Unit – 4
PL/SQL AND TRIGGGERS

Y.A.HATHALIYA

PL/SQL

- SQL does not provide procedural capabilities such as conditional checking, branching and looping, oracle provides **PL/SQL(Procedural Language/Structured Query Language)** to overcome the disadvantage of SQL.
- It supports all the facility of SQL along with procedural capabilities.
- SQL can be included in PL/SQL program block.
- SQL data definition statement such as CREATE is not allowed in PL/SQL.

PL/SQL Advantages

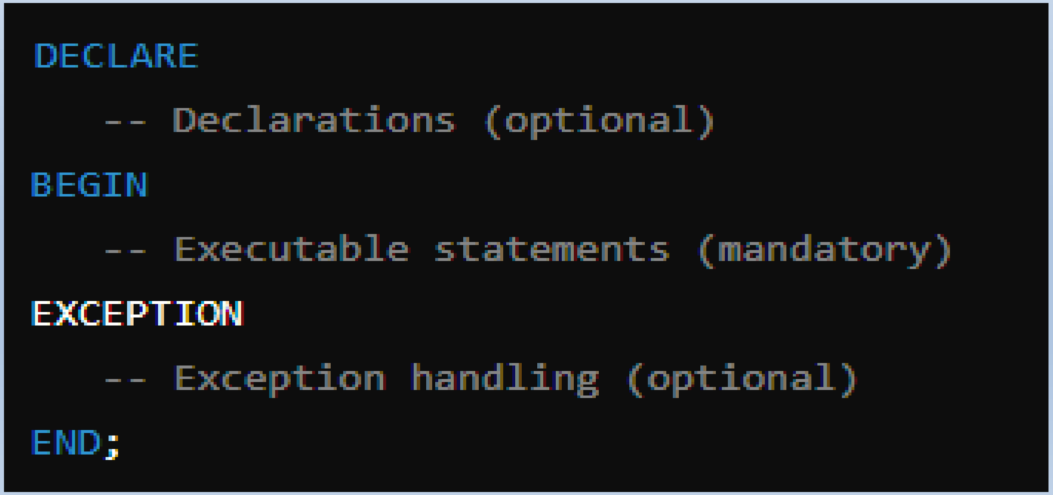
- Procedural Capabilities
- Support to Variable
- Support to OOP
- Error Handling Support
- Support User Defined Function
- Sharing of Code
- Portability
- Efficient Execution

SQL vs PL/SQL

SQL	PL/SQL
Structured Query Language	Procedural Language/Structured Query Language
Mainly used for database manipulation and querying	Extend functionality of SQL with Procedural capabilities.
SQL Statements are executed one at a time	It is executed as a block which contain many SQL statements
Does not provide exception handling	Provide exception handling
Does not provide error handling	Provide error handling
Not Support OOP	Support OOP
Not Support Control Structures	Support Control Structures
Not Support Variables	Support Variables
Not Support I/O Operations	Support I/O Operations

PL/SQL Generic Block Structure

- PL/SQL code or a program grouped into structure called a block.
- Block can be
  - Named block (if some name is given)
  - Anonymous block (if no name is given)
- PL/SQL block contain three section
  1. **Declarations Section**
  2. **Executable Command Section**
  3. **Exception Handling Section**



```
DECLARE
   -- Declarations (optional)
BEGIN
   -- Executable statements (mandatory)
EXCEPTION
   -- Exception handling (optional)
END;
```

- **Declaration Section**
  - 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.
- **Executable Command Section**
  - This section is enclosed between the keyword **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.
- **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**.

- **Example:**

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

```
# OUTPUT
Hello, World
```

PL/SQL Datatypes

- PL/SQL supports a wide range of data types, enabling developers to store and manipulate different kinds of data.
- Here are some of the commonly used PL/SQL data types:

Data Type	**Sub Category / Type**
Character	CHAR VARCHAR2 VARCHAR
Date	DATE
Binary	RAW LONG RAW
Boolean	BOOLEAN (Can have TRUE,FALSE,NULL values)
RowID	ROWID (Stores value of address location of each record)
Numerical	NUMBER

PL/SQL Variables

- Variables are declared in the declaration section of the PL/SQL along with valid datatype.
- Some of the basic examples are given below,

```
v_char CHAR(10) := 'PLSQL';
v_varchar VARCHAR2(20) := 'Oracle';
v_num NUMBER(5) := 123;
v_num NUMBER(5,2) := 123.45;
v_date DATE := SYSDATE;
v_flag BOOLEAN := TRUE;
```

PL/SQL Constants

- To declare a constant, you need to use the keyword CONSTANT in the DECLARE section, followed by the data type and the initial value.
- The value must be assigned at the time of declaration and cannot be changed later.
- Syntax:

```
constant_name CONSTANT data_type := value;
```

- Some of the basic examples are given below,

```
pi CONSTANT NUMBER := 3.14159;
```

PL/SQL Displaying Message

- In PL/SQL, you can display messages or output to the console using the built-in package DBMS_OUTPUT.
- The procedure DBMS_OUTPUT.PUT_LINE is commonly used to print or display text messages during the execution of PL/SQL code.
- Syntax:

```
DBMS_OUTPUT.PUT_LINE(message);
```

- Basic examples are given below,

```
SET SERVEROUTPUT ON;  -- Enable output in your SQL environment
BEGIN
        DBMS_OUTPUT.PUT_LINE('Hello, welcome to PL/SQL! ');
        DBMS_OUTPUT.PUT_LINE('Hello=' || 21);
END;
```

PL/SQL Comments

- In PL/SQL, comments are used to add explanations or notes to your code, which are ignored by the PL/SQL compiler.
- PL/SQL Supports two types of comments,

1. Single-line comments

```
-- This is a single-line comment
```

2. Multi-line comments

```
/* This is a multi-line comment.
It can span multiple lines.
Everything within these symbols is treated as a comment. */
```

