PL/SQL

- The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database.
- **Features of PL/SQL**
- PL/SQL has the following features −

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

- PL/SQL has the following advantages −
- 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 is not a standalone programming language; it is a tool within the Oracle programming environment.
- **SQL* Plus** is an interactive tool that allows you to type SQL and PL/SQL statements at the command prompt.
- These commands are then sent to the database for processing. Once the statements are processed, the results are sent back and displayed on screen.

Basic Syntax of PL/SQL

- Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts.
- Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block.
- DECLARE
- <declarations section>
- BEGIN
- <executable command(s)>
- EXCEPTION
- <exception handling>
- END;

S.No	Sections & Description
1	**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.
2	**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.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

- **The 'Hello World' Example**
- DECLARE
- message varchar2(20):= 'Hello, World!';
- BEGIN
- dbms_output.put_line(message);
- END;
- /

- Output:
- Hello World

- PL/SQL procedure successfully completed.

Identifiers and Comments

- Identifiers

- PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.
- By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

- Comments

- The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the **delimiter -- (double hyphen)** and multi-line comments are enclosed by **/* and */**

Data Types

1. Scalar

- Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.

2. Large Object (LOB)

- Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.

3. Composite

- Data items that have internal components that can be accessed individually. For example, collections and records.

4. Reference

- Pointers to other data items

Syntax:

```
DECLARE
   num1 INTEGER;
   num2 REAL;
   num3 DOUBLE PRECISION;
BEGIN
   null;
END;
/
```

Scalar Datatypes:

- BOOLEAN
- CHAR
- DATE
- NUMBER
- The BOOLEAN data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.
- **Large Object (LOB) Data Types:**
- Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms.

Variable Declaration in PL/SQL

EX: sales number(10, 2);

name varchar2(25);

address varchar2(100);

- When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations as shown above.

Initializing Variables in PL/SQL

- Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −
- The DEFAULT keyword
- The assignment operator

EX: counter binary_integer := 0;

greetings varchar2(20) DEFAULT 'Have a Good Day’;

EX:

```
DECLARE
   a integer := 10;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
/
```

OP:

Value of c: 30

Value of f: 23.333333333333333

PL/SQL procedure successfully completed.

Assigning SQL Query Results to PL/SQL Variables

- You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

```
CREATE TABLE CUSTOMERS(
   ID   INT NOT NULL,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

- Table Created

Let us now insert some values in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
```

Declaring constant

```
DECLARE
    pi constant number := 3.141592654;    -- constant declaration
   radius number(5,2);                    -- other declarations
   dia number(5,2);
   circumference number(7, 2);
   area number (10, 2);
BEGIN
   radius := 9.5;
   dia := radius * 2;
   circumference := 2.0 * pi * radius;
   area := pi * radius * radius;
   dbms_output.put_line('Radius: ' || radius);
   dbms_output.put_line('Diameter: ' || dia);
   dbms_output.put_line('Circumference: ' || circumference);
   dbms_output.put_line('Area: ' || area);
END;
/
```

OP:

Radius: 9.5

Diameter: 19

Circumference: 59.69

Area: 283.53

Operators

PL/SQL language is rich in built-in operators and provides the following types of operators −

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

Arithmetic and Relational Operators

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

Conditions

1. IF - THEN statement
2. IF -THEN-ELSE statement
3. IF-THEN-ELSIF statement
4. Case statement
5. Nested IF-THEN-ELSE

1. IF - THEN statement

Syntax:

```
IF condition THEN
   S;
END IF;
```

EX:

```
DECLARE
   a number(2) := 10;
BEGIN
   a:= 10;
  -- check the 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;
/
```

OP:

```
a is less than 20
value of a is : 10
PL/SQL procedure successfully completed.
```

Loops

1. Basic LOOP
2. While
3. For
4. Nested loop

Basic Loop statement:

```
LOOP
   Sequence of statements;
END LOOP;
```

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

OP:

10 20 30 40 50

After Exit x is: 60

PL/SQL procedure successfully completed.

- You can use the EXIT WHEN statement instead of the EXIT statement:
- 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

```
DECLARE
a NUMBER :=1;
BEGIN
dbms_output.put_line('Program started');
WHILE (a <= 5)
LOOP
dbms_output.put_line(a);
a:=a+1;
END LOOP;
dbms_output.put_line(‘Program completed' );
END:
/
```

Cursors

- Sql Server creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for 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.**
- There are two types of cursors −

1. Implicit cursors
2. Explicit cursors

Implicit Cursors

- Implicit cursors are automatically created by server whenever an SQL statement is executed.
- Whenever any DML operations occur in the database, an implicit cursor is created that holds the rows affected, in that particular operation. These cursors cannot be named and, hence they cannot be controlled or referred from another place of the code. We can refer only to **the most recent cursor** through the cursor attributes, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.

The following program will update the table and increase the salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( ' customers selected ‘ || total_rows);
  END IF;
END;
/
```

OP:

6 customers selected.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+---------
```

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

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.
- Working with an explicit cursor includes the following steps −

1. **Declaring** the cursor for initializing the memory
2. **Opening** the cursor for allocating the memory
3. **Fetching** the cursor for retrieving the data
4. **Closing** the cursor to release the allocated memory

1. Declaring the Cursor

- Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

- CURSOR c_customers is
- SELECT id, name, address FROM customers;

2. Opening the Cursor

- Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −
- OPEN c_customers;

3. Fetching the Cursor

- Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −
- FETCH c_customers INTO c_id, c_name, c_addr;

4. Closing the Cursor

- Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −
- CLOSE c_customers;

```
DECLARE
   c_id customers.id%type;                --variable declaration with formal parameter
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is                  --cursor declaration
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;       --open
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;     --fetch
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;     --close
END;
/
```

OP:

1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

PL/SQL procedure successfully completed.

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 −

1. A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
2. A database definition (DDL) statement (CREATE, ALTER, or DROP).
3. A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

- **Benefits of Triggers :**

1. Generating some derived column values automatically
2. Enforcing referential integrity
3. Event logging and storing information on table access
4. Imposing security authorizations
5. Preventing invalid transactions

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;
```

1. CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
2. {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
3. {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
4. [OF col_name] − This specifies the column name that will be updated.
5. [ON table_name] − This specifies the name of the table associated with the trigger.
6. [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.
7. [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.
8. WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

```
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;
/
```

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

Above program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.

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

Trigger created.

- The following points need to be considered here −

1. OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

2. The above trigger has been written in such a way that it will 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.

- **Triggering a Trigger**
- Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

- When a record is created in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

- Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

- When a record is updated in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```