

What is PL/SQL?

PL/SQL stands for Procedural Language extension of 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.

A Simple PL/SQL Block:

Each PL/SQL program consists of SQL and PL/SQL statements which form a PL/SQL block.

A PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception (or Error) Handling section (optional).

Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

Execution Section:

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of the execution section.

Exception Section:

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Block terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors. Every statement in the above three sections must end with a semicolon. PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

This is how a sample PL/SQL Block looks.

```
DECLARE
  Variable declaration
BEGIN
  Program Execution
EXCEPTION
  Exception handling
END;
```

Advantages of PL/SQL

These are the advantages of PL/SQL.

- **Block Structures:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
- **Procedural Language Capability:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
- **Better Performance:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
- **Error Handling:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

PL/SQL Placeholders

Placeholders are temporary storage area. Placeholders can be any of Variables, Constants and Records. Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block. Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below. Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

PL/SQL Variables

These are placeholders that store the values that can change through the PL/SQL Block.

The General Syntax to declare a variable is:

```
variable_name datatype [NOT NULL := value ];
```

- *variable_name* is the name of the variable.
- *datatype* is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable.
- *value* or DEFAULT *value* is also an optional specification, where you can initialize a variable.

Each variable declaration is a separate statement and must be terminated by a semicolon. For example, if you want to store the current salary of an employee, you can use a variable.

```
DECLARE
salary number (6);
* "salary" is a variable of datatype number and of length 6.
```

When a variable is specified as NOT NULL, you must initialize the variable when it is declared. For example: The below example declares two variables, one of which is a not null.

```
DECLARE
salary number(4);
dept varchar2(10) NOT NULL := "HR Dept";
```

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.

1) We can directly assign values to variables. The General Syntax is:

```
variable_name := value;
```

2) We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:

```
SELECT column_name
INTO variable_name
FROM table_name
[WHERE condition];
```

Example: The below program will get the salary of an employee with id '1116' and display it on the screen.

```
DECLARE
var_salary number(6);
var_emp_id number(6) = 1116;
BEGIN
SELECT salary
INTO var_salary
FROM employee
WHERE emp_id = var_emp_id;
```

```
dbms_output.put_line(var_salary);
dbms_output.put_line('The employee ' || var_emp_id || ' has salary ' ||
var_salary);
END;
/
```

NOTE: The backward slash '/' in the above program indicates to execute the above PL/SQL Block.

Scope of Variables

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- *Local* variables - These are declared in a inner block and cannot be referenced by outside Blocks.
- *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

For Example: In the below example we are creating two variables in the outer block and assigning their product to the third variable created in the inner block. The variable 'var_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block.

```
1> DECLARE
2>  var_num1 number;
3>  var_num2 number;
4> BEGIN
5>  var_num1 := 100;
6>  var_num2 := 200;
7>  DECLARE
8>   var_mult number;
9>   BEGIN
10>    var_mult := var_num1 * var_num2;
11>   END;
12> END;
13> /
```

PL/SQL Constants

As the name implies a *constant* is a value used in a PL/SQL Block that remains unchanged throughout the program. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.

For example: If you want to write a program which will increase the salary of the employees by 25%, you can declare a constant and use it throughout the program. Next time when you want to increase the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.

The General Syntax to declare a constant is:

```
constant_name CONSTANT datatype := VALUE;
```

- *constant_name* is the name of the constant i.e. similar to a variable name.
- The word *CONSTANT* is a reserved word and ensures that the value does not change.
- *VALUE* - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

For example, to declare `salary_increase`, you can write code as follows:

```
DECLARE

salary_increase CONSTANT number (3) := 10;
```

You *must* assign a value to a constant at the time you declare it. If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get an error. If you execute the below PL/SQL block you will get an error.

```
DECLARE

    salary_increase CONSTANT number(3);

BEGIN

    salary_increase := 100;

    dbms_output.put_line (salary_increase);

END;
```

PL/SQL Records

What are records?

Records are another type of datatypes which Oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc. Each scalar data type in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

Declaring a record:

To declare a record, you must first define a composite datatype; then declare a record for that type.

