STORED PROCEDURES AND FUNCTIONS

```
CREATE OR REPLACE PROCEDURE sal_raise (ID IN
NUMBER, sal_incr IN NUMBER) AS
BEGIN
        UPDATE emp
        SET sal = sal + sal_incr
        WHERE empno = id;
        IF SQL%NOTFOUND THEN
           raise_application_error (-20002,'Invalid emp');
        END IF;
END sal_raise;
```

When you finish writing the procedure type (‘/’) as the only character on the last line

TO execute the above procedure from SQL*PLUS

SQL> ***execute sal_raise***(7499,300);

To execute the procedure from within a PL/SQL Block
Just write the name of the procedure without the reserved
word EXECUTE

```
DECLARE
        x         number(4);
        y         number(4)
BEGIN
        Select sal,empno into x,y from emp
        where ename='KING';
        IFx< 6000 THEN
                  sal_raise (y,1000);
        END IF
END
```

NOTE : TO CREATE a Procedure you must be granted the **CREATE PROCEDURE** privilege.

TO EXECUTE a Procedure you must be granted the **EXECUTE PROCEDURE** privilege.

When a procedure is created it is stored in the Database
in Both Source code and Object Code

NEXT LET US LOOK AT FUNCTIONS

```
CREATE FUNCTION get_bal(acc_no NUMBER)
          RETURN  NUMBER
IS  (or AS)
          acc_bal  NUMBER;            -- Local Variable
BEGIN
          SELECT balance INTO acc_bal FROM Account
          WHERE acc_id=acc_no;
          RETURN (acc_bal);
END;
```

Valid Statement in Procedures and Functions are the Same as valid statement in PL/SQL programming

•DML (INSERT, UPDATE, DELETE)
•Calls to Stored procedures and Functions
•Calls to Other procedures stored in remote database
•Dynamic SQL (ONLY in Version 7.1 of ORACLE)

When errors are generated during creation (compilation) of procedures, one can see these error by

SQL> ***SHOW ERRORS;*** SQL*PLUS command

Alternatively, by querying the following data dictionary tables

•**USER_ERRORS**
•**ALL_ERRORS**
•**DBA_ERRORS**

Therefore, the errors are stored in the data dictionary as part of the database. These errors are automatically stored in the database and will be deleted when the procedure is dropped.

The original source code can be retrieved from the data dictionary as well. This can be done by querying the following Views

- **USER_SOURCE**
- **ALL_SOURCE**
- **DBA_SOURCE**

The procedure’s source code is removed from the data dictionary when the procedure is dropped by the following command

```
SQL> DROP PROCEDURE sal_raise;
```

A Procedure depends on

- Every database object it refers to in its Body (Tables..).
- The database objects those objects depend on.

When the definition of those object changes, its dependent objects (i.e procedures) are marked for recompilation.

The data dictionary can tell which procedures are Invalid and need recompilations

USER_OBJECTS

Recompilation of dependent objects (procedures) occurs automatically at run time if needed.

