PL/SQL

PL/SQL programming language was **developed by Oracle Corporation** in the late 1980s as **procedural extension language for SQL** and the Oracle relational database.

PL/SQL is a **completely portable**, **high-performance transaction-processing language**.

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.

Advantages

- SQL is the standard database language and PL/SQL is strongly integrated with SQL.
- PL/SQL **supports 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 a database.
- PL/SQL **saves time on design and debugging** by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- 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**.

PL/SQL Block Structured

Declarations –

This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

Executable Commands –

This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.

Exception Handling -

This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.

```
DECLARE
        <declarations section>
BEGIN
        <executable command(s)>
EXCEPTION
        <exception handling>
END;
```

Variable Declaration in PL/SQL

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Example :

```
name varchar2(25);
```

Example :

```
DECLARE
        message varchar2(20):= 'Hello, World!’;
BEGIN
        dbms_output.put_line(message);
END;
/
```

Output :

```
Hello World

PL/SQL procedure successfully completed.
```

PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments. The PL/SQL supports single-line and multi-line comments.

All characters available inside any comment are ignored by the PL/SQL compiler.

Single Line Comment : To create a single line comment , the symbol – – is used.

Multi Line Comment : To create comments that span over several lines, the symbol /* and */ is used.

Example :

```
DECLARE
      -- variable declaration
      message varchar2(20):= 'Hello, World!';
BEGIN
      /*
      PL/SQL executable statement(s)
      */
      dbms_output.put_line(message);
END;
/
```

```
DECLARE

message varchar2(30):= ‘Hello’;

BEGIN

dbms_output.put_line(message);

END;

/
```

PL/SQL Execution Environment

The PL/SQL engine resides in the Oracle engine.
The Oracle engine can process not only single SQL statement but also block of many statements.
The call to Oracle engine needs to be made only once to execute any number of SQL statements if these SQL statements are bundled inside a PL/SQL block.

SET SERVEROUTPUT ON

It is used to display the buffer used by the dbms_output.

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.

dbms_output.put_line

This command is used to direct the PL/SQL output to a screen.

Display Output

The outputs are displayed by using **DBMS_OUTPUT which is a built-in package that enables the user to display output**, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
     var varchar2(40) :=‘Hello friends’;

  BEGIN
     dbms_output.put_line(var);

  END;
  /
```

PL/SQL code to print sum of two numbers taken from the user

```
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;
    /
```

%TYPE Attribute (Use to declare variables that hold table columns)

The **%TYPE attribute** provides the **datatype of a variable** or database column.

This is particularly useful when declaring variables that will hold database values.

Syntax:

identifier Table.column_name%TYPE;

Example: Suppose column name **title** in a table named **books**. To declare a variable named **my_title** that has the same datatype as column **title**.

my_title books.title%TYPE;

Declaring my_title with %TYPE has two advantages.

- You **need not know the exact datatype** of title.
- if you **change the database definition of title the datatype of my_title changes** accordingly at run time.

```
DECLARE
  l_last_name employees.last_name%TYPE;
  l_department_name departments.department_name%TYPE;
BEGIN
  SELECT last_name, department_name INTO l_last_name, l_department_name
    FROM employees e, departments d WHERE e.department_id=d.department_id
        AND e.employee_id=138;


DBMS_OUTPUT.put_line (l_last_name || ' in ‘ || l_department_name);
END;
```

```
DECLARE
c_id customers.id%type := 1;
c_name customers.name%type;
c_addr customers.address%type;
c_sal customers.salary%type;

BEGIN
SELECT name, address, salary INTO c_name, c_addr, c_sal
FROM customers
WHERE id = c_id;

dbms_output.put_line
('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);

END;
/
```

%ROWTYPE Attribute (used to declare variables that hold table rows)

A record consists of a number of related fields in which data values can be stored.
The %ROWTYPE attribute provides a record type that represents a row in a table.
The record can store an entire row of data selected from the table or fetched from a cursor. Columns in a row and corresponding fields in a record have the same names and datatypes.

Example : you declare a record variable **dept_rec**.
Its fields have the same names and datatypes as the columns in the dept table.

```
DECLARE
dept_rec dept%ROWTYPE;        -- declare record variable
```

You use dot notation to reference fields

```
my_deptno := dept_rec.deptno;
```

Example:

Fetch an entire row from the employee table for a specific employee ID.

