

PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database.


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

- ❖ 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.

TEXT EDITOR

Running large programs from the command prompt may land you in inadvertently losing some of the work. It is always recommended to use the command files.

To use the command files –

- Type your code in a text editor, like **Notepad**, **Notepad+**, or **EditPlus**, etc.
- Save the file with the **.sql** extension in the home directory.
- Launch the **SQL*Plus command prompt** from the directory where you created your PL/SQL file.
- Type **@file_name** at the SQL*Plus command prompt to execute your program.

If you are not using a file to execute the PL/SQL scripts, then simply copy your PL/SQL code and right-click on the black window that displays the SQL prompt; use the **paste** option to paste the complete code at the command prompt. Finally, just press **Enter** to execute the code, if it is not already executed.

BASIC SYNTAX OF PL/SQL

- The Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code.
- Each block consists of three sub-parts –



Declarations



**Executable
Commands**



**Exception
Handling**

S.No	Sections & Description
1	<p>Declarations</p> <p>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.</p>
2	<p>Executable Commands</p> <p>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.</p>
3	<p>Exception Handling</p> <p>This section starts with the keyword EXCEPTION. This optional section contains exception(s) that handle errors in the program.</p>

- 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.

Execution Script

SQL> ed d:\p1.sql → Notepad will get open

DECLARE

message varchar2(20):='Hello , World!';

BEGIN

dbms_output.put_line(message);

END;

/

SQL> @ d:\p1.sql

OUTPUT:

Procedure successfully completed

To get Output

```
SQL> set serveroutput on
```

```
SQL> @ d:\p1.sql
```

OUTPUT:

Hello , World!

PL/SQL procedure successfully completed

Note:

“set serveroutput on” is an environment setting which is session based.

We can write procedure directly in the command

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.
- By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '!=, ~=, ^=	Different versions of NOT EQUAL

THE PL/SQL DELIMITERS

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL –

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. All programming languages allow some form of 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 */.

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 Program Units

A PL/SQL unit is any one of the following –

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

Data Types in PL/SQL

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

S.No	Category & Description
1	Scalar Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
2	Large Object (LOB) Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	Composite Data items that have internal components that can be accessed individually. For example, collections and records.
4	Reference Pointers to other data items.

Following is a valid declaration –

DECLARE

num1 INTEGER;

num2 REAL;

num3 DOUBLE PRECISION;

BEGIN

 null;

END;

/

When the above code is compiled and executed,
it produces the following result –

PL/SQL procedure successfully completed

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]`**

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type.

For example –

`sales number(10, 2);`

`name varchar2(25);`

`address varchar2(100);`

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

For example –

```
counter binary_integer := 0;
```

```
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a NULL value using the NOT NULL constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

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;

/

When the above code is executed, it produces the following result

—

Value of c: 30

Value of f: 23.33333333333333333333

PL/SQL procedure successfully completed.

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;
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;
/
```

When the above code is executed, it produces the following result –

Outer Variable num1: 95

Outer Variable num2: 85

Inner Variable num1: 195

Inner Variable num2: 185

PL/SQL procedure successfully completed.

To reference the global variable we must use a label and dot notation.
Referencing Global Variable with Same Name as Loop Counter

```
SQL> <<main>>
DECLARE
    i NUMBER := 5;
BEGIN
    FOR i IN 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE ('local: ' || TO_CHAR(i) || ', global: ' ||
            TO_CHAR(main.i));
    END LOOP;
END main;
```

OUTPUT:

```
local: 1, global: 5
local: 2, global: 5
local: 3, global: 5
```

PL/SQL procedure successfully completed.

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 is 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:

1. **CREATE** [OR **REPLACE**] **PROCEDURE** procedure_name
2. [(parameter [,parameter])]
3. **IS**
4. [declaration_section]
5. **BEGIN**
6. executable_section
7. [EXCEPTION
8. exception_section]
9. **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

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 FUNCTIONS

The PL/SQL function is very similar to PL/SQL procedure.

The main difference between Procedure and Function is,

- A function must always return a value and on the other hand a procedure may or may not return a value.
- All the other things of PL/SQL Procedures are true for PL/SQL Functions too.

Creating a Function

CREATE [OR REPLACE] FUNCTION

function_name [(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype

{IS | AS}

BEGIN

< function_body >

END [function_name];

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.

Example(1/2)

- **Create a Function** (STEP 1)

Create or replace function adder(n1 in number, n2 in number)

return number

is

n3 number(8);

begin

n3:=n1+n2;

return n3;

end;

/

Example(2/2)

- **Call the Function** (STEP 2)

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

OUTPUT:

Addition is : 33

Statement processed

0.05 seconds

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

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

Creating a Function

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

OUTPUT:

Function created.

