# P L / S Q L  I N T R O D U C T I O N

## PL/SQL

- PL/SQL bridges the gap between database technology and procedural programming languages.
- PL/SQL uses the facilities of the sophisticated RDBMS and extends the standard SQL database language
- Not only PL/SQL allow you to insert, delete, update and retrieve data, it lets you use procedural techniques such as looping and branching to process the data.
- Thus PL/SQL provides the data manipulating power of SQL with the data processing power of procedural languages

## Advantage of PL/SQL

PL/SQL is a completely portable, high performance transaction processing language. It provides the following advantages :

- Procedural Capabilities
  - It supports many of constructs like constants, variable, selection and iterative statements
- Improved Performance
  - Block of SQL statements can be executed at a time
- Enhanced Productivity
  - PL/SQL brings added functionality to non procedural tools such as SQL Forms.
- Portability
  - PL/SQL can run anywhere the RDBMS can run
- Integration with RDBMS
  - Most PL/SQL variables have data types native to the RDBMS data dictionary. Combined with the direct access to SQL, these native data type declarations allow easy integration of PL/SQL with RDBMS.

## Character Set

It is either ASCII or EBCDIC format

## Identifiers

It begins with a letter and can be followed by letters, numbers, $ or #. Maximum size is 30 characters in length.

**Variable Declaration**

The data types (number, varchar2, real, date, …) discussed in SQL are all applicable in PL/SQL.

Ex. Salary *Number(7,2);*
Sex *Boolean;*
Count *smallint :=0;*
Tax *number default 750;*
Name *varchar2(20) not null;*

**Constant declaration**

Ex. Phi *Constant Number(7,2)* := 3.1417;

**Comment**

Line can be commented with double hyphen at the beginning of the line.

Ex. - - This is a comment line

**Assignments**

Variable assignment sets the current value of a variable. You can assign values to a variable as follows

(i) Assignment operator (:=)

Ex. d := b*b – 4*a*c;

(ii) Select … into statement

Ex. *Select* sal *into* salary *from* emp *where* empno=7655;

**Operators**

Operators used in SQL are all applicable to PL/SQL also.

**Block Structure**

PL/SQL code is grouped into structures called blocks. If you create a stored procedure or package, you give the block of PL/SQL code a name. If the block of PL/SQL code is not given a name, then it is called an anonymous block.

The PL/SQL block divided into three section: declaration section, the executable section and the exception section

The structure of a typical PL/SQL block is shown in the listing:

```
declare
        < declaration section >
begin
        < executable  commands>
exception
        <exception handling>
end;
```

*Declaration Section :*

Defines and initializes the variables and cursor used in the block

*Executable commands :*

Uses flow-control commands (such as IF command and loops) to execute the commands and assign values to the declared variables

*Exception handling :*

Provides handling of error conditions

**Declaration Using attributes**

***(i) %type attribute***

The %TYPE attribute provides the data type of a variable, constant, or database column. Variables and constants declared using %TYPE are treated like those declared using a data type name.

For example in the declaration below, PL/SQL treats debit like a REAL(7,2) variable.

```
credit  REAL(7,2);
debit   credit%TYPE;
```

The %TYPE attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference an owner, table, and column.

```
my_dname     dept.dname%TYPE;
```

Using %TYPE to declare my_dname has two advantages.

- First, you need not know the exact datatype of dname.
- Second, if the database definition of dname changes, the datatype of my_dname changes accordingly at run time.

***(ii) %rowtype attribute***

The %ROWTYPE attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table or fetched by a cursor.

```
DECLARE
    emp_rec   emp%ROWTYPE;
    ...
BEGIN
    SELECT  *  INTO  emp_rec  FROM  emp  WHERE ...
    ...
END;
```

Columns in a row and corresponding fields in a record have the same names and data types.

The column values returned by the SELECT statement are stored in fields. To reference a field, you use the dot notation.

```
IF   emp_rec.deptno = 20  THEN ...
```

In addition, you can assign the value of an expression to a specific field.

```
emp_rec.ename := 'JOHNSON';
```

A %ROWTYPE declaration cannot include an initialization clause. However, there are two ways to assign values to all fields in a record at once.

