VIPS

योगः कर्मसु कौशलम्

IN PURSUIT OF PERFECTION

Database Management Systems

PL/SQL

What is PL/SQL

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

Basics of PL/SQL

- PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL).
- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- Oracle uses a PL/SQL engine to processes the PL/SQL statements.
- PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers.

Features of PL/SQL:

- PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
- PL/SQL can execute a number of queries in one block using single command.
- One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
- PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
- Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
- PL/SQL Offers extensive error checking.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQLsupports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in adatabase.
- PL/SQL saves time on design and debugging by strong features, such as exception handling encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

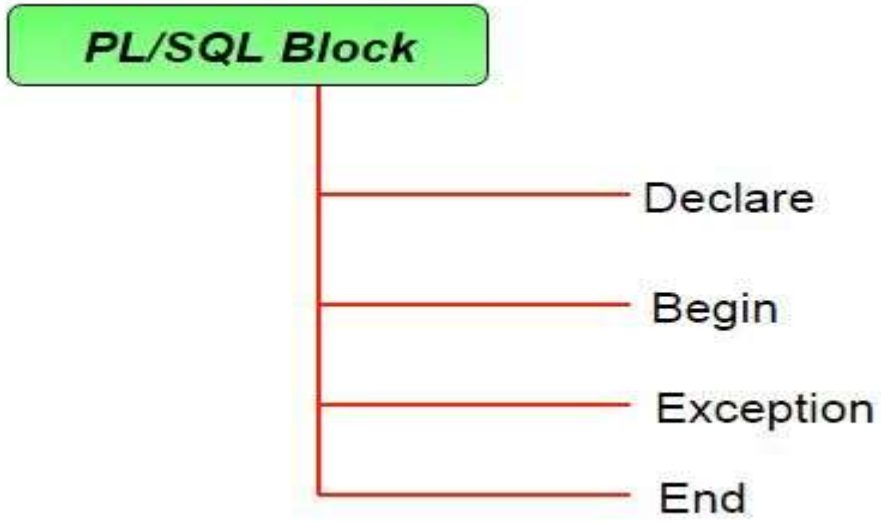
Structure of PL/SQL Block


```
DECLARE
    declaration statements;

BEGIN
    executable statements

EXCEPTIONS
    exception handling statements

END;
```



PL/SQL program units organize the code into blocks. A block without a name is known as an anonymous block. The anonymous block is the simplest unit in PL/SQL. It is called anonymous block because it is not saved in the Oracle database.

Structure of PL/SQL Block


```
DECLARE
    declaration statements;

BEGIN
    executable statements

EXCEPTIONS
    exception handling statements

END;
```

- Declare section starts with DECLARE keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. **This part of the code is optional.**
- Execution section starts with BEGIN and ends with END keyword.**This is a mandatory section** and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.
- Exception section starts with EXCEPTION keyword.**This section is optional** which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

Structure of PL/SQL Block


```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var varchar2(40) := 'Hello PL/SQL' ;

  BEGIN
    dbms_output.put_line(var);

  END;
  /
```

Output:

```
Hello PL/SQL
PL/SQL procedure successfully
completed.
```

Explanation:

SET SERVEROUTPUT ON: It is used to display the buffer used by the dbms_output.

var varchar2 : It is the declaration of variable, named var1 which is of integer type. There are many other data types that can be used like float, int, real, smallint, long etc. It also supports variables used in SQL as well like NUMBER(prec, scale), varchar etc.

PL/SQL procedure successfully completed: It is displayed when the code is compiled and executed successfully.

Slash (/) after END;: The slash (/) tells the SQL*Plus to execute the block.

Assignment operator (:=) : It is used to assign a value to a variable.

Structure of PL/SQL Block


```
SQL> SET SERVEROUTPUT ON;

SQL> DECLARE
    -- taking input for variable a
    a integer := &a ;
    -- taking input for variable b
    b integer := &b ;
    c integer ;
  BEGIN
    c := a + b ;
    dbms_output.put_line('Sum of
'||a||' and '||b||' is = '||c);
  END;
  /
```

Output

Enter value for a: 2

Enter value for b: 3

Sum of 2 and 3 is = 5

PL/SQL procedure successfully completed.

PL/SQL Variables

In PL/SQL, a variable is a meaningful name of a temporary storage location that supports a particular data type in a program.

- **It needs to declare the variable first in the declaration section of a PL/SQL block before using it.**
- **By default, variable names are not case sensitive. A reserved PL/SQL keyword cannot be used as a variable name.**

PL/SQL variables naming rules

- The variable name must be less than 31 characters. Try to make it as meaningful as possible within 31 characters.
- The variable name must begin with an ASCII letter. It can be either lowercase or uppercase.
- Followed by the first character are any number, underscore (_), and dollar sign ($) characters.

Example:

Radius Number := 5;

Date_of_birth date;

PL/SQL Variables

PL/SQL variables naming convention

Prefix	Data Type
v_	VARCHAR2
n_	NUMBER
t_	TABLE
r_	ROW
d_	DATE
b_	BOOLEAN

PL/SQL Variables

Declaration Restrictions:

- Forward references are not allowed i.e. you must declare a constant or variable before referencing it in another statement even if it is a declarative statement.

val number := Total - 200;

Total number := 1000;

The first declaration is illegal because the TOTAL variable must be declared before using it in an assignment expression.

- Variables belonging to the same datatype cannot be declared in the same statement.

