
dbms_output.put_line ('elasped time for
 normal fetch' || (z2-z1) || ' ' || 'hsec' );
z1 := dbms_utility.get_time;
open c1;
fetch c1 bulk collect into v_t1, v_t2;
close c1;
z2 := dbms_utility.get_time;
dbms_output.put_line ('elasped time for bulk
 fetch' || ' ' || 'hsec' ) - (z2-z1) || ' ' || 'hsec' );
end;
/
```

(iii) bulk collect used in dml ..... returning clauses

Returning clauses used in dml statements only. These clauses are used to return values from table after processing & store it into variables.

Syntax: dml statement returning columnname into variablename;

ex: variable z varchar2(10);

```
sql> Update emp set sal = sal + 100 where
     ename = 'KING' returning job into :z ;

Sql> print z ;
```

FRIDAY

JUN 12

15

167-199 WK-24

(output) z

PRESIDENT

When we are processing bulk of data using implicit cursor we can also use bulk collect clause in dml ... returning clauses.

? Write a pl/sql stored procedure, modify salaries of the clerk from emp table & also these modified value immediately stored into index by table using dml... returning clause & also display content from index by table.

Sol^n

```
ex: create or replace procedure p1
    is
    type t1 is table of emp%rowtype
    index by binary_integer;
    v_t t1;
    begin
    update emp set sal=sal+100 where
    job='clerk' returning empno, ename, job,
    mgr, hiredate, sal, comm, deptno bulk
    collect into v_t;
    dbms_output.put_line ('affected no. of clerks
    are :' || sql%rowcount);
    for i in v_t.first .. v_t.last
    loop
```

```
dbms_output.put_line ( v_t(i).ename ||' '
v_t(i).sal ||' '|| v_t(i).job );
end loop;
end;
/

sql > exec p1;
```

Bulk Bind

In bulk bind process we are performing bulk of operations using collection i.e in this process we are using bulk update, bulk delete, bulk insert using forall statement.

Before we are using bulk bind process we are fetching data from dlb into coll^n using bulk collect clause.

Syntax:

```
forall indexvar in collectionvar.first ..
    collectionvar.last
update tablename set columname= newvalue
    where columnname = collectionvar (indexvar);
```

Ex:

```
declare
type t1 is varray (10) of number(10);
v_t
t1:= t1 ( 20, 30, 40, 50, 60, 70, 80 );
begin
forall i in v_t.first .. v_t.last
```

```
update emp set sal=sal+100 where
deptno= v_t(i);
end;
/
```

? write a pl/sql prog to modify salaries of all employees from emp table using bulk bind process in this case first we are fetching empno from emp table & store it into index by table & then only we are using bulk bind process.

Soln

```
declare
type t1 is table of emp%rowtype
index by binary_integer;
v_t t1;
begin
select empno bulk collect into v_t from emp;
forall i in v_t.First .. v_t.last
update emp set sal= sal+100 where
empno= v_t(i);
end;
/
```

ex:

```
declare
type t1 is table of emp%rowtype
index by binary_integer;
```

```
v_t t1;
begin
select empno bulk collect into
v_t from emp;
v_t delete (3);
forall i in v_t.first .. v_t.last
update emp set sal = sal +100 where
empno = v_t(i);
end;
/
```

error-: element at index [3] does not exist

Whenever collections having gaps then we are not allow to use bulk bind process.
To overcome this problem oracle 10g introduce indices of clase in bulk bind Process.
indices of clause internally works based on array indexes.

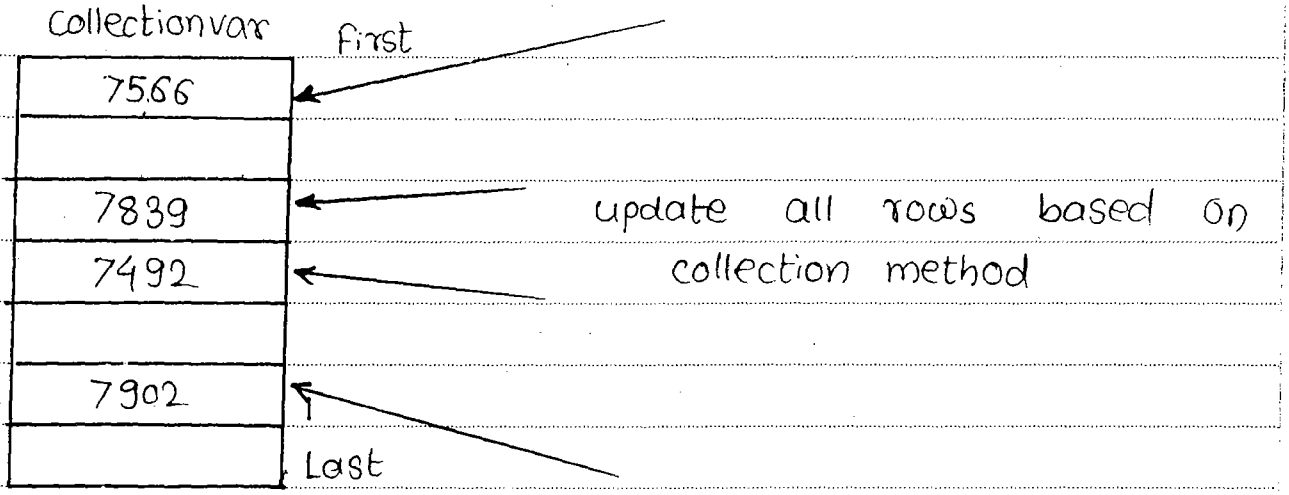
Syntax: forall indexvar in indices of collectionvar
update tablename set columnname = new value
where columnname = collectionvar (indexvar);

TUESDAY

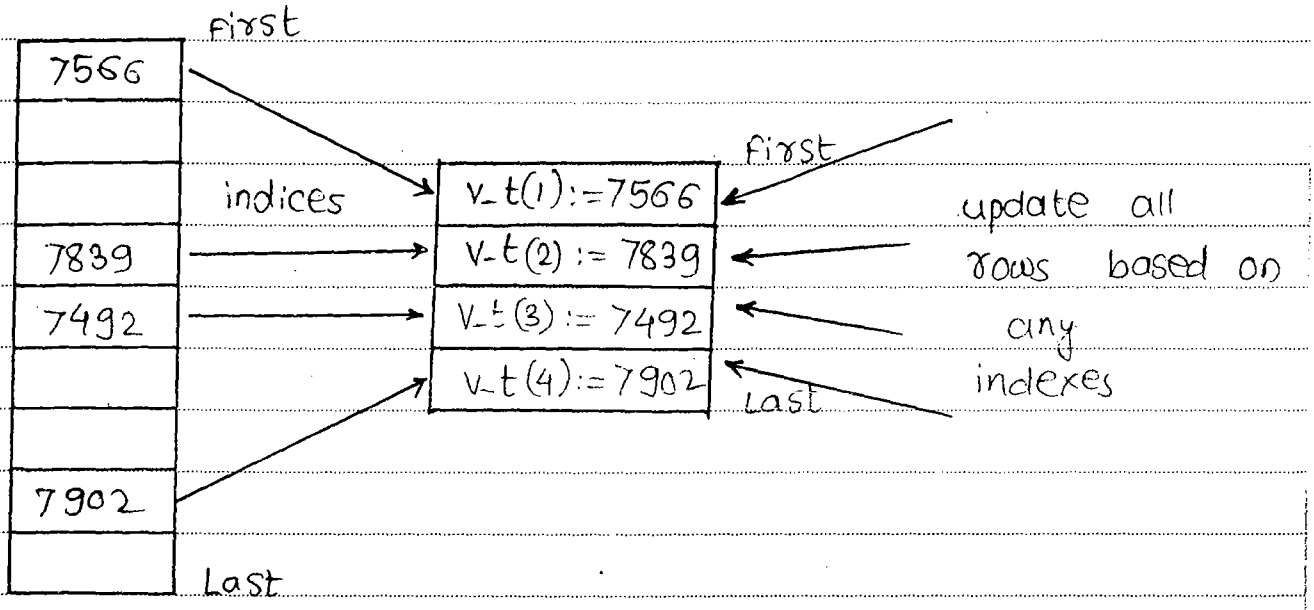
JUN 12

19

171-195 WK-25



Oracle 10g (indices of)



```
declare
type t1 is table of emp.empno%type
index by binary_integer ;
v_t t1;
begin
select empno bulk collect into v_t from emp;
v_t.delete(3);
forall i in indices of v_t
update emp set sal = sal +100 where
empno= v_t(i);
end;
/
```

sql%.bulk_rowcount

— 12 Sep —

oracle introduce sql%.bulk_rowcount attributes to interact number of rows in each itration in bulk bind process.

syntax: sql%.bulk_rowcount (indexvariablename)

ex:

```
declare
type t1 is varray(10) of number(10);
v_t t1 := t1 (20, 30, 50, 70);
begin
forall i in v_t.first .. v_t.last
update emp set sal = sal+100 where
deptno = v_t(i);
for i in v_t.first .. v_t.last
```

THURSDAY
JUN 12
21
173-193 WK-25

```
loop
dbms_output.put_line ('affected number
of rows in:' || ' ' || v_t(i) || ' ' || 'are' ||
' ' || sql %. ~~row~~ bulk_rowcount (i));
end loop;
end;
/
```

bulk delete

```
ex: declare
    type t1 is varray (10) of number (10);
    v_t t1 := t1 (20, 30, 40, 50, 60, 70);
    begin
    forall i in v_t.first .. v_t.last
    delete from emp where ~~v_t i~~ empno= v_t(i);
    end;
    /
```

bulk insert

```
ex: sql> create table test ( sno number(10) );

    sql> declare
         type t1 is table of number (10)
         index by binary_integer;
         v_t t1;
```

```
begin
for i in 1..5000
loop
v_t(i) := i ;
end loop;

forall i in v_t.first .. v_t.last
insert into test values (v_t(i));
end;
/

Sql> select * from test;
```

dbms_utility package

This package used by either database administrator or database developers.

This package having get_~~and~~ time method which returns elasped time.

This method returns always number datatype.

This method is used by database administrator to calculate elasped time.

Syntax: variablename := dbms_utility.glt_time ;

This package used by developers because this package internally having index by table.

SATURDAY

JUN 12

23

175-191 WK-25

This package also having two methods

i) Comma_to_table

ii) table_to_comma

These methods are used to transfer pl/sql values into comma seperated Strings & comma seperated Strings into pl/sql table values. Before we are using these methods we must create an index by table variable using uncl_array type from dbms_utility package.

Syntax: variablename dbms_utility*.uncl_array ;

i) Comma_to_table

This method is used to convert comma seperated strings into index by table values.

Syntax: dbms_utility.comma_to_table (stringname,
binary_integervariablename,
indexby tablevariable);

ex:
```
declare
    v_t  dbms_utility*.uncl_array ;
     z   binary_integer;
    str  varchar2(200);
begin
    str := 'a,b,c,d,e,f' ;
```

June 2012

Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
dbms_utility. comma_to_table (str, z, v_t);

for i in v_t. first .. v_t. last
 loop
dbms_output. put_line ( v_t (i));
end loop;
end;
 /
```

ii) table_to_comma

This method is used to convert index by table values into comma seperated string.