The General Syntax to define a composite datatype is:

```
TYPE record_type_name IS RECORD
(first_col_name column_datatype,
second_col_name column_datatype, ...);
```

- *record_type_name* – it is the name of the composite type you want to define.
- *first_col_name, second_col_name, etc.,* - it is the names the fields/columns within the record.
- *column_datatype* defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.

- 1) You can declare the field in the same way as you declare the fields while creating the table.
- 2) If a field is based on a column from database table, you can define the field_type as follows:

```
col_name table_name.column_name%type;
```

By declaring the field datatype in the above method, the datatype of the column is dynamically applied to the field. This method is useful when you are altering the column specification of the table, because you do not need to change the code again.

NOTE: You can use also *%type* to declare variables and constants.

The General Syntax to declare a record of a user-defined datatype is:

```
record_name record_type_name;
```

The following code shows how to declare a record called *employee_rec* based on a user-defined type.

```
DECLARE
TYPE employee_type IS RECORD
(employee_id number(5),
 employee_first_name varchar2(25),
 employee_last_name employee.last_name%type,
 employee_dept employee.dept%type);
employee_salary employee.salary%type;
employee_rec employee_type;
```

If all the fields of a record are based on the columns of a table, we can declare the record as follows:

```
record_name table_name%ROWTYPE;
```

For example, the above declaration of *employee_rec* can be as follows:

```
DECLARE
employee_rec employee%ROWTYPE;
```

The advantages of declaring the record as a ROWTYPE are:

- 1) You do not need to explicitly declare variables for all the columns in a table.
- 2) If you alter the column specification in the database table, you do not need to update the code.

The disadvantage of declaring the record as a ROWTYPE is:

- 1) When you create a record as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields. So use ROWTYPE only when you are using all the columns of the table in the program.

NOTE: When you are creating a record, you are just creating a datatype, similar to creating a variable. You need to assign values to the record to use them.

The following table consolidates the different ways in which you can define and declare a pl/sql record.

Syntax	Description
TYPE record_type_name IS RECORD (column_name1 datatype, column_name2 datatype, ...);	Define a composite datatype, where each field is scalar.
col_name table_name.column_name%type;	Dynamically define the datatype of a column based on a database column.
record_name record_type_name;	Declare a record based on a user-defined type.
record_name table_name%ROWTYPE;	Dynamically declare a record based on an entire row of a table. Each column in the table corresponds to a field in the record.

Passing Values To and From a Record

When you assign values to a record, you actually assign values to the fields within it. The General Syntax to assign a value to a column within a record directly is:

```
record_name.col_name := value;
```

If you used %ROWTYPE to declare a record, you can assign values as shown:

```
record_name.column_name := value;
```

We can assign values to records using SELECT Statements as shown:

```
SELECT col1, col2  
INTO record_name.col_name1, record_name.col_name2  
FROM table_name  
[WHERE clause];
```

If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each columns separately. In this case, you must SELECT all the columns from the table into the record as shown:

```
SELECT * INTO record_name  
FROM table_name  
[WHERE clause];
```

Lets see how we can get values from a record. The General Syntax to retrieve a value from a specific field into another variable is:

```
var_name := record_name.col_name;
```

The following table consolidates the different ways you can assign values to and from a record:

Syntax	Usage
<code>record_name.col_name := value;</code>	To directly assign a value to a specific column of a record.
<code>record_name.column_name := value;</code>	To directly assign a value to a specific column of a record, if the record is declared using %ROWTYPE.
<code>SELECT col1, col2 INTO record_name.col_name1, record_name.col_name2 FROM table_name [WHERE clause];</code>	To assign values to each field of a record from the database table.
<code>SELECT * INTO record_name FROM table_name [WHERE clause];</code>	To assign a value to all fields in the record from a database table.
<code>variable_name := record_name.col_name;</code>	To get a value from a record column and assigning it to a variable.

Conditional Statements in PL/SQL

As the name implies, PL/SQL supports programming language features like conditional statements, iterative statements.

