

Unit – 5

PL/SQL Objects:

Prepared By: Prof. Meghna K. Bhatt

Topic to be discuss

- ❑ Fundamentals of PL/SQL
- ❑ PL/SQL block structure
- ❑ Stored Functions and Procedures
- ❑ Triggers

WHAT IS PL/SQL?

- ❑ PL/SQL stands for Procedural Language extensions to the Structure
- ❑ 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.
- ❑ Oracle uses a PL/SQL engine to processes the PL/SQL statements.

Advantages of PL/SQL

- 1) **Procedural Capabilities**
- 2) **Support to variables**
- 3) **Error Handling**
- 4) **User Defined Functions**
- 5) **Portability**
- 6) **Sharing of Code**
- 7) **Efficient Execution**

Advantages of PL/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**.
- ❑ **Execution of program can be controlled** instead of abruptly terminating the program.

Advantages of PL/SQL

4) User Defined Functions

- ❑ 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:

- ❑ Allows user to **store compiled code** in database.
- ❑ PL/SQL code can be **accessed and shared by different applications.**
- ❑ PL/SQL code can be **executed by other programming language like JAVA.**

Advantages of PL/SQL

7) Efficient Execution

- ❑ PL/SQL sends an entire block of SQL statements to the Oracle engine, where these statements are executed in one go.
- ❑ Reduces **network traffic and improves efficiency of execution**.
- ❑ In SQL, all statements are transferred one by one.

SQL VS PL/SQL

| SQL | PL/SQL |
|-----------------------------------------------------------------|-------------------------------------------------------------------------------------|
| SQL stands for structured query language | PL/SQL stands procedural language/ structure query language. |
| Can not use variables, constants | Use variables & constants |
| Allow you to use built in functions only | Allow you to use built in functions as well as create your own user define function |
| SQL commands can not be sharable | You can share PL/SQL code |
| In SQL we can not use conditional statement & looping statement | In PL/SQL we can use conditional statement & looping statement |
| We can use commands to solve queries | We can write a program/group of statements |
| SQL is syntax/statement oriented language | PL/SQL is structure oriented language |

SQL VS PL/SQL

| SQL | PL/SQL |
|-------------------------------------------|---------------------------------------------|
| In SQL you can not create or use package | Create a package |
| There is no facility to handle exceptions | Allows you to write exception handling code |
| There is no facility of trigger. | You can create trigger. |
| SQL support DML & DDL compiler | PL/SQL is compiled with PL/SQL compiler |

Generic PL/SQL Block

DECLARE

Declaration of variables, constants

BEGIN

PL/SQL Executable statements

EXCEPTION

PL/SQL Exception Handler Code

END;

Generic PL/SQL Block

1) Declarations (**Optional**)

- ❑ This section starts with the keyword 'DECLARE'.
- ❑ It defines and initializes **variables and cursors used in the block**.

2) Executable Part (**Mandatory**)

- ❑ This section starts with the keyword 'BEGIN'.
- ❑ It contains various SQL and PL/SQL statements providing functionalities like **data retrieval, manipulation, looping and branching**.

3) Exception Handling (**Optional**)

- ❑ This section starts with the keyword 'EXCEPTION'.
- ❑ It **handles errors** that arise during the execution of data manipulation statements in 'executable commands' section.
- ❑ This section specifies that **end** of PL / SQL block.

The PL/SQL Literals

- ❑ A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier.
- ❑ For example, TRUE, 786, NULL, 'marwadi university' are all literals of type Boolean, number, or string.
- ❑ PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals –
 - ❑ Numeric Literals
 - ❑ Character Literals
 - ❑ String Literals
 - ❑ BOOLEAN Literals
 - ❑ Date and Time Literals

The PL/SQL Literals

- Numeric Literals:

| | | | | | |
|--------|------|-------|---------|----------|----|
| 25 | 6.34 | 7g2 | .1 | +17 | -5 |
| 050 | 78 | -14 | 0 | +32767 | |
| 6.6667 | 0.0 | -12.0 | 3.14159 | +7800.00 | |

- Character Literals

| | | | | |
|-----|-----|-----|-----|-----|
| 'A' | '%' | '*' | 'z' | '(' |
|-----|-----|-----|-----|-----|

The PL/SQL Literals

☐ String Literals

☐ 'Hello, world!'

☐ 'Marwadi Univerisity'

☐ '19-NOV-12'

☐ BOOLEAN Literals

☐ TRUE, FALSE, and NULL.

☐ Date and Time Literals

☐ DATE '25-DEC-78';

☐ TIMESTAMP '2012-10-29 12:01:01';

Data Types

- ❑ PL/SQL is **super set** of the SQL.
- ❑ So, it supports all the data types provided by SQL.
- ❑ In PL/SQL Oracle provides **subtypes of the data types**.
- ❑ **For example**, the data type NUMBER has a subtype called INTEGER.

Data Types

| 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 |

PL/SQL CONSTANT

- ❑ A constant holds a value that once declared, does not change in the program.
- ❑ A constant declaration specifies its name, data type, and value, and allocates storage for it.
- ❑ The declaration can also impose the **not null constraint**.
- ❑ A constant is declared using the **constant** keyword.
- ❑ Syntax:
`<constant name> CONSTANT <datatype>:=value;`
- ❑ E.G.
`PI CONSTANT number:=3.14;`

Variables

- Variables are used to store values that may be change during the execution of program.
- In PL/SQL, variables contain values resulting from **queries or expression**.
- Variables are declared in **Declaration section** of the PL/SQL block.
- It can assign valid data type and can also be initialized if necessary.

Variables

Declare a Variable

Syntax:

variableName **datatype** [NOT NULL] := initialValue;

- ❑ **datatype** can be any valid data type supported by PL/SQL.
- ❑ ‘:=’ used as **assignment operator**.
- ❑ If variable need to be initialized then **initialValue** can be assigned at declaration time.
- ❑ If **NOT NULL** is included in declaration, variable cannot have NULL value during program execution and such variable must be initialized.

Variables

Example:

```
no    NUMBER(3);  
value DECIMAL;  
city  CHAR(10);  
name  VARCHAR(10);  
counter NUMBER(2) NOTNULL := 0
```

Anchored Data Type Variable

- A variable can be declared as **anchored data type**.
- It means, **datatype** for variable is **determined based on the data type of the other object**.
- This object can be other **variable** or a **column of the table**.
- This provides ability to match the **data types of the variables** with the **data types of the columns** defined in the database.
- If data type of column is changed, then the data type of variable will also **changed automatically**.

Advantage: It **reduces maintenance cost** and allow a program to adapt changes made in tables.

Anchored Data Type Variables

Syntax:

variableName **object%TYPE** [NOTNULL] := initialValue ;

- ❑ **object** can be any variable declared previously or column of a database.
- ❑ To refer of a column of particular table, column name must be combined with table name.

Example

```
no    Account.Acc_No%TYPE := 101;
bal    Account.Balance%TYPE;
name    Customer.name%TYPE;
```

Assign Value to Variable

1) By using Assignment Operator

Syntax:

`variableName := value ;`

- A value can be a constant value or result of some expression or return value of some function.