Syntax: dbms_utility. table_to_comma (Index by tablevar iable, binary-integervariable, Stringname);

? write a pl/sql program using dbms_utility package, display all dept-names from dept table into seperate string.

Soln

```
declare
    v_t   dbms_utility. uncl-array;
    z   binary_integer;
     str   varchar2 (200);
     begin
    select dname, bulk collect into v_t from dept;
   dbms_utility. table_to_comma (v_t, z, str);
   dbms_output. put_line (str);
   end;
```

MONDAY

JUN 12

25

177-189 WK-26

LOBS (Large objects)

Oracle 8.0 introduced large objects.

If we want to store more than 4000 bytes of alphanumeric data, we are using long datatype.

This datatype stores upto 2GB data but there can be only one long column for a table & also we can not create primary key on that column.

To overcome these problems Oracle 8.0 introduces CLOB datatype.

Syntax: Columnname long

Syntax: Columnname clob

ex: sql> create table k1(col1 long);

If we want to store binary data we are using RAW datatype, but raw support upto 2000 bytes binary data.

If you want to store more than 2000 bytes of binary data we are using long raw datatype.

Syntax: columnname raw (maxsize);

Syntax: columnname long raw;

ex: Sql> create table k2 (col1 raw(100));

ex: Sql> create table k3 (col1 long raw);

If we want to store more than 2000 bytes of binary data we are using blob datatype

Syntax: Columnname blob

There are two types of large objects supported by oracle.

1. Internal Large Objects
2. External Large objects

Internal large objects are stored within database There are two types of internal large objects supported by oracle

(a) Clob (character large object)
(b) blob (binary large object)

External large objects are stored outside of the database.
This is a b-file datatype

before we are handling large object we must create an alise directory related to physical directory using

WEDNESDAY
JUN 12

27

179-187 WK-26

following Syntax

Syntax: Create or replace directory
directoryname as 'path of physicalfile';

Before we are creating alise directory admin user must give create any directory privileage using following syntax

Syntax: grant create any directory to
username1, username2, ...

ex: sql> conn sys as sysdba;
Enter password: sys

sql> grant create any directory to scott;

sql> conn scott/tiger;

sql> create or replace directory zzz as 'E:\';

Storing large amount of data or in image into database

Step 1 :- Create a table in oracle database.

step 2 :- develop a pl/sql block using dbms_lob Package

step-i) before we are storing actual data into lob table we must initialize using empty_clob() or empty_blob() functions & also using returning clauses we are returning point to appropriate lob column.

Syntax: Insert into tablename values (empty_clob())
returning columnname into variablename;

Step-ii) we must concatenate actual file with alise directory using b-filenam-e function.
This fun also returns bfile datatype.

Syntax: variablename := bfilename ('alisedirectory', 'actual filename');

step 3 :- Using load from file method from dbms_lob package we are loading large amount of data into appropriate column.

Syntax: dbms_lob.loadfromfile (clobvariable, bfilevar, length of bfilevar);

Step 4 :- Before we are loading data we must open the file using fileopen method from dbms_lob package, And also after processing close the file using fileclose method from

FRIDAY

JUN 12

29

181-185 WK-26

dbms_lob package.

Syntax: dbms_lob.fileopen (bfilevar);

dbms_lob.fileclose (bfilevar);

```
Sql> create table test (col1 number(10), col2 clob);
ex: declare
    v_clob clob;
    v_bfile bfile;
    begin
    insert into test values (1, empty_clob())
    returning col2 into v_clob;
    v_bfile := bfilename ('ZZZ', 'first.txt');
    dbms_lob.fileopen (v_bfile);
    dbms_lob.loadfromfile (v_clob, v_bfile, dbms_lob.
    getlength (v_bfile));
    dbms_lob.fileclose (v_bfile);
    end;
    /

sql> Select * from test;
```

UTL_File package

Oracle 7.3 introduce utl_file package.

This package used to write a data into a file & read data from a file.

Before we are performing operations admin user must give read, write privileges on alise

directory using following syntax

syntax: grant read, write on directory directoryname to username;

ex: sql> conn sys as sysdba;
Enter password: sys

sql> grant read, write on directory zzz to scott;

Writing data into a file

Step-1> Before we are performing file operations we must create a file pointer oph from file_type in utl_file package.

syntax: filepointervar UTL_file.file_type;

Step-11> we must open the file before we are performing read, write operation using fopen method from utl_file package. This method accepts three parameters & also return file_type datatype.

syntax: filepointervar := utl_file.fopen ('alisedirectoryname', 'filename', 'mode');

Notes

step-III) Using putf method from utl_file package we are storing actual data into a file.