The programming constructs are similar to how you use in programming languages like Java and C++. In this section I will provide you syntax of how to use conditional statements in PL/SQL programming.

IF THEN ELSE STATEMENT

```
1)
IF condition
THEN
  statement 1;
ELSE
  statement 2;
END IF;
```

```
2)
IF condition 1
THEN
  statement 1;
  statement 2;
```



```
ELSIF condtion2 THEN
    statement 3;
ELSE
    statement 4;
END IF
```

```
3)
IF condition 1
THEN
    statement 1;
    statement 2;
ELSIF condtion2 THEN
    statement 3;
ELSE
    statement 4;
END IF;
```

```
4)
IF condition1 THEN
ELSE
    IF condition2 THEN
        statement1;
    END IF;
ELSIF condition3 THEN
    statement2;
END IF;
```

Iterative Statements in PL/SQL

An iterative control Statements are used when we want to repeat the execution of one or more statements for specified number of times. These are similar to those in

There are three types of loops in PL/SQL:

- Simple Loop
- While Loop
- For Loop

1) Simple Loop

A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates. An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations. When the EXIT condition is satisfied the process exits from the loop.

The General Syntax to write a Simple Loop is:

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

These are the important steps to be followed while using Simple Loop.

- 1) Initialise a variable before the loop body.
- 2) Increment the variable in the loop.
- 3) Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

2) While Loop

A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true. The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

The General Syntax to write a WHILE LOOP is:

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

Important steps to follow when executing a while loop:

- 1) Initialise a variable before the loop body.
- 2) Increment the variable in the loop.
- 3) EXIT WHEN statement and EXIT statements can be used in while loops but it's not done oftenly.

3) FOR Loop

A FOR LOOP is used to execute a set of statements for a predetermined number of times. Iteration occurs between the start and end integer values given. The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer.

The General Syntax to write a FOR LOOP is:

```
FOR counter IN val1..val2  
  LOOP statements;  
END LOOP;
```

- val1 - Start integer value.
- val2 - End integer value.

Important steps to follow when executing a while loop:

- 1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly.
- 2) The counter variable is incremented by 1 and does not need to be incremented explicitly.
- 3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done oftenly.

NOTE: The above Loops are explained with a example when dealing with Explicit Cursors.

What are Cursors?

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active set*.

There are two types of cursors in PL/SQL:

Implicit cursors:

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Explicit cursors:

They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

Stored Procedures

What is a Stored Procedure?

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages. A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists of declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

We can pass parameters to procedures in three ways.

- 1) IN-parameters
- 2) OUT-parameters
- 3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]
IS
    Declaration section
BEGIN
    Execution section
```

```
EXCEPTION
    Exception section
END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

The below example creates a procedure 'employer_details' which gives the details of the employee.

```
1> CREATE OR REPLACE PROCEDURE employer_details
2> IS
3>   CURSOR emp_cur IS
4>   SELECT first_name, last_name, salary FROM emp_tbl;
5>   emp_rec emp_cur%rowtype;
6> BEGIN
7>   FOR emp_rec in sales_cur
8>   LOOP
9>     dbms_output.put_line(emp_rec.first_name || ' ' ||emp_rec.last_name
10>      || ' ' ||emp_rec.salary);
11> END LOOP;
12>END;
13> /
```

How to execute a Stored Procedure?

There are two ways to execute a procedure.

1) From the SQL prompt.

```
EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name.

```
procedure_name;
```

NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

PL/SQL Functions

What is a Function in PL/SQL?

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

The General Syntax to create a function is:

```

CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS
Declaration_section
BEGIN
Execution_section
Return return_variable;
EXCEPTION
exception_section
Return return_variable;
END;

```

- 1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
- 2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a function called "employer_details_func" similar to the one created in stored proc

```

1> CREATE OR REPLACE FUNCTION employer_details_func
2>   RETURN VARCHAR(20);
3> IS
4>   emp_name VARCHAR(20);
5> BEGIN
6>   SELECT first_name INTO emp_name
7>   FROM emp_tbl WHERE empID = '100';
8>   RETURN emp_name;
9> END;
10> /

```

In the example we are retrieving the 'first_name' of employee with empID 100 to variable 'emp_name'. The return type of the function is VARCHAR which is declared in line no 2. The function returns the 'emp_name' which is of type VARCHAR as the return value in line no 9.

How to execute a PL/SQL Function?

A function can be executed in the following ways.

- 1) Since a function returns a value we can assign it to a variable.