- **Example: PL/SQL Block for addition of two numbers**

```
DECLARE
        num1 NUMBER(3) := 10;
        num2 NUMBER(3) := 20;
        ans NUMBER(5);
BEGIN
        ans:= num1+num2;
        DBMS_OUTPUT.PUT_LINE('Addition is:'||ans);

END;
```

```
# OUTPUT
30
```

Anchor Data type

- In PL/SQL, the anchor datatype is a feature used to declare variables or parameters with the same datatype as an existing database column, cursor, or another variable.
- The %TYPE attribute allows you to declare a variable or parameter that has the same datatype as a specific database column or another variable.
- Syntax:

```
variable_name column_name%TYPE;
```

- Example:

```
emp_name employees.first_name%TYPE;
        -- emp_name will have the same datatype as first_name in
        employees table
```

PL/SQL User Input

- In PL/SQL, user input is taken by using following way.
- Example:

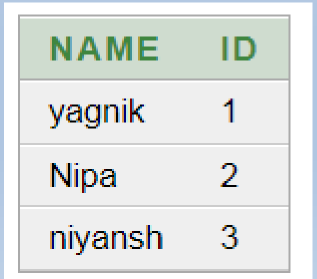
```
DECLARE
    NO NUMBER(3);
BEGIN
    NO:=:NO;   or  NO:=&NO;
    DBMS_OUTPUT.PUT_LINE(NO);
END;
```

Example: PL/SQL Block for addition of two numbers

```
DECLARE
    num1 NUMBER;
    num2 NUMBER;
    result NUMBER;
BEGIN
    -- Prompt the user to input the first number
    num1 := :first_number;
    -- Prompt the user to input the second number
    num2 := :second_number;
    result := num1 + num2;
    -- Display the result
    DBMS_OUTPUT.PUT_LINE('The sum of ' || num1 || ' and ' || num2 || ' is: ' || result);
END;
# OUTPUT
First Number:10
Second Number:20
The sum of 10 and 20 is: 30
```

Example: PL/SQL Block with SQL Commands in it.

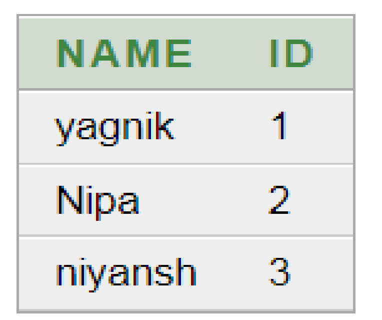
```
create table pl1(name varchar2(20),id number(10));

insert into pl1 values(‘yagnik',1);

insert into pl1 values(‘Nipa’,2);


-- PL/SQL Block that insert data into pl1

BEGIN

        insert into pl1 values (‘niyansh',3);

END;


select * from pl1;
```



NAME	ID
yagnik	1
Nipa	2
niyansh	3

Example: PL/SQL Block with SQL Commands in it.

```
select * from pl1;
```



NAME	ID
yagnik	1
Nipa	2
niyansh	3

```
--PL/SQL Block to retrieve all data from a table pl1.
BEGIN
    -- Using a FOR loop to iterate through all rows
    FOR pl_rec IN (SELECT name, id FROM pl1)
    LOOP
        DBMS_OUTPUT.PUT_LINE('ID: ' || pl_rec.id || ', Name: ' || pl_rec.name);
    END LOOP;
END;
```

PL/SQL Control Structure

- PL/SQL offers several control structures that allow you to control the flow of execution in your program.
- The primary control structures in PL/SQL are:
  - Conditional Control
  - Iterative Control (Loops)
  - Sequential Control

Conditional Control (IF…THEN)

- Executes a block of code if a condition is true.
- Syntax:

```
IF condition THEN
        -- Statements to execute if condition is true
END IF;
```

- Example:

```
SET SERVEROUTPUT ON;
DECLARE
        salary NUMBER := 4000;
BEGIN
        IF salary > 3000 THEN
                DBMS_OUTPUT.PUT_LINE('Salary is above 3000');
        END IF;
END;
# OUTPUT
Salary is above 3000
```

Conditional Control (IF…THEN..ELSE)

- Executes one block of code if the condition is true, and another if it is false.
- Syntax:

```
IF condition THEN
        -- Statements if condition is true
ELSE
        -- Statements if condition is false
END IF;
```

- Example:

```
DECLARE
    salary NUMBER := 2500;
BEGIN
    IF salary > 3000 THEN
        DBMS_OUTPUT.PUT_LINE('Salary is above 3000');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Salary is 3000 or below');
    END IF;
END;
# OUTPUT
Salary is 3000 or below
```

- Example: PL/SQL Block to find whether the given no is even or odd

```
DECLARE
        NO NUMBER(3);
BEGIN
        NO:=:NO;
        IF MOD(NO,2)=0 THEN
                DBMS_OUTPUT.PUT_LINE(NO||' IS EVEN');
        ELSE
                DBMS_OUTPUT.PUT_LINE(NO||' IS ODD');
        END IF;
END;
# OUTPUT
NO: 20
20 IS EVEN
```

Conditional Control (IF…THEN..ELSEIF..ELSE)

- Executes different blocks of code for multiple conditions.
- Syntax:

```
IF condition1 THEN
        -- Statements if condition1 is true
ELSIF condition2 THEN
        -- Statements if condition2 is true
ELSE
        -- Statements if none of the conditions are true
END IF;
```

- Example:

```
DECLARE
        grade CHAR(1) := 'B’;
BEGIN
        IF grade = 'A' THEN
                DBMS_OUTPUT.PUT_LINE('Excellent');
        ELSIF grade = 'B' THEN
                DBMS_OUTPUT.PUT_LINE('Good');
        ELSE
                DBMS_OUTPUT.PUT_LINE('Needs Improvement');
        END IF;
END;
# OUTPUT
Good
```

Conditional Control (CASE)

- Similar to the IF statement but more readable when multiple conditions are involved.
- Syntax:

```
CASE expression
    WHEN value1 THEN
        -- Statements
    WHEN value2 THEN
        -- Statements
    ELSE
        -- Statements
END CASE;
```