2) By Reading from the Keyword:

Syntax:

`variableName := &variableName ;`

- This is similar to `scanf()` function.
- Whenever **'&'** is encountered, a value read from the keyboard and assign it to variable.

`no := &no;`

Assign Value to Variable

3) Selecting or Fetching table data values into Variables:

Syntax:

```
SELECT col1, col2, ..., colN INTO var1, var2, ..., varN  
FROM tableName WHERE condition ;
```

- This statement retrieves values for specified column and stores them in given variable.
- Data type and size of variables must be compatible with the relative columns.
- A condition in WHERE clause must be such that it selects only single record.
- This statement cannot work if multiple records are selected.

Assign Value to Variable

3) Selecting or Fetching table data values into Variables:

Example: Write PL/SQL block stores account number and balance for account 'A01' into variable 'no' and 'bal'.

Input:

```
DECLARE
    no    Account.Acc_No%TYPE;
    bal   Account.Balance%TYPE;
BEGIN
    SELECT  Acc_No, Balance  INTO  no, bal  FROM
Account      WHERE  Acc_No = 'A01' ;
END;
/
```

Display Messages

- ❑ Syntax:

- ❑ `dbms_output.put_line (message);`

- ❑ A `dbms_output` is a **package**, which provides functions to accumulate information in a buffer.

- ❑ A `put_line` is a **function**, which display messages on the screen.

- ❑ A message is a **character string to be displayed**.

- ❑ To display data of other data type, they must be **concatenated with some character string**.

- ❑ The environment parameter, `SERVEROUTPUT` must be **ON** to display messages on screen.

- ❑ `SET SERVEROUTPUT ON;`

Display Messages

Example:

```
dbms_output.put_line ( 'Hi Hello World...' );
```

Hi Hello World...

Concatenation Operator

```
dbms_output.put_line ( 'Sum =' || 25 );
```

Sum = 25

```
dbms_output.put_line ( 'Square of ' || 3 || ' is ' || 9 );
```

Square of 3 is 9

Comments

- ❑ Comments are statement that will **not get executed** even though they are present in the program code.
- ❑ Comments are used to **increase readability** of a program.
- ❑ There are two types of comments:
 - 1) **-- (Double hyphen or double dash) (Single Line Comment):**
 - Treats single line as a comment.

 -- This single line is a comment
 - 2) **/* ... */ (Multiple Line Comment):**
 - Treats multiple lines as comment.

 /* This statement is spread over two line and both lines are treated as comments */

Control Structures

❑ There are 3 types of Control Structures in PL/SQL:

- 1) Conditional Control
- 2) Iterative Control
- 3) Sequential Control

Control Structures - Conditional Control

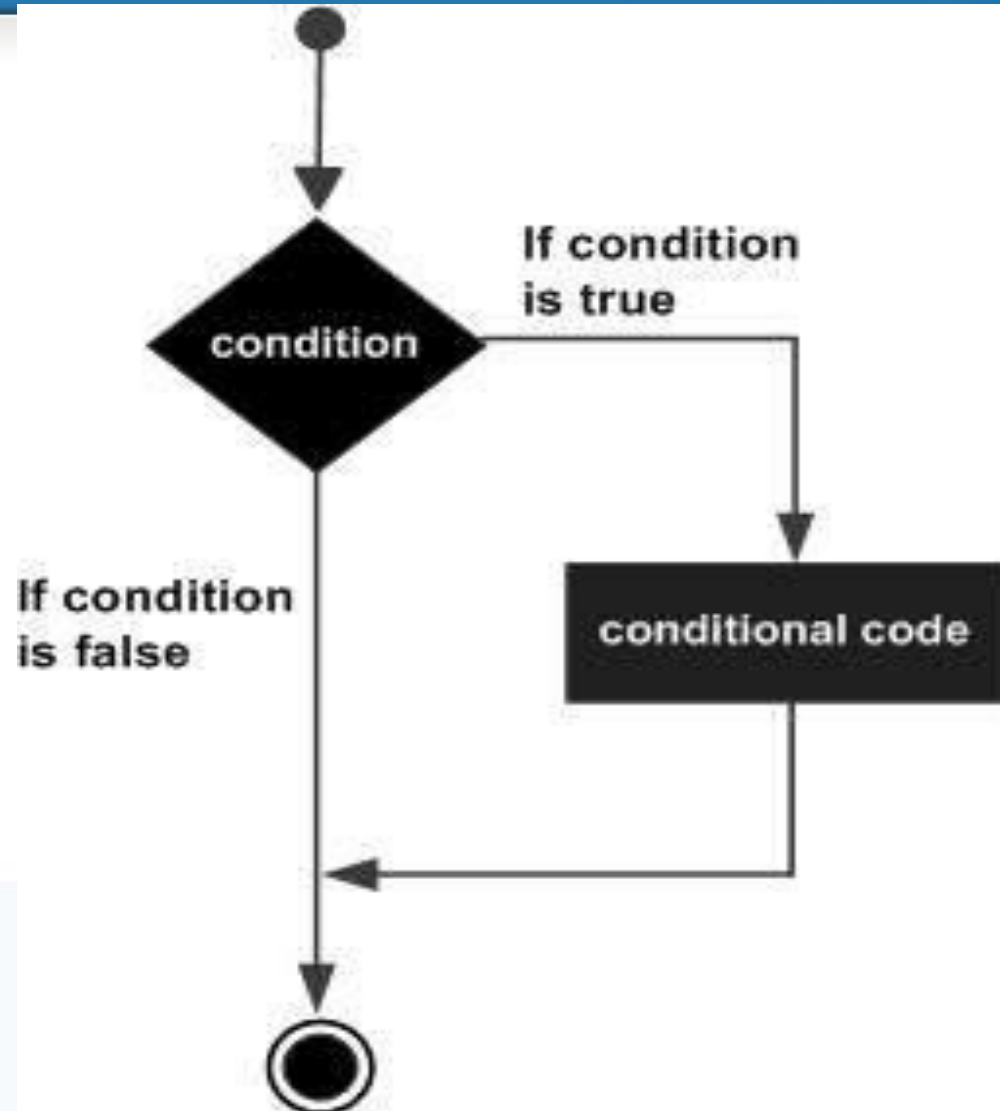
- ❑ To control the **execution of block of code** based on some condition, PL/SQL provides the **IF statement**.
- ❑ Three form of conditional controls
 - ❑ Simple if
 - ❑ Else if
 - ❑ Ifthen....elseifelse

Conditional Control

□ Simple if

Syntax

If <condition> Then
 <statements>
End if;



