Unit 4 : PL/SQL Programming

PL/SQL (Procedural Language/SQL) is an extension of SQL developed by Oracle. It combines SQL's power for database operations with procedural programming constructs like loops, conditions, and error handling. PL/SQL is used to create applications, automate tasks, and add business logic to database operations.

Advantages of PL/SQL:

1. **Integration with SQL**: Tight coupling with SQL enables powerful database operations.
2. **Improved Performance**: PL/SQL reduces network traffic by executing multiple SQL statements in a single block.
3. **Portability**: Works seamlessly across Oracle databases.
4. **Error Handling**: Built-in exception handling provides robust error management.

PL/SQL Block Structure:

A PL/SQL program is organized into blocks, each consisting of three main sections:

1. **Declaration Section** (Optional): Used to declare variables, constants, cursors, etc.
2. **Execution Section** (Mandatory): Contains the executable statements.
3. **Exception Section** (Optional): Handles errors during execution.

Basic Syntax

```
DECLARE
    -- Declaration section
    variable_name datatype [DEFAULT value];
BEGIN
    -- Execution section
    -- SQL and PL/SQL statements
EXCEPTION
    -- Exception handling section
    WHEN exception_name THEN
        -- Handle the exception
END;
```

Example: Basic PL/SQL Block

```
DECLARE
    v_employee_name VARCHAR2(50);
BEGIN
    SELECT FirstName INTO v_employee_name FROM Employees
    WHERE EmployeeID = 101;
    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_employee_name);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No employee found with the given ID.');
END;
```

This block retrieves an employee's name with EmployeeID = 101 and prints it. If no employee is found, it handles the exception.

PL/SQL Data Types:

PL/SQL provides a wide range of data types to define variables, constants, and parameters. These data types are:

Numeric Types: Used for storing numbers.

Data Type	Description	Example
NUMBER(p, s)	Fixed or floating-point numbers.	NUMBER(10, 2)
FLOAT	Floating-point number.	3.14

Example

```
DECLARE
    v_salary NUMBER(10, 2) := 50000.50;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END;
```

Character Types: Used for storing alphanumeric data.

Data Type	Description	Example
CHAR(size)	Fixed-length string.	CHAR(10)
VARCHAR2(size)	Variable-length string.	VARCHAR2(50)
NCHAR(size)	Fixed-length string for Unicode.	NCHAR(10)
NVARCHAR2(size)	Variable-length string for Unicode.	NVARCHAR2(50)

Example

```
DECLARE
    v_name VARCHAR2(50) := 'John Doe';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
END;
```

Date/Time Types: Used for storing date and time values.

Data Type	Description	Example
DATE	Stores date and time.	'07-DEC-2024'
TIMESTAMP	Stores date, time, and fractional seconds.	'07-DEC-2024 10:30:00'
INTERVAL YEAR TO MONTH	Stores interval of years and months.	INTERVAL '1-2' YEAR TO MONTH
INTERVAL DAY TO SECOND	Stores interval of days, hours, minutes, and seconds.	INTERVAL '10 12:30:00' DAY TO SECOND

Example

```
DECLARE
    v_today DATE := SYSDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Today: ' || v_today);
END;
```

Boolean Type: Stores TRUE, FALSE, or NULL.

Data Type	Description
BOOLEAN	Stores logical values.

Example

```
DECLARE
    v_is_active BOOLEAN := TRUE;
BEGIN
    IF v_is_active THEN
        DBMS_OUTPUT.PUT_LINE('Status: Active');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Status: Inactive');
    END IF;
END;
```

Anchored Data Types: PL/SQL allows you to create variables based on the type of a table column or another variable using %TYPE and %ROWTYPE.