Syntax: utl_file.putf (filepointervar, 'actualdata');

step-IV) After processing file operations we must close the file using fClose method from utl_file package.

Syntax: utl_file.fclose (variablename);

ex:
```
declare
 fp utl_file.file_type;
 begin
 fp:=utl_file.fopen ('zzz', 'fan.txt', 'w');
 utl_file.putf (fp, 'tommorow closs shifed
  into block3');
 utl_file.fclose(fp);
end;
/
```

? write a pl/sql program using utl_file package store all employee names into an external file using emp table.

```
declare
 fp utl_file.file_type;
```

ACTION PLAN

JULY '12

WK	Mon	Tue	Wed	Thu	Fri	Sat	Sun
	30	31					1
27	2	3	4	5	6	7	8
28	9	10	11	12	13	14	15
29	16	17	18	19	20	21	22
30	23	24	25	26	27	28	29

OBJECTIVES THIS MONTH

DATE	DESCRIPTION	REMARKS

JUNE '12

M	T	W	T	F	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

2012

AUGUST '12

M	T	W	T	F	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

SUNDAY
JUL 12
1
183-183 WK-26

```
Cursor c1 is select ename from emp;
begin
fp := utl_file.fopen ('zzz', 'fan.txt', 'w');
for i in c1 
loop
utl_file. putf (fp, i.ename);
end loop;
utl_file. fclose (fp);
end;
/
```

Reading data from file

If we want to read data from a file we are using get_line() from utl_file package

Syntax: utl_file. getline (filepointervariable, buffervariable);

```
ex: declare
    fp utl_file. file_type;
    x varchar (200);
    begin
    fp:= utl_file. fopen ('zzz', 'first.txt', 'r');
    utl_file. get_line (fp, x);
    dbms_output. put_line ('data from file' || ' ' || x );
    utl_file. fclose (fp);
    end;
    /
```

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

MONDAY
JUL 12

SQL Loader

2

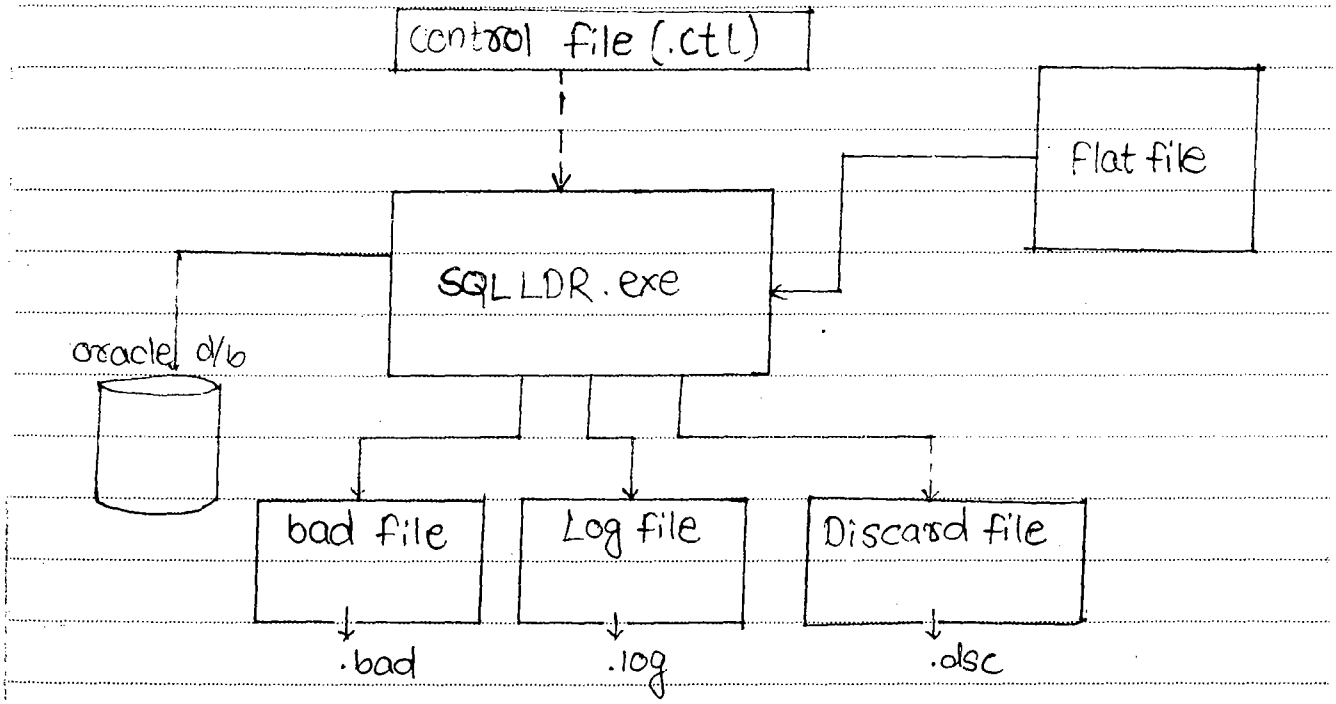
184-182 WK-27

SQL Loader is an utility program which is used to transfer data from flat file & store it into oracle database.

SQL Loader is also called as Bulk Loader.

