PROCEDURES

A stored procedure is a group of SQL and PL/SQL commands that execute certain task. In contrast to trigger, which is automatically executes when a trigger event is occurred, the user must call a procedure either from a program or manually. While a trigger is associated to only one table or user, several users or applications can use a procedure.

ADVANTAGES OF USING A PROCEDURE OR FUNCTION:

1. **Security:**

   Stored procedures and functions can help enforce data security. For e.g. by giving permission to a procedure or function that can query a table and granting the procedure or function to users, permissions to manipulate the table itself need not be granted to users.

2. **Performance:**

   It improves database performance in the following way:

   - Amount of information sent over a network is less.
   - No compilation step is required to execute the code.
   - Once the procedure or function is present in the shared pool of the SGA retrieval from disk is not required every time different users call the procedure or function i.e. reduction in disk Input / Output.

3. **Memory Allocation:**

   The amount of memory used reduces as stored procedures or functions have shared memory capabilities. Only one copy of procedure needs to be loaded for execution by multiple users. Once a copy of the procedure or function is opened in the Oracle engine's memory, other users who have permissions may access them when required.

4. **Productivity:**

   By writing procedures and functions redundant coding can be avoided, increasing productivity

5. **Integrity:**

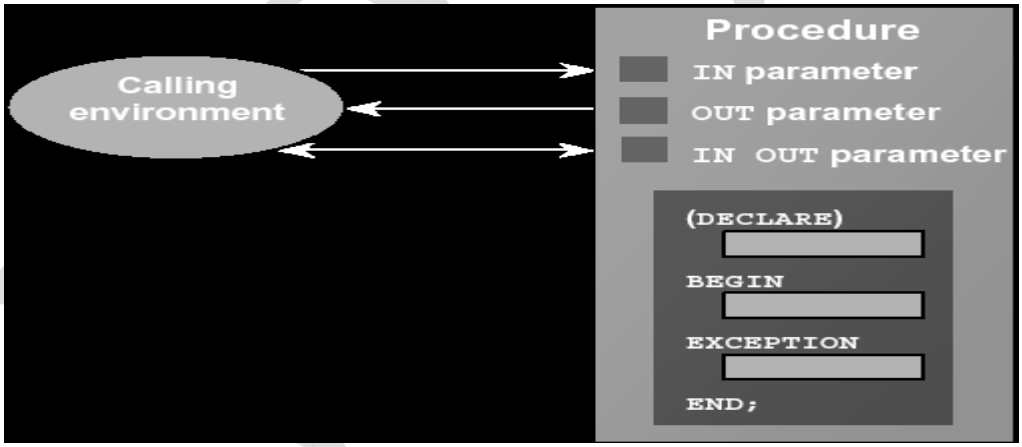
   A procedure or function needs to be tested only once to guarantee that it returns an accurate result. Since procedures and functions are stored in the Oracle engine they

become a part of the engine's resource. Hence the responsibility of maintaining their integrity rests with the Oracle engine. The Oracle engine has high level of in-built security and hence integrity of procedures or functions can be safely left to the Oracle engine.

- **SYNTAX FOR CREATING A PROCEDURE:**

```
CREATE [OR REPLACE] PROCEDURE <PROCEDURE_NAME>
[Parameter1 {IN, OUT, IN OUT}, [Parameter2 {IN, OUT, IN OUT},...]]
{IS/AS}
        [CONSTANT / VARIABLE Declaration;]
BEGIN
        Executable statements;
[EXCEPTION
        Exception handling statements;]
END;
```

- **TYPES OF PARAMETERS:**



- **IN -** This parameter passes a value into the program. This value is read only type of value. It cannot be change.
- **OUT -** The OUT parameter passes a value back from the program. This value is write only value (printable value).
- **IN OUT -** This parameter passes the value into the program and returns the value from the program. This value is read and then it written.

➢ **COMPARISON BETWEEN THE PARAMETERS:**

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

- **HOW TO EXECUTE OR RUN A PROCEDURE:**

SQL > SELECT * FROM <Table-name>;

SQL > @ <Procedure-file name>.SQL;

Procedure is successfully created…

OR

Procedure is created with compilation error…

SQL > SHOW ERRORS;

OR

SQL > SELECT * FROM USER_ERRORS;

SQL > EXECUTE <Procedure-name> (Parameter Value);

SQL > SELECT * FROM <Table-name>;

 NOTE: - If your user doesn't have EXECUTE permission then write the following command.

