NATIONAL UNIVERSITY OF COMPUTER & EMERGING SCIENCES

CL 203-Database Systems Lab

Lab Session 11

PL/SQL FUNCTION: is a named block that returns a value. PL/SQL functions are also known as subroutines or subprograms. To create a PL/SQL function, you use the following syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter[,parameter])]

RETURN return_datatype IS

[Declaration_section]

BEGIN

executable_section

EXCEPTION
[exception_section]
END [function_name];
```

Example:

```
CREATE [OR REPLACE] FUNCTION simple RETURN varchar2 IS
BEGIN
RETURN 'SIMPLE FUNCTION';
END simple;
```

Function Without Parameters Used In a SELECT Clause

```
CREATE [OR REPLACE] FUNCTION getemp RETURN emp.ename%TYPE IS
Var1 emp.ename%TYPe;

BEGIN
SELECT ename
INTO var1
FROM emp
WHERE empno = (SELECT empno FROM emp WHERE rownum=1);

RETURN var1;

EXCEPTION
When OTHERS THEN
```

```
RETURN 'employee not present';
END getemp;
```

TO Execute the Function:

```
SELECT getosuser FROM dual;
```

Simple Function Used in An INSERT Statement

```
INSERT INTO emp VALUES ('7946',getemp,'CLERK','7782',sysdate,'1300.00',NULL,'10');

Select * FROM emp WHERE empno = '7946';
```

Simple function used in a WHERE Clause

```
SELECT * FROM emp WHERE ename = getemp;
```

Simple Function Used In A View

```
CREATE OR REPLACE VIEW emp_view AS
SELECT * FROM emp WHERE ename = getemp;
```

Find out the output yourself!

Function with OUT parameter

```
CREATE OR REPLACE FUNCTION out_func (outparm OUT VARCHAR2)
RETURN VARCHAR2 IS

BEGIN
outparm:='out param';
RETURN 'return param';
END out_func;
/

set serveroutput on;

DECLARE
retval VARCHAR2(20);
outval VARCHAR2(20);

BEGIN
retval:=out_func(outval);
dbms_output.Put_line(outval);
dbms_output.Put_line(retval);
END;
/
```

Function with IN OUT parameter

```
CREATE OR REPLACE FUNCTION inout_func (outparm IN OUT VARCHAR2)
RETURN VARCHAR2 IS

BEGIN
outparm:='coming on';
RETURN 'return param';
END inout_func;
/

set serveroutput on;

DECLARE
retval VARCHAR2(20);
ioval VARCHAR2(20):='Going in';

BEGIN

dbms_output.Put_line('In:'||ioval);
retval:= inout_func(ioval);
dbms_output.Put_line('Out:'||ioval);
dbms_output.Put_line('Return:'|| retval);
END;
/
```

Procedures

PL/SQL procedures behave much like procedures in other programming language.

```
CREATE OR REPLACE PROCEDURE p_procedure1 IS
BEGIN
null;
END p_procedure1;
```

Example of Stored Procedure with Parameters:

```
CREATE OR REPLACE PROCEDURE p_procedure2 (p_name VARCHAR2) IS
BEGIN
dbms_output.put_line(p_name);

END p_procedure2;
```

The Parameters that you pass to a stored procedure can be of three types (IN, OUT, IN OUT).
IN: is the default type. So if you don't specify any parameter IN is used. Thi is telling that I am supplying a parameter to the stored procedure.

OUT: is used to get values back from the stored procedure.
IN OUT: is used to pass values to a stored procedure and to get values from the stored procedure using the same parameter.

Unlike oracle functions we can return multiple values using the out parameter.

For Example: if you want to get back the employee name of an employee and you have just the emp_no (employee number) then you can pass the emp_no to a stored procedure using an IN parameter and then get the employee name using an out parameter.