%TYPE: Links a variable to the data type of a table column or another variable.

```
DECLARE
    v_emp_name Employees.FirstName%TYPE;
BEGIN
    SELECT FirstName INTO v_emp_name FROM Employees WHERE EmployeeID = 101;
    DBMS_OUTPUT.PUT_LINE(v_emp_name);
END;
```

%ROWTYPE: Links a variable to the structure of a table row.

```
DECLARE
    v_employee Employees%ROWTYPE;
BEGIN
    SELECT * INTO v_employee FROM Employees WHERE EmployeeID = 101;
    DBMS_OUTPUT.PUT_LINE(v_employee.FirstName || ' ' || v_employee.LastName);
END;
```

PL/SQL Variables and Constants

Variables and constants in PL/SQL are used to store data temporarily during the execution of a program.

1. Variables in PL/SQL

Variables in PL/SQL store data values that can change during the execution of the program.

Syntax

```
variable_name datatype [NOT NULL] [:= | DEFAULT initial_value];
```

In the above syntax:

- **variable_name**: The name of the variable.
- **datatype**: The data type of the variable (e.g., NUMBER, VARCHAR2).
- **NOT NULL**: Ensures that the variable cannot store NULL.
- **:= or DEFAULT**: Assigns an initial value.

Example: Declaring Variables

```
DECLARE
    v_emp_name VARCHAR2(50); -- Variable without initialization
    v_salary NUMBER(10,2) := 50000; -- Initialized variable
    v_bonus NUMBER DEFAULT 1000; -- Using DEFAULT for initialization
BEGIN
    v_emp_name := 'John Doe'; -- Assigning a value to the variable
    v_salary := v_salary + v_bonus; -- Modifying the variable
    DBMS_OUTPUT.PUT_LINE('Total Salary: ' || v_salary);
END;
```

2. Constants in PL/SQL

Constants are similar to variables but their values cannot be changed once assigned.

Syntax

```
constant_name CONSTANT datatype [NOT NULL] := value;
```

In the above syntax:

- **constant_name**: The name of the constant.
- **CONSTANT**: Keyword indicating it is a constant.
- **value**: The constant's value, which must be assigned at the time of declaration.

Example: Declaring Constants

```
DECLARE
    c_tax_rate CONSTANT NUMBER(5,2) := 15.00; -- Tax rate
constant
    v_price NUMBER(10,2) := 1000;
    v_tax NUMBER(10,2);
BEGIN
    v_tax := (v_price * c_tax_rate) / 100; -- Using the constant
    DBMS_OUTPUT.PUT_LINE('Tax Amount: ' || v_tax);
END;
```

Control Structures in PL/SQL:

Control structures in PL/SQL allow you to control the flow of execution within a block, procedure, or function. They include conditional statements, loops, and sequential control.

Conditional Control:

Conditional control structures execute different statements based on specific conditions.

IF-THEN : Executes a set of statements if a condition is TRUE.

Syntax:

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

Example:

```
DECLARE
    v_salary NUMBER := 50000;
BEGIN
    IF v_salary > 40000 THEN
        DBMS_OUTPUT.PUT_LINE('High Salary');
    END IF;
END;
```

IF-THEN-ELSE : Executes one set of statements if the condition is TRUE, and another set if it is FALSE.

Syntax:

```
IF condition THEN
    -- Statements if the condition is TRUE
ELSE
    -- Statements if the condition is FALSE
END IF;
```

Example:

```
DECLARE
    v_salary NUMBER := 30000;
BEGIN
    IF v_salary > 40000 THEN
        DBMS_OUTPUT.PUT_LINE('High Salary');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Low Salary');
    END IF;
END;
```

IF-THEN-ELSIF: Tests multiple conditions sequentially.

Syntax:

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

Example:

```
DECLARE
    v_marks NUMBER := 85;
BEGIN
    IF v_marks >= 90 THEN
        DBMS_OUTPUT.PUT_LINE('Grade: A');
    ELSIF v_marks >= 75 THEN
        DBMS_OUTPUT.PUT_LINE('Grade: B');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Grade: C');
    END IF;
END;
```