Always SQL Loader executes control file based on the type of flat file we are creating control file & submit to the sql loader then only sql loader fetch data from flat file & store it into target table. During this process sql loader generates following types of files

1. Logfile (.log)
2. Bad file (.bad)
3. Discard file (.dsc)



August	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

02-07

TUESDAY

JUL 12

3

185-181 WK-27

Log file stores all other files information and also stores loaded, skipped, rejected record numbers and also stores elasped time.

Bad, discard file stores rejected records.

Flat File

Flat file is a structured file, which contains data.

There are two types of flat files supported by all system.

(i) variable Record FlatFile –

(ii) fixed Record FlatFile

A flatfile which contains delemeters is called variable record flat file.

ex: 101, abc, 2000

102, xyz, 3000

A flatfile which does not contain delemeters is called fixed record flat file.

ex: 101 abc 2000

102 xyz 3000

July 2012	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

03-07

Control file

This file extension is .ctl, based on the type of flat file we are generating control file

Creating a control file for variable record flat file

Always control file execution start with load data clause.

After load data clause we must specify path of the flat file using in file clause.

Syntax: load data
infile ' path of flatfile'

Note

If control file having a flat file then we must use * in place of flat file path in in file clause & also use begin data clause above of the flat file

ex: load data
infile *
——..——
——..——
begin data
101, abc, 2000
102, xyz, 4000

THURSDAY

JUL 12

5

187-179 WK-27

Using into table tablename clauses we are loading data into target table.

Before into table tablename clauses we are using either insert or truncate or replace or append clauses.

If target table is an empty table then only we are using insert and also by default clause is insert.

After into table tablename clauses we are using following clauses

a) fields terminated by 'delimetername'

b) optionally enclosed by 'delimetername'

After these clauses we must specify target table column within paranthesis.

```
load data
infile 'path of flatfile'
badfile 'path of badfile'
discardfile 'path of discardfile'

insert/truncate/replace/append into table
  tablename
  fields terminated by 'delimetername'
  optionally enclosed by 'delimetername'
    (col1, col2, ... coln)
```

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

05-07

FRIDAY
JUL 12
6
188-178 WK-27

invoking sqlloader

d:\>sqlldr userid = scott/tiger
control = ——.._ filename.ctl

101, abc, 2000
102, xyz, 3000

sql> create table target (empno number(10),
ename varchar2(10), sal number(10));

load data
infile 'E:\one.txt'
insert into table target fields terminated
by',' (~~ename~~ mpno, ename, sal)

D:\> sqlldr userid=scott/tiger

control= E:\murali.ctl

Storing Default Values

Using constant clause we are storing default values into database.

When flat file having less no. of fields & target table having more no. of fields then only we are using constant clause.

SATURDAY

JUL '12

7

189-177 WK-27

Syntax: columnname constant 'actualvalue'

If we want to skip no. of columns within flatfile then we are using filler clause, i.e if flat file having more no. of field & target table having less no. of columns then we are using filler clause.

ex.

```
load data
infile 'E:\one.txt'
insert
into table target fields terminated by ','
(empno, ename, loc constant 'ammerpet')
```

ex:

```
load data
infile 'E:\one.txt'
insert
into table target fields terminated by ','
(empno, ename filler, sal)
```

Bad files stores rejected records that cause an error based on following condition

a) Data type mismatch
b) Business rule violation

This file extension is .bad.

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

07-07

we can also create bad file explicitly using bad file clause & also we are specifying .xls or .doc.
↓
badfile extension

a) Data type mismatch

101, abc, 2000
'102', xyz, 3000
'103', pqr, 4000
104, zzz, 5000

Sql> create table target (empno number(10),
ename varchar2(10), sal number(10));

load data infile 'E\:one.txt'
insert into target ~~ta~~ fields terminated by ','
↓
table
(empno, ename, sal)

'102', xyz, 3000
'103', pqr, 4000

Note

when we are transfering data from flat file into oracle d/b using sql loader automatically log file is created as same name as control filename & also bad file is created as same

MONDAY

JUL 12

9

191-175 WK-28

name as ~~flat~~bad file filename

b) Business Rules Violation

ex: 101, abc, 7000
102, xyz, 3000
103, zzz, 4000
104, ggg, 8000

sql> create table target (empno number(10),
ename varchar2(10), sal number(10)
check (sal > 6000));

102, xyz, 3000
103, zzz, 4000

RECNUM

This clause is used to assign no. to records in d/b table.
This clause internally assign no. to loaded, skipped, rejected records.

Syntax: Columnname recnum

ex: 101, abc, 7000
'102', xyz, 3000
'103', zzz, 4000
104, ggg, 8000

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

sql> select * from target;

Dates used in Control file

i] Using date datatype
ii] Using to_date() function

i] Using date datatype

Syntax: Columnname date "FlatFiledateformat"

ex: 101, abc, 09/05/03
102, xyz, 06/04/12
103, zzz, 07/08/11

sql> create table target (empno number(10),
ename varchar2(10), doj date);

load data infile 'E:\one.txt'
insert into table target fields terminated
by ',' (empno, ename, doj date
"DD/MM/YY").

