Unit – IV
PL/ SQL and Triggers

4.1 Basics of PL / SQL

- Oracle programming language – SQL , provides various functionalities required to manage a database.
- SQL is so much powerful in handling data and various database objects. But it lacks some of basic functionalities provided by other programming language.
- **For example**, SQL does not provide basic procedural capabilities such as conditional checking, branching and looping.
- In SQL, it is not possible to control execution of SQL statements based on some condition or user inputs.
- Oracle provides **PL/SQL** (**Procedural Language / Structured Query Language**) to overcome disadvantages of SQL.
- PL/SQL is super set of SQL.
- PL/SQL supports all the functionalities provided by SQL along with its own **procedural capabilities**.
- Any SQL statements can be used in PL/SQL program with no change, except SQL's **data definition statements such as CREATE TABLE**.
- Data definition statements are not allowed because PL/SQL code is **compile time**. So, **it cannot refer to objects that do not yet exist**.

4.1.1 Data types

- PL/SQL is super set of the SQL. So, it supports all the data types provided by SQL.
- Along with this, in PL/SQL Oracle provides **subtypes of the data types**.
- **For example**, the data type **NUMBER** has a subtype called **INTEGER**.
- These subtypes can be used in PL/SQL block to make the data type compatible with the data types of the other programming languages.

The various data types can be given as below:

Category	**Data Type**	**Sub types/values**
Numerical	NUMBER	BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NATURAL, POSITIVE, REAL, SMALLINT
Character	CHAR, LONG, VARCHAR2	CHARACTER,VARCHAR, STRING, NCHAR, NVARCHAR2
Date	DATE	
Binary	RAW, LONG RAW	
Boolean	BOOLEAN	Can have value like TRUE, FALSE and NULL.
RowID	ROWID	Stores values of address of each record.

4.2 Advantages of PL/SQL over SQL

1) **Procedural Capabilities:**
- PL/SQL provides procedural capabilities such as condition checking, branching and looping.
- This enables programmer to control execution of a program based on some conditions and user inputs.

2) **Support to variables:**
- PL/SQL supports declaration and use of variables.
- These variables can be used to store intermediate results of a query or some expression.

3) **Error Handling:**
- When an error occurs, user friendly message can be displayed.
- Also, execution of program can be controlled instead of abruptly terminating the program.

4) **User Defined Functions:**
- Along with a large set of in-build functions, PL/SQL also supports user defined functions and procedures.

5) **Portability:**
- Programs written in PL/SQL are portable.
- It means, programs can be transferred and executed from any other computer hardware and operating system, where Oracle is operational.

6) **Sharing of Code:**
- PL/SQL allows user to store compiled code in database. This code can be accessed and shared by different applications.
- This code can be executed by other programming language like JAVA.

❖ Creating and Executing a PL/SQL Block

- To create and execute a PL/SQL block, follow the steps given below:
- Open any editor like as notepad. An EDIT command can be used on SQL prompt to open a notepad from the SQL * PLUS environment.
- The following syntax creates and opens a file:

```
EDIT   filename
```

Example:

```
EDIT   D:/PLSQL/test.sql
```

- Create and open a file named ‘test.sql’.
- Write a program code or statements in a file and save it.
- File should have ‘.sql’ extension and last statement in file should be ‘/’.

- To execute this block, use any of the following commands on prompt.
    1. RUN filename
    2. START fileName
    3. @ FILENAME

Example:

```
@ D:/PLSQL/test.sql
```

4.3 Control Structures: Conditional, Iterative, Sequential

- In PL/SQL , the flow of execution can be controlled in three different manners as given below:
    - Conditional Control
    - Iterative Control
    - Sequential Control

1) Conditional Control:

- To control the execution of block of code based on some condition, PL/SQL provides the IF statement.
- The IF – THEN – ELSEIF – ELSE – END IF construct can be used to execute specific part of the block based on the condition provided.

Syntax:

```
IF      condition       THEN
-- Execute commands
ELSIF condition         THEN
--- Execute commands
ELSE
--- Execute commands
END IF;
```

Example:

```
Declare
        n number:=&n;

Begin
        if mod(n,2)=0 then
                dbms_output.put_line('number is even');
        else
                dbms_output.put_line('number is odd');
        end if;
End;
/
```

2) Iterative Control:

- Iterative control allows a group of statements to execute **repeatedly** in a program. It is called **Looping**.