Conditional Control

□ If ...else

Syntax

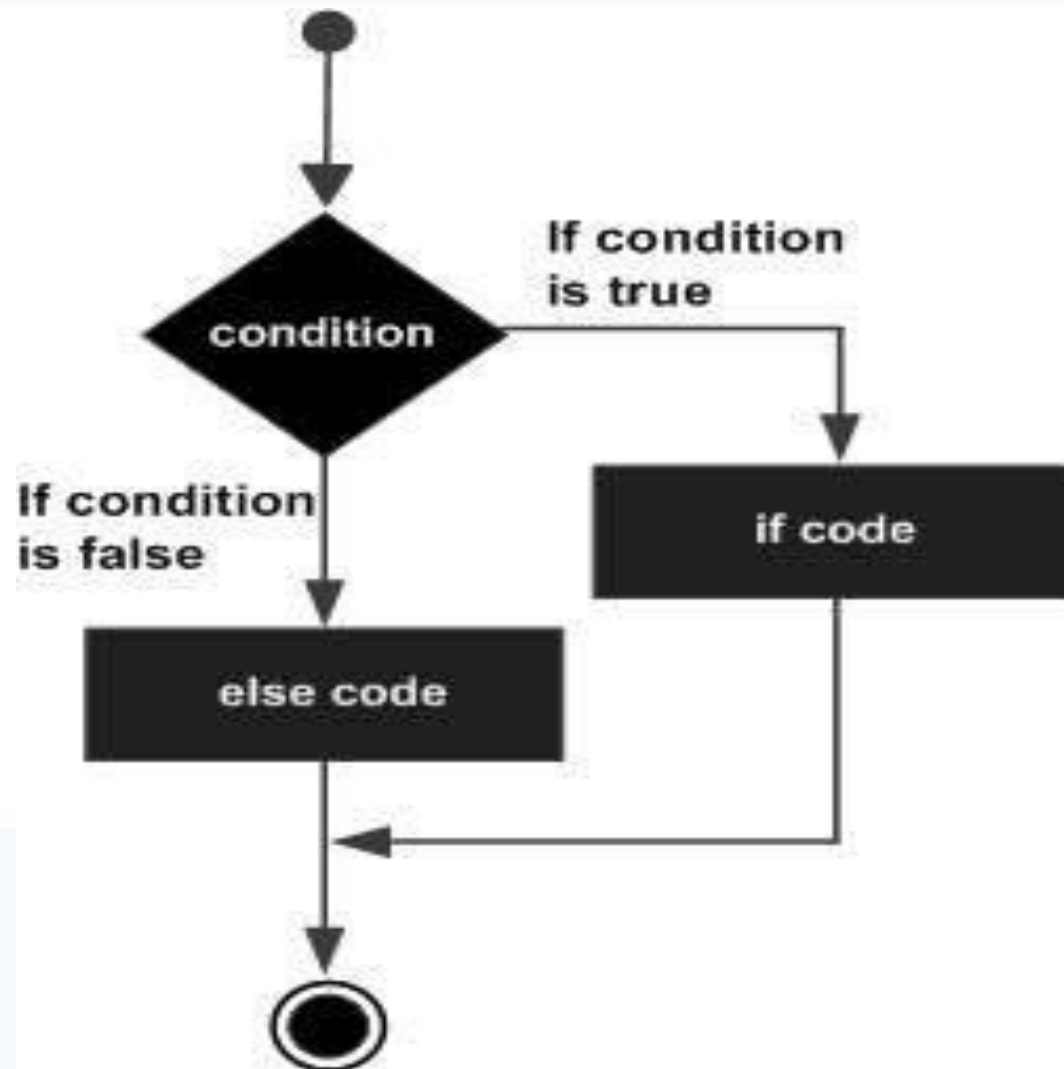
If <condition> Then

 <statement 1>

Else

 <Stement2>

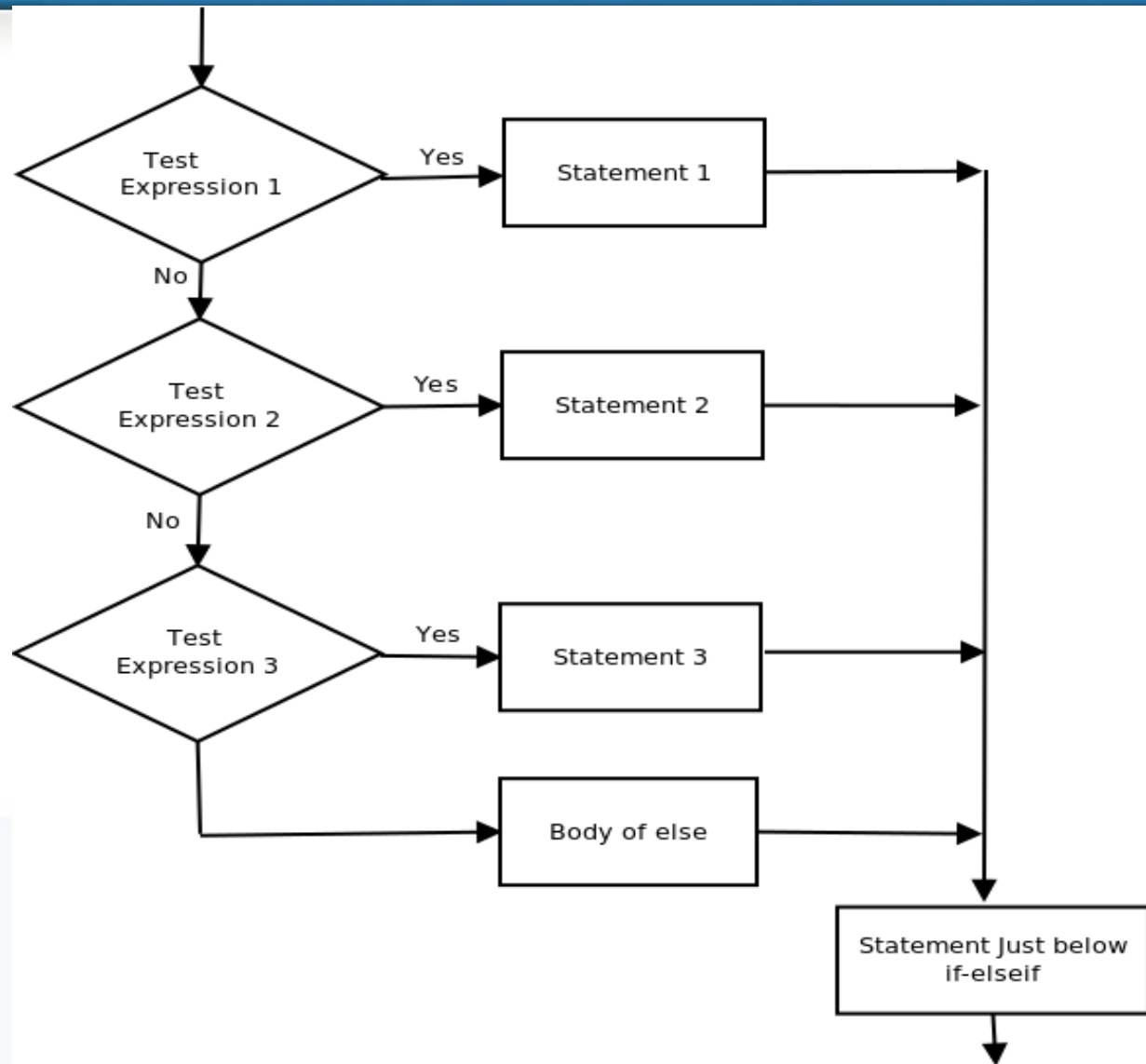
End if;



Conditional Control

❑ If ...then...elseif

```
If <Condition> then  
    <Statement1>  
Elsif <condition2>  
then  
    <statment2>  
Elsif <condition3>  
Then  
    <Stement3>  
Else  
    ...  
End if;
```



Control Structures- 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.