N1, N2, N3 Number; **It is an illegal declaration.**

Initializing Variables in PL/SQL

- **The DEFAULT keyword**
- **The assignment operator**

counter binary_integer := 0;

greetings varchar2(20) DEFAULT 'Hello JavaTpoint';

PL/SQL Variables


```
DECLARE
   a integer := 30;
   b integer := 40;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 100.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
```

Output

```
Value of c: 70
Value of f: 33.3333333333333333

PL/SQL procedure successfully completed.
```

PL/SQL Variables

Variable Scope in PL/SQL:

- **Local Variable:**Local variables are the inner block variables which are not accessible to outer blocks.

- **Global Variable:**Global variables are declared in outermost block.

Output

Outer Variable num1: 95

Outer Variable num2: 85

Inner Variable num1: 195

Inner Variable num2: 185

PL/SQL procedure successfully completed.

```
DECLARE
 -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
/
```

PL/SQL Variables anchors

PL/SQL provides you with a very useful feature called variable anchors. It refers to the use of the %TYPE keyword to declare a variable with the data type is associated with a column's data type of a particular column in a table.

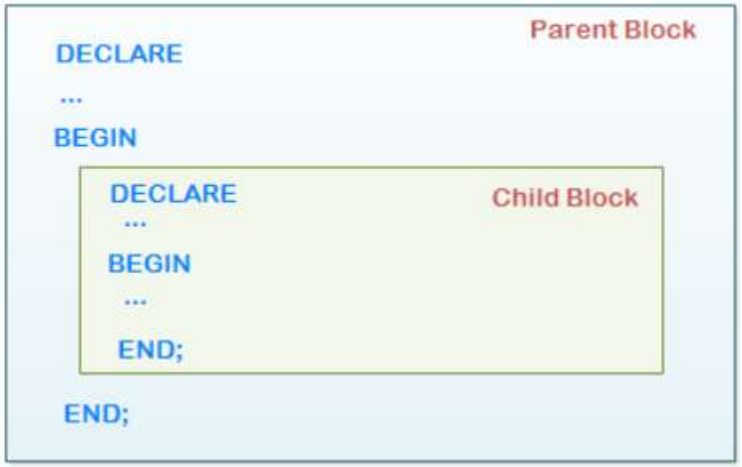
```
DECLARE
  v_first_name  EMPLOYEES.FIRST_NAME%TYPE;
  v_last_name   EMPLOYEES.LAST_NAME%TYPE;
  n_employee_id EMPLOYEES.EMPLOYEE_ID%TYPE;
  d_hire_date   EMPLOYEES.HIRE_DATE%TYPE;
BEGIN
  NULL;
END;
/
```

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key
EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	1
FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	(null)
LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	(null)
EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	(null)
PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	(null)
HIRE_DATE	DATE	No	(null)	6	(null)
JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	(null)
SALARY	NUMBER(8,2)	Yes	(null)	8	(null)
COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	(null)
MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	(null)
DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	(null)

Employees Table

PL/SQL Nested Block


```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
 n_emp_id EMPLOYEES.EMPLOYEE_ID%TYPE := &emp_id1;
BEGIN
  DECLARE
    n_emp_id employees.employee_id%TYPE := &emp_id2;
    v_name   employees.first_name%TYPE;
  BEGIN
    SELECT first_name
    INTO v_name
    FROM employees
    WHERE employee_id = n_emp_id;
    DBMS_OUTPUT.PUT_LINE('First name of employee ' ||
n_emp_id ||  ' is ' || v_name);
    EXCEPTION
      WHEN no_data_found THEN
       DBMS_OUTPUT.PUT_LINE('Employee ' || n_emp_id || '
not found');
  END;
END;
/
```



PL/SQL Nested Block


```
SET SERVEROUTPUT ON SIZE 1000000;
<<parent>>
DECLARE
  n_emp_id EMPLOYEES.EMPLOYEE_ID%TYPE := &emp_id1;
BEGIN
  <<child>>
  DECLARE
    n_emp_id employees.employee_id%TYPE := &emp_id2;
    v_name   employees.first_name%TYPE;
  BEGIN
    SELECT first_name
    INTO v_name
    FROM employees
    WHERE employee_id = parent.n_emp_id;
    DBMS_OUTPUT.PUT_LINE('First name of employee
' || parent.n_emp_id ||   ' is ' || child.v_name);

    EXCEPTION
      WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' ||
parent.n_emp_id || ' not found');
  END;
END;
/
```

PL/SQL If

he PL/SQL IF statement has three forms: **IF-THEN, IF-THEN-ELSE and IF-THEN-ELSIF**.

```
DECLARE
   a number(3) := 500;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
    END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

PL/SQL IF-THEN-ELSE

he PL/SQL IF statement has three forms: **IF-THEN, IF-THEN-ELSE and IF-THEN-ELSIF**.

