PL/SQL

PL/SQL (Procedural Language/SQL) is Oracle's extension of SQL that adds procedural features like loops, conditions, and error handling. It allows developers to write powerful programs that combine SQL queries with logic to control how data is processed. With PL/SQL, complex operations, calculations, and error handling can be performed directly within the Oracle database, making data manipulation more efficient and flexible.

PL/SQL allows developers to:

- Execute SQL queries and DML commands inside procedural blocks.
- Define variables and perform complex calculations.
- Create reusable program units, such as procedures, functions, and triggers.
- Handle exceptions, ensuring the program runs smoothly even when errors occur.

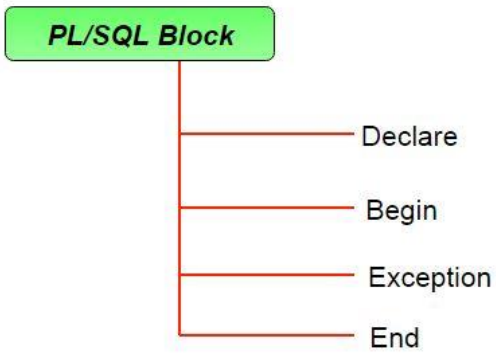
Key Features of PL/SQL

PL/SQL brings the benefits of procedural programming to the relational database world. Some of the most important features of PL/SQL include:

- **Block Structure:** PL/SQL can execute a number of queries in one block using single command.
- **Procedural Constructs:** One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
- **Error Handling:** PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
- **Reusable Code**: Create stored procedures, functions, triggers, and packages, which can be executed repeatedly.
- **Performance**: Reduces network traffic by executing multiple SQL statements within a single block

Structure of PL/SQL Block

PL/SQL extends SQL by adding constructs found in **procedural languages**, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



Typically, each block performs a logical action in the program. A block has the following structure:

DECLARE

declaration statements;

BEGIN

executable statements

EXCEPTIONS

exception handling statements

END;

PL/SQL code is written in blocks, which consist of three main sections:

- Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
- Execution section starts with **BEGIN** and ends with **END** keyword. This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.
- Exception section starts with **EXCEPTION** keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

PL/SQL Identifiers

In PL/SQL, identifiers are names used to represent various program elements like variables, constants, procedures, cursors, triggers etc. These identifiers allow you to store, manipulate, and access data throughout your PL/SQL code.

1. Variables in PL/SQL

Like several other programming languages, variables in PL/SQL must be declared prior to its use. A variable is like a container that holds data during program execution. Each variable must have a valid name and a specific **data type**.

Syntax for declaration of variables:

variable_name datatype [NOT NULL := value];

- **variable_name**: The name of the variable.
- **datatype**: The data type of the variable (e.g., INTEGER, VARCHAR2).
- **NOT NULL**: This optional constraint means the variable cannot be left empty.
- **:= value**: This optional assignment assigns an initial value to the variable.

Declare a Variable in PL/SQL

When writing PL/SQL code it is important to declare variables properly to **store** and manipulate data effectively. Variables act as containers for values and enable various operations on the stored data.

Variables in PL/SQL are declared using the **DECLARE** keyword within an anonymous block or a named program unit such as a procedure, function, or package.

Common Methods for Declaring Variables in PL/SQL

The below methods are used to **declare a variable in PL/SQL** are as follows:

Table of Content

1. Using Declare Variables in PL/SQL

To declare a variable in PL/SQL, use the **DECLARE** keyword followed by the variable name and its data type. Optionally, you can also assign an initial value to the variable using the '**:=**' operator.

Syntax:

```
DECLARE
```

```
variable_name datatype := initial_value;
```

here,

- **variable_name:** It is the name of the variable.
- **datatype:** It is the data type of the variable.
- **:= initial_value:** It is an optional assignment of an initial value to the variable.

Example:

```
DECLARE
  name VARCHAR2(20) := 'GeeksForGeeks';
BEGIN
  DBMS_OUTPUT.PUT_LINE(name);
END;
```

2. Using Initializing Variables in PL/SQL

In this method Variables can be initialized in two ways either during declaration or later in the code.

a. Initializing during declaration

Variables can be assigned values when declared, as shown below:

Syntax:

```
DECLARE
  my_variable NUMBER := value;
BEGIN
  -- PL/SQL code
END;
```

Example:

```
DECLARE
  name VARCHAR2(20) := 'GeeksForGeeks';
BEGIN
  DBMS_OUTPUT.PUT_LINE(name);
END;
```

b. Initialization After Declaration

You can also assign a value to a variable later in the code using the := operator.