GRANT EXECUTE ON <PROCEDURE-NAME> TO <USER-NAME>;

- **SYNTAX FOR DELETING A PROCEDURE:**

DROP PROCEDURE <PROCEDURE_NAME>;

- **EXAMPLES OF PROCEDURE:**

1. Write a simple procedure without any parameter that updates the values in the EMP table. (P_RAISE_SAL.SQL)

```
CREATE OR REPLACE PROCEDURE RAISE_SAL
IS
BEGIN
        UPDATE EMP SET SAL =SAL+ SAL*0.10;
END;
/

SQL>@P_RAISE_SAL.SQL;
or
SQL>EXECUTE RAISE_SAL;
```

2. Write a simple procedure that increases the salary of employees for the given department no by percentage inputted by the user using IN parameter. (P_RAISE_SAL2.SQL)

```
CREATE OR REPLACE PROCEDURE RAISE_SAL2 (VDEPT IN EMP.DEPTNO %TYPE, VPER IN
NUMBER)
IS
BEGIN
        UPDATE EMP SET SAL =SAL+ SAL*(VPER/100) WHERE DEPTNO=VDEPT;
END;
/

SQL>@P_RAISE_SAL2.SQL;
SQL>EXECUTE RAISE_SAL2 (10,20);
```

3. Write a procedure that search's whether the given employee number is present or not in the table. (P_SEARCH_EMP.SQL)

```
SET SERVEROUTPUT ON

CREATE OR REPLACE PROCEDURE SEARCH_EMP
(VEMPNO IN EMP.ENO%TYPE, VNAME OUT VARCHAR2)
IS
BEGIN
                SELECT ENAME INTO VNAME FROM EMP WHERE ENO = VEMPNO;
        EXCEPTION
                WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE (TO_CHAR(VEMPNO) || 'NOT FOUND');
END;
/
```

SQL>@P_SEARCH_EMP.SQL;
SQL>VARIABLE EMP_NAME VARCHAR2;
SQL>EXECUTE SEARCH_EMP(101,EMP_NAME);
SQL>PRINT EMP_NAME;

4. Write a PL/SQL block to call the SEARCH_EMP procedure.

```
SET SERVEROUTPUT ON
DECLARE
        M_NAME EMP.ENAME%TYPE;
BEGIN
        SEARCH_EMP (1000, M_NAME);
        DBMS_OUTPUT.PUT_LINE (‘ THE EMPLOYEE NO 1000’ || M_NAME);

EXCEPTION
        WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE (TO_CHAR (VEMPNO) || 'NOT FOUND');
END;
/
```

5. Write a procedure that explains the use of IN OUT parameter by updating the value of salary. (P_UPDADTESAL)

```
CREATE OR REPLACE PROCEDURE UPDATESAL (NO IN NUMBER, INC_SAL OUT NUMBER)
IS

        SAL1 NUMBER;

BEGIN

        SELECT SAL INTO SAL1 FROM EMP WHERE ENO = NO;
        INC_SAL: = INC_SAL + SAL1;
        UPDATE EMP SET SAL = INC_SAL WHERE ENO = NO;

EXCEPTION

        WHEN NO_SALARY_FOUNE THEN
                RAISE_APPLICATION_ERROR (-20250, ‘ NO SALARY FOUND’);
END;
/
```

6. Write a PL/SQL block that calls the UPDATESAL procedure.

```
SET SERVEROUTPUT ON
```

```
DECLARE

        ENO NUMBER: = 100;
        INC_SAL NUMBER: = 500;

BEGIN

        UPDATESAL (ENO, INC_SAL);
        DBMS_OUTPUT.PUT_LINE ('UPDATED SALARY' || INC_SAL);

END;
/
```

FUNCTIONS

Functions are also collection of SQL and PL/SQL code, which can return a value to the caller or to caller PL/SQL block. A function is similar to a stored procedure. The main difference between them is that function returns a value and a stored procedure doesn't return a value.