```
DECLARE
   a number(3) := 500;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   ELSE
      dbms_output.put_line('a is not less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

PL/SQL IF-THEN-ELSE


```
DECLARE
   a number(3) := 500;
BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   ELSE
      dbms_output.put_line('a is not less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

Output

a is not less than 20

value of a is : 500

PL/SQL procedure successfully completed.

PL/SQL IF-THEN-ELSIF


```
DECLARE
  bonus    NUMBER(6,2);
  empid    NUMBER(6) := 120;
  hiredate DATE;
BEGIN
-- retrieve the date that employee was hired, the date is checked
-- to determine the amount of the bonus for the employee
  SELECT hire_date INTO hiredate FROM employees
WHERE employee_id = empid;
  IF hiredate > TO_DATE('01-JAN-98') THEN
      bonus := 500;
 ELSIF hiredate > TO_DATE('01-JAN-96') THEN
      bonus := 1000;
   ELSE
      bonus := 1500;
   END IF;

DBMS_OUTPUT.PUT_LINE('Bonus for employee: ' || empid || ' is: ' || bonus );
END;
/
```

PL/SQL case statement


```
DECLARE
   grade char(1) := 'A';
BEGIN
   CASE grade
      when 'A' then dbms_output.put_line('Excellent');
      when 'B' then dbms_output.put_line('Very good');
      when 'C' then dbms_output.put_line('Good');
      when 'D' then dbms_output.put_line('Average');
      when 'F' then dbms_output.put_line('Passed with Grace');
      else dbms_output.put_line('Failed');
   END CASE;
END;
/
```

PL/SQL case statement


```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  n_pct    employees.commission_pct%TYPE;
  v_eval   varchar2(10);
  n_emp_id employees.employee_id%TYPE := 145;
BEGIN
  -- get commission percentage
  SELECT commission_pct
  INTO n_pct
  FROM employees
  WHERE employee_id = n_emp_id;
```

```
CASE n_pct
   WHEN 0 THEN
     v_eval := 'N/A';
   WHEN 0.1 THEN
     v_eval := 'Low';
   WHEN 0.4 THEN
     v_eval := 'High';
   ELSE
     v_eval := 'Fair';
 END CASE;
 -- print commission evaluation
 DBMS_OUTPUT.PUT_LINE('Employee ' || n_emp_id ||
                      ' commission ' || TO_CHAR(n_pct) ||
                      ' which is '   || v_eval);
END;
/
```

PL/SQL LOOP Statement

PL/SQL LOOP statement with EXIT and EXIT-WHEN statements:

```
LOOP
    ...
    EXIT;
END LOOP;
```

```
LOOP
    ...
    EXIT WHEN condition;
END LOOP;
```

Using Exit Statement

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE n_counter NUMBER := 0;
BEGIN
 LOOP
   n_counter := n_counter + 1;
   DBMS_OUTPUT.PUT_LINE(n_counter);
   IF n_counter = 5 THEN
     EXIT;
   END IF;
 END LOOP;
END;
/
```

PL/SQL LOOP Statement

PL/SQL LOOP statement with EXIT and EXIT-WHEN statements:

```
LOOP
    ...
    EXIT;
END LOOP;
```

```
LOOP
    ...
    EXIT WHEN condition;
END LOOP;
```

Using Exit When Statement

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE n_counter NUMBER := 0;
BEGIN
 LOOP
   n_counter := n_counter + 1;
   DBMS_OUTPUT.PUT_LINE(n_counter);
   EXIT WHEN n_counter = 5;
 END LOOP;
END;
/
```

PL/SQL WHILE Loop

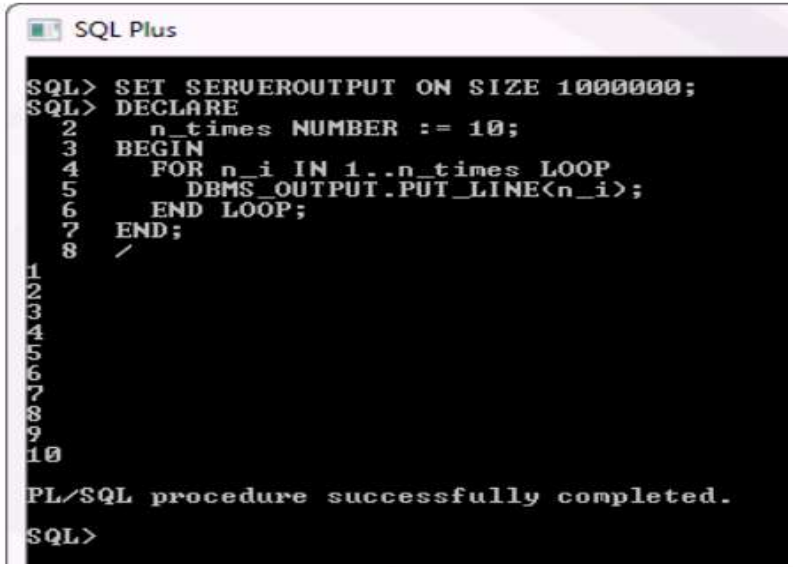

```
WHILE condition
LOOP
   sequence_of_statements;
END LOOP;
```

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  n_counter   NUMBER := 10;
  n_factorial NUMBER := 1;
  n_temp      NUMBER;
BEGIN
  n_temp := n_counter;
  WHILE n_counter > 0
  LOOP
    n_factorial := n_factorial * n_counter;
    n_counter   := n_counter - 1;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('factorial of ' || n_temp ||
                       ' is ' || n_factorial);
END;
/
```

PL/SQL FOR Loop


```
FOR loop_counter IN [REVERSE] lower_bound .. higher_bound
LOOP
   sequence_of_statements;
END LOOP;
```