Syntax:

```
DECLARE
  my_variable NUMBER;
BEGIN
  my_variable := value;
END;
```

Example:

```
DECLARE
   num1 NUMBER;
   num2 NUMBER;
   result NUMBER;
BEGIN
   num1 := 5;
   num2 := 3;
   result := num1 + num2;
   DBMS_OUTPUT.PUT_LINE('Sum: ' || result);
END;
```

3. Using Variable Scope in PL/SQL

Variable scope determines where a variable can be accessed within a program. In PL/SQL, variable scope can be either local or global.

- **Local Variables:** Declared within a block or subprogram, accessible only inside that block or subprogram.
- **Global Variables:** Declared in the outermost block and accessible by nested blocks.

Example:

```
DECLARE
  global_var NUMBER; -- global variable
BEGIN
  -- PL/SQL code using global_var
  DECLARE
    local_var NUMBER; -- local variable
  BEGIN
    -- PL/SQL code using local_var and global_var
  END;
```

```
  -- Here you can't access local_var
END;
```

Explanation: The **global_var** can be accessed throughout the entire program, while the **local_var** is only accessible within the inner block

4. Using Variable Attributes (%TYPE and %ROWTYPE)

PL/SQL provides two powerful attributes, **%TYPE** and **%ROWTYPE**, which allow variables to inherit data types from existing columns or entire rows**.**

- **%TYPE:** It defines a variable with the same data type as another variable or column.
- **%ROWTYPE:** It defines a record with the same structure as a table or cursor.

1. **Using %TYPE Attribute**

In this example, we have declared a variable **salary_var** using **%TYPE** to match the **data type** of the **salary** column in the **employees table** then we have assigned a value to **salary_var** and displayed the assigned value using **DBMS_OUTPUT.PUT_LINE**.

```
DECLARE
  salary_var employees.salary%TYPE;
BEGIN
  -- Assign a value to the variable
  salary_var := 70000;

  -- Display the assigned value
  DBMS_OUTPUT.PUT_LINE('Assigned Salary: ' || salary_var);
END;
```

2. Using %ROWTYPE Attribute

In this example, We have declared a record variable **employee_record** using **%ROWTYPE** to match the structure of the **employees table** and fetched the data from the **employees** table into the **employee_record** variable using a **SELECT INTO statement** then we have displayed the retrieved data from the **employee_record** using **DBMS_OUTPUT.PUT_LINE.**

```
DECLARE
  employee_record employees%ROWTYPE;
BEGIN
  -- Fetch data from the table into the record variable
```

```
  SELECT * INTO employee_record FROM employees WHERE employee_id = 2;

  -- Display the retrieved data
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || employee_record.employee_id);
  DBMS_OUTPUT.PUT_LINE('First Name: ' || employee_record.first_name);
  DBMS_OUTPUT.PUT_LINE('Last Name: ' || employee_record.last_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || employee_record.salary);
END;
```

Executable Commands Section in a PL/SQL Block

PL/SQL (Procedural Language/Structured Query Language) is an extension of SQL used in Oracle databases to write procedural code such as loops, conditions, and exception handling along with SQL statements. A PL/SQL block is the basic unit of execution and consists of three main sections: the **declaration section**, the **executable section**, and the **exception-handling section**. Among these, the **executable commands section** is the most important part because it contains the actual logic and instructions that the program performs.

Structure of a PL/SQL Block

A typical PL/SQL block has the following structure:

```
DECLARE
   -- Declaration section (optional)
   -- Variable, constant, and cursor declarations
BEGIN
   -- Executable section (mandatory)
   -- Statements that perform actions
EXCEPTION
   -- Exception-handling section (optional)
END;
/
```

Out of these sections, **only the executable section is mandatory**, as it defines what the block will actually do.

Executable Section

The **executable section** begins with the keyword BEGIN and ends with the keyword END. It contains all the **executable statements** that perform the program's main tasks, such as data processing, arithmetic operations, conditional execution, loops, and database manipulation through SQL commands.

1. SQL Statements

Within the executable section, SQL statements such as SELECT INTO, INSERT, UPDATE, and DELETE are used to interact with the database. For example:

```
BEGIN
  INSERT INTO employees (emp_id, emp_name) VALUES (101, 'John');
  UPDATE employees SET salary = salary + 1000 WHERE emp_id = 101;
END;
/
```

These statements allow the PL/SQL program to modify and retrieve data from database tables.