- PL/SQL provides three constructs to implement loops, as listed below:

1. **LOOP**
2. **WHILE**
3. **FOR**

- In **PL/SQL**, any loop starts with a **LOOP** keyword and it terminates with an **END LOOP** keyword.
- Each loop requires a **conditional statement** to control the number of times a loop is executed.

1. LOOP

Syntax:

```
LOOP
        -- Execute commands
END LOOP
```

- **LOOP** is an infinite loop. It executes commands in its body infinite times.
- So, it requires an **EXIT** statement within its body to terminate the loop after executing specific iteration.

Example:

```
DECLARE
                i NUMBER := 1;
BEGIN
                LOOP
                        EXIT WHEN i>10;
                        DBMS_OUTPUT.PUT_LINE(i);
                        i := i+1;
                END LOOP;
END;
```

2. WHILE

Syntax:

```
WHILE  Condition
LOOP
        -- Execute commands
END LOOP
```

- The **WHILE** loop executes commands in its body as long as the **condition** remains **TRUE.**
- The loop terminates when the condition evaluates to **FALSE or NULL.**
- The **EXIT** statement can also be used to exit the loop.

Example:

```
DECLARE
    i INTEGER := 1;
BEGIN
    WHILE i <= 10
    LOOP
            DBMS_OUTPUT.PUT_LINE(i);
            i := i+1;
    END LOOP;
END;
```

3. FOR

Syntax:

```
FOR counter IN initial_value .. final_value
 LOOP
         statements;
END LOOP;
```

Example:

```
DECLARE
    k INTEGER;
BEGIN
    FOR k IN 1..10
         LOOP
            DBMS_OUTPUT.PUT_LINE (k);
        END LOOP;
END;
```

3) Sequential Control:

- Normally, execution proceeds sequentially within the block of code.
- Sequence can be changed conditionally as well as unconditionally.
- To alter the sequence unconditionally, the GOTO statement can be used.

Syntax:

```
GOTO label_name;
 ..
```

```
..
<<label_name>>
Statement;
```

Example:

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

BEGIN
                  <<loopstart>>

                           WHILE a < 20
                            LOOP
                                       dbms_output.put_line ('value of a: ' || a);
                                       a := a + 1;
                            IF a = 15 THEN
                                       a := a + 1;
                                       GOTO loopstart;
                           END IF;
                  END LOOP;
END;
/
```

4.4 Exceptions: Predefined Exceptions, User defined exceptions

- Run-time errors can be handled in some useful way rather than getting system specific message and terminating program directly. It's called exception handling.

There are two types of exception:

1) **System Exception:** In PL/SQL, various run-time errors are associated with different exceptions. These types of exceptions are known as system exception.

2) **User-defined Exception:** User also can define their own exception is known as user-defined exception.

- Exception handler scans the PL/SQL block to check existence of the **Exception Handling** section within block.

- If it is available then it is checked to find the code to handle exception.

Syntax:

EXCEPTION

WHEN exceptionName **THEN**
-- code to handle exception.

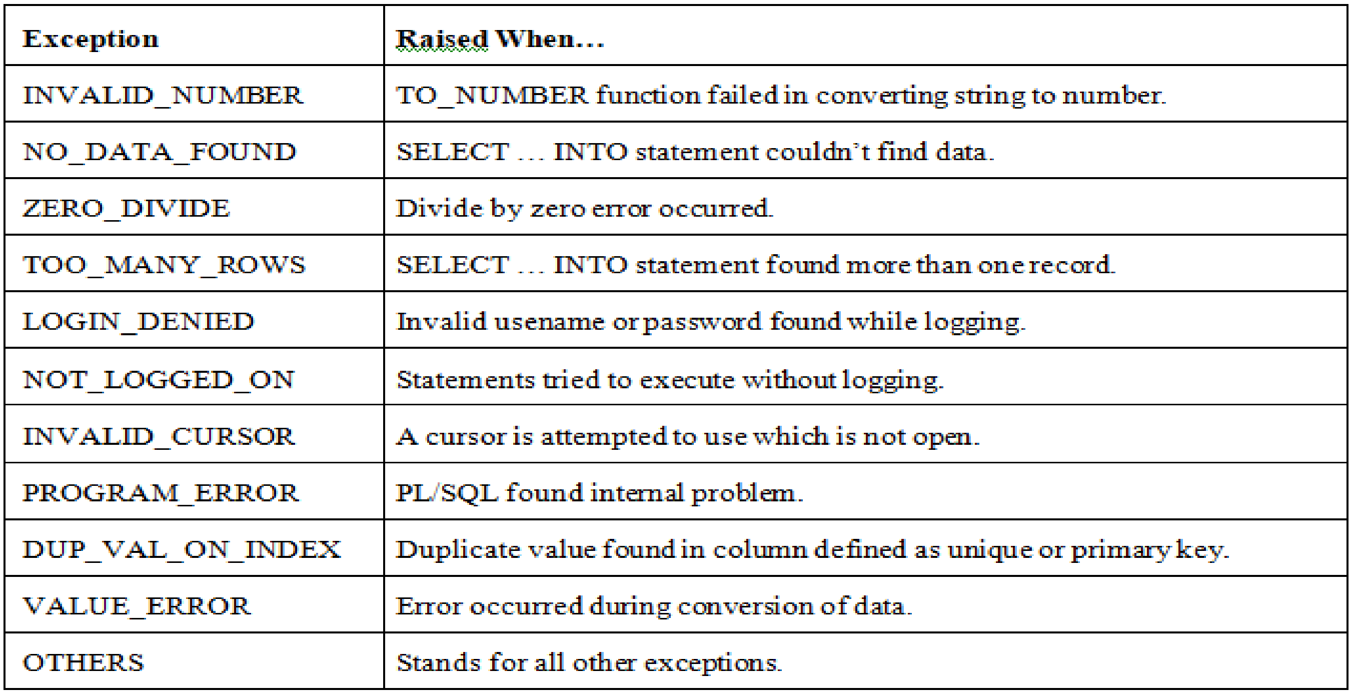
- Here, it contains more than on **WHEN** clauses.
- An **exceptionName** is a character string represents an exception to be handled.
- If **exception handling section is available** and an exception is **raised** then the appropriate code is executed. **Otherwise** an exception is handled using **default exception handling** code that is simply displaying an error message or terminating the program.

❖ Types of Exceptions

- Exception can be either **System Exception (Pre-defined Exception)** or **User-define Exception**.
- System Exceptions can be further divide in to two parts:
  - **Named Exceptions**
  - **Numbered Exceptions**

1) Named Exceptions:

- Particular name given to some common system exceptions **is known as Named Exception.**
- Oracle has defined 15 to 20 named exceptions.



Exception	**Raised When...**
INVALID_NUMBER	TO_NUMBER function failed in converting string to number.
NO_DATA_FOUND	SELECT … INTO statement couldn't find data.
ZERO_DIVIDE	Divide by zero error occurred.
TOO_MANY_ROWS	SELECT … INTO statement found more than one record.
LOGIN_DENIED	Invalid usename or password found while logging.
NOT_LOGGED_ON	Statements tried to execute without logging.
INVALID_CURSOR	A cursor is attempted to use which is not open.
PROGRAM_ERROR	PL/SQL found internal problem.
DUP_VAL_ON_INDEX	Duplicate value found in column defined as unique or primary key.
VALUE_ERROR	Error occurred during conversion of data.
OTHERS	Stands for all other exceptions.

Handling Named Exceptions

Create an Account with Acc_No as a primary key. Write a PL/SQL block to insert a record in this table. Also handle named exceptions DUP_VAL_ON_INDEX. Which is raised on encountering duplicate value for primary or unique key. (Assume table is available)

```
DECLARE
-- declare required variable
no      Account.Acc_No%TYPE;
bal     Account.Balance%TYPE;
branch          Account.B_Name%TYPE;
BEGIN

--read an account number, balance and branch name for new record no := &no;

bal  := &bal;
branch := &branch;
--insert record into Account table

INSERT INTO Account VALUES (no, bal, branch); --commit and display message confirming
insertion

COMMIT;
dbms_output.put_line('Record inserted successfully.');
EXCEPTION
--handle named exception

WHEN        DUP_VAL_ON_INDEX       THEN
dbms_output.put_line('Duplicate value found for primary
key.');
END;
/
```

Output:

```
Enter value for no: 'A01'
Enter value for bal: 5000
Enter value for no: 'RJT'
Record inserted successfully
```

2) Numbered Exceptions:

- These exceptions are identified by using negative signed number, such as -1200.
- Oracle has defined more than 20000 numbered exceptions.

Handling Numbered Exception

- A WHEN clause in exception handling section required a character string representing exception name to be handled.
- So, numbered exceptions cannot be handled directly like named exception.
- To handle numbered exceptions, they need to be bound with some names. This binding is provided in declaration section.
- After that it can be handle like named exception in exception section.

Syntax:

```
DECLARE

        exceptionName           EXCEPTION;

        PRAGMA      EXCEPTION_INIT (exceptionName, errorName);

BEGIN

        --execute commans . . .

