PL/SQL

UNIT III

INTRODUCTION

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

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. Following are certain notable facts about PL/SQL −

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL*Plus interface.
- Direct call can also be made from external programming language calls to database.

Features of PL/SQL

- 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.

BASIC SYNTAX

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 –

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.

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;
```

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

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code.

DATA TYPES

S.No	Date Type & Description
1	**Numeric** Numeric values on which arithmetic operations are performed.
2	**Character** Alphanumeric values that represent single characters or strings of characters.
3	**Boolean** Logical values on which logical operations are performed.
4	**Datetime** Dates and times

S.No	Data Type & Description
1	**PLS_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	**BINARY_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	**BINARY_FLOAT** Single-precision IEEE 754-format floating-point number
4	**BINARY_DOUBLE** Double-precision IEEE 754-format floating-point number
5	**NUMBER(prec, scale)** Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0
6	**DEC(prec, scale)** ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	**DECIMAL(prec, scale)** IBM specific fixed-point type with maximum precision of 38 decimal digits
8	**NUMERIC(pre, secale)** Floating type with maximum precision of 38 decimal digits
9	**DOUBLE PRECISION** ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
10	**FLOAT** ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	**INT** ANSI specific integer type with maximum precision of 38 decimal digits
12	**INTEGER** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	**SMALLINT** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	**REAL** Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

```
DECLARE
      num1 INTEGER;
      num2 REAL;
      num3 DOUBLE
PRECISION;

BEGIN
null;

END;
/
```

VARIABLE

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

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

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type

Some valid variable declarations along with their definition are shown below –

```
Sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

```
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;
/
```

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

```
Value of c: 30

Value of f: 23.333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope

- **Local variables** – Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** – Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form –

```
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;
/
```

```
Outer Variable num1: 95
Outer Variable num2: 85

Inner Variable num1: 195
Inner Variable num2: 185
```

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) );
```

```
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;
/
```

```
Customer Ramesh from Ahmedabad earns 2000
```

LOOPS IN PL/SQL

PL/SQL provides the following types of loop to handle the looping requirements.

1. PL/SQL Basic LOOP

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.

```
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;
/
```

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

```
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;
/
```

PL/SQL - WHILE LOOP Statement

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;
```

```
DECLARE
   a number(2) := 10;
BEGIN
  WHILE a < 20 LOOP
        dbms_output.put_line('value of a: ' || a);
        a := a + 1;
  END LOOP;
END;
/
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

PL/SQL - 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;
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
 PL/SQL procedure successfully completed.
```

Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this –

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

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
 value of a: 12
value of a: 11
value of a: 10
PL/SQL procedure successfully completed.
```

Following is the flow of control in a **For Loop** –

•The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.

•Next, the condition, i.e., *initial_value .. final_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.

•After the body of the for loop executes, the value of the counter variable is increased or decreased.

•The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

Following are some special characteristics of PL/SQL for loop –

- The *initial_value* and *final_value* of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The *initial_value* need not be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows the determination of the loop range dynamically at run time.

PL/SQL - Nested Loops

PL/SQL allows using one loop inside another loop. Following section shows a few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows –

```
LOOP
   Sequence of statements1
LOOP
  Sequence of statements2
END LOOP;
END LOOP;
```

```
FOR counter1 IN initial_value1 .. final_value1LOOP
    sequence_of_statements1
FOR counter2 IN initial_value2 .. final_value2 LOOP
    sequence_of_statements2
END LOOP;
END LOOP;
```

```
WHILE condition1 LOOP
     sequence_of_statements1
WHILE condition2 LOOP
    sequence_of_statements2
END LOOP;
END LOOP;
```

The following program uses a nested basic loop to find the prime numbers from 2 to 100 −

```
DECLARE
i number(3);
j number(3);
BEGIN
i := 2;
LOOP
j:= 2;
LOOP
exit WHEN ((mod(i, j) = 0) or (j = i));
 j := j +1;
END LOOP;
IF (j = i ) THEN dbms_output.put_line(i || ' is prime');
END IF; i := i + 1;
exit WHEN i = 50;
END LOOP;
END;
 /
```

Functions & Procedures

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms –

- **Functions** – These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
< procedure_body >
 END procedure_name;
```

Where,

•*procedure-name* specifies the name of the procedure.

•[OR REPLACE] option allows the modification of an existing procedure.

•The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

•*procedure-body* contains the executable part.

•The AS keyword is used instead of the IS keyword for creating a standalone procedure.

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
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.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways –

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as –

```
EXECUTE greetings;
```

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

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

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement.

Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement −

DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
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; /
```

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
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; /
```

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
 BEGIN < function_body >
END [function_name];
```

Where,

•*function-name* specifies the name of the function.

•[OR REPLACE] option allows the modification of an existing function.

•The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

•The function must contain a **return** statement.

•The *RETURN* clause specifies the data type you are going to return from the function.

•*function-body* contains the executable part.

•The AS keyword is used instead of the IS keyword for creating a standalone function.

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.
We will use the CUSTOMERS table.

```
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 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers

  RETURN number IS
    total number(2) := 0;

BEGIN
  SELECT count(*) into total FROM customers;
  RETURN total;

END;
/
```

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

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block –

```
DECLARE
 c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

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

Total no. of Customers: 6 PL/SQL procedure successfully completed.

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
        a number;
        b number;
        c number;

FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
            IF x > y THEN
               z:= x;
            ELSE
               z:= y;
            END IF;
 RETURN z;
END;
BEGIN
        a:= 23;
        b:= 45;
        c := findMax(a, b);
        dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
 /
```

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

Maximum of (23,45): 45 PL/SQL procedure successfully completed.