---

2. Procedural Statements

PL/SQL extends SQL by providing procedural constructs such as:

- **Conditional statements** (IF...THEN...ELSE) for decision-making.
- **Loops** (FOR, WHILE, LOOP) for repetition.
- **Assignments** to variables using the := operator.

Example:

```
BEGIN
```

```
  IF salary < 5000 THEN
    salary := salary + 500;
  ELSE
    salary := salary + 200;
  END IF;
END;
/
```

These constructs make PL/SQL more powerful than standard SQL by allowing logical control over program flow.

---

3. Function and Procedure Calls

The executable section can call **procedures** and **functions** defined elsewhere in the database or within the same block. For example:

```
BEGIN
  calculate_bonus(emp_id => 101);
END;
/
```

This enables modular programming and code reuse.

---

4. Input and Output Operations

To display or check the results of execution, PL/SQL uses the **DBMS_OUTPUT.PUT_LINE** procedure. This is often used in the executable section for debugging or displaying messages:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Employee record inserted successfully.');
END;
/
```

Importance of the Executable Section

The executable section is crucial because it represents the **action part** of the PL/SQL block. Without it, the block would not perform any task. It integrates SQL with procedural features, allowing developers to:

- Process and manipulate database data.
- Implement business logic inside the database.
- Handle complex decision-making using control structures.
- Automate repetitive database operations.

Exception Handling in PL/SQL

An exception is an error which disrupts the normal flow of program instructions. PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

Syntax:

```
DECLARE
-- Declaration statements;
BEGIN
        -- SQL statements;
-- Procedural statements;
EXCEPTION
-- Exception handling statements;
END;
```

There are two types of exceptions defined in PL/SQL

1. User defined exception.
2. System defined exceptions.

Types of Exception Handling

There are two types of exceptions defined in PL/SQL :

1. System-Defined Exceptions

These are predefined exceptions that occur when Oracle rules or constraints are violated. They include NO_DATA_FOUND, ZERO_DIVIDE, TOO_MANY_ROWS, etc.

```
DECLARE
a NUMBER := 10;
b NUMBER := 0;
c NUMBER;
BEGIN
c := a / b;
-- Division by zero  DBMS_OUTPUT.PUT_LINE('Result: ' || c);
EXCEPTION
WHEN ZERO_DIVIDE THEN
DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed.');
END;
```

Output:

```
Error: Division by zero is not allowed.
```

In this example:

- Variable Declaration: a = 10, b = 0.
- Program tries to divide a / b, which causes an error.
- The EXCEPTION block catches the ZERO_DIVIDE error.
- It prints: "Error: Division by zero is not allowed.

1. Named System Exception:

These exceptions have predefined names such as ACCESS_INTO_NULL, DUP_VAL_ON_INDEX, LOGIN_DENIED, etc**.**

2. Unnamed System Exception:

Unnamed system exceptions are predefined by Oracle, but they don’t have a specific name like NO_DATA_FOUND. They occur less frequently and are identified by Oracle error codes **.**

2. User Define Exception

User-defined exceptions are custom exceptions created by the programmer to handle specific business logic errors that are not covered by Oracle's predefined exceptions. These exceptions must be declared explicitly and are raised using the RAISE keyword.

Syntax:

```
DECLARE
  exception_name EXCEPTION;  -- Declaration of user-defined exception
BEGIN
-- Logic
```

```
IF condition THEN
RAISE exception_name;  -- Raising the exception explicitly
END IF;
EXCEPTION
 WHEN exception_name THEN      -- Handling code
    DBMS_OUTPUT.PUT_LINE('Exception handled');
END;
```

PL/SQL Triggers

PL/SQL triggers are block structures and predefined programs **invoked automatically** when some event occurs. They are stored in the database and **invoked repeatedly** in a particular scenario. There are two states of the triggers, they are **enabled** and **disabled**. When the trigger is created it is enabled. **CREATE TRIGGER** statement creates a trigger. A triggering event is specified on a table, a view, a schema, or a database. **BEFORE** and **AFTER** are the trigger Timing points.

They are associated with response-based events such as a

- **Database Definition Language** statements such as CREATE, DROP or ALTER.
- **Database Manipulation Language** statements such as UPDATE, INSERT or DELETE.
- Database operations such as LOGON, LOGOFF, STARTUP, and SHUTDOWN .

Types of Triggers in PL/SQL with Examples

Based on a variety of factors, triggers in PL/SQL may be divided into distinct categories. To help you understand, let's go through each type of trigger in PL SQL with examples.

1. Row-Level Triggers

A row-level trigger occurs once for each row that a triggering event affects.

A. Before Row Triggers

This trigger occurs before the insertion, update, or deletion of a row. It may be used to change the values of the currently processed row.

B. After-Row Triggers

Following an INSERT, UPDATE, or DELETE operation on a row, this type of trigger occurs. It may be used to conduct actions based on the row's modifications.

2. Statement-Level Triggers

No matter how many rows are impacted, a trigger event on a table always fires a statement-level trigger.

A. Before Statement Triggers

This trigger occurs before the execution of a SQL query. It can be used to take actions or validations before processing the statement.

Example:

```
CREATE OR REPLACE TRIGGER before_statement_trigger
BEFORE INSERT ON student
BEGIN
  -- Perform some validation or action before the insert statement is executed
  IF : NEW.fees < 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Fees cannot be negative.');
  END IF;
END;
```

B. After Statement Triggers

Upon execution of a SQL statement, this trigger occurs. It can be used to conduct actions based on the statement's overall outcome.

Example:

```
CREATE OR REPLACE TRIGGER after_statement_trigger
AFTER INSERT OR DELETE ON student
BEGIN
  -- Update the student count in a separate table
  UPDATE student_count_table
  SET count = (SELECT COUNT(*) FROM student);
END;
```

3. Database-Level Triggers

No matter which user or application provides the statement, database triggers in PL SQL are specified on a table, saved in the corresponding database, and performed as a result of an INSERT, UPDATE, or DELETE statement being made against a table.

A. Startup Triggers

This trigger activates after the initialization of the database. It can be used to undertake setup activities or to carry out particular operations during startup.

Enabling or Disabling an Individual Trigger:

To enable or disable a single trigger, use the ALTER TRIGGER statement followed by the trigger name and either ENABLE or DISABLE.

Code

```
ALTER TRIGGER trigger_name ENABLE;
ALTER TRIGGER trigger_name DISABLE;
```

Example:

Code

```
ALTER TRIGGER trg_employee_audit ENABLE;
ALTER TRIGGER trg_employee_audit DISABLE;
```

2. Enabling or Disabling All Triggers on a Table:

To enable or disable all triggers associated with a particular table, use the ALTER TABLE statement followed by the table name and either ENABLE ALL TRIGGERS or DISABLE ALL TRIGGERS.

Code

```
ALTER TABLE table_name ENABLE ALL TRIGGERS;
ALTER TABLE table_name DISABLE ALL TRIGGERS;
```

Example:

Code

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE employees DISABLE ALL TRIGGERS;
```

3. Creating a Disabled Trigger:

When creating a new trigger, you can specify that it should be created in a disabled state by including the DISABLE keyword in the CREATE TRIGGER statement. This can be useful for testing or when you don't want the trigger to fire immediately after creation.

Code

```
CREATE OR REPLACE TRIGGER trg_new_feature
BEFORE INSERT ON new_table
DISABLE
BEGIN
  -- Trigger logic here
END;
/
```

Replacing Objects (using OR REPLACE)

The OR REPLACE clause is used with CREATE statements (e.g., CREATE PROCEDURE, CREATE FUNCTION, CREATE PACKAGE, CREATE VIEW, CREATE TRIGGER) to modify an existing object without explicitly dropping and re-creating it. Syntax Example (Procedure).

Dropping Objects (using DROP)

The DROP statement is used to permanently remove an object from the database.

Syntax Examples:

Procedure:

Code

```
DROP PROCEDURE my_procedure;
```

package.

Code

```
DROP PACKAGE my_package;
```

view.

Code

```
DROP VIEW my_view;
```

trigger.

Code

```
DROP TRIGGER my_trigger;
```

Procedures in PL/SQL

PL/SQL procedures are reusable code blocks that perform specific actions or logic within a database environment. They consist of two main components such as the procedure header which defines the procedure name & optional parameters and the procedure body which contains the executable statements implementing the desired business logic.

The procedure contains two parts:

Procedure Header

- The procedure header includes the procedure name and optional parameter list.
- It is the first part of the procedure and specifies the name and parameters

Procedure Body

- The procedure body contains the executable statements that implement the specific business logic.
- It can include declarative statements, executable statements, and exception-handling statements

Create Procedures in PL/SQL

To create a procedure in PL/SQL, use the **CREATE PROCEDURE** command:

Syntax:

```
CREATE PROCEDURE procedure_name
@Parameter1 INT,
@Parameter2 VARCHAR(50) = NULL,
@ReturnValue INT OUTPUT
AS
BEGIN
END
GO
```

Parameters in Procedures

In PL/SQL, parameters are used to pass values into procedures. There are three types of parameters used in procedures:

IN parameters

- Used to pass values into the procedure
- Read-only inside the procedure

- Can be a variable, literal value, or expression in the calling statement.