PL/SQL Program to find largest number among 3 numbers:

```
DECLARE
    num1 NUMBER; -- First number
    num2 NUMBER; -- Second number
    num3 NUMBER; -- Third number
    largest NUMBER; -- Variable to store the largest number
BEGIN
    -- Accept values for the numbers
    num1 := &num1; -- Replace &num1 with an input value during execution
    num2 := &num2; -- Replace &num2 with an input value during execution
    num3 := &num3; -- Replace &num3 with an input value during execution
    IF (num1 >= num2) AND (num1 >= num3) THEN
        largest := num1;
    ELSIF (num2 >= num1) AND (num2 >= num3) THEN
        largest := num2;
    ELSE
        largest := num3;
    END IF;
    DBMS_OUTPUT.PUT_LINE('The largest number is: ' || largest);
END;
```

Iterative Control (Loops)

Loops execute a block of statements repeatedly.

Basic Loop: Repeats a block of code indefinitely until an EXIT statement is reached.

Syntax:

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

Example:

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 5;
    END LOOP;
END;
```

WHILE Loop : Executes a block of code as long as a condition is TRUE.

Syntax:

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

Example:

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    WHILE v_counter <= 5 LOOP
        DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;
END;
```

FOR Loop : Executes a block of code for a fixed number of iterations.

Syntax:

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

Example:

```
BEGIN
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration: ' || i);
    END LOOP;
END;
```

PL/SQL Program to Find the Sum of the First 100 Numbers

This program calculates the sum of the first 100 natural numbers using a **loop** in PL/SQL.

```
DECLARE
    v_sum NUMBER := 0; -- Variable to store the sum
    v_counter NUMBER;  -- Loop counter
BEGIN
    FOR v_counter IN 1..100 LOOP
        v_sum := v_sum + v_counter; -- Adding the current number to the sum
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('The sum of the first 100 numbers is: ' || v_sum);
END;
```

Output:

```
The sum of the first 100 numbers is: 5050
```

Exception Handling in PL/SQL

In PL/SQL, exceptions are runtime errors that occur during the execution of a program. Exception handling ensures that these errors are captured and managed, preventing the program from crashing.

Structure of Exception Handling:

Exceptions are handled using the EXCEPTION block in a PL/SQL program.

Basic Syntax

```
DECLARE
    -- Variable declarations
BEGIN
    -- Executable statements
EXCEPTION
    WHEN exception_name THEN
        -- Actions to handle the exception
    WHEN OTHERS THEN
        -- Actions for all other exceptions
END;
```

Types of Exceptions

PL/SQL exceptions are categorized into **Predefined Exceptions and User-Defined Exceptions**

1. Predefined Exceptions: PL/SQL provides several predefined exceptions for common errors, such as division by zero or invalid data.

Exception	Description
ZERO_DIVIDE	Raised when dividing by zero.
NO_DATA_FOUND	Raised when a SELECT INTO query returns no rows.
TOO_MANY_ROWS	Raised when a SELECT INTO query returns multiple rows.
INVALID_NUMBER	Raised when a conversion to a number fails.
VALUE_ERROR	Raised for arithmetic, conversion, or size errors.
LOGIN_DENIED	Raised when login credentials are invalid.

Example: Handling ZERO_DIVIDE

```
DECLARE
    v_num NUMBER := 10;
    v_den NUMBER := 0;
    v_result NUMBER;
BEGIN
    v_result := v_num / v_den; -- Causes a division by zero error
EXCEPTION
    WHEN ZERO_DIVIDE THEN
```

```
            DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed.');
    END;
```

2. User-Defined Exceptions: You can define and handle custom exceptions using the EXCEPTION block.

Steps to Declare and Handle User-Defined Exceptions:

1. Declare the exception in the DECLARE block.
2. Raise the exception using the RAISE statement.
3. Handle the exception in the EXCEPTION block.

Example: User-Defined Exception

```
DECLARE
    insufficient_balance EXCEPTION; -- Declare exception
    v_balance NUMBER := 500;
    v_withdraw NUMBER := 1000;
BEGIN
    IF v_withdraw > v_balance THEN
        RAISE insufficient_balance; -- Raise exception
    END IF;
    DBMS_OUTPUT.PUT_LINE('Withdrawal successful.');
EXCEPTION
    WHEN insufficient_balance THEN
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient balance for withdrawal.');
END;
```

Cursors in PL/SQL:

A **cursor** in PL/SQL is a pointer or a handle to a query result set. It is used to retrieve, process, and traverse records in a query one row at a time.