- **DIFFERENTIATE BETWEEN A FUNCTION AND A PROCEDURE:**

  Unlike a procedure, functions can return a value to the caller whereas the procedures cannot return a value. This value is return by using RETURN keyword within the function. A function can return a single value to the caller.

	A Function	A Procedure
1.	Function must return a value.	Procedure it is optional and it does not return a value.
2.	Functions can have only input parameters for it.	Procedures can have input/output parameters.
3.	Function takes only one input parameter and it is mandatory	Procedure may take o to n input parameters.
4.	Functions can be called from Procedure.	Procedures cannot be called from Function.

- **SYNTAX FOR CREATING A FUNCTION:**

```
CREATE [OR REPLACE] FUNCTION <PROCEDURE_NAME>
        [Parameter1 {IN}, [Parameter2 {IN},...]]
```

```
RETURN DataType
{IS/AS}
        [CONSTANT / VARIABLE Declaration;]
BEGIN
        Executable statements;
[EXCEPTION
        Exception handling statements;]
RETURN Return_value;
END;
```

- **SYNTAX FOR DELETING A FUNCTION:**

```
DROP FUNCTION <FUNCTION_NAME>;
```

- **HOW TO EXECUTE OR RUN A FUNCTION:**

```
SQL >   @ <function-filename>.SQL;
SQL >   VARIABLE <VARIABLE_NAME> datatype;
SQL>    EXECUTE: <VARIABLE_NAME>: =<function-name>(Parameter Value);
SQL >   PRINT <VARIABLE_NAME>;
OR
                Write a PL/SQL block that calls the function.
SQL >   SHOW ERRORS;
OR
SQL >   SELECT * FROM USER_ERRORS;
```

NOTE: - If your user doesn't have EXECUTE permission then write the following command.

```
GRANT EXECUTE ON <FUNCTION-NAME> TO <USER-NAME>;
```

- **EXAMPLES OF FUNCTION:**

7. Write a function that returns the square of the given number. (F_SQUARE.SQL)

```
CREATE OR REPLACE FUNCTION SQUARE (NO IN NUMBER)
RETURN NUMBER
IS

BEGIN
```

```
        RETURN NO*NO;

END;
/
SQL>@F_SQUARE.SQL;
SQL>VARIABLE ANS NUMBER;
SQL>EXECUTE: ANS: = SQUARE (5);
SQL>PRINT ANS;
        OR
SQL>SELECT SQUARE (5) FROM DAUL;
```

8. Write a function display the string in reverse order. (F_REVERSE.SQL)

```
CREATE OR REPLACE FUNCTION STR_REVERSE (V_STR IN VARCHAR2)
RETURN VARHCAR2
IS
R_STR VARCHAR2 (80): = ' ';
BEGIN
        FOR I IN REVERSE 1..LENGTH (V_STR) LOOP
        R_STR: = RSTR || SUBSTR (V_STR, I, 1);
        END LOOP;
        RETURN R_STR;
END;
/
```

9. Write a function that calculates total number of employees that are working in the given department. (F_CONTEMP.SQL)

```
CREATE OR REPLACE FUNCTION COUNTEMP (V_DEPTNO IN EMP.DEPTNO%TYPE)
RETURN NUMBER
IS
TOTAL_EMP NUMBER;

BEGIN
        SELECT COUNT ( * ) INTO TOTAL_EMP FROM EMP WHERE DEPTNO = V_DEPTNO;
        RETURN TOTAL_EMP;
END;
```

10. Write a function that returns balance for given account number. (F_BANK.SQL)

```
CREATE OR REPLACE FUNCTION BANK (M_NO IN NUMBER)
RETURN NUMBER
IS
        M_BALANCE BANK_MASTER.BALANCE%TYPE;
```

```
BEGIN
        SELECT BALANCE INTO M_BALANCE FROM BANK_MASTER WHERE ACC_NO =
M_NO;
        RETURN M_BALANCE;
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                RETURN O;
END;
```