```
CREATE OR REPLACE PROCEDURE p_procedure3 (p_emp_nno IN number, p_name OUT
varchar2) IS
BEGIN
SELECT ename
INTO p_name
FROM emp
Where emp_no = p_emp_no;
END p_procedure3;

DECLARE
I_name varchar2(20)

BEGIN
p_procedure3(7946,I_name);
dbms_output.put_line(I_name);

END;

Ex:
CREATE OR REPLACE PROCEDURE Get_emp_names(Dept_num IN NUMBER) IS
Emp_name VARCHAR(10);
CURSOR c1(Deptno NUMBER) IS
SELECT EnameFROM Emp_tab WHERE deptno= Deptno;

BEGIN
OPEN c1 (Dept_num);
LOOP
        FETCH c1 INTO Emp_name;
        EXIT WHEN C1%NOTFOUND;
        dbms_output.put_line(Emp_name);
END LOOP;
CLOSE c1;
END;
```

Using %TYPE keyword in parameters of stored procedures

```
CREATE OR REPLACE PROCEDURE EMP_NAMES (Dept_Num IN Emp.Deptno%TYPe)
IS
Emp_name VARCHAR2(10);
```

```
CURSOR c1 (Deptno NUMBER) IS SELECT Ename FROM Emp WHERE deptno = Deptno;
BEGIN
OPEN c1 (Dept_num);
LOOP
FETCH c1 INTO Emp_name;
EXIT WHEN C1 %NOTFOUND;
dbms_output.put_line(Emp_name);
END LOOP;
CLOSE c1;
END;
```

Using OUT parameter with % ROWTYPE

```
CREATE OR REPLACE PROCEDURE Get_emp_rec (Emp_number IN Emp.Empno%TYPE,
emp_ret OUT Emp%ROWTYPE) IS
BEGIN
SELECT Empno,Ename, Job, mgr, Hiredate,Sal,comm,Deptno
INTO Emp_rec
FROM Emp
WHERE Empno=Emp_number;
END;
```

You could call this procedure from a PL/SQL block as follows:

```
DECLARE
Emp_ro Emp_tab%ROWTYPE; --declare a record matching a row in the Emp Table

BEGIN
Get_Emp_rec(7499,Emp_row); call for Emp # 7499
dbms_output.put(Emp_row.Ename||" ||Emp_row.Empno);
dbms_output.put(" ||Emp_row.Job||"||Emp_row.Mgr);
dbms_output.put(" ||Emp_row.Comm||"||Emp_row.Deptno);
dbms_output.new_line;
END;
```

Stored functions can also return values that are declared using %ROWTYPE. For Example:

```
FUNCTION Get_emp_rec(Dept_num IN Emp_tab.Deptno%TYPE)
RETURN Emp_tab%ROWTYPE IS...
```

Default Parameter Values

Parameters can take default values. Use the DEFAULT keyword or the assignment operator to give a parameter a default value. For example, the specification for the Get_emp_names procedure could be as following:

```
Procedure get_emp_names (Dept_num IN NUMBER DEFAULT 20) IS...
```

or
Procedure get_emp_names (Dept_num IN NUMBER:=20) IS...
when a parameter takes a default value, it can be omitted from the actual parameter list when you call the procedure. When you do specify the parameter value on te call, it overrides the default value.

Triggers

A trigger is an even within the DBMS that can cause some code to execute automatically.

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER|INSTEAD OF}
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REPLACING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
--SQL Statements
END;
```

For Example:
The price of a product changes constantly. It is important to maintain the history of the prices of the producs.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the product table and product_price_history

```
CREATE TABLe product_price_history
(product_id number(5),
product_name VARCHAR(20),
supplier_name VARCHAR (20),
unit_price number(7,2) );

CREATE TABLE product
(product_id number(5),
product_name VARCHAR(20),
supplier_name VARCHAR (20),
unit_price number(7,2) );
```

2) Create the product_history_trigger and execute it

```
CREATE OR REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
```

```
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES (:old.product_id,
:old.product_name,
:old.supplier_name,
:old.unit_price);
END;
```