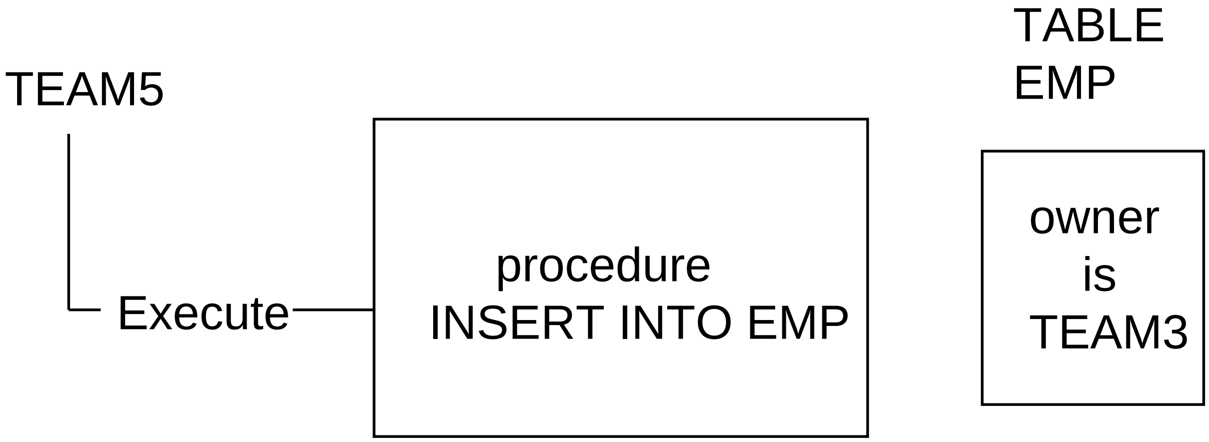
One can recompile manually by

SQL> **ALTER PROCEDURE sal_raise COMPILE**;

To execute a procedure on a remote server user

EXECUTE proc_name@JORDAN(*Parameter list*);

Where JORDAN is the remote server DATE LINK name.
Consider the following Diagram



TEAM3 is Owner of TABLE EMP and the Procedure Proc1.
TEAM5 Has execute permission on Procedure Proc1, but has no INSERT Permission to TABLE TEAM3.EMP.
What will happen at execution Time.

INHERITANCE is The Answer

PACKAGES

Packages are nothing but a collection of procedures and Functions that the programmer chooses to collect together perhaps because these procedures or function are somehow related.

The declaration of a package has two parts:-

•Package Specification
•Package Body

The ***package specification*** contains the declaration of procedures (and functions), variables, constants, and exceptions that are accessible outside the package.

The ***package body*** defines procedures (and functions), and exceptions that are declared in the package specification. The package body can also define procedures, variables, constants, cursors, and exceptions not declared in the package specification; however, these objects are only accessible within the scope of the package.

We have already used a package called DBMS_OUTPUT and executed the procedure PUT_LINE and therefore the syntax DBMS_OUTþPUT.PUT_LINE

EXAMPLE

```
CREATE PACKAGE sud_tel AS
FUNCTION inser_rec (sub_no  NUMBER, tel_no NUMBER)
        RETURN NUMBER;
PROCEDURE  remove_rec (sub_no NUMBER);
PROCEDURE change_rec (sub_no NUMBER, tel_no NUMBER);
END sud_tel;

CREATE PACKAGE BODY SUD_TEL  AS
FUNCTION inser_rec (sub_no NUMBER, tel_no NUMBER)
        RETURN NUMBER
IS
BEGIN
        INSERT INTO sudatel VALUES(sub_no,tel_no);
        RETURN(0);
END inser_rec;
PROCEDURE remove_rec (sub_no NUMBER) IS
BEGIN
        DELETE FROM sudatel where sub_no=sub_no;
        IF SQL%NOTFOUND THEN
          RAISE_APPLICATION_ERROR (-20002,'msg');
        END IF;
END remove_rec;
PROCEDURE change_rec (sub_no NUMBER,tel_no NUMBER) IS
        UPDATE sudatel SET telno=tel_no
        WHERE sub_no = subno;
        IF SQL%NOTFOUND THEN  ...
        END IF;
END change_rec;
END sud_tel;
```

TO change the number of a sub. ***SUD_TEL.change_rec***(1,22);

Any variable declared in the package specification section is global for all procedures and function withing that package

Example

```
create or replace package test_global is
                procedure get_time;
                 procedure  put_time;
                  x         date;
                end test_global;

/

create or replace package body test_global is procedure
get_time is
                begin
                                select sysdate into x from dual;
                end get_time;
                procedure put_time is
                begin
                                dbms_output.put_line(sysdate-x);
                end put_time;
                end test_global;

/
```

```
SQL>Set ServerOuptut On
                SQL> Execute test_global.get_time  -- x is
initialized             SQL> Execute test_global.put_time
-- x is still defined

.001226
```

Which is the time difference between the execution of both statement (0.001226 fraction of a day)