Iterative control- Loop

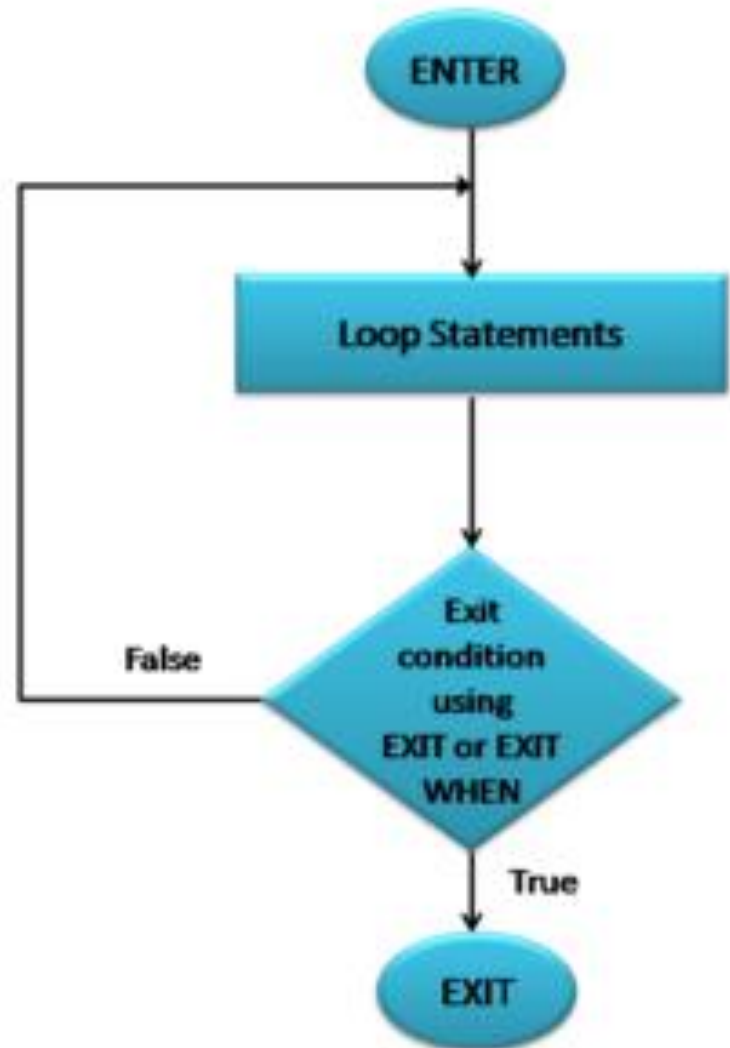
Syntax:

Loop

<Statements>;

[Exit When <Condition>];

End Loop;

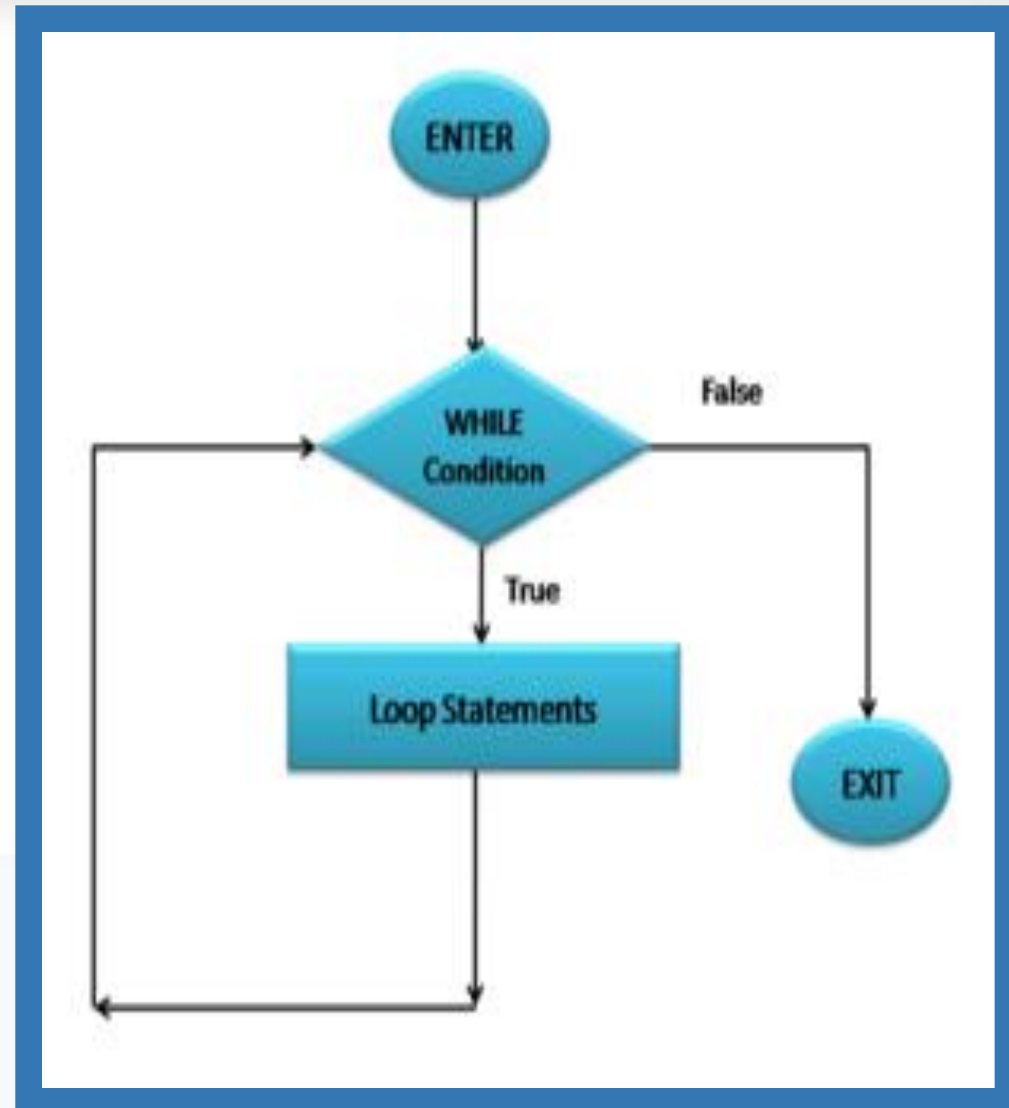


Iterative control- While

Syntax:

While <condition>
Loop

 <Statements>;
End Loop;