- Example:

```
DECLARE
    grade CHAR(1) := 'A’;
BEGIN
    CASE grade
        WHEN 'A' THEN
            DBMS_OUTPUT.PUT_LINE('Excellent');
        WHEN 'B' THEN
            DBMS_OUTPUT.PUT_LINE('Good');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Needs Improvement');
    END CASE;
END;
# OUTPUT
Excellent
```

Iterative Control (LOOP)

- Executes a block of statements repeatedly until explicitly terminated using EXIT
- Syntax:

```
LOOP
        -- Statements
        EXIT WHEN condition;
END LOOP;
```

- Example: PL/SQL Block to print 1 to 5.

```
DECLARE
        i NUMBER(3):=1;
BEGIN
        LOOP
                EXIT WHEN i>5;
                DBMS_OUTPUT.PUT_LINE(i);
                i:= i + 1;
        END LOOP;
END;
# OUTPUT
1
2
3
4
5
```

Iterative Control (WHILE)

- Executes a block of statements as long as a condition is true.
- Syntax:

```
WHILE condition LOOP
        -- Statements
END LOOP;
```

- Example: PL/SQL Block to print 1 to 5.

```
DECLARE
        i NUMBER(3):=1;
BEGIN
        while  i<=5
        LOOP
                DBMS_OUTPUT.PUT_LINE(i);
                i:= i + 1;
        END LOOP;
END;
```

```
# OUTPUT
1
2
3
4
5
```

Iterative Control (FOR)

- Executes a block of code a specific number of times.
- Syntax:

```
FOR loop_counter IN lower_bound..upper_bound LOOP
        -- Statements
END LOOP;
```

- Example: PL/SQL Block to print 1 to 5.

```
BEGIN
        FOR counter IN 1 .. 5
        loop
                dbms_output.put_line(counter);
        end loop;
END;
# OUTPUT
1
2
3
4
5
```

Sequential Control (GOTO)

- Transfers control to another part of the program identified by a label.
- This is not recommended in structured programming, but it is supported in PL/SQL.
- Syntax:

```
<<label>>
       -- Statements
GOTO label;
```

- Example:

```
DECLARE
    x NUMBER := 1;
BEGIN
    <<start_loop>>
    IF x < 5 THEN
        DBMS_OUTPUT.PUT_LINE('x is: ' || x);
        x := x + 1;
        GOTO start_loop;
    END IF;
END;
# OUTPUT
x is: 1
x is: 2
x is: 3
x is: 4
```

Function

- Function is a subprogram that performs a specific task and returns a value.
- Functions are typically used in SQL statements and PL/SQL blocks to return a single value.
- The function can be either user-defined or predefined.
- Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name (argument IN datatype) RETURN datatype
IS
        -- Declarations
BEGIN
        -- Statements
        RETURN some_value;
END function_name;
```

- Syntax to drop a function is:

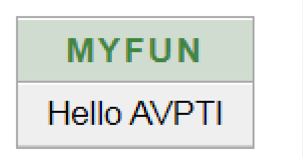
```
        DROP FUNCTION function_name;
```

- Example: Write a Function to print Hello AVPTI.

```
CREATE OR REPLACE FUNCTION myfun RETURN VARCHAR2
IS
BEGIN
        RETURN ('Hello AVPTI’);
END myfun;
```

- How to Call Function: Method1: Using Select Commands

```
SELECT myfun from dual;
```



MYFUN
Hello AVPTI

- How to Call Function: Method2: Using PL/SQL Block

```
        DECLARE
                message VARCHAR2(25);
        BEGIN
                message := myfun;
                DBMS_OUTPUT.PUT_LINE(message);
        END;
#OUTPUT: Hello AVPTI
```

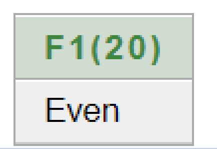
Example: Write a Function to check whether given number is odd or even, also write the PL/SQL block to invoke/call it.

- Function Creation

```
CREATE FUNCTION f1 (n IN NUMBER) RETURN VARCHAR2
IS
    result VARCHAR2(10);
BEGIN
    IF MOD(n, 2) = 0 THEN
        result := 'Even';
    ELSE
        result := 'Odd';
    END IF;
    RETURN result;
END f1;
```

- Calling the Function using select commands

```
SELECT f1 (20) from dual;
# OUTPUT
```



F1(20)
Even

- Calling the Function Using PL/SQL Block

```
DECLARE
	n NUMBER := 7;
	result VARCHAR2(10);
BEGIN
	-- Call the function and store the result in result
	result := f1(n);
	-- Display the result
	DBMS_OUTPUT.PUT_LINE('The number ' || n || ' is ' || result);
END;
```

```
#OUTPUT
The number 7 is Odd
```

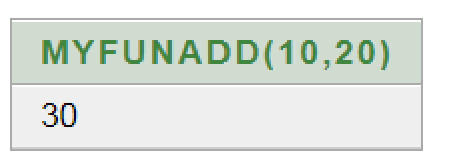
Example: Write a Function to calculate addition of two numbers, also write the PL/SQL block to invoke/call it.

- Function Creation

```
CREATE FUNCTION myfunadd (n1 IN NUMBER,n2 IN NUMBER) RETURN NUMBER
IS
        n3 NUMBER(8);
BEGIN
        n3:= n1+n2;
        RETURN n3;
END myfunadd;
```

- Calling the Function using select commands

```
SELECT myfunadd (10,20) from dual;
# OUTPUT
```



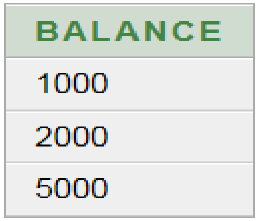
MYFUNADD(10,20)
30

- Calling the Function Using PL/SQL Block

```
DECLARE
        n1 NUMBER := 10;
        n2 NUMBER := 20;
        n3 NUMBER(8);
BEGIN
        n3 := myfunadd(n1,n2);
        DBMS_OUTPUT.PUT_LINE('Addition is:' ||n3);
END;
```