Calling a Function

```
DECLARE
```

```
    c number(2);
```

```
BEGIN
```

```
    c := totalCustomers();
```

```
    dbms_output.put_line('Total no. of  
Customers: ' || c);
```

```
END;
```

```
/
```

OUTPUT:

Total no. of Customers: 6 PL/SQL
procedure successfully completed.

```
DECLARE
```

```
    a number;
```

```
    b number;
```

```
    c number;
```

```
FUNCTION findMax(x IN number, y IN number)
```

```
RETURN number
```

```
IS
```

```
z number;
```

```
BEGIN
```

```
    IF x > y THEN
```

```
        z:= x;
```

```
    ELSE
```

```
        Z:= y;
```

```
    END IF;
```

```
    RETURN z;
```

```
END;
```

```
BEGIN
```

```
    a:= 23;
```

```
    b:= 45;
```

```
    c := findMax(a, b);
```

```
    dbms_output.put_line(' Maximum of (23,45): ' || c);
```

```
END;
```

```
/
```

Example

The following example
Demonstrates Declaring,
Defining, and Invoking
a Simple PL/SQL
Function that computes
and returns the
maximum of two values.

OUTPUT :

Maximum of (23,45): 45
PL/SQL procedure
successfully
completed.

```

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;
/

```

PL/SQL Recursive Functions

When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

OUTPUT:

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;
```

PL/SQL Cursor

- A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time.
- There are two types of cursors:
 - ❖ Implicit Cursors
 - ❖ Explicit Cursors

PL/SQL Implicit Cursors

- The implicit cursors are automatically generated by Oracle while an SQL statement is executed.
- If you don't use an explicit cursor for the statement. These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.
- Oracle provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are:
 - ❖ %FOUND
 - ❖ %NOTFOUND,
 - ❖ %ROWCOUNT
 - ❖ %ISOPEN.

The following table specifies the status of the cursor with each of its attribute.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

PL/SQL Implicit Cursor Example

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

Create procedure:

DECLARE

```
total_rows number(2);
```

BEGIN

```
UPDATE customers
```

```
SET salary = salary + 5000;
```

```
IF sql%notfound THEN
```

```
    dbms_output.put_line('no customers updated');
```

```
ELSIF sql%found THEN
```

```
    total_rows := sql%rowcount;
```

```
    dbms_output.put_line( total_rows || ' customers  
                           updated ');
```

```
END IF;
```

```
END;
```

```
/
```

Output:

6 customers updated PL/SQL procedure successfully completed. Now, if you check the records in customer table, you will find that the rows are updated.

select * **from** customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	Chandan	25	Noida	31000
5	Alex	21	Paris	33000
6	Sunita	20	Delhi	35000

PL/SQL Explicit Cursors

- The Explicit cursors are defined by the programmers to gain more control over the context area.
- These cursors 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.
- **Syntax of explicit cursor**
CURSOR cursor_name **IS** select_statement;;
- You must follow these steps while working with an explicit cursor.
 - ❖ Declare the cursor to initialize in the memory.
 - ❖ Open the cursor to allocate memory.
 - ❖ Fetch the cursor to retrieve data.
 - ❖ Close the cursor to release allocated memory.

Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

Syntax for explicit cursor declaration

```
CURSOR name IS  
SELECT statement;
```

Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax for cursor open:

```
OPEN cursor_name;
```

Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

Syntax for cursor fetch:

```
FETCH cursor_name INTO variable_list;
```

Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

Syntax for cursor close:

```
Close cursor_name;
```

PL/SQL Explicit Cursor Example

- Create customers table and have records

Create procedure:

Execute the following program to retrieve the customer name and address.

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;
```

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;
```

```
/
```

Output:

1	Ramesh	Allahabad
2	Suresh	Kanpur
3	Mahesh	Ghaziabad
4	Chandan	Noida
5	Alex	Paris
6	Sunita	Delhi

PL/SQL procedure successfully completed.

PL/SQL Exception Handling

- An error occurs during the program execution is called Exception in PL/SQL.
- PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.
- There are two type of exceptions:
 - ❖ System-defined Exceptions
 - ❖ User-defined Exceptions

Syntax for exception handling

DECLARE

<declarations **section**>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 **THEN**

exception1-handling-statements

WHEN exception2 **THEN**

exception2-handling-statements

WHEN exception3 **THEN**

exception3-handling-statements

.....

WHEN others **THEN**

exception3-handling-statements

END;

Example of exception handling

SELECT* FROM COUSTOMERS;

DECLARE