Functions using Control file

we can also use predefined, user defined
oracle fun within control file but in this case
we must specify fun functionality withine double

WEDNESDAY

JUL 12

11

193-173 WK-28

quotes & also use a colon operator infront of the columnname in function functionality.

syntax: columnname "functionname (:columnname)"

ex: 101, abc, m
102, xyz, f
103, zzz, m

sql> create table target (empno number(10), ename varchar2(10), gender varchar2(10));

load data infile 'E:\one.txt'
insert into table target fields terminated by ',' (empno, ename, gender "decode (:gender, 'm', 'male', 'f', 'female')")

Discard File

Discard file stores rejected records that cause error then we are using when clause

But discard file we must specify explicity using discard file clause.

This file extension is .dsc

ex: 101, abc, 10
102, xyz, 20

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

103, zzz, 30
104, aaa, 10

```
load data infile 'E:\one.txt'
discard file 'E:\sunday.dsc'
insert into table target when deptno='10'
Fields terminated by ','
(empno, ename, deptno)
```

```
sql> create table target (empno number(10),
     ename varchar2(10), deptno number(10));
```

102, xyz, 20
103, zzz, 30

Note

In when clause we must specify condition value within single quote.
In when clause we are using "=", "!=" relational operators only.
In when clause we can use only logical operator 'and' we are not allow to use logical operator 'or'.

Creating control file for fixed Record Flatfile

A flatfile which does not have delimeter is

FRIDAY

JUL 12

13

195-171 WK-28

called fixed record flat file.

When we are using fixed record flat file we must use position clause in control file.

In position clause we must specify starting, ending position of the every field within the Flat file using color operator. Alongwith position clause we must specify sql loader datatypes.

SQL Loader supports 3 datatypes:

1. integer external
2. decimal external
3. Char

Syntax: Columnname position (startingposition: endingposition) sqlloaderdatatype

Note

When we are using fun^ns within control file then we are not allow to use sqlloader datatype in place of this one we are using fun^ns functionality.

Syntax: columnname position (startingpos: endingpos) " function (:columnname)"

ex: 101 abc 1000

102 xyz 2000

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

103 zzz 3000
104 aaa 1000

```
sql> create table target (empno number(10),
     ename varchar2(10), sal number(10));

load data infile 'E:\one.txt'
insert into table target (empno
position (01:03) integer external,
ename position (04:06) char,
sal position (07:10) integer external)
```

Sequences Used in Control file

Syntax: columnname "sequencename.nextval"

ex:
101
102
103
104
105
106

```
sql> create table target (sno number(10));
sql> create sequence s1 start with 1;

load data infile 'E\:one.txt'
insert into table target (sno "s1.nextval")
```

SUNDAY

JUL 12

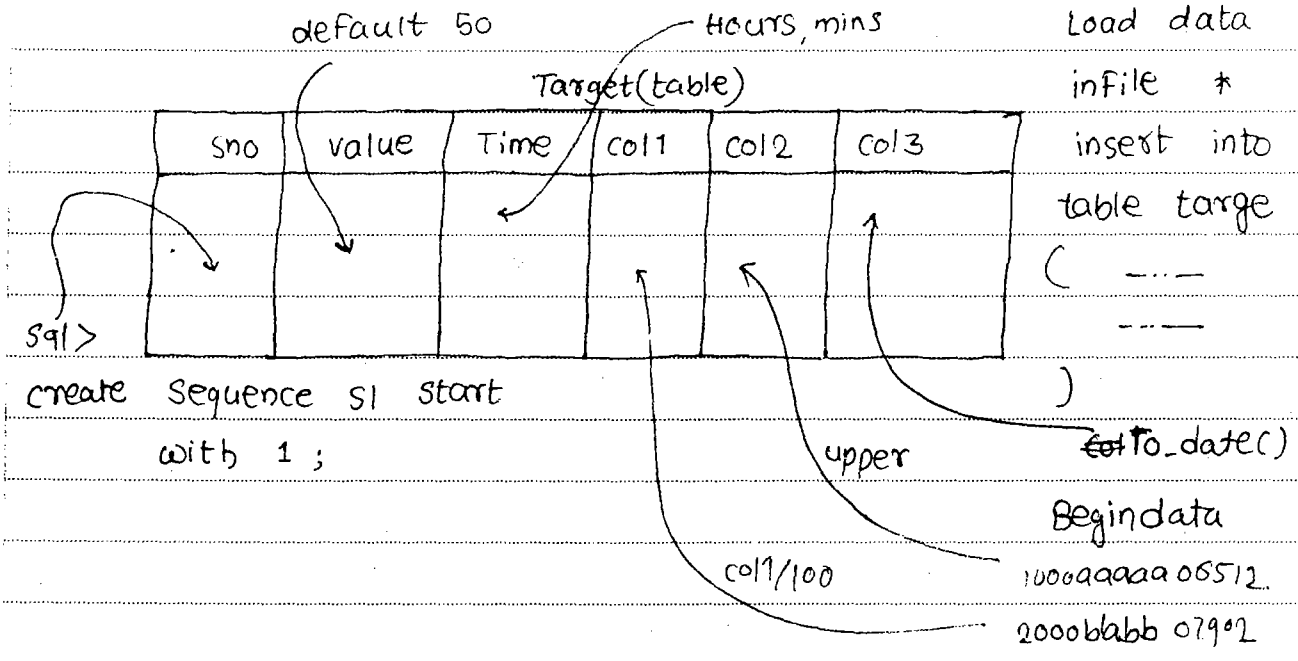
15

197-169 WK-28

Flat file used in control file self

ex: Sql> create table target (empno number (10), ename varchar2 (10), sal number (10));

```
load data infile * insert into table target
        fields terminated by ','
        (empno, ename, sal)
        begindata
        1021, abc, 1000
        1022, xyz, 2000
        1033, xxx, 5000
```



July 2012	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

MONDAY
JUL 12
16
198-168 WK-29

```
( sno    "sl.nextval",
  value  constant'50',
  time   "to_char (sysdate, 'HH12:MI:SS')",
  col1   position (01:05)  ":col1/100",
  col2   position (06:13)  "upper (:col2)",
  col3   position (14:19)  "To_date (:col3, 'DDMMYY')"
)
```

Note

When flatfile contains quotations then we are using optionally enclosed by clause within control file.