BENEFITS OF PROCEDURES

1. SECURITY

2. INTEGRITY

3. PERFORMANCE
   - Reduce no of calls to Database;
   - Decrease network traffic;
   - Compiled SQL statement or Pre-parsed

4. MEMORY SAVING
   - requires one copy of code only.
   - Takes advantage of ORACLE SHARED SQL

5. PRODUCTIVITY
   - Avoids redundant code for multiple applications.
   - Reduces errors.

6. MAINTAINABILITY
   - Dependency tracking by ORACLE
   - System wide changes

NEXT SECTþIOþN DATABASE TRIGGERS

DATABASE TRIGGERS

Database Triggers are procedures that are stored in the database and implicitly executed (‘fired’) when a table is modified

COMPLETE EXAMPLE

```
CREATE TABLE balance (acc_id NUMBER ,
                                bal NUMBER);
CREATE TABLE daily_trans (acc_id NUMBER
                    amount NUMBER, cr_db NUMBER);
```

Requirements: When a transaction is inserted into table

daily_trans, an update statement should automatically change the balance of that account id in the balance table, when cr_db is positive, it is a credit, when negative it is a debit.

This can be easily accomplished by using a database trigger on table daily_trans. Let see how.

```
CREATE OR REPLACE TRIGGER upd_bal
AFTER INSERT ON daily_trans
FOR EACH ROW
BEGIN
          UPDATE balance  SET bal=nvl(bal,0)+
          :NEW.cr_db*:NEW.amount
          WHERE acc_id =:NEW.acc_id;
          IF SQL%NOTFOUND THEN
             INSERT INTO balance
             VALUES (:NEW.acc_id,:NEW.amount);
          END IF;
END;
```

Table Balance Before Row is inserted in table daily_trans

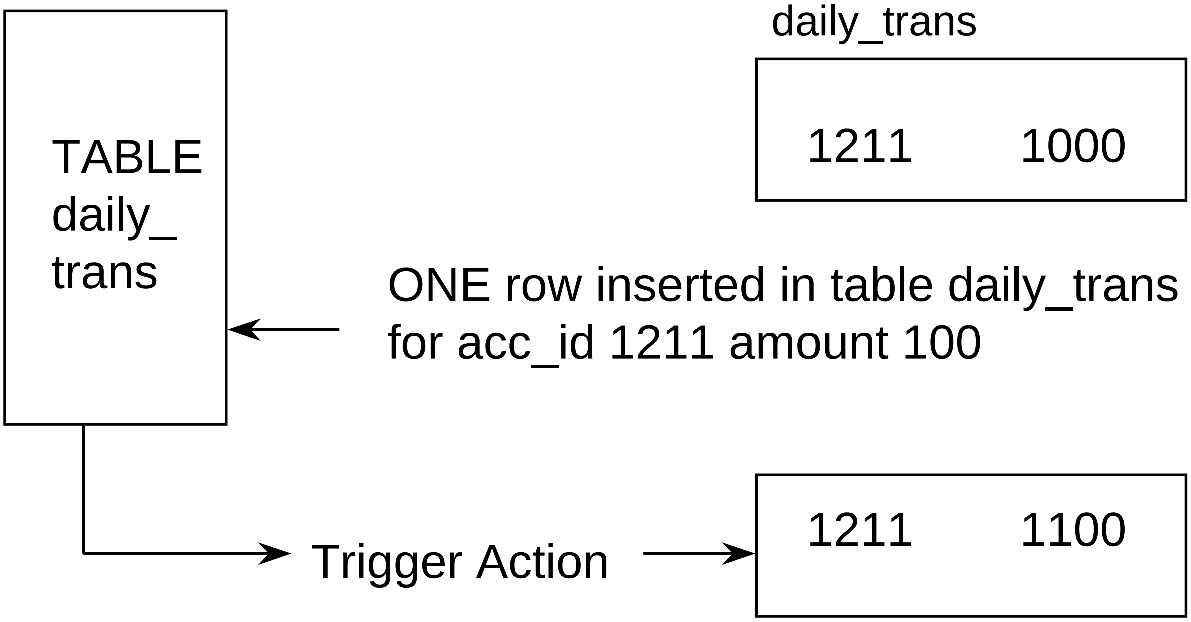


TABLE daily_ trans

1211	1000

ONE row inserted in table daily_trans for acc_id 1211 amount 100

Trigger Action

1211	1100

NOTE that trigger executes (or Fires) automatically after inserting the record on daily_trans

•Use triggers to guarantee that when a specific operation is performed, related actions are performed.