Why Use Cursors?

1. To process individual rows returned by a query.
2. To handle query results that are too large to process all at once.
3. To perform operations on data row-by-row in a controlled manner.

Types of Cursors

Cursors in PL/SQL are broadly classified into:

1. **Implicit Cursors**
2. **Explicit Cursors**

1. Implicit Cursors: Automatically created by Oracle whenever a SELECT, INSERT, UPDATE, or DELETE statement is executed. Following are attributes of Implicit Cursors:

Attribute	Description
%FOUND	Returns TRUE if the DML statement affects rows.
%NOTFOUND	Returns TRUE if no rows are affected.
%ROWCOUNT	Returns the number of rows affected by the DML.
%ISOPEN	Always returns FALSE for implicit cursors.

Example of Implicit cursor with Attributes:

```
BEGIN
    UPDATE employees
    SET salary = salary + 1000
    WHERE department_id = 50;
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('No rows updated.');
    END IF;
  END;
```

2. Explicit Cursors: Defined explicitly in the PL/SQL block for queries that return more than one row. Explicit cursors offers more control over query execution and result processing.

Steps to Use Explicit Cursors:

1. **Declare** the cursor:
   CURSOR cursor_name IS query;
2. **Open** the cursor:
   OPEN cursor_name;
3. **Fetch** data from the cursor:
   FETCH cursor_name INTO variables;
4. **Close** the cursor:
   CLOSE cursor_name;

Example: Using an Explicit Cursor

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, first_name, salary FROM employees
        WHERE department_id = 50;
    v_emp_id employees.employee_id%TYPE;
    v_emp_name employees.first_name%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_emp_id, v_emp_name, v_salary;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('ID:'||v_emp_id ||',Name:'||v_emp_name ||',
Salary: ' || v_salary);
    END LOOP;
    CLOSE emp_cursor;
END;
```

3. Cursor and FOR Loop

Example: Cursor FOR Loop

```
DECLARE
    CURSOR emp_cursor IS
        SELECT employee_id, first_name, salary
        FROM employees
        WHERE department_id = 50;
BEGIN
    FOR emp_rec IN emp_cursor LOOP
        DBMS_OUTPUT.PUT_LINE('ID: ' || emp_rec.employee_id || ', Name: ' ||
emp_rec.first_name || ', Salary: ' || emp_rec.salary);
    END LOOP;
END;
```

Parameterizing Cursors

Cursors can accept parameters to make them more dynamic.

Example: Parameterized Cursor

```
DECLARE
    CURSOR dept_cursor(p_dept_id NUMBER) IS
        SELECT employee_id, first_name
        FROM employees
        WHERE department_id = p_dept_id;
    v_emp_id employees.employee_id%TYPE;
    v_emp_name employees.first_name%TYPE;
BEGIN
    OPEN dept_cursor(50); -- Pass department ID as parameter
    LOOP
        FETCH dept_cursor INTO v_emp_id, v_emp_name;
        EXIT WHEN dept_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('ID: ' || v_emp_id || ', Name: ' || v_emp_name);
    END LOOP;
    CLOSE dept_cursor;
END;
```

Advantages of Using Cursors

1. Allows row-by-row processing for complex operations.
2. Useful for handling queries returning multiple rows.
3. Parameterized cursors enable dynamic execution.

Stored Procedure in PL/SQL

A **stored procedure** is a named PL/SQL block that performs a specific task and can be invoked multiple times. Stored procedures are stored in the database, making them reusable and efficient.

Advantages of Stored Procedures

1. **Improved Performance**: Reduces client-server communication overhead.
2. **Security**: Code is stored and executed on the server, ensuring sensitive logic is not exposed.
3. **Reusability**: Common tasks can be centralized in one place.
4. **Scalability**: Simplifies application logic by moving processing to the database server.

Syntax of a Stored Procedure:

```
CREATE OR REPLACE PROCEDURE procedure_name (
   parameter1 IN/OUT/IN OUT data_type,
   parameter2 IN/OUT/IN OUT data_type,
   parameter3 IN/OUT/IN OUT data_type
)
AS
   -- Local variable declarations
BEGIN
   -- PL/SQL block with executable statements
EXCEPTION
   -- Exception handling
END procedure_name;
```

Types of Parameters of Procedure in PL/SQL

Parameters in PL/SQL procedures allow you to pass data into and out of the procedure. Parameters can be classified into three types:

1. IN Parameters : It is the default parameter type. It is used to pass input values to the procedure. The value is read-only inside the procedure hence cannot be modified within the procedure.

Example: Using IN Parameter

```
CREATE OR REPLACE PROCEDURE greet_user(p_name IN VARCHAR2)
AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, ' || p_name || '!');
END;
```

Calling the Procedure

```
BEGIN
    greet_user('John');
END;
```

Output:

Hello, John!

2. OUT Parameters: It is used to return values (output) to the calling program. It acts as a writable variable within the procedure. It must be assigned a value in the procedure before it is returned.

Example: Using OUT Parameter

```
CREATE OR REPLACE PROCEDURE calculate_square(p_number IN NUMBER,
p_result OUT NUMBER)
AS
BEGIN
    p_result := p_number * p_number;
END;
```

Calling the Procedure

```
DECLARE
    v_result NUMBER;
BEGIN
    calculate_square(5, v_result);
    DBMS_OUTPUT.PUT_LINE('Square: ' || v_result);
END;
```

Output:

Square: 25

3. IN OUT Parameters: It is used to pass a value to the procedure, modify it, and return the updated value. It acts as both an input and output variable. The procedure can read and modify the value.

Example: Using IN OUT Parameter

```
CREATE OR REPLACE PROCEDURE update_salary(p_salary IN OUT NUMBER,
p_increment IN NUMBER)
AS
BEGIN
    p_salary := p_salary + p_increment;
END;
```

Calling the Procedure

```
DECLARE
    v_salary NUMBER := 5000;
BEGIN
    update_salary(v_salary, 500);
    DBMS_OUTPUT.PUT_LINE('Updated Salary: ' || v_salary);
END;
```

Output:

Updated Salary: 5500

Deleting a Stored Procedure in PL/SQL

In PL/SQL, you can delete a stored procedure from the database using the DROP PROCEDURE statement.

Function in PL/SQL

A **Function** or **Stored Function** in PL/SQL is similar to a stored procedure but differs in one key aspect: a function **must return a value**. Functions are typically used to perform calculations or transformations and return the result to the calling program.

Features of a Stored Function

1. **Returns a Value**: A function always returns a single value using the RETURN statement.
2. **Usage in SQL**: Functions can be used in SQL queries (unlike procedures).
3. **Modularity**: Helps in modularizing and reusing logic.
4. **Call Context**: Can be invoked directly in PL/SQL blocks, queries, or expressions.

Syntax of a Stored Function

```
CREATE OR REPLACE FUNCTION function_name (
    parameter1 IN data_type,
    parameter2 IN data_type
)
RETURN return_data_type
AS
    -- Local variable declarations
BEGIN
    -- Executable statements
    RETURN value; -- Returning a value
EXCEPTION
    -- Exception handling
END function_name;
```

Example: A Simple Stored Function

Function to Calculate Square of a Number

```
CREATE OR REPLACE FUNCTION calculate_square(p_number IN NUMBER)
RETURN NUMBER
AS
BEGIN
    RETURN p_number * p_number;
END calculate_square;
```

Calling the Function in PL/SQL

```
DECLARE
    v_square NUMBER;
BEGIN
    v_square := calculate_square(5);
    DBMS_OUTPUT.PUT_LINE('Square: ' || v_square);
END;
```

Output:

```
Square: 25
```

Differences Between Procedure and Function

Feature	Function	Procedure
Return Value	Always returns a single value.	Does not return a value (can use OUT parameters).
Usage in SQL	Can be used in SQL statements.	Cannot be used in SQL statements.
Call Context	Used in expressions or assignments.	Called independently in PL/SQL blocks.
Output	Returns a value via RETURN.	Outputs via OUT or IN OUT parameters.

Deleting a Stored Function in PL/SQL

In PL/SQL, you can delete a stored function from the database using the DROP FUNCTION statement.

PL/SQL TRIGGERS:

In PL/SQL, **triggers** are automatically executed in response to certain events on a table or view. In PL/SQL, triggers can be defined as either **row-level** or **statement-level** based on how they operate when a DML (Data Manipulation Language) operation is performed.

1. Row-Level Triggers: A **row-level trigger** executes once for each row affected by the DML operation (INSERT, UPDATE, DELETE). When a DML statement affects multiple rows, the trigger is fired for each row individually.

Characteristics of Row-Level Triggers:

- **Fires once per row**: It is executed for every row affected by the DML operation.
- **Uses FOR EACH ROW**: This clause indicates that the trigger will operate on each row individually.
- **Can access: NEW and :OLD**: These special record variables hold the new and old values for each row.
- **Usage**: Often used when you need to perform actions on each row individually, such as validation or data modification.

Syntax of Row-Level Trigger

```
CREATE OR REPLACE TRIGGER trigger_name
   BEFORE | AFTER {INSERT | UPDATE | DELETE} ON table_name
   FOR EACH ROW
BEGIN
   -- Trigger logic
   -- Access :NEW for new row data (in INSERT/UPDATE)
   -- Access :OLD for old row data (in UPDATE/DELETE)
END;
```

Example: Row-Level Trigger for Validation

Let's say we want to create a trigger that checks the salary of an employee before insertion. If the salary is less than $500, the insertion should be prevented.

```
CREATE OR REPLACE TRIGGER validate_salary
   BEFORE INSERT ON employees
   FOR EACH ROW
BEGIN
   IF :NEW.salary < 500 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Salary must be at least $500');
   END IF;
END;
```

In the above program :NEW.salary refers to the new value of the salary column being inserted. If the salary is less than $500, the trigger raises an error and prevents the insertion.

2. Statement-Level Triggers

A **statement-level trigger** executes **once per statement**, no matter how many rows are affected. It does not iterate over the rows and cannot access :NEW and :OLD because it doesn’t deal with individual rows. Statement-level triggers are often used for actions that should happen once regardless of how many rows the DML statement affects.

Characteristics of Statement-Level Triggers:

- **Fires once per statement**: It is executed only once for the entire DML operation, regardless of how many rows are affected.
- **No FOR EACH ROW**: No need to specify FOR EACH ROW because the trigger works on the entire statement.
- **Cannot access :NEW or :OLD**: These special variables are not available in statement-level triggers.

Syntax of Statement-Level Trigger

```
CREATE OR REPLACE TRIGGER trigger_name
   BEFORE | AFTER {INSERT | UPDATE | DELETE} ON table_name
BEGIN
   -- Trigger logic
END;
```

Difference Between Row-Level and Statement-Level Triggers:

Feature	Row-Level Trigger	Statement-Level Trigger
Trigger Execution	Executes once for each row affected.	Executes once per DML statement.
Access to :NEW and :OLD	Yes, can use :NEW and :OLD to access row data.	No, cannot use :NEW and :OLD.
Trigger Timing	Can be BEFORE or AFTER for individual rows.	Can be BEFORE or AFTER for the whole statement.
Use Case	When actions depend on individual rows, like validation or calculation.	When actions should happen once per statement, like logging or auditing.
Performance Considerations	More resource-intensive if many rows are affected, as it executes for each row.	More efficient for bulk operations, but may not be suitable for row-specific actions.

In PL/SQL, you can **drop a trigger** using the `DROP TRIGGER` statement.