11. Write a PL/SQL block that calls the BANK function. (F_CALL2.SQL)

```
SET SERVEROUTPUT ON
SET FEEDBACK OFF
SET VERIFY OFF
DECLARE
        M_ACCNO BANK_MASTER.ACC_NO%TYPE;
        TOTAL_BALANCE NUMBER (8,2);
BEGIN
        M_ACCNO: = &M_ACCNO;
        TOTAL_BALANCE: = BANK (M_ACCNO);

        IF TOTAL_BALANCE > 0 THEN
                DBMS_OUTPUT.PUT_LINE (‘ BALANCE IS ‘ || TOTAL_BALANCE);
        ELSE
                DBMS_OUTPUT.PUT_LINE (‘ ACCOUNT NUMBER IS NOT FOUND’);
        END IF;
END;
```

TRIGGERS

The oracle engine allows the user to define the procedures that are executed by oracle engine itself, when an insert, update or delete statement is issued against a table. The triggers are standalone procedures that are fired implicitly (internally) by the oracle engine and not explicitly by the user.

USE OF DATABASE TRIGGERS:

1. A trigger can permit the DML statement against a table only if they are issued during the regular business hours or on pre-mentioned weekdays.
2. The triggers can also be used to prevent invalid transactions & to enforce security to the data.

- **DIFFERENTIATE BETWEEN THE TRIGGER & PROCEDURE:**

1. The triggers don't accept the parameters whereas procedure can accept.
2. The oracle engine executes a trigger implicitly whereas to execute a procedure it must explicitly called by the users.

- **HOW TO APPLY THE DATABASE TRIGGERS?**

A trigger has three basic parts. The following are the three basic parts:

1. **Triggering Event or Statement:**

   It is a SQL statement that causes a trigger to be fired. It can be INSERT, UPDATE or DELETE statement for a specific table.

2. **Trigger Restriction:**

   A trigger restriction specifies a Boolean Expression that must be TRUE for the trigger to fire. It is an option available for the triggers that are fired for each row. A trigger restriction is specified using a WHEN clause.

3. **Trigger Action:**

   A trigger action is the PL/SQL code to be executed when triggering statement is encountered and the value of trigger restriction is TRUE. The PL/SQL block contains SQL and PL/SQL statements.

- **TYPES OF TRIGGERS:**

1. **Row Trigger:**

   A Row Trigger is fired each time a row in the table is affected by the triggering statement. For example, if the UPDATE statement updates multiple rows of a table a Row Trigger is fired once for each row affected by the UPDATE statement. If the triggering statement affects no rows, the trigger is not executed at all.

2. **Statement Trigger:**

   A Statement Trigger is fired once on behalf of the triggering statements, independent of the number of rows the triggering statement affects (even if no rows are affected).

- **DIFFERENTIATE BETWEEN THE BEFORE AND AFTER TRIGGERS:**

While defining a trigger it is necessary to specify the trigger timing that we must have to specify when the triggering action is to be executed in relation to the triggering statement. BEFORE and AFTER apply to both row and the statement triggers.

1. The BEFORE triggers execute the trigger action before the triggering statement is executed.
2. The AFTER trigger executes the triggering action after the execution of triggering statement.

Syntax:

CREATE OR REPLACE TRIGGER <TRIGGER_NAME>

{BEFORE/AFTER}

{INSERT/UPDATE/DELETE} [OF COLUMN] ON <TABLE_NAME>

[FOR EACH ROW]

[WHEN CONDITION]

[PL/SQL BLOCK]

- **Syntax to drop a trigger:**

DROP TRIGGER <TRIGGER_NAME>;

- **GUIDELINE FOR CREATING A TRIGGER:**

1. There can be only one trigger of a particular type that is for UPDATE, for INSERT, or for DELETE.
2. Only one table can be specified in the triggering statement.
3. The triggers cannot include COMMIT, ROLLBACK and SAVEPOINT statements.
4. Inside the trigger the correlation name :NEW and :OLD can be made use of to refer to data on the command line and data in the table respectively.