EXCEPTION

WHENexcepetionName          THEN

        -- code to Handle Exception . . .

END ;

/
```

- A PRAGMA is a call to pre-compiler that binds the numbered exception to some name.
- A function EXCEPTION_INIT takes two parameters: one is exception name and number of the exception to be handled.
- Once binding is provided, exception can be handle in exception handling section using WHEN clause.

Example: Along with named exception, in above example also handle numbered exception with number -1200, which is raised on encountering for primary or NOT NULL key. (Assume table is available)

```
DECLARE

        exNull EXCEPTION;
        PRAGMA   EXCEPTION_INIT (exNull, -1200);

        no      Account.Acc_No%TYPE;
        bal     Account.Balance%TYPE;
        branch  Account.B_Name%TYPE;

BEGIN
        --read an account number, balance and branch name for new record
        no  := &no;
        bal  := &bal;
        branch := &branch;

        --insert record into Account table
        INSERT INTO Account VALUES (no, bal, branch);
        COMMIT;
        dbms_output.put_line('Record inserted successfully.');

EXCEPTION

        --handle named exception
        WHEN        DUP_VAL_ON_INDEX        THEN
                dbms_output.put_line('Duplicate value found for primary key.');
        --handle numbered exception
        WHEN        exNull THEN
                dbms_output.put_line('Null value found for primary key.')
        END;
/
```

Output 1 :

```
                Enter value for no: 'A02'
                Enter value for bal: 6000
                Enter value for no: 'RJT'
                Record inserted successfully.
```

Output 2 :

```
                Enter value for no: null
                Enter value for bal: 10000
                Enter value for no: 'SRT'
                Null value found for primary key.
```

3) User-defined Exceptions:

- User also can define their own exceptions are known as **user define exceptions**.
- These exceptions are used to **validate business rules** like balance for any account should not be negative value.
- User-defined exceptions need to be **declared, raised and handled explicitly**.

Syntax:

```
        DECLARE
                exceptionName           EXCEPTION ;
```

```
      BEGIN
          --SQL and PL/SQL statement

          IF      condition       THEN
                  RAISE           exceptionName
          END IF ;

  EXCEPTION

      WHEN     exceptionName           THEN
      -- code to Handle Exception

  END ;
```

Handling User-defined Exceptions

```
DECLARE
        exNull EXCEPTION;
        PRAGMA      EXCEPTION_INIT (exNull, -1200);
        myEx   EXCEPTION;

        no      Account.Acc_No%TYPE;
        bal     Account.Balance%TYPE;
        branch  Account.B_Name%TYPE;

BEGIN

--read an account number, balance and branch name for new record no := &no;

bal  := &bal;
branch := &branch;

--check balance, if negative, raise 'myEx' exception
        IF      bal > 0 THEN
                RAISE myEx;
        END IF ;

--insert record into Account table
INSERT INTO Account VALUES (no, bal, branch);

--commit and display message confirming insertion
COMMIT;

dbms_output.put_line('Record inserted successfully.');
```

```
EXCEPTION

--handle named exception
WHEN            DUP_VAL_ON_INDEX        THEN
        dbms_output.put_line('Duplicate value found for primary key.');

--handle numbered exception
WHEN exNull THEN
        dbms_output.put_line('Null value found for primary key.')

--handle user-defined exception
WHENmyEx   THEN
dbms_output.put_line('Balance cannot be negative value.')

END;
/
```

Output 1 :

```
                Enter value for no: 'A03'
                Enter value for bal: 6000
                Enter value for no: 'RJT'
                Record inserted successfully.
```

Output 2 :