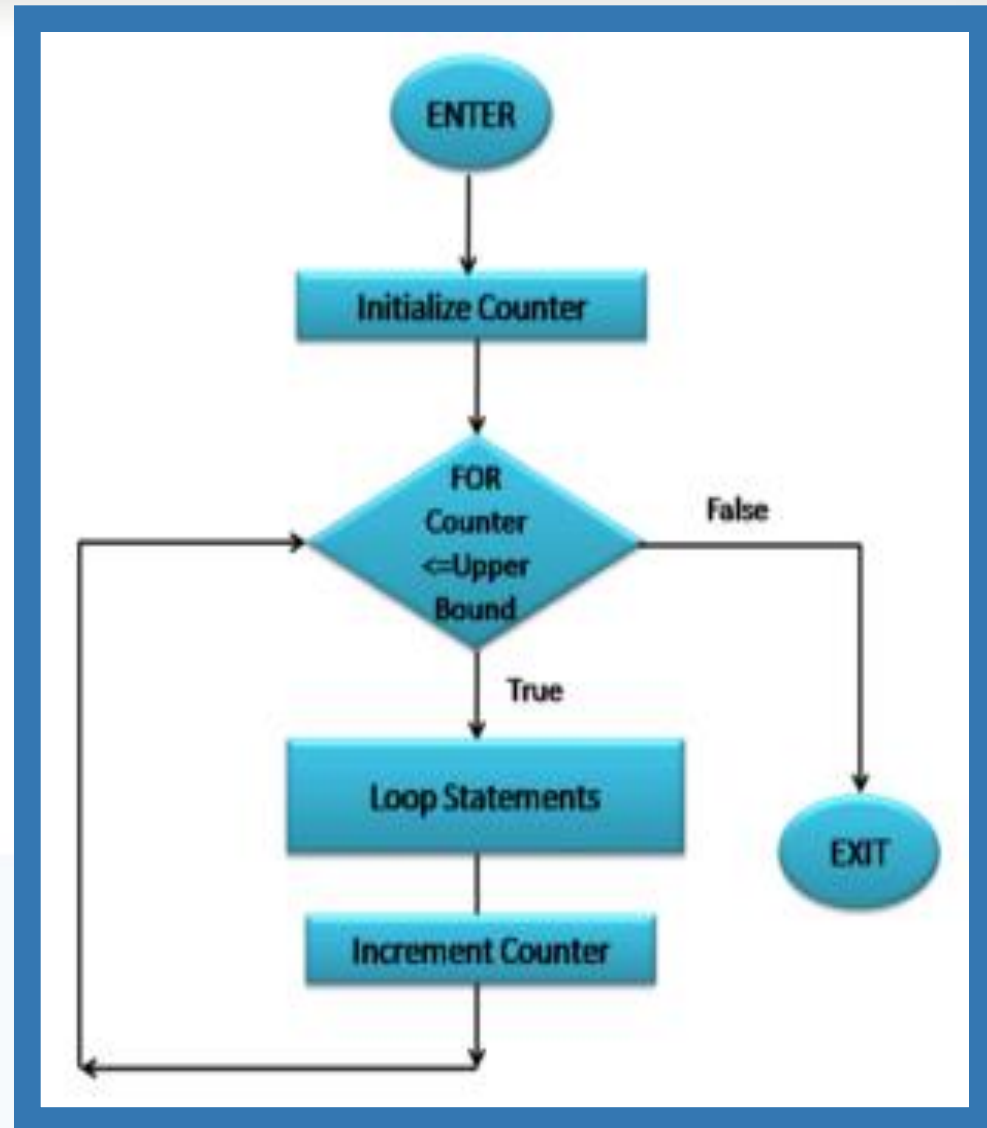
Iterative control- for

Syntax:

For Variable In [Reverse]
Start..End
Loop

<Statement>

End Loop;



Control Structures- 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.

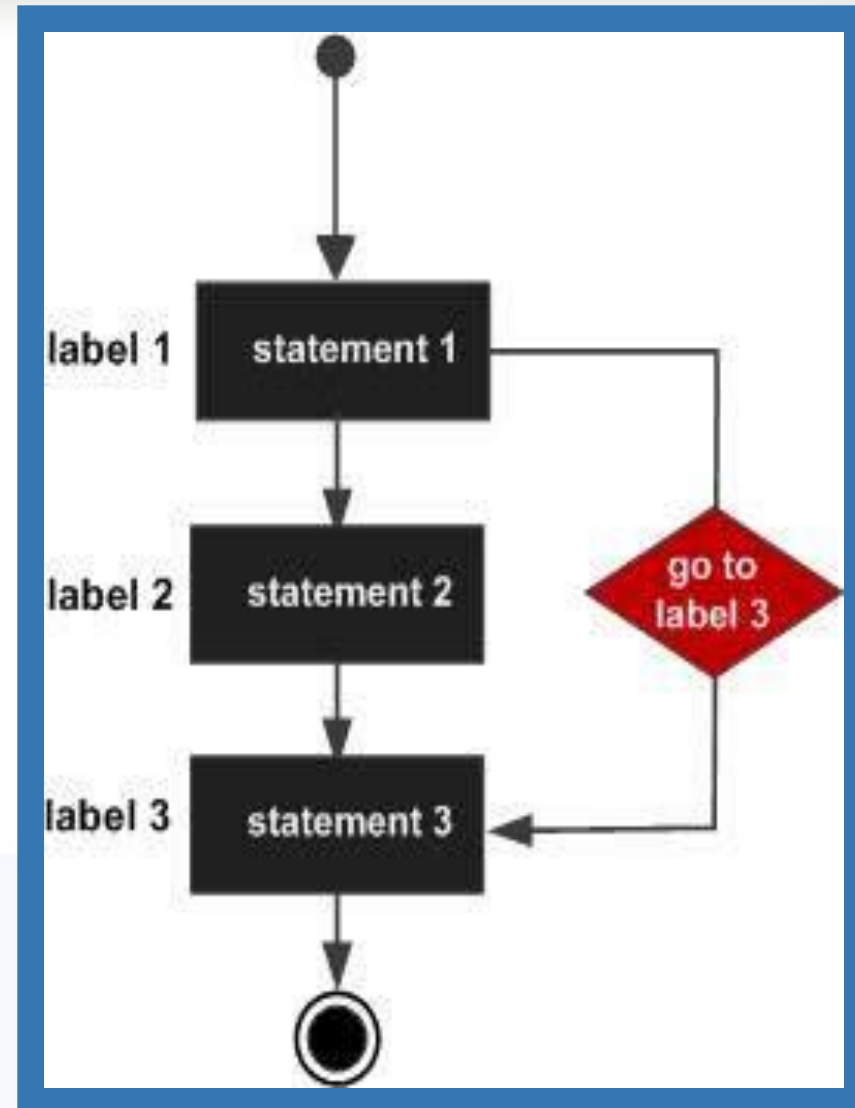
Syntax:

```
GOTO    jumpHere ;  
      :  
      :  
      << jumpHere >>
```

- ❑ The **GOTO** statement makes flow of execution to jump at **<< jumpHere >>**.