- **EXAMPLES OF TRIGGERS:**

12. Write a trigger to insert the existing values of the EMP table into NEWEMP table when the record is deleted from EMP table.

```
CREATE OR REPLACE TRIGGER TR_EMPDELETE
BEFORE DELETE ON EMP
FOR EACH ROW

DECLARE
```

```
        V_NO EMP.EMPNO%TYPE;
        V_NAME EMP.ENAME%TYPE;
        V_DEPTNO EMP.DEPTNO%TYPE;

BEGIN
        V_NO: =:OLD.EMPNO;
        V_NAME: =:OLD.ENAME;
        V_DEPTNO: =:OLD.DEPTNO;
        INSERT INTO NEWEMP (EMPNO, ENAME, DEPTNO) VALUES
        (V_NO, V_NAME, V_DEPTNO);
END;
/
```

- **HOW TO EXECUTE A TRIGGER:**

```
SQL>   START TR_EMPDELETE; OR  @TR_EMPDELETE;

        Trigger is created…….

SQL>   DELETE FROM EMP WHERE EMPNO = 100;
SQL>   SELECT * FROM NEWEMP;
```

If trigger is created with compilation error then execute the following command:

```
SQL>   SHOW ERROR;
```

To view the list of triggers created by user follow following steps:

```
SQL>   DESCRIBE USER_TRIGGERS;
SQL>   SELECT TRIGGER_NAME, TRIGGER_BODY FROM USER_TRIGGERS WHERE
        TRIGGER_NAME = 'TR_EMPDELETE'.
```

13. Write a trigger to insert the existing values of the EMP table into NEWEMP table when the record is updated in EMP table.

```
CREATE OR REPLACE TRIGGER TR_EMPUPDATE
BEFORE UPDATE ON EMP
FOR EACH ROW

BEGIN
        INSERT INTO NEWEMP (EMPNO, ENAME, DEPTNO) VALUES
        (:OLD.EMPNO, :OLD.ENAME, :OLD.DEPTNO);
END;
/
```

14. Write a trigger to insert the values into the NEWEMP table when the records are inserted into the EMP table.

```
CREATE OR REPLACE TRIGGER TR_EMPINSERT
BEFORE INSERT ON EMP
FOR EACH ROW

BEGIN
        INSERT INTO NEWEMP (EMPNO, ENAME, DEPTNO) VALUES
        (: NEW.EMPNO, : NEW.ENAME, : NEW.DEPTNO);

END;
/
```

15. Write a trigger for INSERT, UPDATE and DELETE operation in one program.

```
CREATE OR REPLACE TRIGGER TR_EMPALL
BEFORE INSERT OR UPDATE OR DELETE ON EMP
FOR EACH ROW

BEGIN

        IF INSERTING THEN
                INSERT INTO EMP_BACKUP (ENO, ENAME, SAL) VALUES
                (: NEW.ENO,  : NEW.ENAME, : NEW.SAL);

        ELSIF UPDATING THEN
                INSERT INTO EMP_BACKUP (ENO, ENAME, SAL) VALUES
                (:OLD.ENO, : OLD.ENAME, : OLD.SAL);

        ELSIF DELETING THEN
                INSERT INTO EMP_BACKUP (ENO, ENAME, SAL) VALUES
                (:OLD.ENO, : OLD.ENAME, : OLD.SAL);
        END IF;

END;
/
```

16. Write a trigger to restrict user form using the table on Sunday.

```
SET SERVEROUTPUT ON
SET FEEDBACK OFF
SET VERIFY OFF


CREATE OR REPLACE TRIGGER TR_HOLIDAY
BEFORE INSERT OR UPDATE OR DELETE ON ITEM
FOR EACH ROW
```

```
BEGIN

        IF TRIM (TO_CHAR (SYSDATE, 'Day')) = 'Sunday' THEN
                RAISE_APPLICATION_ERROR (-20420, ' You cannot modify data on
Sunday');
        END IF;

END;
/
```

17. Write a trigger that identifies the gender of the employee and according to the gender sets MR. in front of MALE employees and MS. in front of FEMALE employee.

```
SET SERVEROUTPUT ON
SET FEEDBACK OFF
SET VERIFY OFF

CREATE OR REPLACE TRIGGER TR_GENDER
BEFORE UPDATE OR INSERT ON EMP
FOR EACH ROW

BEGIN

        IF : NEW.GENDER = 'M' THEN
                : NEW.ENAME: = 'MR.' || : NEW.ENAME;
        ELSE
                : NEW.ENAME: = 'MS.' || : NEW.ENAME;
        END IF;

END;
/
```