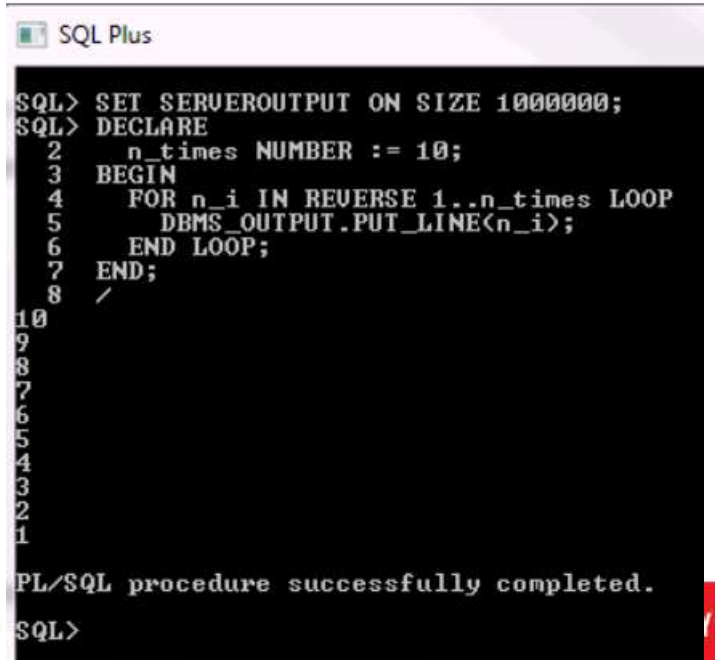
```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  n_times NUMBER := 10;
BEGIN
  FOR n_i IN 1..n_times LOOP
    DBMS_OUTPUT.PUT_LINE(n_i);
  END LOOP;
END;
/
```



PL/SQL FOR Loop

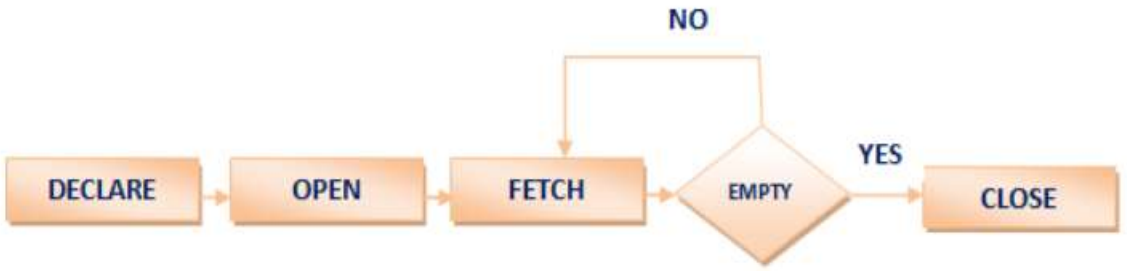

```
FOR loop_counter IN [REVERSE] lower_bound .. higher_bound
LOOP
   sequence_of_statements;
END LOOP;
```

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  n_times NUMBER := 10;
BEGIN
  FOR n_i IN REVERSE 1..n_times LOOP
    DBMS_OUTPUT.PUT_LINE(n_i);
  END LOOP;
END;
/
```



PL/SQL Cursor

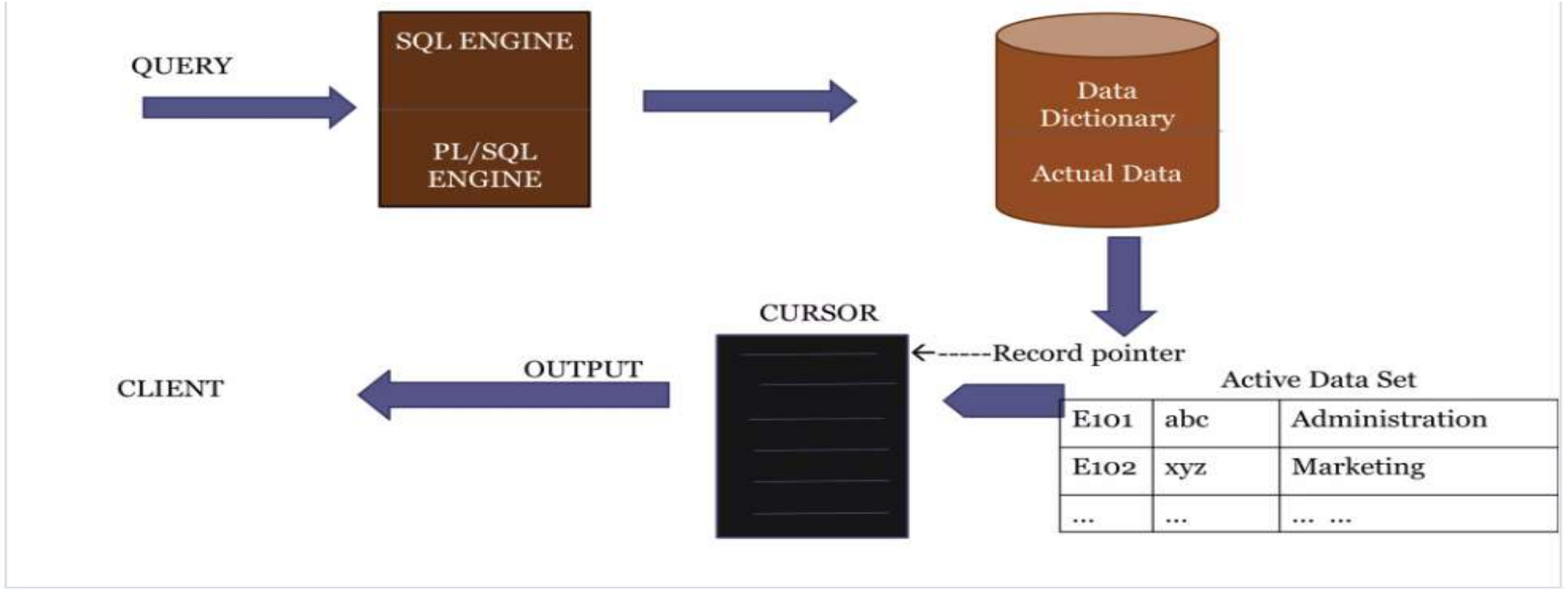
When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor.



A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

1. Implicit Cursors
2. Explicit Cursors

PL/SQL Cursor



PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

PL/SQL Cursors Attributes

Attribute	Description
cursor_name%FOUND	returns `TRUE` if record was fetched successfully by cursor `cursor_name`
cursor_name%NOTFOUND	returns `TRUE` if record was not fetched successfully by cursor `cursor_name`
cursor_name%ROWCOUNT	returns the number of records fetched from the cursor `cursor_name` at the time we test `%ROWCOUNT` attribute
cursor_name%ISOPEN	returns `TRUE` if the cursor `cursor_name` is open

PL/SQL Implicit Cursors