First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.

```
DECLARE
        dept_rec1    dept%ROWTYPE;
        dept_rec2    dept%ROWTYPE;
        …..
BEGIN
..      …..
        dept_rec1  :=  dept_rec2;
        …..
END;
```

Second, you can assign a list of column values to a record by using the SELECT and FETCH statement, as the example below shows. The column names must appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement.

```
DECLARE
        dept_rec    dept%ROWTYPE;
        ….
BEGIN
        SELECT  deptno, dname, loc INTO  dept_rec  FROM dept
        WHERE deptno = 30;
        ….
END;
```

However, you cannot assign a list of column values to a record by using an assignment statement. Although you can retrieve entire records, you cannot insert them.

For example, the following statement is illegal:

```
INSERT INTO  dept  VALUES (dept_rec); -- illegal
```

**Creating and Executing PL/SQL Programs**

Edit your PL/SQL program in your favourite editor as text file.

Execute the following command once for a session to get displayed the output.

SQL> *set serveroutput on;*

Now execute the program using the following command.

SQL> *start* filename; (or) SQL> @filename;

Note : Give absolute path of the filename if you saved the file in some directory.

Ex. SQL> *start* z:\plsql\ex11; (or) SQL> @ z:\plsql\ex11;

**Control Structures**

***(i) IF Statements***

There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The third form of IF statement uses the keyword ELSIF (NOT ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
        sequence_of_statements1;
ELSIF condition2 THEN
        sequence_of_statements2;
ELSE
        sequence_of_statements3;
END IF;
```

***(ii) LOOP and EXIT Statements***

There are three forms of LOOP statements. They are LOOP, WHILE-LOOP, and FOR-LOOP.

***LOOP***

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
        sequence_of_statements3;
        ...
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use the EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

```
LOOP
        ...
        IF ... THEN
                ...
                EXIT; -- exit loop immediately
        END IF;
END LOOP;
-- control resumes here
```

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition evaluates to TRUE, the loop completes and control passes to the next statement after the loop.

```
LOOP
        ….
        EXIT WHEN i>n; -- exit loop if condition is true
        ….
END LOOP;
….
```

Until the condition evaluates to TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition.

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
        sequence_of_statements;
        ...
END LOOP [label_name];
```

Optionally, the label name can also appear at the end of the LOOP statement.

With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete, then use the label in an EXIT statement.

```
<<outer>>
LOOP
        ...
        LOOP
                ...
                EXIT outer WHEN ... -- exit both loops
        END LOOP;
...
END LOOP outer;
```

***(iii) WHILE-LOOP***

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

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

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. Since the condition is tested at the top of the loop, the sequence might execute zero times.

***(iv) FOR-LOOP***

Whereas the number of iteration through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

```
FOR counter IN [REVERSE] lower_bound..upper_bound LOOP
        sequence_of_statements;
        ...
END LOOP;
```

The lower bound need not be 1. However, the loop counter increment (or decrement) must be 1. PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR  i  IN  1..emp_count   LOOP
        ...
END LOOP;
```

The loop counter is defined only within the loop. You cannot reference it outside the loop. You need not explicitly declare the loop counter because it is implicitly declared as a local variable of type INTEGER.

The EXIT statement allows a FOR loop to complete prematurely. You can complete not only the current loop, but any enclosing loop.

***(v) GOTO and NULL statements***

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can make the meaning and action of conditional statements clear and so improve readability.

```
BEGIN
        ...
        GOTO insert_row;
        ...
        <<insert_row>>
        INSERT INTO emp VALUES ...
END;
```

A GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block. A GOTO statement cannot branch from one IF statement clause to another. A GOTO statement cannot branch out of a subprogram. Finally, a GOTO statement cannot branch from an exception handler into the current block.

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. Also, the NULL statement is a handy way to create stubs when designing applications from the top down.

* * *

| Ex. No. 11 | B A S I C  P L / S Q L | Date : |
|---|---|---|

**Q1)** Write a PL/SQL Block to find the maximum of 3 Numbers

```
Declare
        a  number;
        b  number;
        c  number;
Begin
        dbms_output.put_line('Enter a:');
        a:=&a;
        dbms_output.put_line('Enter b:');
        b:=&b;
        dbms_output.put_line('Enter c:');
        c:=&c;
        if (a>b) and (a>c) then
                dbms_output.putline('A is Maximum');
        elsif (b>a) and (b>c) then
                dbms_output.putline('B is Maximum');
        else
                dbms_output.putline('C is Maximum');
        end if;
End;
/
```

**Q2)** Write a PL/SQL Block to find the sum of odd numbers upto 100 using loop statement

```
Declare


Begin









End;
/
```