CURSORS

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:

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 and the information in it.

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.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**.

The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes –

S.No	Attribute & Description
1	**%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.
2	**%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.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

We will be using the CUSTOMERS table :

```
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 |
 +----+----------+-----+-----------+----------+
```

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( total_rows || ' customers selected ');
END IF;
END;
 /
```

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

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

If you check the records in customers table, you will find that the rows have been updated –

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.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

- 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

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;
```

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;
```

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;
```

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;
```

Example

Following is a complete example to illustrate the concepts of explicit 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;
/
```

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

```
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

•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.

Benefits of Triggers

Triggers can be written for the following purposes –

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is –

```
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.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
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 |
+----+----------+-----+-----------+----------+
```

The following 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. 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; /
```

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 –

•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.

•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
```

Type of Triggers

1. BEFORE Trigger : BEFORE trigger execute before the triggering DML statement (INSERT, UPDATE, DELETE) execute. Triggering SQL statement is may or may not execute, depending on the BEFORE trigger conditions block.

2. AFTER Trigger : AFTER trigger execute after the triggering DML statement (INSERT, UPDATE, DELETE) executed. Triggering SQL statement is execute as soon as followed by the code of trigger before performing Database operation.

3. ROW Trigger : ROW trigger fire for each and every record which are performing INSERT, UPDATE, DELETE from the database table. If row deleting is define as trigger event, when trigger file, deletes the five rows each times from the table.

4. Statement Trigger : Statement trigger fire only once for each statement. If row deleting is define as trigger event, when trigger file, deletes the five rows at once from the table.

5. Combination Trigger : Combination trigger are combination of two trigger type,

    1. Before Statement Trigger : Trigger fire only once for each statement before the triggering DML statement.
    2. Before Row Trigger : Trigger fire for each and every record before the triggering DML statement.
    3. After Statement Trigger : Trigger fire only once for each statement after the triggering DML statement executing.
    4. After Row Trigger : Trigger fire for each and every record after the triggering DML statement executing.

Inserting Trigger

This trigger execute BEFORE to convert ename field lowercase to uppercase.

```
CREATE or REPLACE TRIGGER trg1
BEFORE
INSERT ON emp1 FOR EACH ROW
BEGIN
:new.ename := upper(:new.ename);
END;
 /
```

Restriction to Deleting Trigger

This trigger is preventing to deleting row.
Delete Trigger Code:

```
CREATE or REPLACE TRIGGER trg1
AFTER
DELETE ON emp1
FOR EACH ROW
BEGIN
 IF :old.eno = 1 THEN
raise_application_error(-20015, 'You can't delete this row');
END IF;
END;
/
```

Concept Of Error Handling

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition.

What is Exception Handling?

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is recieved

.

PL/SQL Exception message consists of three parts.

1) Type of Exception
2) An Error Code
3) A message

By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

Syntax for Exception Handling The General Syntax for exception handling is as follows.
Here you can list down as many as exceptions you want to handle.
The default exception will be handled using WHEN others THEN:

```
DECLARE
<declaration section>
BEGIN
<executable command(s)>
EXCEPTION
<Exception handling goes here>
WHEN exception1 THEN
   exception1-handling-statements
WHEN exception2 THEN
   exception2-handling-statements
WHEN exception3 THEN
  exception3-handling-statements
    ........
WHEN others THEN
  exception3-handling-statements
 END;
```

Example Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table:

```
 DECLARE
c_id customers.id%type := 8;
c_name customers.name%type;
c_addr customers.address%type;
 BEGIN
    SELECT name, address INTO c_name, c_addr
    FROM customers
    WHERE id = c_id;

    DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
    WHEN no_data_found THEN
        dbms_output.put_line('No such customer!');
    WHEN others THEN
        dbms_output.put_line('Error!');
 END; /
```

When the above code is executed at SQL prompt, it produces the following result:
No such customer!
PL/SQL procedure successfully completed.

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

Raising Exceptions
Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command RAISE. Following is the simple syntax of raising an exception:

Following is the simple syntax of raising an exception:

```
DECLARE
exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
  EXCEPTION
```

```
WHEN exception_name THEN
statement;
END;
```

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A userdefined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception is:

```
DECLARE
my-exception EXCEPTION;
```

Example:
The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE
 c_id customers.id%type := &cc_id;
c_name customers.name%type;
c_addr customers.address%type;
 -- user defined exception
ex_invalid_id EXCEPTION;
BEGIN
IF c_id <= 0 THEN
   RAISE ex_invalid_id;
ELSE
   SELECT name, address INTO c_name, c_addr FROM customers WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
END IF;
EXCEPTION
```

```
WHEN ex_invalid_id THEN
   dbms_output.put_line('ID must be greater than zero!'); WHEN no_data_found THEN
  dbms_output.put_line('No such customer!');
WHEN others THEN
  dbms_output.put_line('Error!');
END;
/
```

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

```
 Enter value for cc_id: -6 (let's enter a value -6)
old 2: c_id customers.id%type := &cc_id;
new 2: c_id customers.id%type := -6;
ID must be greater than zero!
PL/SQL procedure successfully completed
```

For any queries email at: romanariyazuok@gmail.com