Control Structures- sequential control

```
BEGIN  
GOTO second_message; <<first_message>>  
    DBMS_OUTPUT.PUT_LINE( 'Hello' );  
GOTO the_end;  
<<second_message>>  
    DBMS_OUTPUT.PUT_LINE( 'PL/SQL GOTO  
Demo' );  
GOTO first_message;  
<<the_end>>  
    DBMS_OUTPUT.PUT_LINE( 'and good bye...' );  
END;
```

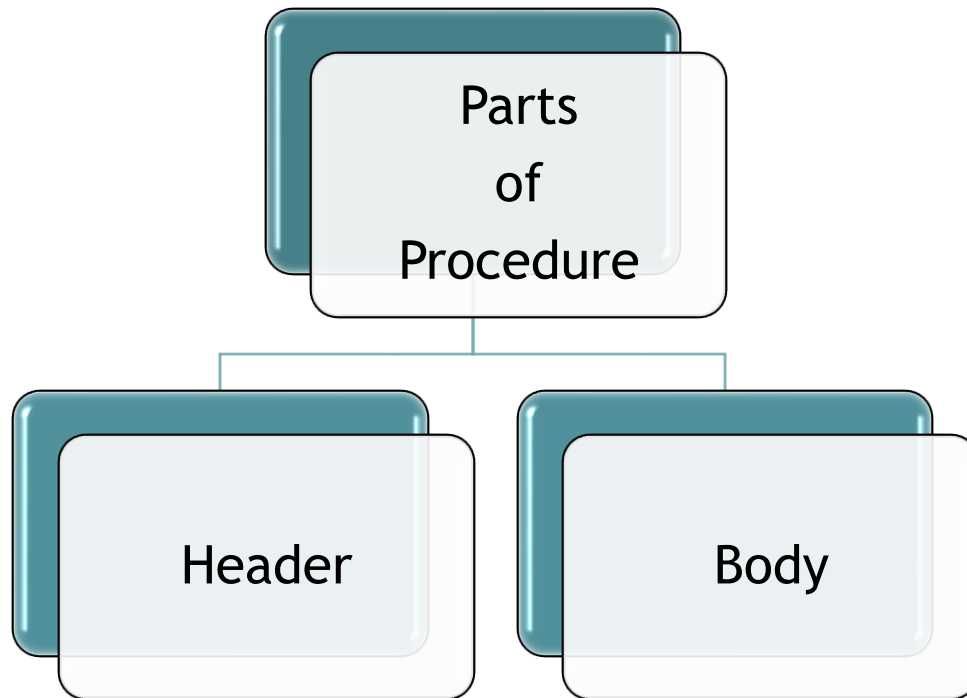


Stored procedure & functions

What is procedure?

- A procedure is a group of PL/SQL statements that you can call by name.
- A procedure is a subprogram that can take parameters and be called.
- A procedure is a module performing one or more actions , it does not need to return any values
- The user must call a procedure either from a program or manually.
- Generally, you use a procedure to perform an action.

What is procedure?



The header contains the name of the procedure and the parameters or variables passed to the procedure.

The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

Procedure Syntax:

```
CREATE [OR REPLACE] PROCEDURE <PROCEDURE_NAME>
    [Parameter1 {IN, OUT, IN OUT}, [Parameter2 {IN, OUT,
    IN OUT},...]]
    {IS/AS}
        [CONSTANT / VARIABLE Declaration;]
    BEGIN
        Executable statements/ <procedure body>;
    [EXCEPTION
        Exception handling statements;]
    END < PROCEDURE_NAME>;
```

Procedure Syntax:

| | |
|----------------------------------------|----------------------------------------------------------------|
| [OR REPLACE] | Allows the modification of an existing procedure |
| <PROCEDURE_NAME> | Specifies the name of the procedure. |
| Parameter | Parameter list contains name, mode and types of the parameters |
| AS/IS | For creating a standalone procedure |
| <procedure body> | Contains the executable part. |
| EXCEPTION | Contains exception handling statements |
| END <PROCEDURE_NAME>; | End of the procedure. |

Types of parameters

- The parameter is variable of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code.
- There are 3 types of parameters:
 - IN Parameter
 - OUT Parameter
 - IN OUT Parameter

IN Parameter

- ❑ The parameters are of IN type.
- ❑ It is used for giving input to the subprograms.
- ❑ This value is read only type of value. It cannot be change.

OUT Parameter

- ❑ This parameter is used for getting output from the subprograms.
- ❑ It is a read-write variable inside the subprograms.
- ❑ Their values can be changed inside the subprograms.

IN OUT Parameter

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms.
- Their values can be changed inside the subprograms.

Difference between IN, OUT , IN OUT

| IN | OUT | IN OUT |
|----------------------------------------------------------------------------------|------------------------------------|---------------------------------------------------------|
| Default mode | Must be specified | Must be specified |
| Value is passed into subprogram | Returned to calling environment | Passed into subprogram; returned to calling environment |
| Formal parameter acts as a constant | Uninitialized variable | Initialized variable |
| Actual parameter can be a literal, expression, constant, or initialized variable | Must be a variable | Must be a variable |
| Can be assigned a default value | Cannot be assigned a default value | Cannot be assigned a default value |

HOW TO EXECUTE OR RUN A PROCEDURE ?

SQL > @ <Procedure-file name>.SQL;
Procedure is successfully created...

OR

Procedure is created with compilation error...

SQL > SHOW ERRORS;

OR

SQL > SELECT * FROM USER_ERRORS;

SQL > EXECUTE <Procedure-name> (Parameter Value);

SQL > SELECT * FROM <Table-name>;

NOTE: - If your user doesn't have EXECUTE permission then write the following command.

SQL>GRANT EXECUTE ON <PROCEDURE-NAME> TO <USER-NAME>;

Deleting a Procedure

- Use the DROP PROCEDURE statement to remove a standalone stored procedure from the database.
- Syntax:

```
Drop Procedure <procedure_name>;
```

Example: 1

-- The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed. (pe1.sql)

```
CREATE OR REPLACE PROCEDURE greetings
AS

BEGIN
    dbms_output.put_line('Hello World!');
END;
/
```

```
SQL>execute greetings;
```

Function

What is function?

- ❑ Function - User Define Functions is similar to stored procedure.
- ❑ The PL/SQL function is a named PL/SQL block which performs one or more specific tasks and must returns a value.
- ❑ It is similar to procedure only the difference between procedure & function is function returns a value.
- ❑ **Functions** can accept one, many, or no parameters, but a function must have a **return clause** in the executable section of the function

Procedure Syntax:

```
CREATE [OR REPLACE] FUNCTION < <FUNCTION_NAME >  
[Parameter1 {IN, OUT, IN OUT}, [Parameter2 {IN, OUT,  
IN OUT},...]]
```

```
RETURN <RETURN_DATA TYPE>
```

```
{IS/AS}
```

```
[CONSTANT / VARIABLE Declaration;]
```

```
BEGIN
```

```
Executable statements/ <procedure body>;
```

```
RETURN <return_value>;
```

```
[EXCEPTION
```

```
Exception handling statements;]
```

```
END <FUNCTION_NAME>;
```

Function Syntax:

| | |
|-----------------------------------|----------------------------------------------------------------|
| [OR REPLACE] | Allows the modification of an existing procedure |
| <FUNCTION_NAME> | Specifies the name of the function |
| Parameter | Parameter list contains name, mode and types of the parameters |
| RETURN return_Datatype | Specifies the data type that we are going to return. |
| AS/IS | For creating a standalone FUNCTION |
| <function body> | Contains the executable part. |
| EXCEPTION | Contains exception handling statements |
| END <FUNCTION_NAME>; | End of the function. |

HOW TO EXECUTE OR RUN A FUNCTION ?