```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Output:

```
6 customers updated
PL/SQL procedure successfully completed.
```

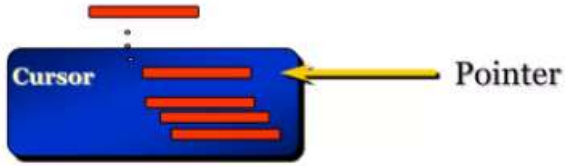
PL/SQL Explict Cursors

Open the cursor

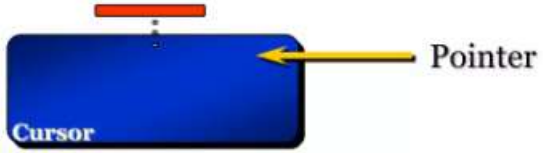
Fetch

Open the cursor.

Fetch a row from the cursor.



Continue until empty.



Close

Close the cursor.

PL/SQL Explict Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors should be defined in the **declaration section** of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Steps:

- Declare the cursor to initialize in the memory.
- Open the cursor to allocate memory.
- Fetch the cursor to retrieve data.
- Close the cursor to release allocated memory.

1) Declare the cursor:

CURSOR name IS
SELECT statement;

2) Open the cursor:

OPEN cursor_name;

3) Fetch the cursor:

FETCH cursor_name INTO variable_list;

4) Close the cursor:

Close cursor_name;

PL/SQL Explict Cursors


```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

PL/SQL Cursors PL/SQL Cursors with Parameters


```
SET SERVEROUTPUT ON;

DECLARE

CURSOR GFG (Min_rank NUMBER) IS

SELECT Id, name, rank

FROM Geeks

WHERE rank > Min_rank;

— Declare variables

cur_id Geeks.Id%TYPE;

cur_name Geeks.name%TYPE;

cur_rank Geeks.rank%TYPE;

BEGIN

— Open and fetch data using the cursor

OPEN GFG(951);

LOOP

FETCH GFG INTO cur_id, cur_name, cur_rank;

EXIT WHEN GFG%NOTFOUND;

— Process fetched data

DBMS_OUTPUT.PUT_LINE('ID: ' || cur_id || ', Name: ' ||
cur_name || ', Rank: ' || cur_rank);

— Close the loop

END LOOP;

— Close the cursor

CLOSE GFG;

END;
```

PL/SQL - Triggers

Triggers in PL/SQL. Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)

A database definition (DDL) statement (CREATE, ALTER, or DROP).

A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Creating Triggers in PL SQL


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

```
DECLARE
Declaration-statements
BEGIN
Executable-statements
EXCEPTION
Exception-handling-statements
END;
```

Creating Triggers in PL SQL


```
1  CREATE TABLE superheroes (
2    sh_name VARCHAR2(20)
3  );
4  --Example 1
5  SET SERVEROUTPUT ON;
6  CREATE OR REPLACE TRIGGER bi_superheroes
7  BEFORE INSERT ON superheroes
8  FOR EACH ROW
9  ENABLE
10 DECLARE
11   v_user  VARCHAR2 (20);
12 BEGIN
13   SELECT user INTO v_user FROM dual;
14   DBMS_OUTPUT.PUT_LINE ('You Just Inserted A Line Mr. '||v_user);
15 END;
16 /
```

```
INSERT INTO superheroes VALUES ('Ironman');
```

Output

```
1 row inserted.

You Just Inserted A Line Mr. HR
```

Types of Triggers in PL/SQL

1. Row-Level Triggers

A row-level trigger occurs once for each row that a triggering event affects.

A. Before Row Triggers

This trigger occurs before the insertion, update, or deletion of a row. It may be used to change the values of the currently processed row.

B. After-Row Triggers

Following an INSERT, UPDATE, or DELETE operation on a row, this type of trigger occurs. It may be used to conduct actions based on the row's modifications.

C. Instead of Row Triggers

This trigger is used with views and fires instead of the view's default DML actions. It enables you to create custom actions for DML operations.

Types of Triggers in PL/SQL

2. Statement-Level Triggers

No matter how many rows are impacted, a trigger event on a table always fires a statement-level trigger.

A. Before Statement Triggers

This trigger occurs before the execution of a SQL query. It can be used to take actions or validations before processing the statement.

B. After Statement Triggers

Upon execution of a SQL statement, this trigger occurs. It can be used to conduct actions based on the statement's overall outcome.

Types of Triggers in PL/SQL

3. Database-Level Triggers

No matter which user or application provides the statement, database triggers in PL SQL are specified on a table, saved in the corresponding database, and performed as a result of an INSERT, UPDATE, or DELETE statement being made against a table.

A. Startup Triggers

This trigger activates after the initialization of the database. It can be used to undertake setup activities or to carry out particular operations during startup.

B. Shutdown Triggers

This trigger starts when the database is shutting down. It can be used to undertake cleaning tasks or to carry out particular operations upon shutdown.

Types of Triggers in PL/SQL

4. DDL Triggers

DDL (Data Definition Language) triggers are actions that occur in reaction to DDL statements, such as CREATE, ALTER, or DROP. It enables you to capture and control DDL activities in the database.

5. Instead of Triggers

This trigger is used with views and fires instead of the view's usual DML actions. It enables you to create custom actions for DML activities.

6. Compound Triggers

To increase flexibility and effectiveness, this trigger combines row-level and statement-level triggers. It enables you to specify actions at various levels and phases of the triggering event.

Types of Triggers in PL/SQL

7. System Triggers

The Oracle database defines and invokes these triggers in response to specified system events.Server problems, log-in or log-off events, and particular user activities are examples of these events. The behavior and examples of system triggers are dependent on the precise event to which they are related.

Types of Triggers in PL/SQL

Row-Level Triggers