•Do not define triggers to duplicate the functionality already built into ORACLE. For example, do not define triggers to enforce data integrity rules that can be enforced using declarative integrity constraints.

•Be careful not to create recursive triggers. For example creating BEFORE UPDATE statement trigger on DEPT, that itself issues an update statement on DEPT, causes the trigger to fire recursively.

Warning: Because a trigger must be compiled when it is first fired, it is a good idea to limit the size of triggers to roughly 60 lines). Compilation of small triggers have negligible effect on the system. To handle large triggers you can write its code using a procedure and let the trigger call the procedure. Remember that procedures are stored in compiled format. Therefore, one can avoid compilation time of large segments of code.

To create a trigger in your account (schema) you must have a CREATE TRIGGER system privilege.

You can also let the triggering action take place when you DELETE OR INSERT OR UPDATE a table.
For example:

```
CREATE OR REPLACE TRIGGER my_trig
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
DECLARE
            -- variables, cursors, etc ..
BEGIN
            -- PL/SQL BLOCK
END;
```

The above trigger will execute (fire) whenever any record is updated or deleted or inserted in the table emp.

Please note that the trigger will fire if any field of the emp is updated. One can restrict the firing to take place only if a specific field in updated.

EXAMPLE

```
CREATE TRIGGER my_trig
BEFORE DELETE OR UPDATE OF sal ON emp
FOR EACH ROW
    .... etc.
```

The statement FOR EACH ROW is OPTIONAL. If it is included the trigger is called a row trigger, if not then trigger is called a statement trigger. The absence of this FOR EACH ROW option implies that the trigger should only be fired once for the triggering statement. Its presence dictates, however, that the trigger body is fired individually for each row of the table affected by the triggering statement.

Optionally, one can add a Boolean SQL expression using a WHEN clause. If included, the expression in the WHEN clause is evaluated for each row that the trigger affects. If the expression evaluates to TRUE for a row, the trigger body is fired on behalf of that row. If FALSE or NOT TRUE (As in case of NULLS)

```
CREATE TRIGGER DUMMY
AFTER DELETE ON EMP
FOR EACH ROW
WHEN (NEW.JOB != ‘MANAGER’)
DECLARE
            ...
BEGIN
```

The WHEN clause will restrict the action of the trigger to those employees who are not MANAGERS.

:NEW and :OLD

When you update a row in a table you change the current values to the new values. In triggers one can refer to the old values and the new values by using the qualifier **:NEW** and **:OLD**

Please note that depending on the type of the triggering statement, certain referencing to **:NEW** and **:OLD** might not be logical.

A trigger fired by an INSERT statement has only **:NEW** values.

A trigger fired by an ýUPDATE statement has both of these values defined.

A trigger fired by DELETE statement has only **:OLD** values defined.

The undefined values are NULL

IMPORTANT: DO NOT USE COLON : WITH **NEW** AND **OLD** IF USED WITH WHEN CLAUSE

SEE EXAMPLE ON NEXT PAGE

```
CREATE OR REPLACE TRIGGER increase_chk
BEFORE UPDATE OF sal ON EMP
FOR EACH ROW
WHEN (NEW.sal <OLD.sal or NEW.sal >1.1*OLD.sal)
BEGIN
            RAISE_APPLICATION_ERROR(-20022,’msg’);
END;
```

NOTE The following :-

The trigger will fail and return this user defined error code which is equal to -20022 and a message. The firing statement which is the original update of emp will rollback automatically. OF course the application can capture this error code and act accordingly.