- ❑ SQL > @ <function-filename>.SQL;

- ❑ SQL > SHOW ERRORS;

OR

- ❑ SQL > SELECT * FROM USER_ERRORS;

- ❑ SQL > VARIABLE <VARIABLE_NAME> datatype;

- ❑ SQL> EXECUTE :<VARIABLE_NAME>: =<function-name>(Parameter Value);

- ❑ SQL > PRINT <VARIABLE_NAME>;

OR

- ❑ Write a PL/SQL block that calls the function

- ❑ **NOTE:** - If your user doesn't have EXECUTE permission then write

- ❑ the following command.

- ❑ SQL> GRANT EXECUTE ON <FUNCTION-NAME> TO <USER-NAME>;

Example

-- Write a function that returns the total number of EMPLOYEES in the emp table. (f1.sql):

```
CREATE OR REPLACE FUNCTION total_emp  
RETURN number  
IS  
    total number(2) := 0;  
BEGIN  
    SELECT count(*) into total FROM emp;  
    RETURN total;  
END;  
/
```

How to call a function from command line ?

```
SQL> @ F:/PLSQL/f1.sql;
```

```
SQL> var mno number;
```

```
SQL> execute :mno:=total_emp;
```

PL/SQL procedure successfully completed.

```
SQL> print mno;
```

MNO

14

Difference between procedure & function

| Procedure | Function |
|----------------------------------------------------|---------------------------------------------------------|
| A procedure used to perform certain task in order. | A function used to calculate result using given inputs. |
| A procedure cannot be called by a function | A function can called by a procedure. |
| Uses IN, OUT, IN OUT parameter. | Uses only IN parameter. |
| Returns a value using “ OUT” parameter. | Returns a value using “RETURN”. |
| DML statements can executed within a procedure. | DML statements cannot executed within a function. |
| Procedures always executes as PL SQL statement | Functions executes as part of expression |
| Procedures always executes as PL SQL statement | Functions executes as part of expression |

Difference between procedure & function

| Procedure | Function |
|-----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| It does not contain return clause in header section | It must contain return clause in header |
| Does not specify the datatype of the value if it is going to return after a calling made to it. | Necessarily specifies the datatype of the value which it is going to return after a calling made to it. |
| A procedure need not deal with expressions. | A function must deal with expressions. |
| Procedures will not return the value | Functions must return the value. |
| Procedures need to be compiled once, and if necessary, we can call them repeatedly without compiling them every time. | Functions are compiled whenever they are called. |

Trigger

What is Trigger ?

- *Trigger* is a *series of PL/SQL statements* attached to a database table that execute whenever a *triggering event (select, update, insert, delete)* occurs.
- The *triggers* are *standalone procedures* that are *fired implicitly (internally)* by the oracle.
- Unlike *stored procedures* and *functions*, they *not explicitly called*, but they are activated (triggered) when a *triggering event* occurs.

Syntax for Trigger

Syntax to create a trigger:

```
CREATE OR REPLACE TRIGGER <TRIGGER_NAME>
{BEFORE/AFTER}
{INSERT/UPDATE/DELETE} [OF COLUMN] ON
    <TABLE_NAME>
[FOR EA CH ROW]
[WHEN CONDITION]
[PL/SQL BLOCK]
```

Syntax to drop a trigger:

```
DROP TRIGGER <TRIGGER_NAME>;
```


Syntax for trigger

| | |
|------------------------------------------|----------------------------------------------------------------------------------|
| CREATE OR REPLACE TRIGGER | Creates or replaces an existing trigger with the trigger_name. |
| BEFORE/AFTER | Specifies when the trigger will be executed |
| INSERT/UPDATE/ DELETE | Specifies the DML operation. |
| ON <TABLE_NAME > | Specifies the name of the table associated with the trigger |
| FOR EACH ROW | This specifies a row-level trigger |
| PL/SQL BLOCK | PL/SQL block that provides the operation to be performed as the trigger is fired |

HOW TO APPLY THE DATABASE TRIGGERS?

A *trigger* has *three* basic parts. The following are the three basic parts:

1. Triggering Event or Statement:

2. Trigger Restriction:

3. Trigger Action:

HOW TO APPLY THE DATABASE TRIGGERS?

1. Triggering Event or Statement:

It is a SQL statement that causes a trigger to be fired. It can be INSERT, UPDATE or DELETE statement for a specific table.

2. Trigger Restriction:

A trigger restriction specifies a Boolean Expression that must be TRUE for the trigger to fire. It is an option available for the triggers that are fired for each row. A trigger restriction is specified using a WHEN clause.

3. Trigger Action:

A trigger action is the PL/SQL code to be executed when triggering statement is encountered and the value of trigger restriction is TRUE. The PL/SQL block contains SQL and PL/SQL statements.

:OLD and :NEW Values

- When a DML statement changes a column the old and new values are visible to the executing code.
- This is done by prefixing the table column with :old or :new
- :new is useful for INSERT and UPDATE
- :old is useful for DELETE and UPDATE

