PL/SQL

- 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.
- Following are certain notable facts about PL/SQL −
    - 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's general syntax is based on that of ADA and Pascal programming language.
    - Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

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.

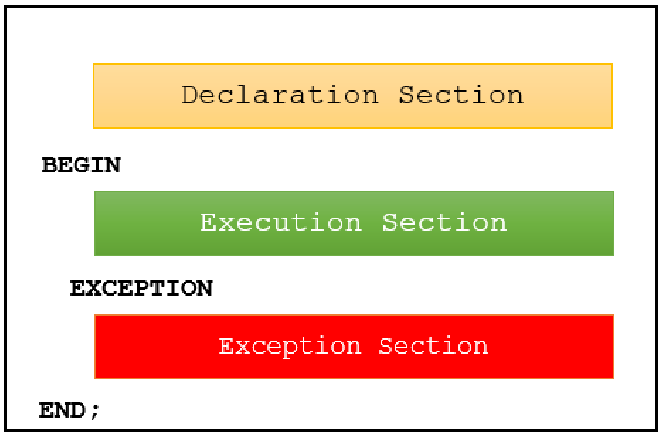
Advantages of PL/SQL

PL/SQL has the following advantages −

- 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 types.
- Applications written in PL/SQL are fully portable.

- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

PL/SQL BLOCK Structure:



1) Declaration section
A PL/SQL block has a declaration section where you declare variables, allocate memory for cursors, and define data types.
2) Executable section
A PL/SQL block has an executable section. An executable section starts with the keyword **BEGIN** and ends with the keyword **END.** The executable section must have a least one executable statement, even if it is the NULL statement which does nothing.

3) Exception-handling section
A PL/SQL block has an exception-handling section that starts with the keyword EXCEPTION. The exception-handling section is where you catch and handle exceptions raised by the code in the execution section.
Note a block itself is an executable statement, therefore you can nest a block within other blocks.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block –

```
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;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result –

```
Hello World

PL/SQL procedure successfully completed.
```

The PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code.

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 */.

```
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.
```

Example of initilizing variable

Let's take a simple example to explain it well:

```
DECLARE
    a integer := 30;
     b integer := 40;
     c integer;
     f real;
BEGIN
     c := a + b;
     dbms_output.put_line('Value of c: ' || c);
     f := 100.0/3.0;
     dbms_output.put_line('Value of f: ' || f);
END;
```

After the execution, this will produce the following result:

```
Value of c: 70
Value of f: 33.333333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL:

PL/SQL allows nesting of blocks. A program block can contain another inner block. If you declare a variable within an inner block, it is not accessible to an outer block. There are two types of variable scope:

- Local Variable: Local variables are the inner block variables which are not accessible to outer blocks.
- Global Variable: Global variables are declared in outermost block.

Example of Local and Global variables

Let's take an example to show the usage of Local and Global variables in its simple form:

```
DECLARE
 -- Global variables
  num1 number := 95;
  num2 number := 85;
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;
  /
```

After the execution, this will produce the following result:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

PL/SQL If

PL/SQL supports the programming language features like conditional statements and iterative statements.

Syntax for IF Statement:

There are different syntaxes for the IF-THEN-ELSE statement.

Syntax: (IF-THEN statement):

```
IF condition
THEN
Statement: {It is executed when condition is true}
END IF;
```

This syntax is used when you want to execute statements only when condition is TRUE.

Syntax: (IF-THEN-ELSE statement):

```
IF condition
THEN
  {...statements to execute when condition is TRUE...}
ELSE
  {...statements to execute when condition is FALSE...}
END IF;
```

This syntax is used when you want to execute one set of statements when condition is TRUE or a different set of statements when condition is FALSE.

Syntax: (IF-THEN-ELSIF statement):

```
IF condition1
THEN
  {...statements to execute when condition1 is TRUE...}
ELSIF condition2
THEN
  {...statements to execute when condition2 is TRUE...}
END IF;
```

This syntax is used when you want to execute one set of statements when condition1 is TRUE or a different set of statements when condition2 is TRUE.

Syntax: (IF-THEN-ELSIF-ELSE statement):

```
IF condition1
THEN
  {...statements to execute when condition1 is TRUE...}
ELSIF condition2
THEN
  {...statements to execute when condition2 is TRUE...}
ELSE
  {...statements to execute when both condition1 and condition2 are FALSE...}
END IF;
```

It is the most advance syntax and used if you want to execute one set of statements when condition1 is TRUE, a different set of statement when condition2 is TRUE or a different set of statements when both the condition1 and condition2 are FALSE.

When a condition is found to be TRUE, the IF-THEN-ELSE statement will execute the corresponding code and not check the conditions any further.

If there no condition is met, the ELSE portion of the IF-THEN-ELSE statement will be executed.

ELSIF and ELSE portions are optional.