```
c_id customers.id%type := 8;  
c_name customers.name%type;  
c_addr customers.address%type;
```

BEGIN

```
SELECT name, address INTO c_name, c_addr FROM customers  
WHERE id = c_id;
```

```
DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
```

```
DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
```

EXCEPTION

```
WHEN no_data_found THEN
```

```
    dbms_output.put_line('No such customer!');
```

```
WHEN others THEN
```

```
    dbms_output.put_line('Error!');
```

```
END;
```

```
/
```

After the execution of above code at SQL Prompt, it produces the following result:

No such customer!

PL/SQL procedure successfully completed.

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception `NO_DATA_FOUND`, which is captured in `EXCEPTION` block.

DECLARE

```
c_id customers.id%type := 5;  
c_name customers.name%type;  
c_addr customers.address%type;
```

BEGIN

```
SELECT name, address INTO c_name, c_addr
```

```
FROM customers
```

```
WHERE id = c_id;
```

```
DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
```

```
DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
```

EXCEPTION

```
WHEN no_data_found THEN
```

```
    dbms_output.put_line('No such customer!');
```

```
WHEN others THEN
```

```
    dbms_output.put_line('Error!');
```

```
END;
```

```
/
```

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. here, we use the id 5.

OUTPUT:

Name: alex
Address: paris
PL/SQL procedure
successfully completed.

Raising Exceptions

- In the case of any internal database error, exceptions are raised by the database server automatically.
- But it can also be raised explicitly by programmer by using command RAISE.
- **Syntax for raising an exception:**

DECLARE

exception_name EXCEPTION;

BEGIN

IF condition **THEN**

RAISE exception_name;

END IF;

EXCEPTION

WHEN exception_name **THEN**

statement;

END;

PL/SQL User-defined Exceptions

- PL/SQL facilitates their users to define their own exceptions according to the need of the program.
- A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.
- **Syntax for user define exceptions**

DECLARE

my-exception EXCEPTION;

PL/SQL Pre-defined Exceptions

- There are many pre-defined exception in PL/SQL which are executed when any database rule is violated by the programs.
- **For example:** NO_DATA_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.
- Following is a list of some important pre-defined exceptions:

Exception	Oracle Error	SQL Code	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a NULL object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the "WHEN" clauses of a CASE statement is selected, and there is no else clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than exists to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a select into statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

PL/SQL Trigger

- Trigger is invoked by Oracle engine automatically whenever a specified event occurs.
- Trigger is stored into database and invoked repeatedly, when specific condition match.
- Triggers are stored programs, which are automatically executed or fired when some event occurs.
- Triggers are 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 could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers

- ❑ Trigger generates some derived column values automatically
- ❑ Enforces referential integrity
- ❑ Event logging and storing information on table access
- ❑ Auditing
- ❑ Synchronous replication of tables
- ❑ Imposing security authorizations
- ❑ Preventing invalid transactions

Creating a trigger

Syntax for creating trigger:

```
CREATE [OR REPLACE ] TRIGGER trigger_name  
{BEFORE | AFTER | INSTEAD OF }  
{INSERT [OR] | UPDATE [OR] | DELETE}  
[OF col_name]  
ON table_name  
[REFERENCING OLD AS o NEW AS n]  
[FOR EACH ROW]  
WHEN (condition)  
DECLARE  
    Declaration-statements  
BEGIN  
    Executable-statements  
EXCEPTION  
    Exception-handling-statements  
END;
```

Here,

- `CREATE [OR REPLACE] TRIGGER trigger_name`: It creates or replaces an existing trigger with the `trigger_name`.
- `{BEFORE | AFTER | INSTEAD OF}` : This specifies when the trigger would 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 would 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, like `INSERT`, `UPDATE`, and `DELETE`.
- `[FOR EACH ROW]`: This specifies a row level trigger, i.e., the trigger would 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.
- `WHEN (condition)`: This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example

Create trigger:

Let's take a program to create 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);
    dbms_output.put_line('New salary: ' || :NEW.salary);
    dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
OUTPUT:
Trigger created
```

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

DECLARE

total_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 5000;

IF sql%notfound **THEN**

dbms_output.put_line('no customers updated');

ELSIF sql%found **THEN**

total_rows := sql%rowcount;

dbms_output.put_line(total_rows || ' customers updated ');

END IF;

END;

/

OUTPUT:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

- **Note:** As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.
- Following are the two very important point and should be noted carefully.
 - ❖ OLD and NEW references are used for record level triggers these are not available for table level triggers.
 - ❖ If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.