```
                Enter value for no: 'A04'
                Enter value for bal: -10000
                Enter value for no: 'SRT'
                Balance cannot be negative value.
```

4.5 Cursors: Static (Implicit & Explicit), Dynamic

- Whenever an SQL statement is executed, Oracle reserves a private **SQL area in memory**.
- The data required to execute the statement are **loaded** in this memory area from the **hard disk**.
- Once data are stored in memory, they are processed as per the operation.
- After processing is finished, updated data are stored back to the hard disk and **memory is freed**.
- Cursor comes into picture for this kind of processing.
- **A Cursor is an area in memory where the data required to execute SQL statement.**
- So, a cursor referred as work area.
- So, the **size of the cursor** will be the same as a size to hold this data.
- **Active Data Set:** The data (Set of rows) that is stored in the cursor is called Active Data Set.
- **Result Set:** Data is stored in cursor because of some SQL statement. So, it is called Result Set.
- **Current Row:** The row that is being processed is called the Current Row.

- **Row Pointer:** A pointer that is used to track the current row is known as Row Pointer.
- **Cursor Attributes:** Multiple cursor variables are used to indicate the current status of the processing being done by the cursor. These kinds of variables are known as Cursor Attributes.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

- **There are two types of cursors in PL/SQL:**
  - **Implicit Cursor**
  - **Explicit Cursor**

1) Implicit Cursor:

- A cursor is called an **Implicit Cursor**, if it is opened by **Oracle itself** to execute SQL Statement like **SELECT**, **INSERT**, **UPDATE** or **DELETE**.
- It is opened and managed by Oracle itself. So, user needs not to care about it. o We cannot use implicit cursors for **user defined work**.
- Oracle performs following operation to **manage** an implicit cursor:
  - Reserve an area in memory to store data required to execute SQL statement.
  - Occupy this area with required data.
  - Processes data.
  - Frees memory area by **closes a cursor**, when processing is completed.
- The syntax to use attributes of implicit cursor can be given as:

SQL%AttributeName

- The value of the cursor attribute always refers to the SQL command that was **executed most recently**.
- **Before open** implicit cursor, its attribute contains **NULL** as value.

Example:

```
DECLARE

--      Declare required variables branch
        Account.B_Name%TYPE;

BEGIN

--      read a number from the user
        branch := &branch;

--      display number of record updated if any
        IF  SQL%FOUND  THEN
                dbms_output.put_line(‘ Total ’|| SQL%ROWCOUNT || ‘ records are
                updated.’);
        ELSE
                dbms_output.put_line(‘ Given branch not available.‘);
        END IF ;

END;
/
```

Output 1:

```
                Enter value for branch: ‘surat’
                Given branch not available.
```

Output 2 :

```
                Enter value for branch:’RJT’
                Total 2 records are updated.
```

2) Explicit Cursor:

- A cursor is called **Explicit Cursor**, if it is **opened by user** to process data through PL/SQL block.
- It is opened by user. So, user has to take care about managing it.
- It is **used** when there is a need to process **more than one record individually**.
- Even though the cursor stores multiple records, only one record can be processed at a time, which is called as **current row**.

- Following steps required to manage an explicit cursor:
  Declare a cursor
  Open a cursor
  Fetching data
  Processing data
  Closing cursor

❖ **Declare a Cursor:**

Syntax:

CURSOR cursorName **IS** **SELECT …. ;**

- A cursor with **cursorName** is declared.
- It is mapped to a query given by SELECT statement.
- Here, only cursor will be declared. No any memory is allocated yet.

Example:

CURSOR cursorAcc **IS**
SELECT Acc_No, Balance, B_Name **FROM** Account ;

❖ **Open a Cursor:**

- Once cursor is declared we can open it.
- When cursor is opened following operations are performed:
- Memory is allocated to store the data.
- Execute SELECT statement associated with cursor.
- Create active data set by retrieving data from table.

Syntax:

OPEN cursorName ;

❖ **Fetching Data:**

- We cannot process selected row directly. We have **to fetch column values** of a row into **memory variables**.
- This is done by **FETCH** statement.

Syntax:

FETCH cursorName **INTO** variable1, variable2………. ;

- Retrieve data from the current row in the active data set and stores them in given variables.
- Data from a single row are fetched at a time.
- After fetching data, updates row pointer to point the next row in an active data set.
- Variables should be compatible with the columns specified in the SELECT statement.

Example:

```
        FETCH      cursorAcc      INTO  no, balance, bname ;
```

- Fetched account number, balance and branch name from **current row** in active data set and **store** them in respective variables.
- To process **more than one record**, the **FETCH** statement is enclosed within loop like

  LOOP … END LOOP can be used.

❖ **<u>Processing data</u>:**

- This step involves actual processing of current row by using PL/SQL as well as SQL statements..

❖ **<u>Closing Cursor</u>:**

- A cursor should be closed after the processing of data completes. Once you close the cursor it will release memory allocated for that cursor.
- If user forgets to close the cursor, it will be automatically closed after termination of the program.

Syntax:

```
        CLOSE      cursorName ;
```

- The syntax to use attributes of explicit cursor can be given as:

  SQL%AttributeName

Example:

```
DECLARE
-- declare a cursor
        CURSOR  cursorAcc  IS
        SELECT Acc_No, Balance, B_Name FROM Account ;
--declare required variables
        no      Account.Acc_No%TYPE ;
         balance Account.Balance%TYPE;
        branch  Account.B_Name%TYPE;

BEGIN
        --open a cursor
        OPEN  cursorAcc ;

--if cursor is opened successfully then process data
--Else display error message
IF      cursorAcc%ISOPEN  THEN
LOOP
```

```
--fetch data from cursor row into variavbles
FETCH cursorAcc      INTO  no, balance, branch;

--if no record available in active data set then exit from loop
EXIT   WHEN  cursorAcc%NOTFOUND ;

        --process data. If record belongs to ‘RJT’ branch, transfer it
        IF       branch = ‘RJT’          THEN

        -- insert record into Account_RJT table
        INSERT INTO Account_RJT  VALUES(no, balance);

        --delete record from the Account table
        DELETE FROM  Account  WHERE Acc_No = no ;

        END IF ;
END LOOP;

ELSE
        dbms_output.put_line (‘Cursor cannot be opened.’) ;
END IF ;
END ;
/
```

4.6 Procedures & Functions

- A **procedure** or **function** is a group of PL/SQL statements that performs specific task.
- A procedure and function is a **named PL/SQL block of code**. This block can be compiled and successfully compiled block can be stored in Oracle database. This procedure and function is called **Stores Procedure or Function**.
- We can pass parameters to procedures and functions. So that their execution can be changed dynamically.

❖ **Creating a Procedure**

Syntax:

```
CREATE [OR REPLACE] PROCEDURE proc_name (argument [IN, OUT, IN OUT]
datatype)
IS
        Declaration section
BEGIN
        Execution section
EXCEPTION
        Exception section
END ;
```

❖ Executing a Procedure

There are two ways to execute a procedure:

Syntax:

```
        EXECUTE    [or EXEC]    procedure_name        (parameter) ;
```

Example:

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

❖ Creating a Function

Syntax:

```
    CREATE [OR REPLACE] FUNCTION func_name (argument IN dataType…)
    RETURN dataType

    IS
        Declaration section
    BEGIN
        Execution section
    EXCEPTION
        Exception section
    END ;
```

Example:

```
        Create or replace function adder(n1 in number, n2 in number)
        Return number

        Is
            n3 number(8);
        Begin
            n3 :=n1+n2;
            Return n3;
        End;
        /
```

❖ Executing a Function

Syntax:

```
        SELECT  function_name  (parameter)  FROM  dual ;
```

4.7 Fundamentals of Database Triggers

- A **trigger** is a group or set of SQL and PL/SQL statements that are executed by **Oracle itself**.
- The main characteristic of the trigger is that it is **fired automatically** when DML statements like Insert, Delete, and Update is executed on a table.
- **The advantages of triggers are as given below:**
    - To prevent **misuse** of database.
    - To implement **automatic backup** of the database.
    - To implement **business rule constraints**, such as balance should not be **negative**.
      o Based on change in one table, we want to update other table.
    - To **track the operation** being performed on specific tables with details like operation, time when it is performed, user name who performed it, etc..

❖ **<u>:NEW and :OLD Clause</u>**

- In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.
- Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.
- **:NEW –** It holds a new value for the columns of the base table/view during the trigger execution
- **:OLD –** It holds old value of the columns of the base table/view during the trigger execution

4.8 Creating Triggers

Syntax:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER }
{INSERT [OR] | UPDATE [OR] | DELETE}
ON table_name

FOR EACH ROW]

WHEN (condition)

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

Example:

```
CREATE OR REPLACE TRIGGER age_changes
BEFORE DELETE OR INSERT OR UPDATE ON student

FOR EACH ROW

WHEN (NEW.CODE > 0)

DECLARE
   age_diff number;

BEGIN
   age_diff := :NEW.age  - :OLD.age;
   dbms_output.put_line ('Prevoius age: ' || : OLD.age);
   dbms_output.put_line ('Current age: ' || : NEW.age);
   dbms_output.put_line ('Age difference: ' || age_diff);

END;
/
```

4.9 Types of Triggers: Before, after for each row, for each statement

Triggers can be classified based on the following parameters.

- Classification based on the **timing**

  - **BEFORE Trigger:** It fires before the specified event has occurred.
  - **AFTER Trigger:** It fires after the specified event has occurred.

- Classification based on the **level**

  - **STATEMENT level Trigger**: It fires one time for the specified event statement.
  - **ROW level Trigger**: It fires for each record that got affected in the specified event. (only for DML)