3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price=800 WHERE product_id=100
```

Once the above query is executed, the trigger fires and updates the 'product_price_history' table.

4) If you ROLLBACK the transaction before commiting the database, the data inserted to the table is also rolled back.

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.
1) Row Level Trigger- An event is triggered for each row updated, inserted or deleted.
2) Statement Level Trigger- An event is triggered for each sql statement executed.

PL/SQL Triggers Execution Hierarchy

The following hierarchy is followed when a trigger is fired.
1) BEFORE statement trigger fires first.
2) Next BEFORE row level triggers fires, once for each row affected.
3) Then AFTER row level triggervfires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.
4) Finally the AFTER statement level trigger fires.

For Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product_check
(Message VARCHAR(50),
Current_Date NUMBER(32)
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
```

```
BEGIN
INSERT INTO product_check
Values ('Before update, statement level', sysdate);
END;
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is update.

```
CREATE or REPLACE TRIGGER Before_Update_Row_product
BEFORE
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values ('Before update row level', sysdate);
END;
/
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is excuted, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values ('After update, statement level', sysdate);
END;
/
```

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product-check' after each row is updated.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values ('After update, row level', sysdate);
END;
/
```

Now lets execute aupdate statement on table product.

```
UPDATE PRODUCT SET unit_price = 800
WHERE product_id in (100,101);
```

Lets check the data in 'product_check' table to see the order in which the trigger is fired.
SELECT * FROM product_check;

Output:

Message	Current_Date
Before update, statement level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
Before update, Row level	26-Nov-2008
After update, Row level	26-Nov-2008
After update, statement level	26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occurred twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

How to know information about triggers.

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger.

The below statement shows the structure of the view 'USER_TRIGGERS'

DESC USER_TRIGGERS;

NAME	TYPE
TRIGGER_NAME	VARCHAR2(30)
TRIGGER_TYPE	VARCHAR2(16)
TRIGGER_EVENT	VARCHAR2(75)
TRIGGER_OWNER	VARCHAR2(30)
BASE_OBJECT_TYPE	VARCHAR2(16)
TABLE_NAME	VARCHAR2(30)
COLUMN_NAME	VARCHAR2(4000)
REFERENCING_NAMES	VARCHAR2(128)
WHEN_CLAUSE	VARCHAR2(4000)
STATUS	VARCHAR2(8)
DESCRIPTION	VARCHAR2(4000)
ACTION_TYPE	VARCHAR2(11)
TIGGER_BODY	LONG

This view stores information about header and body of the trigger.
SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_Stat_product';

The above sql query provides the header and body of the trigger 'Before_Update_Stat_product'.

You can drop a trigger using the following command.
DROP TRIGGER trigger_name;

CYCLIC CASCADING in a TRIGGER

This is an undesirable situation where more than one trigger enter into an infinite loop. While creating a trigger we should ensure the such a situation does not exist.
The below example shows how trigger's can enter into cyclic cascading.

Let's consider we have two tables 'abc' and 'xyz'. Two triggers are created

1) The INSERT Trigger, triggerA on table 'abc' issues an UPDATE on table 'xyz'.
2) The UPDATE Trigger, triggerB on table 'xyz' issues an INSERT on table 'abc'.

In such a situation, when there is a row inserted in table 'abc', triggerA fires and will update tabele 'xyz'.
When tha table 'xyz' is updated, triggerB fires and will insert in table 'abc'.
This cyclic sitation continues and will enter into a infinite loop, which will crash the database.

Example:

```
CREATE OR REPLACE TRIGGER TRD_EMPLOYEES_SALARY_CHECK
BEFORE UPDAE
ON EMPLOYEEFOR EACH ROW
BEGIN
IF :OLD.SAL > :NEW.SAL THEN
RAISE_APPLICATION_ERROR(-20111,'SORRY! SALARY CAN NOT BE DECREASED!');
END IF;
END;
```