```
1  CREATE OR REPLACE TRIGGER tr_superheroes
2  BEFORE INSERT OR DELETE OR UPDATE ON superheroes
3  FOR EACH ROW
4  ENABLE
5  DECLARE
6    v_user  VARCHAR2(20);
7  BEGIN
8    SELECT user INTO v_user FROM dual;
9    IF INSERTING THEN
10     DBMS_OUTPUT.PUT_LINE ('One Row Inserted By ' ||v_user);
11   ELSIF DELETING THEN
12     DBMS_OUTPUT.PUT_LINE ('One Row Deleted By '|| v_user);
13   ELSIF UPDATING THEN
14     DBMS_OUTPUT.PUT_LINE ('One Row Updated by '|| v_user);
15   END IF;
16 END;
```

```
CREATE TABLE superheroes (
  sh_name VARCHAR2(20)
);
```

```
INSERT INTO superheroes VALUES ('Batman');
UPDATE superheroes SET sh_name = 'Superman' WHERE sh_name = 'Batman';
```

Output

```
One Row Inserted By HR

1 row updated.

One Row Updated by HR
```

Types of Triggers in PL/SQL

Row-Level Triggers

```
2  CREATE TABLE sh_audit(
3     new_name   VARCHAR2(30),
4     old_name   VARCHAR2(30),
5     user_name  VARCHAR2(30),
6     entry_date   VARCHAR2(30),
7     operation VARCHAR2(30)
8  );
```

```
ript Output
Task completed in 0.086 seconds
ble SH_AUDIT created.
```

```
CREATE TABLE superheroes (
  sh_name VARCHAR2(20)
);
```

Types of Triggers in PL/SQL

Row-Level Triggers

```
CREATE OR REPLACE trigger superheroes_audit
BEFORE INSERT OR DELETE OR UPDATE ON superheroes
FOR EACH ROW
ENABLE
DECLARE
  v_user  VARCHAR2(30);
  v_date  VARCHAR2(30);
BEGIN
SELECT user,TO_CHAR(sysdate,'DD/MON/YYYY HH24:MI:SS') INTO v_user, v_date FROM dual;
IF INSERTING THEN
INSERT INTO sh_audit(new_name, old_name, user_name, entry_date, operation)
VALUES  (:NEW.sh_name, NULL, v_user, v_date, 'Insert');
ELSIF DELETING THEN
 INSERT INTO sh_audit (new_name, old_name, user_name, entry_date, operation)
ELSIF UPDATING THEN
  INSERT INTO sh_audit (new_name, old_name, user_name, entry_date, operation)
  VALUES( :NEW.sh_name, :OLD.sh_name, v_user, v_date, 'Update');
END IF;
END;
```

Types of Triggers in PL/SQL

Row-Level Triggers

Output

```
1 SELECT * FROM sh_audit;
2 INSERT INTO superheroes VALUES ('Superman');
3 UPDATE superheroes SET sh_name = 'Ironman' WHERE sh_name = 'Superman';
4 DELETE FROM superheroes WHERE sh_name = 'Ironman';
5
6
```

Script Output × Query Result ×

SQL | All Rows Fetched: 3 in 0.002 seconds

	NEW_NAME	OLD_NAME	USER_NAME	ENTRY_DATE	OPERATION
1	Superman	(null)	HR	20/NOV/2015 15:51:12	Insert
2	Ironman	Superman	HR	20/NOV/2015 15:54:19	Update
3	(null)	Ironman	HR	20/NOV/2015 15:58:23	Delete

Types of Triggers in PL/SQL

Instead of Triggers

Using Instead-of trigger you can control the default behavior of Insert, Update, Delete and Merge operations on *Views* but not on tables.