```
#OUTPUT
Addition is: 30
```

Example: How to Use Function to retrieve some data from table, consider person1 table displayed here



BALANCE
1000
2000
5000

- Function Creation

```
CREATE FUNCTION myf RETURN NUMBER
IS
        maxbal NUMBER;
BEGIN
        select max(balance) into maxbal from person1;
        RETURN maxbal;
END myf;
```

- Calling the Function using select commands

```
SELECT myf from dual;
#OUTPUT
5000
```

- How to Call Function Using PL/SQL Block

```
DECLARE
        maxbal NUMBER;
BEGIN
        maxbal := myf;
        DBMS_OUTPUT.PUT_LINE(maxbal);
END;
#OUTPUT
5000
```

Procedure

- Procedure is a subprogram that performs a specific task but does not return a value directly, however, you can use OUT parameters to return multiple values from a procedure, as it is stored in database server it is also called Stored Procedure
- Generally Procedure is used to perform an action and function is used to compute a value.
- Syntax:

```
CREATE [OR REPLACE] PROCEDURE p_name (argument  [IN,OUT,INOUT] datatype)
IS
        -- Declarations
BEGIN
        -- Statements
END p_name;
```

- Here in above syntax:

  IN: Passes a value to the procedure (default).

  OUT: Returns a value from the procedure.

  IN OUT: Passes a value to the procedure and returns an updated value.
- Syntax to drop a Procedure is:

```
        DROP PROCEDURE procedure_name;
```

Example: Write a procedure to print hello, Also write the PL/SQL block to invoke it.

- Creation of Procedure:

```
CREATE PROCEDURE myproc
IS
BEGIN
            DBMS_OUTPUT.PUT_LINE(‘hello');
END myproc;
```

- Execute Procedure using ‘EXECUTE’ command (Supports only on Command line)

```
EXEC myproc;
#OUTPUT
Hello
```

- Execute Procedure using PL/SQL Block:

```
BEGIN
        myproc;
END;
#OUTPUT
Hello
```

Example: Write a procedure to check whether given number is odd or even. Also write the PL/SQL block to invoke it.

- Creation of Procedure:

```
CREATE PROCEDURE p1 (n IN NUMBER)
IS
BEGIN
        IF MOD(n, 2) = 0 THEN
                DBMS_OUTPUT.PUT_LINE(n || ' is even');
        ELSE
                DBMS_OUTPUT.PUT_LINE(n || ' is odd');
        END IF;
END p1;
```

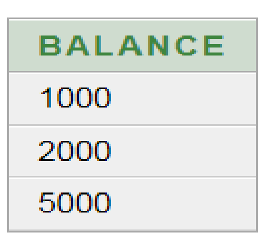
- Execute Procedure using ‘EXECUTE’ command.

```
EXEC p1(20)
#OUTPUT
20 is even
```

- Execute Procedure Using PL/SQL Block:

```
DECLARE
        n NUMBER := 7;
BEGIN
        p1(n);
END;

#OUTPUT
7 is odd
```

Example: How to Use Procedure to insert some data to the table, consider person1 table displayed here



BALANCE
1000
2000
5000

- Procedure Creation

```
CREATE PROCEDURE myf1(balance IN NUMBER)
IS
BEGIN
        insert into person1 values(balance);
END myf1;
```

- Execute Procedure using ‘EXECUTE’ command.

```
exec myf1(6000);
#OUTPUT
PL/SQL Procedure successfully completed
```

- Execute Procedure Using PL/SQL Block:

```
DECLARE
BEGIN
        myf1(7000);
END;
```

```
#OUTPUT
Statement Processed
```

Procedure vs Function

Procedure	Function
It may or may not return a value	It must return a value
Can Return Multiple Values	Can Return only one value
Mainly used for performing action	Mainly used for computation
Procedure can call a function	Function cannot call a procedure
A block of code that performs a task without returning a value	A block of code that performs a task and returns a value
Example:	Example:

Exceptions/Exceptions Handling

- In PL/SQL, Exceptions are used to handle errors and other exceptional conditions that arise during execution of PL/SQL Block.
- When an error occurs during the execution of a PL/SQL block, an exception is raised, and control is transferred to the exception-handling part of the block of PL/SQL which allows the program to respond to the error gracefully instead of crashing.
- Types of Exceptions:
  - Predefined/System Defined Exception
  - User-defined Exceptions

- Basic Syntax for Exception Handling:

```
DECLARE
        -- Declaration
BEGIN
        -- Executable statements
EXCEPTION
        WHEN exception_name1 THEN
                -- Handle the exception
        WHEN exception_name2 THEN
                -- Handle the exception
        ........
        ........
        ........
        WHEN OTHERS THEN
                -- Handle all other exceptions
END;
```

Predefined/System Defined Exceptions

- Exception that are automatically raised by the PL/SQL runtime engine when certain common errors occur is known as predefined exception.
- It is a predefined error conditions that oracle database raises in response to some specific error/exception conditions, you can catch them in your EXCEPTION block.
- These exceptions have pre defined names to identify them.
- For example: NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE etc.
- It can be further divided into two types,
  1. Named Exceptions
  2. Unnamed (Numbered) Exceptions

- Named Exceptions:
  - Named Exceptions are exceptions that are explicitly defined by the Oracle database and have predefined names, these named exceptions correspond to certain error codes, which are raised when specific situations occur.
  - By using named exceptions, you can write more readable and maintainable error handling code.
  - Here are a few common named exceptions in PL/SQL:
    - NO_DATA_FOUND: Raised when a SELECT INTO statement does not return any rows.
    - TOO_MANY_ROWS: Raised when a SELECT INTO statement returns more than one row.
    - ZERO_DIVIDE: Raised when an attempt is made to divide a number by zero.
    - DUP_VAL_ON_INDEX: Raised when you insert duplicate value in the field where primary or unique key is declared.
    - INVALID_NUMBER: Raised when invalid numeric operation is performed.
    - NOT_LOGGED_ON: Raised when you are trying to operation before login.
    - LOGON_DENIED: Raised when you are trying to login with wrong user_id/password.

Example of Named Exception:

```
CREATE TABLE ex1 (id NUMBER(10) PRIMARY KEY, name  VARCHAR2(50), salary NUMBER(10));
INSERT INTO ex1 VALUES (1, 'Yagnik', 150000);
INSERT INTO ex1 VALUES (2, 'Nipa', 260000);

DECLARE
        l_id NUMBER(10) := 103;      -- Non-existing employee ID
        l_salary NUMBER(10);
BEGIN
        -- Attempt to fetch the salary for an employee who may not exist
        SELECT salary INTO  l_salary FROM   ex1 WHERE  id = l_id;
        DBMS_OUTPUT.PUT_LINE('Salary: ' || l_salary);
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('No employee found with ID ' || l_id);
END;
# OUTPUT: No employee found with ID 103
```

Unnamed Exceptions:

- Unnamed Exceptions are exceptions that are not predefined by Oracle (like named exceptions), but are instead handled using the WHEN OTHERS clause in the EXCEPTION block.
- This allows you to catch any exception that is not specifically handled by a named exception handler.
- Key Points:
  - WHEN OTHERS: This is a catch-all clause that can handle any exception that is raised and not explicitly caught by other handlers.
  - SQLCODE and SQLERRM: These functions allow you to retrieve the error code and the error message associated with the exception, providing more information about what went wrong.

Example of Named Exception:

```
CREATE TABLE ex1 (id NUMBER(10) PRIMARY KEY, name  VARCHAR2(50), salary NUMBER(10));
INSERT INTO ex1 VALUES (1, 'Yagnik', 150000);
INSERT INTO ex1 VALUES (2, 'Nipa', 260000);

DECLARE
        l_id NUMBER(10) := 1;
        l_salary NUMBER(10);
BEGIN
        -- Attempting to fetch salary, but introducing an error (division by zero)
        SELECT salary / 0 INTO l_salary FROM  ex1 WHERE  id = l_id;
        DBMS_OUTPUT.PUT_LINE('Salary: ' || l_salary);
EXCEPTION
        WHEN OTHERS THEN
        -- Handling any unexpected exception
        DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('Error code: ' || SQLCODE);
END;
# OUTPUT: An error occurred: ORA-01476: divisor is equal to zero
Error code: -1476
```

User Defined Exceptions

- Exceptions that are defined by users are known as User Defined Exceptions.
- PL/SQL allows you to define your own exceptions according to the need of your program, these are explicitly defined by the programmer using the EXCEPTION keyword and raised using the RAISE statement.
- It enhance the code clarity.
- Declaring a User-Defined Exception Steps or Sequence:
  1. Declare Exception
  2. Raise Exception using the RAISE statement
  3. Handle Exception it in the EXCEPTION block of PL/SQL

```
DECLARE
        exception_name EXCEPTION;           -- Declare the exception
BEGIN
        -- Some logic
        IF some_condition THEN
                RAISE exception_name;       -- Raise the exception
        END IF;
EXCEPTION
                WHEN exception_name THEN
                DBMS_OUTPUT.PUT_LINE('User-defined exception occurred.');
END;
```

Example: We’ll create a table for employees, insert data, and define a custom exception that will be raised if an employee’s salary is below a certain threshold.

```
CREATE TABLE employees (employee_id NUMBER(10) PRIMARY KEY, name VARCHAR2(50),
salary NUMBER(10));
```

```
INSERT INTO employees VALUES (101, ‘Yagnik’, 150000);
INSERT INTO employees VALUES (102, ‘Nipa’, 30000);
```

```
DECLARE
        l_employee_id NUMBER(10) := 102;                    -- Choosing employee with a low salary
        l_salary NUMBER(10);
        ex_salary_too_low EXCEPTION;                        -- Define a user-defined exception
        min_salary CONSTANT NUMBER(10) := 40000;            -- Set the minimum salary
BEGIN
        -- Fetch the salary for the chosen employee
        SELECT salary INTO   l_salary FROM   employees WHERE  employee_id = l_employee_id;
        -- Check if the salary is below the threshold and raise the user-defined exception
        IF l_salary < min_salary THEN
                RAISE ex_salary_too_low;
        ELSE
                DBMS_OUTPUT.PUT_LINE('Salary is acceptable: ' || l_salary);
        END IF;
EXCEPTION
        WHEN ex_salary_too_low THEN
                -- Handle the user-defined exception
                DBMS_OUTPUT.PUT_LINE('Error: Salary for employee ID ' || l_employee_id || ' is below the acceptable threshold.');
END;
# OUTPUT: Error: Salary for employee ID 102 is below the acceptable threshold.
```

Cursors

- Cursor is an area in memory where the data required to execute SQL Statements is stored, it holds one or more rows returned by SQL Statements.
- Cursor is a pointer that points to a result of a query.
- The data that is stored in the cursor is called Active Data Set, and then that data is stored due to execution of some SQL Statements is called Result Set
- The row that is being processed is called Current Row, and a Pointer is known as Row Pointer which is keeping the track of current row.
- There are two types of cursors in PL/SQL:
  - Implicit Cursors
  - Explicit Cursors

Implicit Cursors (Static and System Defined)

- Cursor which is opened by oracle itself to execute any SQL Statements is called Implicit Cursor
- These cursors are automatically created whenever a DML statement like INSERT, UPDATE, DELETE, or a SELECT statement is executed, programmers cannot control the implicit cursors and the information in it.
- For INSERT operations, the cursor holds the data that needs to be inserted and for UPDATE and DELETE operations, the cursor identifies the rows that would be affected.
- Syntax to use Implicit Cursor Attribute:

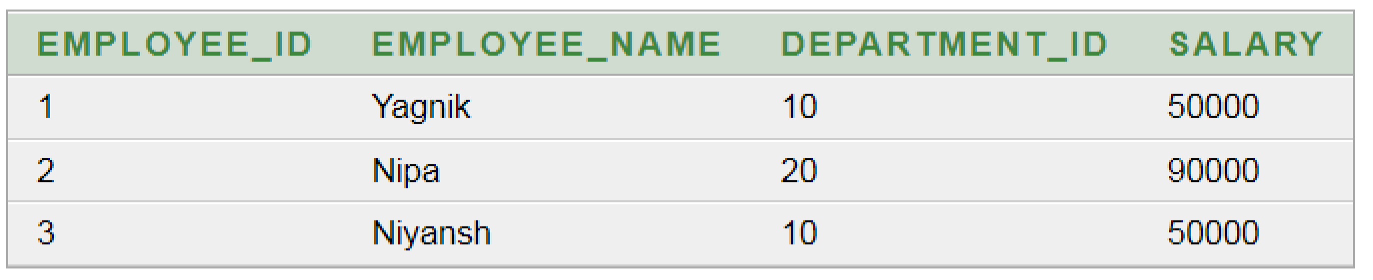
  Name_of_Cursor % Attribute_Name

- Implicit Cursors Attributes:

Name	Use/Impact
SQL%FOUND	Returns TRUE if an SQL statement affected one or more rows, FALSE otherwise.
SQL%NOTFOUND	Returns TRUE if an SQL statement did not affect any rows, FALSE otherwise (Opposite of SQL%FOUND).
SQL%ROWCOUNT	Returns The number of rows affected by an SQL statement.
SQL%ISOPEN	Returns FALSE for implicit cursors, since they are automatically closed after execution.

Example of Implicit Cursor:

```
CREATE TABLE employees (
employee_id NUMBER(10) PRIMARY KEY,
employee_name VARCHAR2(50),
department_id NUMBER(10),
salary NUMBER(10));

INSERT INTO employees VALUES (1, 'Yagnik', 10, 50000);
INSERT INTO employees VALUES (2, 'Nipa', 20, 90000);
INSERT INTO employees VALUES (3, 'Niyansh', 10, 50000);

select * from employees;
```



EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT_ID	SALARY
1	Yagnik	10	50000
2	Nipa	20	90000
3	Niyansh	10	50000

```
-- Here, we'll use an UPDATE statement to increase the salary of employees in department 10 by 500 and then use the implicit cursor attributes to display the result.
BEGIN
        -- Update the salary of employees in department 10
        UPDATE employees SET salary = salary + 500 WHERE department_id = 10;
        -- Output the result using implicit cursor attributes
        IF SQL%ROWCOUNT > 0 THEN
                DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated.');
        ELSE
                DBMS_OUTPUT.PUT_LINE('No rows updated.');
        END IF;
END;
```

# OUTPUT: 2 rows updated

You Can Verify with Select * from employees

Explicit Cursors

- Cursor which are created and managed by user in PL/SQL is called explicit cursor or user defined cursor.
- For queries that return multiple rows, you need to explicitly define a cursor and control its execution.
- The basic steps for using explicit cursors are:
  1. Declare a Cursor (to initialize memory)
  2. Open a cursor (to allocate memory)
  3. Fetch data from cursor (to retrieve data)
  4. Process data (to process data)
  5. Close a cursor (to release allocated memory)

- **Declare a Cursor**
  - Define the SQL query that the cursor will use in declaration section of PL/SQL block.
  - After declaring a cursor it is initializing the memory area, remember still it is not allocated to it.
  - Syntax: CURSOR cursor_name IS Select statements;

- **Open a Cursor**
  - Cursor is opened in executable section of PL/SQL Block.
  - When we open cursor:
    - Memory is allocated to it for storing data.
    - Select statements is executed and associated with cursor.
    - Active Data Set is created by retrieving data from table.
    - Set the cursor row pointer to point the first record in active data set.
  - Syntax: OPEN cursor_name;

- **Fetch Data From Cursor**
  - Retrieve each row from the result set.
  - The FETCH statement retrieves each row from the cursor into the declared variables.

```
FETCH cursor_name INTO variable1,variable2;
```

- **Process Data**
  - Process Retrieved Data.
  - Data already fetch in variables which can be processed accordingly.
  - Use EXIT WHEN cursor_name%NOTFOUND to stop the loop when all rows are processed.

- **Close a Cursor**
  - Cursor should be closed after processing data, and it will release the memory associated with the cursor

```
CLOSE cursor_name;
```

- Syntax to use Explicit Cursor Attribute:

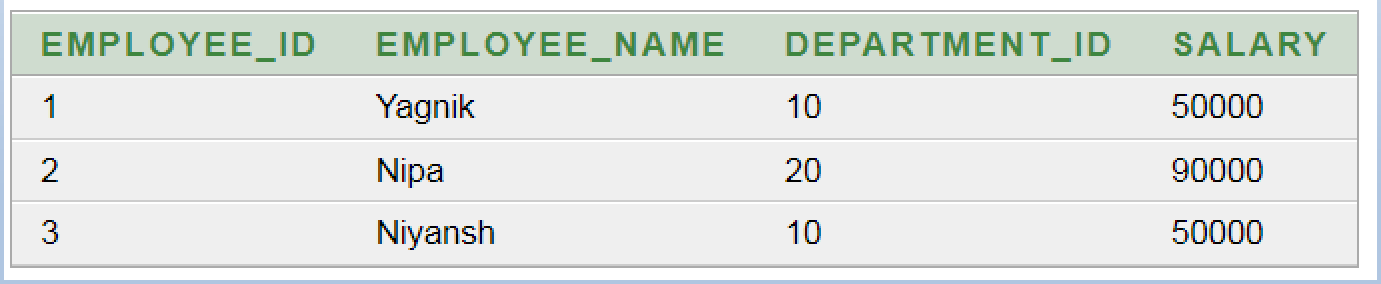
  Name_of_Cursor % Attribute_Name

- Explicit Cursors Attributes:

Name	Use/Impact
SQL%FOUND	TRUE if the last fetch returned a row. FALSE if no row was returned.
SQL%NOTFOUND	TRUE if the last fetch did not return a row. FALSE if a row was returned.
SQL%ROWCOUNT	Returns The number of rows fetched so far by the cursor.
SQL%ISOPEN	TRUE if the cursor is open. FALSE if the cursor is closed.

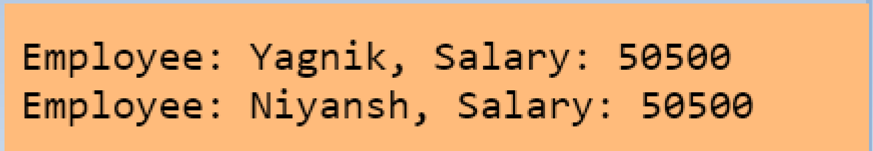
Example of Explicit Cursor:

```
 CREATE TABLE employees (
employee_id NUMBER(10) PRIMARY KEY,
employee_name VARCHAR2(50),
department_id NUMBER(10),
salary NUMBER(10));

INSERT INTO employees VALUES (1, 'Yagnik', 10, 50000);
INSERT INTO employees VALUES (2, 'Nipa', 20, 90000);
INSERT INTO employees VALUES (3, 'Niyansh', 10, 50000);

select * from employees;
```



EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT_ID	SALARY
1	Yagnik	10	50000
2	Nipa	20	90000
3	Niyansh	10	50000

```
--Now, we’ll use an explicit cursor to retrieve and display the names and salaries of employees in department 10
DECLARE
        -- Declare the cursor
        CURSOR emp_cursor IS
        SELECT employee_name, salary FROM employees WHERE department_id = 10;
        -- Declare variables to hold data from the cursor
        v_employee_name employees.employee_name%TYPE;
        v_salary employees.salary%TYPE;
BEGIN
        -- Open the cursor
        OPEN emp_cursor;
        -- Fetch rows from the cursor one by one
        LOOP
                -- Fetch a row
                FETCH emp_cursor INTO v_employee_name, v_salary;
                -- Exit loop when no more rows are returned
                EXIT WHEN emp_cursor%NOTFOUND;
                -- Output the employee name and salary
        DBMS_OUTPUT.PUT_LINE('Employee: ' || v_employee_name || ', Salary: ' || v_salary);
        END LOOP;
        CLOSE emp_cursor;
END;
```



```
Employee: Yagnik, Salary: 50500
Employee: Niyansh, Salary: 50500
```

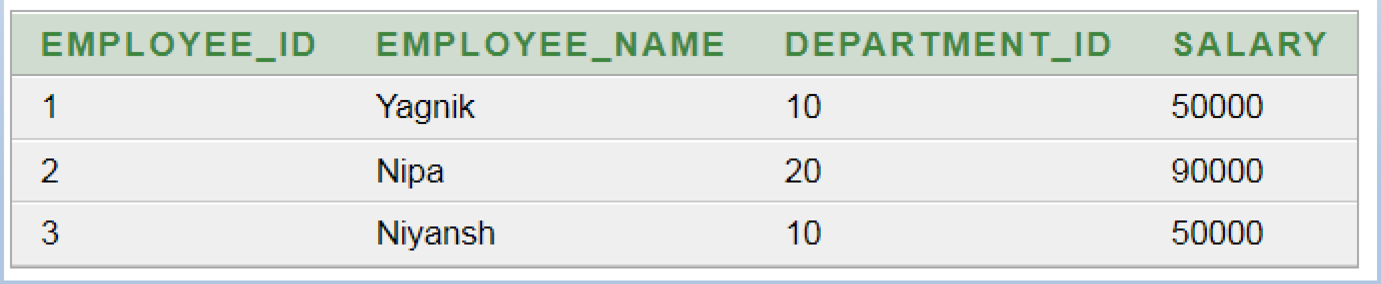
Parameterized Cursors (Dynamic)

- Parameterized cursors in PL/SQL are cursors that accept parameters when they are opened, this allows you to pass values to the cursor's query dynamically at runtime, making the cursor more flexible and reusable for different sets of input.
- Syntax:

```
CURSOR cursor_name (parameter1 datatype, parameter2 datatype, ...) IS
SELECT_statement;
```

Example of Parameterized Cursor:

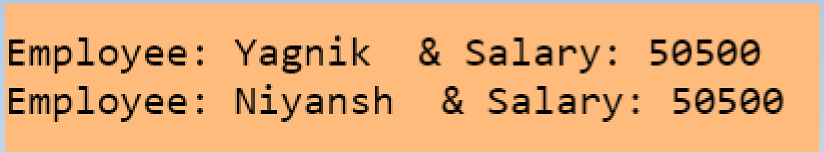
```
 CREATE TABLE employees (
employee_id NUMBER(10) PRIMARY KEY,
employee_name VARCHAR2(50),
department_id NUMBER(10),
salary NUMBER(10));

INSERT INTO employees VALUES (1, 'Yagnik', 10, 50000);
INSERT INTO employees VALUES (2, 'Nipa', 20, 90000);
INSERT INTO employees VALUES (3, 'Niyansh', 10, 50000);

select * from employees;
```



EMPLOYEE_ID	EMPLOYEE_NAME	DEPARTMENT_ID	SALARY
1	Yagnik	10	50000
2	Nipa	20	90000
3	Niyansh	10	50000

-- Now, we’ll declare a parameterized cursor to retrieve employees from a specific department based on a parameter passed at runtime.