18. Write a trigger that restricts the entry of record if salary is greater then 8000 Rs.

```
CREATE OR REPLACE TRIGGER TR_TESTSAL
BEFORE INSERT OR UPDATE OF SAL ON EMP
FOR EACH ROW

BEGIN

        IF : NEW.SAL > 8000 THEN
                RAISE_APPLICATION_ERROR (-20200, 'INCORRECT VALUE');
        END IF;
```

```
END;
/
```

19. Write a trigger that deletes the record from NEWEMP table if the corresponding record is deleted from EMP table.

```
CREATE OR REPLACE TRIGGER TR_DEL
BEFORE DELETE ON EMP
FOR EACH ROW
BEGIN

        DELETE FROM NEWEMP WHERE EMPNO  = :OLD.EMPNO;

END;
/
```

- **GENERATING PRIMARY KEY USING A SEQUENCE AND A TRIGGER:**

20. Write a trigger that generates the primary key using a sequence.

CODE FOR SEQUENCE GENERATION:

```
CREATE SQEQUENCE SEQ_CLIENT
        INCREMENT BY 1
        START WITH 1;
```

CODE FOR PRIMARY KEY GENERATION USING A TRIGGER:

```
SET SERVEROUTPUT ON
SET FEEDBACK OFF
SET VERIFY OFF

CREATE OR REPLACE TRIGGER TR_CLIENT
BEFORE INSERT ON CLIENT_MASTER
FOR EACH ROW

DECLARE
        P_KEY VARCHAR2 (4);

BEGIN

        SELECT LPAD (TO_CHAR (SEQ_CLIENT.NEXTVAL), 4, '0') INTO P_KEY FORM
DUAL;
        : NEW.CLIENT_NO: = 'C' || P_KEY;
```

```
    END;
/
```

NOTE: - Since the CLIENT_NO is generated automatically; the user must insert values in all columns except CLIENT_NO column. See the following insert query.

```
INSERT INTO CLIENT_MASTER (CLIENT_NAME, ADDRESS, CITY)
VALUES ('Amar','100 Station Road', 'Ahmedabad');
```

PACKAGE

1. What is Package? List part of package.

Ans: A package is an Oracle object which holds other objects within it. Objects commonly held with package are Function, Procedures, variables, constants, cursors and exceptions. Packages can contain Pl/sql block of code or a subprogram that requires input from another pl/sql block.

Parts of package:

1) Package Specification : contains name of package, names of the data types of any arguments
2) Package Body : contains definition of public objects

2. Explain advantages of PL/SQL Package.

Ans: Advantages of Package are:

- **Modularity**

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

- **Easier Application Design**

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.

- **Information Hiding**

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

- **Added Functionality**

Packaged public variables and cursors persist for the duration of a session. They can be shared by all subprograms that execute in the environment. They let you maintain data across transactions without storing it in the database.

- **Better Performance**

When you call a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O. Packages stop cascading dependencies and avoid unnecessary recompiling. For example, if you change the body of a packaged function, Oracle does not recompile other subprograms that call the function; these subprograms only depend on the parameters and return value that are declared in the spec, so they are only recompiled if the spec changes.

3. Explain PL/SQL package with syntax and example.

Ans: Creating Package syntax:

Create or replace package <package_name> as
    <declaration of procedure or function>;
    <variable declaration>;
End <package_name>;

Creating package body syntax:

Create or replace package body <package_name> as
    <logic of procedure/function declared in package specification>;
End <package_name>;

Execution (Calling) of package:

SQL> Execute <package_name>.<subprogram_name> or
SQL> Call <package_name>.<subprogram_name>

If Your subprogram written in package return values to user or require input from user then write PL/SQL block to call your package.

SQL> Declare
<variable declaration>;
Begin
<package_name>.<subprogram_name> ;
End;

Example:

Create a package that has a procedure that finds current salary of customer. Take customer id from user.

TableName: Customers (id, salary)

Solution:

-- Package Specification

```
CREATE PACKAGE cust_sal AS
  PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
```

-- Package Body

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
  PROCEDURE find_sal(c_id customers.id%TYPE) IS
```

```
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
```

Execution of package

```
DECLARE
  code customers.id%type := &cc_id;
BEGIN
  cust_sal.find_sal(code);
END;
```