```
--TABLE 1
CREATE TABLE trainer
(
  full_name VARCHAR2(20)
);
--Table 2
CREATE TABLE subject
(
  subject_name  VARCHAR2(20)
);
```

```
1 CREATE VIEW vw_RebellionRider AS
2 SELECT full_name, subject_name FROM trainer, subject;
3 INSERT INTO vw_RebellionRider VALUES ('Manish', 'Java');
```

Output

Error

Types of Triggers in PL/SQL

Instead of Triggers

```
CREATE OR REPLACE TRIGGER tr_io_insert
INSTEAD OF INSERT ON vw_RebellionRider
FOR EACH ROW
BEGIN
  INSERT INTO trainer (full_name) VALUES (:new.full_name);
  INSERT INTO subject (subject_name) VALUES (:new.subject_name);
END;
/
```

Worksheet | Query Builder

```
1  DESC vw_RebellionRider
2
3  Name           Null Type
4  -------------- ---- -------------
5  FULL_NAME           VARCHAR2(20)
6  SUBJECT_NAME        VARCHAR2(15)
7
```

```
INSERT INTO vw_RebellionRider VALUES ('Manish', 'Java');
```

Output

1 row inserted

Types of Triggers in PL/SQL

Instead of Triggers(Update)

```
UPDATE vw_RebellionRider SET full_name = 'Tony Stark' WHERE subject_name = 'Java';
```

Output

Error

```
CREATE OR REPLACE TRIGGER io_update
INSTEAD OF UPDATE ON vw_RebellionRider
FOR EACH ROW
BEGIN
  UPDATE trainer SET full_name = :new.full_name
  WHERE full_name = :old.full_name;

  UPDATE subject SET subject_name = :new.subject_name
  WHERE subject_name = :old.subject_name;

END;
/
UPDATE vw_rebellionrider SET full_name = 'Tony Stark' WHERE SUBJECT_NAME = 'Java';
```

Types of Triggers in PL/SQL

Statement Level Trigger


```
CREATE OR REPLACE TRIGGER customers_credit_trg
    BEFORE UPDATE OF credit_limit
    ON customers
DECLARE
    l_day_of_month NUMBER;
BEGIN
    -- determine the transaction type
    l_day_of_month := EXTRACT(DAY FROM sysdate);

    IF l_day_of_month BETWEEN 28 AND 31 THEN
        raise_application_error(-20100,'Cannot update customer credit from 28th to 31st');
    END IF;
END;
```

```
UPDATE
    customers
SET
    credit_limit = credit_limit * 110;
```

Types of Triggers in PL/SQL

DDL Trigger

```
Worksheet  Query Builder
1  CREATE TABLE schema_audit(
2     ddl_date  DATE,
3     ddl_user  VARCHAR2(15),
4     object_created  VARCHAR2(15),
5     object_name VARCHAR2(15),
6     ddl_operation VARCHAR2(15)
7  );
```

```
1  CREATE OR REPLACE TRIGGER hr_audit_tr
2  AFTER DDL ON SCHEMA
3  BEGIN
4    INSERT INTO schema_audit VALUES(
5      sysdate,
6      sys_context('USERENV','CURRENT_USER'),
7      ora_dict_obj_type,
8      ora_dict_obj_name,
9      ora_sysevent
10   );
11 END;
12 /
```

```
CREATE TABLE rebellionRider(r_num NUMBER);
```

```
SELECT * FROM schema_audit;
```

	DDL_DATE	DDL_USER	OBJECT_CREATED	OBJECT_NAME	DDL_OPERATION
1	07-DEC-15	HR	TABLE	REBELLIONRIDER	CREATE

```
TRUNCATE TABLE RebellionRider;
```

```
SELECT * FROM schema_audit;
```

2	07-DEC-15	HR	TABLE	REBELLIONRIDER	TRUNCATE

Types of Triggers in PL/SQL

Database Trigger

Database event triggers come into action when some system event occurs such as

- database log on
- log off
- start up or
- shut down

Types of Triggers in PL/SQL

Database Trigger Logon

```
CREATE TABLE hr_evnt_audit
  (
    event_type VARCHAR2(20),
    logon_date DATE,
    logon_time VARCHAR2(15),
    logof_date DATE,
    logof_time VARCHAR2(15)
  );
```

```
CREATE OR REPLACE TRIGGER hr_lgon_audit
AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO hr_evnt_audit VALUES(
    ora_sysevent,
    sysdate,
    TO_CHAR(sysdate, 'hh24:mi:ss'),
    NULL,
    NULL
  );
  COMMIT;
END;
/
```

```
Trigger HR_LGON_AUDIT compiled
```

Types of Triggers in PL/SQL

Database Trigger Logon

```
CREATE OR REPLACE TRIGGER hr_lgon_audit
AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO hr_evnt_audit VALUES(
    ora_sysevent,
    sysdate,
    TO_CHAR(sysdate, 'hh24:mi:ss'),
    NULL,
    NULL
  );
  COMMIT;
END;
/
```

```
Trigger HR_LGON_AUDIT compiled
```

```
SELECT * FROM hr_evnt_audit;
DISC;
conn hr/hr;
```

```
SELECT * FROM hr_evnt_audit;
```

```
LOGON 08-JAN-16 16:45:58 (null) (null)
```

Types of Triggers in PL/SQL

Database Trigger Logoff

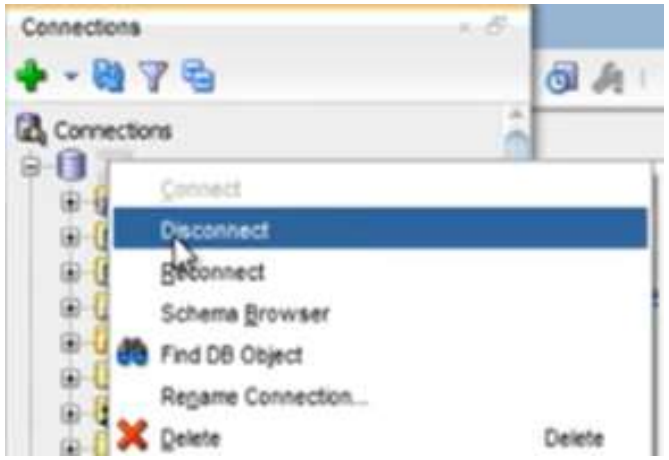
```
CREATE TABLE hr_evnt_audit
  (
    event_type VARCHAR2(20),
    logon_date DATE,
    logon_time VARCHAR2(15),
    logof_date DATE,
    logof_time VARCHAR2(15)
  );
```