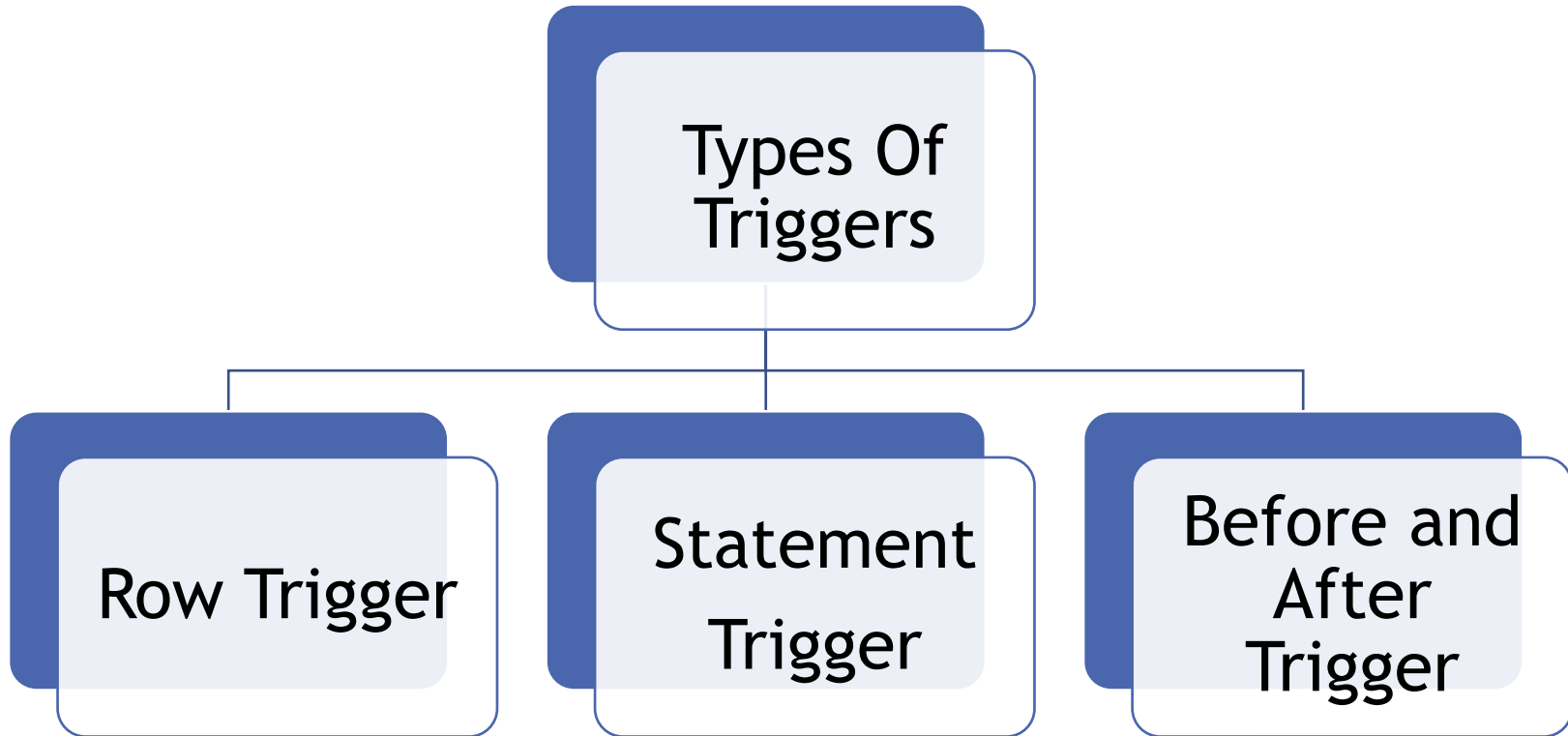
:OLD and :NEW Values

- The OLD and NEW qualifiers are used to reference the values of a column before and after the data change, respectively.
- The OLD and NEW qualifiers can be used only with row triggers.
- They cannot be used with statement triggers.
- The OLD and NEW qualifiers must be prefixed with a colon (:) in every SQL and PL/SQL statement except when they are referenced in a WHEN restricting clause.

Advantages of trigger

- ☐ 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

Types of Triggers




Row Trigger

- A *Row Trigger* is fired *each time* a row in the table is *affected* by the triggering statement.
- For example, if the UPDATE statement updates multiple rows of a table a Row Trigger is fired once for each row affected by the UPDATE statement.
- If the triggering statement affects no rows, the trigger is not executed at all.

Row Trigger

```
CREATE OR REPLACE TRIGGER trig_test
AFTER UPDATE OF SNUM ON PERSONNEL
FOR EACH ROW
BEGIN
    null;                -- write operations here
END;
```



A Row trigger fires once for every row affected by the DML operation.

Statement Trigger

- A *Statement Trigger* is *fired once* on behalf of the triggering statements, independent of the number of rows the triggering statement affects (even if no rows are affected).

Statement Trigger

Inserting, Deleting and Updating can be used to find out which event is occurring

```
CREATE OR REPLACE TRIGGER trig_testTable
AFTER INSERT or UPDATE ON Personnel
BEGIN
  If Inserting Then
    INSERT into testTable values ('insert done', SYSDATE) ;
  Else
    INSERT into testTable values ('update done', SYSDATE) ;
  End If;
END;
```

Test_trigger_1 tests this trigger

Before and After Trigger

- While defining a trigger it is necessary to specify the *trigger timing* that we must have to specify when the *triggering action* is to be executed in relation to the triggering statement.
- **BEFORE** and **AFTER** apply to both row and the statement triggers.

3. Before and After Trigger

- The ***BEFORE triggers*** execute the trigger action before the triggering statement is executed.
- The ***AFTER trigger*** executes the triggering action after the execution of triggering statement.

GUIDELINES FOR CREATING A TRIGGER:

1. There can be *only one trigger* of a particular type that is for *UPDATE*, for *INSERT*, or for *DELETE*.
2. Only *one table* can be specified in the *triggering statement*.
3. The triggers **cannot include COMMIT, ROLLBACK and SAVEPOINT statements.**
4. Inside the trigger the correlation name *:NEW and :OLD* can be made use of to refer to data on the command line and data in the table respectively.

WHEN Clause

- WHEN is *optional additional statement* to control the trigger.
- Takes a **BOOLEAN** SQL expression
 - Trigger fires if **TRUE** and not if **FALSE**
- Operates on a **ROW level trigger**
- To prevent the trigger from firing in specific row cases we use **WHEN (expression)**

WHEN Example

```
create or replace trigger only_nulls
after update on BOOK
for each row
when (old.price is null) -- notice the colon with OLD is not used here
begin
    insert into PriceChange values(:old.isbn,:new.price);
end;
```


HOW TO EXECUTE A TRIGGER ?

```
SQL> START TR_EMPDELETE.SQL; OR @TR_EMPDELETE.SQL;
```

Trigger is created.....

```
SQL> DELETE FROM EMP WHERE EMPNO = 100;
```

```
SQL> SELECT * FROM NEWEMP;
```

If trigger is created with compilation error then execute the following command:

```
SQL> SHOW ERRORS;
```

To view the list of triggers created by user follow following steps:

```
SQL> DESCRIBE USER_TRIGGERS;
```

```
SQL> SELECT TRIGGER_NAME, TRIGGER_BODY FROM USER_TRIGGERS
```

```
WHERE TRIGGER_NAME = 'TR_EMPDELETE';
```

How to Remove / Drop a Trigger ?

- SYNTAX FOR DROPPING A TRIGGER:

DROP TRIGGER <TRIGGER_NAME>;

- SYNTAX FOR DROPPING A TRIGGER:

DROP TRIGGER TR_EMPUPDATE;

Note: If conflicting triggers are created on the same database object then unwanted triggers need to be removed otherwise it will cause other trigger not to execute in a desired manner.

Difference between Trigger and Procedure

| Trigger | Procedure |
|-----------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| Trigger don't accept parameters | Procedure can accept parameters |
| The oracle engine executes a trigger implicitly (automatically fired) | Procedure it must explicitly called by the users. |
| Trigger fired automatically so we need not to use any command. | Execute command used to execute a procedure. |
| The syntax that defines a trigger inside a database is: CREATE TRIGGER TRIGGER_NAME | The syntax that defines a procedure inside a database is: CREATE PROCEDURE PROCEDURE_NAME |
| Triggers cannot be called inside a procedure. | However, you can call a procedure inside a trigger. |