Therefore, the conclusion is : IF a pre-defined or user-defined error condition or exception is raised during the execution of a trigger body, all effects of the trigger body, as well as the triggering statement, are rolled back (Unless handled by an exception.

If a trigger can be fired by more than one type of DML operation (INSERT OR DELETE OR UPDATE OF emp), the trigger body can use conditional predicates to execute specific blocks of code, depending on the type of statement that fires the trigger

EXAMPLE

```
CREATE OR REPLACE TRIGGER tot_sal
AFTER DELETE OR INSERT OR UPDATE OF deptno,
sal ON emp
FOR EACH ROW
BEGIN
            IF DELETING OR UPDATING THEN
                        .....
            END IF;
            IF INSERTING THEN
                        ....
            END IF;
END;
```

Triggers can be **DISABLED** and later **ENABLED**.

```
SQL> ALTER TRIGGER my_trig  ENABLE;
SQL> ALTER TRIGGER my_trig DISABLE;
        OR
SQL> ALTER TABLE emp ENABLE ALL TRIGGERS;
SQL>ALTER TABLE emp DISABLE ALL TRIGGERS;
```

Triggers cannot be ALTERED. They should be dropped and recreated. That is why, you are recommended to use CREATE OR REPLACE TRIGGER.

To execute any of the above statements you must have the appropriate privileges

- **ALTER ANY TRIGGER** system privilege.
- **DROP ANY TRIGGER** system privilege.

As with procedures you can refer to the Data Dictionary tables and views to get information about the source code or your triggers

- **USER_TRIGGERS**
- **ALL_TRIGGERS**
- **DBA_TRIGGERS**

TYPE OF TRIGGERS

BEFORE UPDATE row	AFTER UPDATE row
BEFORE DELETE row	AFTER DELETE row
BEFORE INSERT st.	AFTER INSERT st.
BEFORE INSERT row	AFTER INSERT row
BEFORE UPDATE st.	AFTER UPDATE st.
BEFORE DELETE st.	AFTER DELETE st.

COMPARISON BETWEEN TRIGGERS AND PROCEDURES

- Triggers are only associated with tables and are executed implicitly; procedures are invoked explicitly;
- COMMIT, ROLLBACK are not allowed in triggers.
- Triggers are compiled at least the first time they are loaded and are not stored in compiled format unlike procedures.

EXAMPLE

What does the following trigger do?

```
CREATE OR REPLACE TRIGGER TEST
BEFORE INSERT OR DELETE OR UPDATE ON emp;
DECLARE
            x1          INTEGER;
BEGIN
IF TO_CHAR(SYSDATE,’DY’) IN (‘THU’,’FRI’) THEN
            RAISE_APPLICATION_ERROR (-20020,’msg’);
END IF
SELECT COUNT(*) INTO x1 FROM HOLIDAY
WHERE  day = SYSDATE;
IF X1 > 0 THEN
            RAISE_APPLICATION_ERROR(20021,’MSG’);
END IF;
IF TO_CHAR(SYSDATE,’HH24’) NOT BETWEEN
   ‘08’  AND ‘18’ THEN
            RAISE_APPLICATION_ERROR(20022,’MSG’);
END IF;
END;
```

It checks for security authorization.

Another example

```
CREATE TRIGGER reorder
AFTER UPDATE OF part_on_hand ON inventory
FOR EACH ROW
WHEN (new.part_on_hand < new.reorder_point)
DECLARE
          x NUMBER;
BEGIN
          SELECT COUNT(*) INTO x FROM PENDING
          WHERE part_no = :new.part_no;
          IF X = 0 THEN
            INSERT INTO pending VALUES
            (:new.part_no, :new.reorder_qty, sysdate);
          END IF;
END;
```

TRIGGERS APPLICATIONS:

Provide sophisticated auditing.
Prevent invalid transactions.
Enforce complex business rules.
Enforce complex security authorizations.
Provide transparent event logging.
Maintain synchronous table replication.