# Unit 9-PL/SQL

## Subject Code: 303105203

**Prof. S.W.Thakare**
Assistant Professor,
Computer science & Engineering

Parul®
University

DIGITAL LEARNING CONTENT

# CHAPTER-8

# PL/SQL Concept

# PL/SQL Basic

- 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.
- PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

PL/SQL is a completely portable, high-performance transaction-processing language.

PL/SQL provides a built-in, interpreted and OS independent programming environment.

PL/SQL can also directly be called from the command-line **SQL*Plus interface**.

Direct call can also be made from external programming language calls to database.

# PL/SQL Basic

**Features of PL/SQL**

PL/SQL has the following features

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

# PL/SQL Basic

**Advantages of PL/SQL**

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL.
- Static SQL supports DML operations and transaction control from PL/SQL block.
- In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data

Parul® University

DIGITAL LEARNING CONTENT

# PL/SQL Basic

**PL/SQL-Basic Syntax**

PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts

**Declarations**

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 Commands**

This section is enclosed between the keywords 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

Run PL/SQL in online Portal:

https://youtu.be/7QpjOmc7VSs?si=rlGVHX0AvKrbirnt

How to install PL/SQL

https://youtu.be/Sp_xufmKBsA?si=05aRNs79B1GN0BY7

# PL/SQL Basic

This section starts with the keyword EXCEPTION. This optional section contains exception(s) that handle errors in the program.

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling>
END;
```

The 'Hello World' Example

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

# PL/SQL Basic

**The PL/SQL Identifiers**

- PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words.
- The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

**The PL/SQL Comments**

- The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler.
- The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

# PL/SQL Basic

```
DECLARE
-- variable declaration
message varchar2(20):= 'Hello, World!';

BEGIN
/*
* PL/SQL executable statement(s)
*/
dbms_output.put_line(message);

END;
/
```

When the above code is executed at the SQL prompt, it produces the following result − Hello World

PL/SQL procedure successfully completed.


# PL/SQL Basic

**PL/SQL-Data Types**

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage
format, constraints, and a valid range of values.

**PL/SQL–Variables**

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive.

**Variable Declaration in PL/SQL**

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is −

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

# PL/SQL Basic

**Initializing Variables in PL/SQL**

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

The **DEFAULT** keyword

The assignment operator

```
DECLARE
a integer := 10;
b integer := 20;
c integer;
f real;
BEGIN
c := a + b;
dbms_output.put_line('Value of c: ' || c);
f := 70.0/3.0;
dbms_output.put_line('Value of f: ' || f);
END;
```

# PL/SQL Basic

**Variable Scope in PL/SQL**

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope

- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package./

**DECLARE**

-- Global variables

num1 number := 95;

num2 number := 85;

# PL/SQL Basic

```
BEGIN
dbms_output.put_line('Outer Variable num1: ' || num1);
dbms_output.put_line('Outer Variable num2: ' || num2);
DECLARE
-- Local variables
num1 number := 195;
num2 number := 185;
BEGIN
dbms_output.put_line('Inner Variable num1: ' || num1);
dbms_output.put_line('Inner Variable num2: ' || num2);
END;
END;
/
```

# PL/SQL Cursors

**PL/SQL -Cursors**

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement.

The set of rows the cursor holds is referred to as the active set.

There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit cursors:

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.
- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE

# PL/SQL Cursors

- In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**.
- The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

| S.No | Attribute & Description |
|---|---|
| 1 | %FOUND: Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| 2 | %NOTFOUND: The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| 3 | %ISOPEN: Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| 4 | %ROWCOUNT: Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

# PL/SQL Cursors

Example :We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

# PL/SQL Cursors

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
```

When the above code is executed at the SQL prompt, it produces the following result

# PL/SQL Cursors

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated −

Select * from customers;

| ID | NAME | AGE | ADDRESS | SALARY |

| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |

| 2 | Khilan | 25 | Delhi | 2000.00 |

| 3 | kaushik | 23 | Kota | 2500.00 |

| 4 | Chaitali | 25 | Mumbai | 7000.00 |

| 5 | Hardik | 27 | Bhopal | 9000.00 |

| 6 | Komal | 22 | MP | 5000.00 |


# PL/SQL Cursors

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

# PL/SQL Cursors

**Declaring the Cursor**

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

**Opening the Cursor**

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers; |
```

# PL/SQL Cursors

**Fetching the Cursor**

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

**Closing the Cursor**

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

# PL/SQL Cursors

**Example**

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
```

# PL/SQL Cursors

```
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

# PL/SQL Cursors

When the above code is executed at the SQL prompt, it produces the following result −

1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

PL/SQL procedure successfully completed.

# PL/SQL Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)

A database definition (DDL) statement (CREATE, ALTER, or DROP).

A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

**Benefits of Triggers**

- Triggers can be written for the following purposes −
- Generating some derived column values automatically
- Enforcing referential integrity

# PL/SQL Triggers

- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

**Creating Triggers**

The syntax for creating a trigger is −

CREATE [OR REPLACE ] TRIGGER trigger_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col_name]

ON table_name

# PL/SQL Triggers

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

  Declaration-statements

BEGIN

  Executable-statements

EXCEPTION

  Exception-handling-statements

END;

# PL/SQL Triggers

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

  Declaration-statements

BEGIN

  Executable-statements

EXCEPTION

  Exception-handling-statements

END;

# PL/SQL Triggers

Where,

CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the trigger_name.

{BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.

{INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.

[OF col_name] − This specifies the column name that will be updated.

[ON table_name] − This specifies the name of the table associated with the trigger.

[REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

[FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

# PL/SQL Triggers

WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example: To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

Select * from customers;

| ID | NAME | AGE | ADDRESS | SALARY |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

# PL/SQL Triggers

The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
```