```
CREATE OR REPLACE TRIGGER log_off_audit
BEFORE LOGOFF ON SCHEMA
BEGIN
  INSERT INTO hr_evnt_audit VALUES(
    ora_sysevent,
    NULL,
    NULL,
    SYSDATE,
    TO_CHAR(sysdate, 'hh24:mi:ss')
    );
  COMMIT;
END;
/
```

```
Trigger LOG_OFF_AUDIT compiled
```

Types of Triggers in PL/SQL

Database Trigger Logoff




```
SELECT * FROM hr_evnt_audit;
```

Query Result

All Rows Fetched: 1 in 0.003 seconds

EVENT_TYPE	LOGON_DATE	LOGON_TME	LOGOF_DATE	LOGOF_TME
LOGOFF	(null)	(null)	26-JAN-16	16:01:41

Types of Triggers in PL/SQL

Startup Trigger

Startup triggers execute during the startup process of the database. In order to create a database event trigger for shutdown and startup events we either need to logon to the database as a user with DBA privileges such as sys or we must possess the ADMINISTER DATABASE TRIGGER system privilege.

Step1: Logon to the database

Step 2: Create a Table

Step 3: Create the database Event Startup Trigger

Types of Triggers in PL/SQL

Startup Trigger

```
CREATE TABLE startup_audit
(
  Event_type  VARCHAR2(15),
  event_date  DATE,
  event_time  VARCHAR2(15)
);
```

```
CREATE OR REPLACE TRIGGER startup_audit
AFTER STARTUP ON DATABASE
BEGIN
  INSERT INTO startup_audit VALUES
(
    ora_sysevent,
    SYSDATE,
    TO_CHAR(sysdate, 'hh24:mm:ss')
  );
END;
/
```

Types of Triggers in PL/SQL

Startup Trigger

```
CREATE TABLE startup_audit
(
  Event_type  VARCHAR2(15),
  event_date  DATE,
  event_time  VARCHAR2(15)
);
```

```
CREATE OR REPLACE TRIGGER tr_shutdown_audit
BEFORE SHUTDOWN ON DATABASE
BEGIN
  INSERT INTO startup_audit VALUES(
    ora_sysevent,
    SYSDATE,
    TO_CHAR(sysdate, 'hh24:mm:ss')
  );
END;
/
```

PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

Header: The header contains the name of the procedure and the parameters or variables passed to the procedure.

Body: The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

Pass parameters in procedure

IN parameters: The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.

OUT parameters: The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.

INOUT parameters: The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function

PL/SQL Create Procedure

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (parameter [,parameter]) ]
IS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```

PL/SQL Create Procedure

Table creation:

```
create table user
(id number(10) primary key,
name varchar2(100));
```

Procedure Code:

```
create or replace procedure
INSERTUSER
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

PL/SQL program to call procedure

```
BEGIN
   insertuser(101,'Rahul');

dbms_output.put_line('record inserted successfully');
END;
/
```

Now, see the "USER" table, you will see one record is inserted.

ID	Name
101	Rahul

IN & OUT Mode


```
DECLARE
   a number;
   b number;
   c number;
PROCEDURE findMin(x IN number, y IN
number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
```

```
 BEGIN
  a:= 23;
  b:= 45;
  findMin(a, b, c);
  dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

IN & OUT Mode


```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT
number) IS
BEGIN
  x := x * x;
END;
```

```
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways –

I Using the EXECUTE keyword

II Calling the name of the procedure from a PL/SQL block

I Using the EXECUTE keyword

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

```
Procedure created.
```

```
EXECUTE greetings;
```

The above call will display –

```
Hello World
```

Executing a Standalone Procedure

II Calling the name of the procedure from a PL/SQL block

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

The procedure can also be called from another PL/SQL block –

```
BEGIN
   greetings;
END;
/
```

The above call will display –

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

Syntax

DROP PROCEDURE procedure-name;

DROP PROCEDURE greetings;

PL/SQL blocks

1. Anonymous blocks: In PL/SQL, That's blocks which is not have header are known as anonymous blocks. These blocks do not form the body of a function or triggers or procedure. Example: Here a code example of find greatest number with Anonymous blocks.

Example

```
DECLARE
   -- declare variable a, b and c
   -- and these three variables datatype are integer
  a number;
  b number;
  c number;
BEGIN
  a:= 10;
  b:= 100;
   --find largest number
   --take it in c variable
  IF a > b THEN
    c:= a;
  ELSE
    c:= b;
  END IF;
  dbms_output.put_line(' Maximum number in 10 and 100: ' || c);
END;
/
```

Output:

```
Maximum number in 10 and 100: 100
```

PL/SQL blocks

2. Named blocks: That's PL/SQL blocks which having header or labels are known as Named blocks. These blocks can either be subprograms like functions, procedures, packages or Triggers. Example: Here a code example of find greatest number with Named blocks means using function.

Example

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
S
    z number;
BEGIN
   IF x > y THEN
    z:= x;
   ELSE
  a:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 10;
   b:= 100;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum number in 10 and 100 is: ' || c);
END;
/
```

Output:

```
Maximum number in 10 and 100: 100
```