```
DECLARE
        -- Declare a parameterized cursor
        CURSOR emp_cursor (p_dept_id NUMBER) IS
        SELECT employee_name, salary FROM employees WHERE department_id = p_dept_id;
        -- Variables to hold data fetched from the cursor
        v_employee_name employees.employee_name%TYPE;
        v_salary employees.salary%TYPE;
BEGIN
        -- Open the cursor for department 10
        OPEN emp_cursor(10);
        -- Fetch rows from the cursor one by one
        LOOP
        FETCH emp_cursor INTO v_employee_name, v_salary;
        -- Exit the loop when no more rows are returned
        EXIT WHEN emp_cursor%NOTFOUND;
        -- Output the employee name and salary
        DBMS_OUTPUT.PUT_LINE('Employee: ' || v_employee_name || '  & Salary: ' || v_salary);
        END LOOP;
        CLOSE emp_cursor;
END;
```



```
Employee: Yagnik  & Salary: 50500
Employee: Niyansh  & Salary: 50500
```

Triggers

- Triggers in PL/SQL are special types of stored procedures that automatically execute in response to certain events on a particular table or view.
- Trigger is fired automatically when DML statements are executed or some operations are performed on table.
- Triggers could be defined on the table, view, schema, or database with which the event is associated.
- Triggers are written to be executed in response to any of the following events.
  - DML statement (DELETE, INSERT, or UPDATE).
  - DDL statement (CREATE, ALTER, or DROP).
  - Database Operation (SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN).

- Trigger Advantages:
    - Enforces referential integrity
    - Event logging and storing information on table access
    - Auditing sensitive data
    - Synchronous replication of tables
    - Imposing security authorizations
    - Preventing invalid transactions

- Trigger Disadvantages:
    - Increase Overhead of Database
    - Difficult to Debug

- Types of Trigger:
  - Level Trigger
    - Row Level Trigger
      - It fires on every record affected by triggering statements.
      - For example if UPDATE statement updates multiple rows in a table, Row Trigger is fires once for each row.
      - It always uses FOR EACH ROW clause in triggering statement.
    - Statement Trigger
      - It fires once for each statement.
      - For example if UPDATE statement updates multiple rows in a table, a statement trigger is fired only once.
      - FOR EACH ROW clause not used in triggering statement.
  - Timing Trigger
    - Before Trigger
      - It fires before triggering statements (before the execution of DML statements)
    - After Trigger
      - It fires after triggering statements (after the execution of DML statements)

- Trigger Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER }
{INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW]
DECLARE
        --declarations
BEGIN
        --executable
EXCEPTION
        --exceptions
END;
```

- Syntax to DROP a Trigger is:

```
        DROP TRIGGER trigger_name;
```

Example:

Step:1 Create Tables

```
        CREATE TABLE Employees (
        EmployeeID INT PRIMARY KEY,
        Name VARCHAR(50),
        Salary DECIMAL(10, 2));
```

Step:2 Create the Trigger (This trigger will print a message whenever a new employee is inserted with a salary lower than 20000)

```
CREATE OR REPLACE TRIGGER MinSalaryTrigger
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
        IF :NEW.Salary < 20000 THEN
                dbms_output.put_line('please enter a valid salary');
        END IF;
END;
```

Step:3 Insert data and see the trigger in action

```
-- Insert an employee with a lower salary
INSERT INTO Employees VALUES (1, 'Amit', 15000);
#OUTPUT: please enter a valid salary

-- Insert an employee with a lower salary
INSERT INTO Employees VALUES (2, 'Yagnik', 25000);
#OUTPUT: 1 row(s) inserted.
```

Package

- Package is a container for other database objects, so package can hold other database objects such as variables, constants, cursors, exceptions, procedures, functions and sub-programs.
- Advantages:
  - Code Reusability
  - Sharing of Code
  - Improve Performance
- Package has usually two components,

1. Package Specification
   - It is the public interface of your application, it contains public types, variables, constants, exceptions, cursors, and subprograms (procedures and functions) that are accessible from outside the package.
   - Syntax:

```
CREATE [OR Replace] PACKAGE package_name
IS
        --Package_specification
End package_name;
```

2. Package Body

- It contains the implementation details of the object declared in the package specification.
- It also can define private types, variables, constants, and subprograms that are not accessible outside the package.
- Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS
        --Package_body
End package_name;
```

Example of Package Use: Suppose we have a following table.

```
CREATE TABLE employees (
id NUMBER(10) PRIMARY KEY,
name VARCHAR2(50),
salary NUMBER(10));
INSERT into employees values(1,'yagnik',150000);
INSERT into employees values(2,'nipa',50000);
```

Step:1 Create the Package Specification

```
CREATE PACKAGE employee_pkg IS
            -- Declaration of a function
            FUNCTION get_employee_salary(p_id INT) RETURN NUMBER;
END employee_pkg;
```

Example of Package Use: Suppose we have a following table.

```
CREATE TABLE employees (
id NUMBER(10) PRIMARY KEY,
name VARCHAR2(50),
salary NUMBER(10));
INSERT into employees values(1,'yagnik',150000);
INSERT into employees values(2,'nipa',50000);
```

Step:1 Create the Package Specification

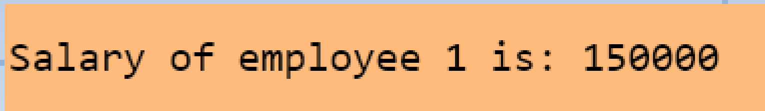
```
CREATE PACKAGE employee_pkg IS
        -- Declaration of a function
        FUNCTION get_employee_salary(p_id INT) RETURN NUMBER;
END employee_pkg;
```

Step:2 Create the Package Body

```
CREATE PACKAGE BODY employee_pkg IS
        -- Implementing the function to get employee salary by ID
        FUNCTION get_employee_salary(p_id INT) RETURN NUMBER
        IS
                v_salary NUMBER;
        BEGIN
                SELECT salary INTO v_salary FROM employees WHERE id = p_id;
        RETURN v_salary;
        END get_employee_salary;
END employee_pkg;
```

Step:3 Use the Package (Retrieving Employee Salary Using the Package Function)

```
        DECLARE
                v_salary NUMBER;
        BEGIN
                v_salary := employee_pkg.get_employee_salary(1);
                DBMS_OUTPUT.PUT_LINE('Salary of employee 1 is: ' || v_salary);
        END;
```



```
Salary of employee 1 is: 150000
```