```
DECLARE
v_emp employee%ROWTYPE;
BEGIN
SELECT * INTO v_emp FROM employee WHERE employee_id = 205;
dbms_output.put_line( v_emp.last_name);
END;
```

```
DECLARE

customer_rec customers%rowtype;

BEGIN

SELECT * into customer_rec FROM customers WHERE id = 5;

dbms_output.put_line('Customer ID: ' || customer_rec.id);

dbms_output.put_line('Customer Name: ' || customer_rec.name);

dbms_output.put_line('Customer Address: ' || customer_rec.address);

dbms_output.put_line('Customer Salary: ' || customer_rec.salary);

END;
/
```

```
CREATE OR REPLACE PROCEDURE emp_sal_query ( p_empno IN emp.empno%TYPE )
IS
r_emp emp%ROWTYPE;
v_avgsal emp.sal%TYPE;

BEGIN
    SELECT ename, job, hiredate, sal, deptno INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal,
    r_emp.deptno FROM emp WHERE empno = p_empno;
        DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
        DBMS_OUTPUT.PUT_LINE('Name : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Hire Date : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Dept # : ' || r_emp.deptno);
    SELECT AVG(sal) INTO v_avgsal FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE(v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE(v_avgsal);
    END IF;
END;
```

Example: How to use a cursor with the %ROWTYPE attribute to retrieve department information about each employee in the EMP table.

```
CREATE OR REPLACE PROCEDURE emp_info
IS
CURSOR empcur IS SELECT ename, deptno FROM emp;
myvar empcur%ROWTYPE;
BEGIN
OPEN empcur;
LOOP
FETCH empcur INTO myvar;
EXIT WHEN empcur%NOTFOUND;
DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department ' || myvar.deptno );
END LOOP;
CLOSE empcur;
END;
```

Cursor

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement.
Example: The number of rows processed, etc.
A **cursor is a pointer** to this context area.
PL/SQL controls the context area through a cursor.
A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.
You can **name a cursor** so that it could be referred to in a program **to fetch and process the rows** returned by the SQL statement, one at a time.
There are two types of cursors:

- Implicit cursors
- Explicit cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors.

Whenever a **DML statement** (**INSERT, UPDATE and DELETE) is issued**, an **implicit cursor is associated with this statement**.

For **INSERT operations**, the **cursor holds the data** that needs to be inserted.

For **UPDATE and DELETE operations**, the **cursor identifies the rows** that would be affected.

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

SQL cursor attribute will be accessed as **sql%attribute_name**

Example:

Update the salary of each customer by 3000 and determine the number of rows affected.

```
DECLARE
        total_rows number(2);
BEGIN
        UPDATE customers SET salary = salary + 3000;
        IF sql%notfound THEN
                dbms_output.put_line('no customers selected');
        ELSIF sql%found THEN
                total_rows := sql%rowcount;
                dbms_output.put_line( total_rows || ' customers selected ');
        END IF;
END;
/
```

Output:

6 customers selected

PL/SQL procedure successfully completed.

SELECT-INTO cursor: **fastest and simplest way to fetch a single row** from a SELECT statement.

SELECT select_list INTO variable_list FROM remainder_of_query;

If the SELECT statement identifies more than one row to be fetched, Oracle Database will raise the TOO_MANY_ROWS exception. If the statement doesn't identify any rows to be fetched, Oracle Database will raise the NO_DATA_FOUND exception.

Get the last name for a specific employee ID (the primary key in the employees table)

```
DECLARE
l_last_name employees.last_name%TYPE;
BEGIN
SELECT last_name INTO l_last_name FROM employees WHERE employee_id = 138;
DBMS_OUTPUT.put_line ( l_last_name);
END;
```

Explicit Cursors:

Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor 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.

Syntax :

```
CURSOR cursor_name IS select_statement;
```

steps of working with an explicit cursor :

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

```
CURSOR c_customers IS
SELECT id, name, address FROM customers;
```

Opening the Cursor

```
OPEN c_customers;
```

Fetching the Cursor

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

```
CLOSE c_customers;
```

```
DECLARE
        c_id customers.id%type;
        c_name customerS.No.ame%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;
/
```

Output:

001 Ram Dehradun
002 Mohan Roorkee
003 Mukesh Haridwar
004 Ashok Meerut
005 Rajkumar Roorkee
006 Shubham Rishikesh

PL/SQL procedure successfully completed.