# PL/SQL Triggers

```
dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

Trigger created.

# PL/SQL Stored Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- Header: The header contains the name of the procedure and the parameters or variables passed to the procedure.
- Body: The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

**How to pass parameters in procedure:**

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

# PL/SQL Stored Procedure

- IN parameters: The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
- OUT parameters: The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
- INOUT parameters: The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

# PL/SQL Stored Procedure

- A stored procedure in PL/SQL is nothing but a series of declarative SQL statements which can be stored in the database catalogue.
- A procedure can be thought of as a function or a method. They can be invoked through triggers, other procedures, or applications on Java, PHP etc.

**Advantages:**

•If a procedure is being called frequently in an application in a single connection, then the result in performance improve the application.

•They reduce the traffic between the database and the application .

•They add to code reusability, similar to how functions and methods work in other languages such as C/C++ and Java.

# PL/SQL Stored Procedure

**Disadvantages:**

•Stored procedures can cause a lot of memory usage.

•MySQL does not provide the functionality of debugging the stored procedures.

Syntax to create a stored procedure

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- Comments --
CREATE PROCEDURE procedure_name
  = ,
  = ,
  =
```

# PL/SQL Stored Procedure

```
AS
BEGIN
-- Query --
END
GO
```

Example:

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE PROCEDURE GetStudentDetails
    @StudentID int = 0
```

# PL/SQL Stored Procedure

```
AS
BEGIN
    SET NOCOUNT ON;
    SELECT FirstName, LastName, BirthDate, City, Country
    FROM Students WHERE StudentID=@StudentID
END
GO
```

**Syntax to drop a Procedure:**

```
DROP PROCEDURE procedure_name
```

Example:

```
DROP PROCEDURE GetStudentDetails
```

# PL/SQL Stored Functions:

- The PL/SQL Function is very similar to PL/SQL Procedure.
- The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.

Syntax to create a function:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

# PL/SQL Stored Funtions

Here:

- Function_name: specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters.
- IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that data type you are going to return from the function.
- Function_body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone

# PL/SQL Functions

**PL/SQL– Functions**

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

**Creating a Function**

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

# PL/SQL Functions

**Where**,

•function-name specifies the name of the function.

•[OR REPLACE] option allows the modification of an existing function.

•The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

•The function must contain a return statement.

•The RETURN clause specifies the data type you are going to return from the function.

•function-body contains the executable part.

•The AS keyword is used instead of the IS keyword for creating a standalone function

# PL/SQL Functions

**Example**

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter −

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME   | AGE | ADDRESS| SALARY |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali | 25  | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
```

# PL/SQL Functions

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Function created.
```

# PL/SQL Functions

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Function created.
```


# PL/SQL Stored Funtions

## PL/SQL Function Example

Let's see a simple example to **create a function**.

```
create or replace function adder(n1 in number, n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
/
```

# PL/SQL Stored Funtions

## PL/SQL Function Example

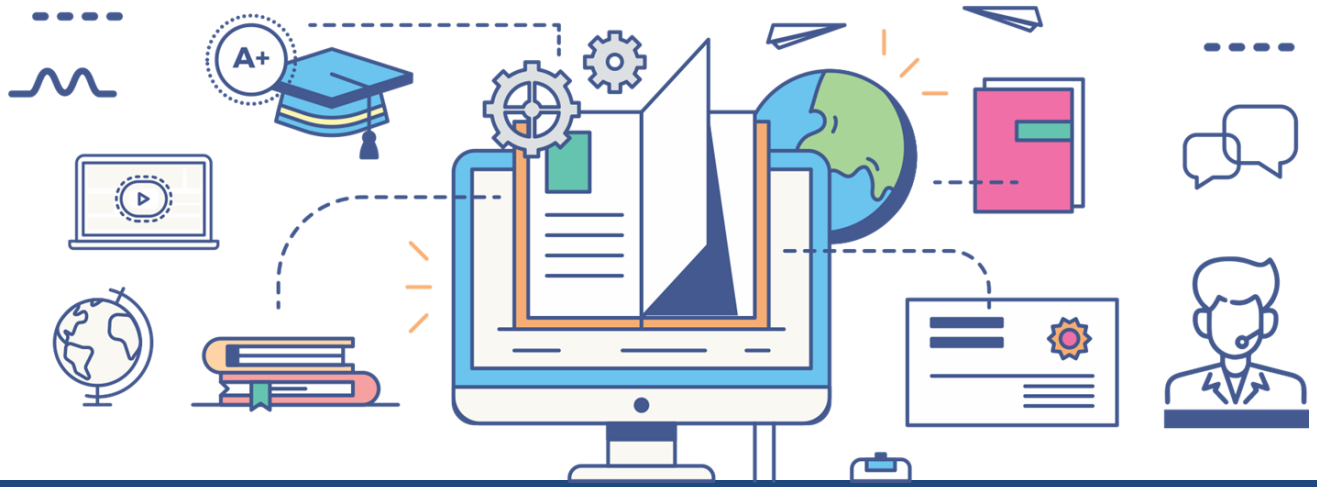
Now write another program to call the function.

```
DECLARE
   n3 number(2);
BEGIN
   n3 := adder(11,22);
   dbms_output.put_line('Addition is: ' || n3);
END;
/
```

Output:

```
Addition is: 33
Statement processed.
0.05 seconds
```

# DIGITAL LEARNING CONTENT




Parul® University

www.paruluniversity.ac.in