Using sql loader we can also transfer data from ~~inse~~ one flat file into multiple target table using multiple into clauses. And also no. of flat files data transfer into single target table using multiple infile clause but using sql loader we can not transfer source as ~~one~~ ~~flat~~file, database source as different databases.

Nested Block

Block within another block is called nested block. Generally child block is also called as nested block. In nested block we can also use same variable name with parent block, child block.

TUESDAY

JUL '12

17

199-167 WK-29

Generally nexted block executable Statement access nested block variables. If we want to access parent block variable we must use parent block name within executable statement. Before that one must supecify parent block name within double angular bracket.

```
ex.  <<parent>>
     declare
     v_empno      emp.empno%type :=&empno1;
     begin
    <<child>>
     declare
     v_empno   emp.empno%type :=&empno2;
     v_ename   emp.empname%type;
     begin
    select  ename  into  v_ename  from  emp
     where  empno = parent.v_empno;
    dbms_output.put_line (v_ename);
    end;
    end;
    /
```

(output) enter value for empno1 : 7902
enter value for empno2 : 7566

FORD

When a pl/sql block contains no. of select -- into clauses & also if you want to display flexible messages than the generalized messages in exception handler we must use nexsted block.

When exceptions are raised in declare section, exception section those exceptions must be handled using outer block only.

ex.
```
declare
v_empno1    emp.empno%type := &empno1;
v_empno2    emp.empno%type := &empno2;
v_ename     emp.ename%type;
begin
begin
select ename into v_ename from emp where
 empno = v_empno1;
dbms_output.put_line ('my first employee'||''||
 v_ename );
exception
 when  no_data_found  then
dbms_output.put_line (' your employee
 does not exist' with employee number'
  ||' '|| v_empno1 );
end;
begin
select ename into v_ename from emp
where   empno = v_empno2;
```

```
dbms_output.put_line ('my second
employee' || '' || v_empname);
exception
when no_data_found then
dbms_output.put_line ('your employee
does not exist with empno' || ' ' ||
v_empno2 );
end;
end;
/
```

Note

Nested blocks also access parent block variables & their own variables.

Local Procedures, Local Functions

Local subprograms are used to solve some perticular task.

These subprograms are not stored in database

These subprograms does not create or replace key words.

These subprograms are created in either annonyms or in nested blocks or in stored procedures.

Always these subprograms are defined in declare section of the pl/sql block, & also call these blocks in immediate executable section.

FRIDAY
JUL 12
20
202-164 WK-29

We must define local subprograms in last of the declare section in pl/sql block.

Syntax:

```
declare
        variable declarations;
        cursor declarations;
        types declarations;
procedure procedurename (formal
 parameters)
is
---
begin
---
[exception]
---
end [procedurename];
function functionname (formal
 parameters) return datatype
is
---
begin
---
return expr;
end [functionname];
begin
procedurename (actual parameters);
var:= functionname (actual params);
end;
```

SATURDAY

JUL '12

21

203-163 WK-29

Passing PL/SQL Table As In Parameter Into Local Procedure

```
ex.   declare
      type t1 is table of emp%rowtype
      index by binary_integer;
      v_t t1;
      procedure p1 (p_t in t1)
      is
      begin
      for i in p_t.first.. p_t.last
      loop
      dbms_output.put_line ( p_t(i). ename);
      end loop;
      end p1;
       begin
      select * bulk collect into v_t from emp;
      p1 (v_t);
       end;
       /
```

Passing PL/SQL Table As Out Parameter Into Local procedure

```
ex.   declare
      type t1 is table of emp%rowtype
      index by binary_integer;
      v_t t1;
```

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
procedure p1 (p_t out t1)
 is
 begin
 select * bulk collect into p_t from emp;
end p1;
begin
 p1 (v_t);
for i in v_t.first.. v_t.last
 loop
 dbms_output.put_line ( v_t(i).ename);
 end loop;
 end;
 /
```

Return PL/SQL Table As Local Function

Return Type

ex:

```
declare
type t1 is table of emp%rowtype
index by binary_integer;
 procedure p1 (p_t t1)
 is
begin
 for i in p_t.first.. p_t.last
loop
 dbms_output.put_line (p_t(i).ename);
end loop;
```

MONDAY

JUL 12

23

205-161 WK-30

```
function f1 return t1
 is
v_t  t1;
begin
select * bulk collect into v_t
from emp;
return v_t;
end f1;
 begin
p1(f1);
end;
/
```

Dynamic SQL

Oracle 7.1 introduced dynamic SQL.
It is the combination of SQL, PL/SQL i.e SQL statements are executed dynamically with PL/SQL block using execute immediate clause.
Generally in PL/SQL block we are not allow to use DDL, DCL statements. Using dynamic SQL DDL, DCL statements within PL/SQL block.

Syntax:

```
begin
execute immediate 'sql statement';
end;
/
```

July 2012

Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```
ex: begin
    execute immediate 'create table
    x1 (sno number(10))';
    end;
    /
```

Passing Values Into Dynamic SQL Statement

through using clause we are passing values into dynamic SQL statement.