Example of PL/SQL If Statement

Let's take an example to see the whole concept:

```
DECLARE
  a number(3) := 500;
BEGIN
  -- check the boolean condition using if statement
  IF( a < 20 ) THEN
    -- if condition is true then print the following
    dbms_output.put_line('a is less than 20 ' );
  ELSE
    dbms_output.put_line('a is not less than 20 ' );
  END IF;
```

```
   dbms_output.put_line('value of a is : ' || a);
END;
```

After the execution of the above code in SQL prompt, you will get the following result:

```
a is not less than 20
value of a is : 500
PL/SQL procedure successfully completed.
```

PL/SQL Loop

The PL/SQL loops are used to repeat the execution of one or more statements for specified number of times. These are also known as iterative control statements.

Syntax for a basic loop:

```
LOOP
  Sequence of statements;
END LOOP;
```

Types of PL/SQL Loops

There are 4 types of PL/SQL Loops.

1. Basic Loop / Exit Loop
2. While Loop
3. For Loop
4. Cursor For Loop

PL/SQL Exit Loop (Basic Loop)

PL/SQL exit loop is used when a set of statements is to be executed at least once before the termination of the loop. There must be an EXIT condition specified in the loop, otherwise the loop will get into an infinite number of iterations. After the occurrence of EXIT condition, the process exits the loop.

Syntax of basic loop:

```
LOOP
  Sequence of statements;
END LOOP;
```

Syntax of exit loop:

```
LOOP
   statements;
   EXIT;
```

```
  {or EXIT WHEN condition;}
END LOOP;
```

Example of PL/SQL EXIT Loop

Let's take a simple example to explain it well:

```
DECLARE
i NUMBER := 1;
BEGIN
LOOP
EXIT WHEN i>10;
DBMS_OUTPUT.PUT_LINE(i);
i := i+1;
END LOOP;
END;
```

After the execution of the above code, you will get the following result:

```
1
2
3
4
5
6
7
8
9
10
```

Note: You must follow these steps while using PL/SQL Exit Loop.

- Initialize a variable before the loop body
- Increment the variable in the loop.
- You should use EXIT WHEN statement to exit from the Loop. Otherwise the EXIT statement without WHEN condition, the statements in the Loop is executed only once.

Syntax of while loop:

```
WHILE <condition>
 LOOP statements;
END LOOP;
```

Example of PL/SQL While Loop

Let's see a simple example of PL/SQL WHILE loop.

```
DECLARE
i INTEGER := 1;
BEGIN
WHILE i <= 10 LOOP
DBMS_OUTPUT.PUT_LINE(i);
i := i+1;
END LOOP;
END;
```

After the execution of the above code, you will get the following result:

```
1
2
3
4
5
6
7
8
9
10
```

Note: You must follow these steps while using PL/SQL WHILE Loop.

- Initialize a variable before the loop body.
- Increment the variable in the loop.
- You can use EXIT WHEN statements and EXIT statements in While loop but it is not done often.

PL/SQL 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 are three ways to pass parameters in procedure:

1. **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.
2. **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.
3. **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.

A procedure may or may not return any value.

PL/SQL Create Procedure

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (parameter [,parameter]) ]
IS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```

Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

```
create table user(id number(10) primary key,name varchar2(100));
```

Now write the procedure code to insert record in user table.

Procedure Code:

```
create or replace procedure "INSERTUSER"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

Output:

```
Procedure created.
```

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
BEGIN
  insertuser(101,'Rahul');
  dbms_output.put_line('record inserted successfully');
END;
/
```

Now, see the "USER" table, you will see one record is inserted.

ID	Name
101	Rahul

PL/SQL Drop Procedure

Syntax for drop procedure

```
DROP PROCEDURE procedure_name;
```

Example of drop procedure

```
DROP PROCEDURE pro1;
```

PL/SQL Function

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. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

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];
```

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 function.

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;
/
```

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
```

PL/SQL function example using table

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

Create customers table and have records in it.

Customers			
Id	**Name**	**Department**	**Salary**
1	alex	web developer	35000
2	ricky	program developer	45000
3	mohan	web designer	35000
4	dilshad	database manager	44000

Create Function:

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

After the execution of above code, you will get the following result.

```
Function created.
```

Calling PL/SQL Function:

While creating a function, you have to give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. Once the function is called, the program control is transferred to the called function.

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

After the execution of above code in SQL prompt, you will get the following result.

```
Total no. of Customers: 4
PL/SQL procedure successfully completed.
```

PL/SQL Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

```
DECLARE
  num number;
  factorial number;

FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
  ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;

BEGIN
```

```
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

After the execution of above code at SQL prompt, it produces the following result.

```
Factorial 6 is 720
PL/SQL procedure successfully completed.
```

PL/SQL Drop Function

Syntax for removing your created function:

If you want to remove your created function from the database, you should use the following syntax.

```
DROP FUNCTION function_name;
```