```
DECLARE
-- %ROWTYPE can include all the columns in a table...
emp_rec employees%ROWTYPE;
-- ...or a subset of the columns, based on a cursor.
CURSOR c1 IS
SELECT department_id, department_name FROM departments;
dept_rec c1%ROWTYPE;
-- Could even make a %ROWTYPE with columns from multiple tables.
CURSOR c2 IS
SELECT employee_id, email, employees.manager_id, location_id
FROM employees, departments
WHERE employees.department_id = departments.department_id;
join_rec c2%ROWTYPE;
BEGIN
-- We know EMP_REC can hold a row from the EMPLOYEES table.
SELECT * INTO emp_rec FROM employees WHERE ROWNUM < 2;
-- We can refer to the fields of EMP_REC using column names
-- from the EMPLOYEES table.
IF emp_rec.department_id = 20 AND emp_rec.last_name = 'JOHNSON' THEN
emp_rec.salary := emp_rec.salary * 1.15;
END IF;
END;
/
```

Example

```
DECLARE
emp_rec employees%ROWTYPE;
empno employees.employee_id%TYPE := 100;
CURSOR c1 IS SELECT department_id, department_name, location_id FROM departments;
dept_rec c1%ROWTYPE;
BEGIN
SELECT * INTO emp_rec FROM employees WHERE employee_id = empno;
        IF (emp_rec.department_id = 20) AND (emp_rec.salary > 2000) THEN
                NULL;
        END IF;
END;
/
```

Triggers

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

Syntax:

```
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;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger will be executed.

  The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that will be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.

- [REFERENCING OLD AS o NEW AS n]:

  This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

- [FOR EACH ROW]: This specifies a **row-level trigger**,

  i.e., the trigger will be executed for each row being affected.

  Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

- WHEN (condition):

  This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff:=NEW.salary-OLD.salary;
   dbms_output.put_line('Old salary: ' || OLD.salary);
   dbms_output.put_line('New salary: ' || NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations,

for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

IF-THEN Statement

Syntax

IF (condition) THEN

statement;

END IF;

Example

```
DECLARE
        a number(2) := 10;
BEGIN
        a:= 10;

        IF( a < 20 ) THEN

                dbms_output.put_line('a is less than 20 ' );

        END IF;
dbms_output.put_line('value of a is : ' || a);
END;

/
```

```
DECLARE
        c_id customers.id%type := 1;
        c_sal customers.salary%type;
BEGIN
        SELECT salary INTO c_sal FROM customers WHERE id = c_id;

        IF (c_sal <= 2000) THEN

                UPDATE customers SET salary = salary + 1000 WHERE id = c_id;

                dbms_output.put_line ('Salary updated’);

        END IF;

END;

/
```

IF-THEN-ELSE Statement

Syntax

IF (condition) THEN

statement1;

ELSE

statement2;

END IF;

IF-THEN-ELSIF Statement

Syntax

```
IF(boolean_expression 1)THEN
S1;
ELSIF( boolean_expression 2) THEN
S2;
ELSIF( boolean_expression 3) THEN
S3;
ELSE
S4;                                  -- executes when the none of the above condition is true
END IF;
```

Case statement

Syntax

```
CASE selector
WHEN 'value1' THEN S1;
WHEN 'value2' THEN S2;
WHEN 'value3' THEN S3;
...
ELSE Sn;                          -- default case
END CASE;
```

Example

```
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('Well done');
            when 'D' then dbms_output.put_line('You passed');
            when 'F' then dbms_output.put_line('Better try again');
            else dbms_output.put_line('No such grade’);
        END CASE;
END;
/
```

Nested IF-THEN-ELSE Statements

It is always legal in PL/SQL programming to nest the **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

Syntax

```
IF( boolean_expression 1)THEN
                -- executes when the boolean expression 1 is true
        IF(boolean_expression 2) THEN
                -- executes when the boolean expression 2 is true
        sequence-of-statements;
        END IF;
ELSE
                -- executes when the boolean expression 1 is not true
        else-statements;
END IF;
```

Basic Loop Statement

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax

LOOP

Sequence of statements;

END LOOP;

Here, the sequence of statement(s) may be a single statement or a block of statements. An **EXIT statement** or an **EXIT WHEN statement** is required to break the loop.

Example

```
DECLARE
x number := 10;
BEGIN
        LOOP
                dbms_output.put_line(x);
                x := x + 10;
                IF x > 50 THEN
                        exit;
                END IF;
        END LOOP;
dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

```
DECLARE
x number := 10;
BEGIN
        LOOP
                dbms_output.put_line(x);
                x := x + 10;
                exit WHEN x > 50;
        END LOOP;
        -- after exit, control resumes here
dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

WHILE LOOP

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

WHILE (condition) LOOP

sequence_of_statements

END LOOP;

FOR LOOP Statement

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

FOR counter IN initial_value .. final_value LOOP

sequence_of_statements;

END LOOP;

Example

```
DECLARE
a number(2);
BEGIN
        FOR a in 10 .. 20 LOOP
                dbms_output.put_line('value of a: ' || a);
        END LOOP;
END;
/
```