OUT parameters

- Used to return values from the procedure to the calling program
- Read-write inside the procedure
- Must be a variable in the calling statement to hold the returned value

IN OUT parameters

- Used for both passing values into and returning values from the procedure
- Read-write inside the procedure
- Must be a variable in the calling statement

Modify Procedures in PL/SQL

To modify an existing procedures in PL/SQL use the **ALTER PROCEDURE command**:

Syntax

ALTER PROCEDURE Syntax is:

SET ANSI_NULLS ON

SET QUOTED_IDENTIFIER ON

GO

ALTER PROCEDURE procedure_name

@Parameter1 INT,

@Parameter2 VARCHAR(50) = NULL,

@ReturnValue INT OUTPUT

AS

BEGIN

-- Query

END

GO

Drop Procedure in PL/SQL

To drop a procedure in PL/SQL use the **DROP PROCEDURE** command

Syntax:

DROP PROCEDURE procedure_name

PL/SQL DROP PROCEDURE Example

In this example, we will delete a procedure in PL/SQL

DROP PROCEDURE GetStudentDetails

Functions in PL/SQL

A PL/SQL function is a named, self-contained block of code that performs a specific task and always returns a single value. Functions can accept zero or more input parameters and are typically used to compute and return a result. They can be called from SQL statements, other PL/SQL blocks, or even within packages.

Syntax Example:

Code

```
CREATE OR REPLACE FUNCTION calculate_square (p_number IN NUMBER)
RETURN NUMBER
IS
  v_square NUMBER;
BEGIN
  v_square := p_number * p_number;
  RETURN v_square;
END;
/
```

PL/SQL Packages

PL/SQL packages are a way to organize and encapsulate related **procedures**, **functions**, **variables**, **triggers**, and other PL/SQL items into a single item. Packages provide a modular approach to write and maintain the code. It makes it easy to manage large codes.

A package is compiled and then stored in the database, which then can be shared with many applications. The package also has specifications, which declare an item to be public or private. Public items can be referenced from outside of the package.

A PL/SQL package is a collection of related **Procedures**, **Functions**, **Variables**, and other elements that are grouped for **Modularity** and **Reusability**.

Key Benefits of Using PL/SQL Packages

The needs of the Packages are described below:

- **Modularity**: Packages provide a modular structure, allowing developers to organize and manage code efficiently.
- **Code Reusability**: Procedures and functions within a package can be reused across multiple programs, reducing redundancy.

- **Private Elements**: Packages support private procedures and functions, limiting access to certain code components.
- **Encapsulation**: Packages encapsulate related logic, protecting internal details and promoting a clear interface to other parts of the code.

Structure of a PL/SQL Package

A **PL/SQL** package consists of two parts:

1. A package Specification
2. A package Body

1. Package Specification

The package specification declares the public interface of the package. It includes declarations of **procedures**, **functions**, **variables**, **cursors**, and other constructs that are meant to be accessible from outside the package. The specification is like a header file that defines what a package can do.

***Example of Package Specification**:*

```
CREATE OR REPLACE PACKAGE my_package AS
  PROCEDURE my_procedure(p_param1 NUMBER);
  FUNCTION calculate_sum(x NUMBER, y NUMBER) RETURN NUMBER;
  -- Other declarations...
END my_package;
```

2. Package Body

The package body contains the implementation of the details of the package. It includes the coding of the procedures or functions which are decalared in the package specification. The body can also contain private **variables** and **procedures** that are not exposed to outside the code.

Example of Package Body:

```
CREATE OR REPLACE PACKAGE BODY my_package AS
  PROCEDURE my_procedure(p_param1 NUMBER) IS
  BEGIN
    -- Implementation code...
  END my_procedure;

  FUNCTION calculate_sum(x NUMBER, y NUMBER) RETURN NUMBER IS
  BEGIN
    -- Implementation code...
```

```
  END calculate_sum;
  -- Other implementation details...
END my_package;
```

Once your create your package in above two steps, you can use it in PL/SQL codes. This allows for modular programming, code reuse, and better maintenance of the the code base.

Using Oracle PL/SQL Packages in Code

```
DECLARE
  result NUMBER;
BEGIN
  -- Call a procedure from the package
  my_package.my_procedure(42);

  -- Call a function from the package
  result := my_package.calculate_sum(10, 20);

  -- Other code...
END;
```