Here we are passing values either statically or dynamically using placeholders.

In oracle place holders are represented using colon operator.

? Write a dynamic SQL program to insert a record into dept table.

Sol?

```
declare
    v_deptno number(10):= 5 ;
    v_dname varchar2(10) := 'a' ;
    v_loc varchar2(10) :='b' ;
    begin
    execute immediate 'insert into dept values
    (:1, :2, :3)' using v_deptno, v_dname, v_loc ;
    End;
    /
```

WEDNESDAY

JUL '12

25

207-159 WK-30

Retriving Values From Dynamic SQL Statements

through into clause we are retriving values from dynamic statement.

? write a dynamic SQL prog to display no. of records no. from emp table.

```
Sol^n  declare
       z number(10);
       begin
       execute immediate ' select count(*) from
       emp' into z ;
       dbms_output.put_line (z);
       end;
```

? write a dynamic SQL prog to retrive all employee names from emp table & store it into index by table & also display the content from index by table.

```
Sol^n  declare
       type t1 is table of varchar2(10)
       index by binary_integer ;
       v_t t1;
       begin
       execute immediate 'select ename from emp'
```

```
bulk collect into v_t;
for i in v_t.first .. v_t.last
loop
dbms_output.put_line ( v_t(i));
end loop;
end;
```

Note

When a dynamic SQL statement contains using, into clauses always into clause preceeds using clause.

? Write a dynamic SQL prog for passing deptno 20 retrive deptname, loc from dept table.

Sol?

```
declare
v_deptno number(10) :=20;
v_dname varchar2 (10);
v_loc varchar2 (10);
begin
execute immediate ' select dname, loc from
dept where deptno:=1 ' into v_dname, v_loc
using v_deptno;
dbms_output.put_line (v_dname||' '||v_loc);
end;
/
```

FRIDAY
JUL 12
27
209-157 WK-30

Oracle Versions

Oracle 2.0 ---> 1979

---> basic sql functionality
---> joins

Oracle 3.0 ---> 1983

---> Commit, rollback
---> rewritten in C language

Oracle 4.0 ---> 1984

---> read consistancy

Oracle 5.0 ---> 1986

---> Client-server architecture

Oracle 6.0 ---> 1988

---> PL/SQL introduced
---> row level locks

Oracle 7.0 ---> 1992

---> datatype varchar changed into varchar2

July	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
2012	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SATURDAY
JUL '12
28
210-156 WK-30

---> integrity constraints
---> stored procedures, functions, triggers;
---> truncate table
---> View Compilation

Oracle 7.1

---> ANSI / ISO SQL92
---> Dynamic SQL

Oracle 7.2

---> ref cursors (or) cursor variables
---> inline views
---> dbms_job package

Oracle 7.3

---> Bitmap indexes
---> utl_File package

Oracle 8.0 ---> 1996

Object technology
nested tables, varrays
large objects
columns increased per a table upto 1000

August 2012	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

SUNDAY

JUL 12

29

211-155 WK-30

Oracle 8i

---> Analytical function
---> instead of triggers
---> materialized views
---> Functionbased indexes

Oracle 9i ---> 2001

---> rename a column
---> merge statements
---> ANSI joins
---> multitable inserts
---> flashback query

Oracle 10g ---> 2005

---> rename tablespace
---> flashback table
---> recycle bin
---> wm_concat()
---> indices of clause

Oracle 11g ---> 2007

11g introduce read only tables along with alter command.

July 2012	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Syntax: alter table tablename read only;

Syntax: alter table tablename read write;

Virtual Column — Before 11g if you want to store stored expr in d/b we are using fun based indexes, view. But 11g introduce virtual column using generated always as clause directly storing stored expression within database.

syntax: columnname datatype (size) generated always as expression [virtual]

ex: sql> Create table w1 (a number (10), b number (10) c number (10) generated always as (a+b) virtual);

sql> insert into w1 values (10, 20);

sql> select * from w1;

If we want to view virtual expression we are using data_default property from user_tab_columns datadictonary.

sql> desc user_tab_columns;

TUESDAY

JUL '12

31

213-153 WK-31

```
sql> Select data_default from
     user_tab_column where
     table_name = 'W1';
```

Simple_integer data in pl/sql

This datatype performance is very high compare to binary_integer, pls_integer_datatype This datatype can not accept null values.

```
ex: declare
    a simple_integer := 50;
    begin
    dbms_output.put_line (a);
    end;
    /
```

Oracle 11g introduce continuous statement in pl/sql loops as same as c-language Continuous Statement.

Oracle 11g introduce follows clause in triggers.

11g introduce variable assignment concept when we are sequences in pl/sql blocks.

Oracle 11g introduce enable, disable Keywords within trigger specification itself.

Oracle 11g introduce named, mixed notations in functions calling in select statements.

- mvmsrinivas@gmail.com 9959430399

July 2012	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31