```
employee_name := employer_details_func;
```

If 'employee_name' is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

- 2) As a part of a SELECT statement

```
SELECT employer_details_func FROM dual;
```

- 3) In a PL/SQL Statements like,

```
dbms_output.put_line(employer_details_func);
```

This line displays the value returned by the function.

Parameters in Procedure and Functions

How to pass parameters to Procedures and Functions in PL/SQL ?

In PL/SQL, we can pass parameters to procedures and functions in three ways.

- 1) IN type parameter:** These types of parameters are used to send values to stored procedures.
- 2) OUT type parameter:** These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
- 3) IN OUT parameter:** These types of parameters are used to send values and get values from stored procedures.

NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

1) IN parameter:

This is similar to passing parameters in programming languages. We can pass values to the stored procedure through these parameters or variables. This type of parameter is a read only parameter. We can assign the value of IN type parameter to a variable or use it in a query, but we cannot change its value inside the procedure.

The General syntax to pass a IN parameter is

```
CREATE [OR REPLACE] PROCEDURE procedure_name (  
    param_name1 IN datatype, param_name12 IN datatype ... )
```

- param_name1, param_name2... are unique parameter names.
- datatype - defines the datatype of the variable.
- IN - is optional, by default it is a IN type parameter.

2) OUT Parameter:

The OUT parameters are used to send the OUTPUT from a procedure or a function. This is a write-only parameter i.e, we cannot pass values to OUT parameters while executing the stored procedure, but we can assign values to OUT parameter inside the stored procedure and the calling program can receive this output value.

The General syntax to create an OUT parameter is

```
CREATE [OR REPLACE] PROCEDURE proc2 (param_name OUT datatype)
```

The parameter should be explicitly declared as OUT parameter.

3) IN OUT Parameter:

The IN OUT parameter allows us to pass values into a procedure and get output values from the procedure. This parameter is used if the value of the IN parameter can be changed in the calling program.

By using IN OUT parameter we can pass values into a parameter and return a value to the calling program using the same parameter. But this is possible only if the value passed to the procedure and output value have a same datatype. This parameter is used if the value of the parameter will be changed in the procedure.

The General syntax to create an IN OUT parameter is

```
CREATE [OR REPLACE] PROCEDURE proc3 (param_name IN OUT datatype)
```

The below examples show how to create stored procedures using the above three types of parameters.

Example1:

Using IN and OUT parameter:

Let's create a procedure which gets the name of the employee when the employee id is passed.

```
1> CREATE OR REPLACE PROCEDURE emp_name (id IN NUMBER, emp_name OUT NUMBER)
2> IS
3> BEGIN
4>     SELECT first_name INTO emp_name
5>     FROM emp_tbl WHERE empID = id;
6> END;
7> /
```

We can call the procedure 'emp_name' in this way from a PL/SQL Block.

```
1> DECLARE
2> empName varchar(20);
3> CURSOR id_cur SELECT id FROM emp_ids;
4> BEGIN
5> FOR emp_rec in id_cur
6> LOOP
7>     emp_name(emp_rec.id, empName);
8>     dbms_output.putline('The employee ' || empName || ' has id ' || emp-
rec.id);
9> END LOOP;
10> END;
11> /
```

In the above PL/SQL Block

In line no 3; we are creating a cursor 'id_cur' which contains the employee id.

In line no 7; we are calling the procedure 'emp_name', we are passing the 'id' as IN parameter and 'empName' as OUT parameter.

In line no 8; we are displaying the id and the employee name which we got from the procedure 'emp_name'.

Example 2:

Using IN OUT parameter in procedures:

```

1> CREATE OR REPLACE PROCEDURE emp_salary_increase
2> (emp_id IN emp_tbl.empID%type, salary_inc IN OUT emp_tbl.salary%type)
3> IS
4>   tmp_sal number;
5> BEGIN
6>   SELECT salary
7>   INTO tmp_sal
8>   FROM emp_tbl
9>   WHERE empID = emp_id;
10>  IF tmp_sal between 10000 and 20000 THEN
11>    salary_inc := tmp_sal * 1.2;
12>  ELSIF tmp_sal between 20000 and 30000 THEN
13>    salary_inc := tmp_sal * 1.3;
14>  ELSIF tmp_sal > 30000 THEN
15>    salary_inc := tmp_sal * 1.4;
16>  END IF;
17> END;
18> /

```

The below PL/SQL block shows how to execute the above 'emp_salary_increase' procedure.

```

1> DECLARE
2>   CURSOR updated_sal is
3>   SELECT empID, salary
4>   FROM emp_tbl;
5>   pre_sal number;
6> BEGIN
7>   FOR emp_rec IN updated_sal LOOP
8>     pre_sal := emp_rec.salary;
9>     emp_salary_increase(emp_rec.empID, emp_rec.salary);
10>    dbms_output.put_line('The salary of ' || emp_rec.empID ||
11>      ' increased from ' || pre_sal || ' to ' || emp_rec.salary);
12>   END LOOP;
13> END;
14> /

```

Exception Handling

In this section we will discuss about the following,

1) What is Exception Handling.

2) Structure of Exception Handling.

3) Types of Exception Handling.

1) What is Exception Handling?

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is recieved.

PL/SQL Exception message consists of three parts.

- 1) **Type of Exception**
- 2) **An Error Code**
- 3) **A message**

By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

2) Structure of Exception Handling.

The General Syntax for coding the exception section

```

DECLARE
    Declaration section
BEGIN
    Exception section
EXCEPTION
WHEN ex_name1 THEN
    -Error handling statements
WHEN ex_name2 THEN
    -Error handling statements
WHEN Others THEN
    -Error handling statements
END;
```

General PL/SQL statments can be used in the Exception Block.

When an exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'ex_name1 ', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing fo the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

If there are nested PL/SQL blocks like this.

```

DELCARE
    Declaration section
BEGIN
    DECLARE
        Declaration section
```

```

BEGIN
    Execution section
EXCEPTION
    Exception section
END;
EXCEPTION
    Exception section
END;

```

In the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block. If none of the blocks handle the exception the program ends abruptly with an error.

3) Types of Exception.

There are 3 types of Exceptions.

- a) Named System Exceptions
- b) Unnamed System Exceptions
- c) User-defined Exceptions

a) Named System Exceptions

System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.

For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions.

Named system exceptions are:

- 1) Not Declared explicitly,
- 2) Raised implicitly when a predefined Oracle error occurs,
- 3) caught by referencing the standard name within an exception-handling routine.

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

For Example: Suppose a NO_DATA_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

```
BEGIN
  Execution section
EXCEPTION
WHEN NO_DATA_FOUND THEN
  dbms_output.put_line ('A SELECT...INTO did not return any row.');
```

b) Unnamed System Exceptions

Those system exception for which oracle does not provide a name is known as unnamed system exception. These exception do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed system exceptions:

1. By using the WHEN OTHERS exception handler, or
2. By associating the exception code to a name and using it as a named exception.

We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION_INIT**. **EXCEPTION_INIT** will associate a predefined Oracle error number to a programmer_defined exception name.

Steps to be followed to use unnamed system exceptions are

- They are raised implicitly.
- If they are not handled in WHEN Others they must be handled explicitly.
- To handle the exception explicitly, they must be declared using Pragma EXCEPTION_INIT as given above and handled referring the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

```
DECLARE
  exception_name EXCEPTION;
  PRAGMA
  EXCEPTION_INIT (exception_name, Err_code);
BEGIN
  Execution section
EXCEPTION
  WHEN exception_name THEN
    handle the exception
END;
```

For Example: Lets consider the product table and order_items table from sql joins.

Here product_id is a primary key in product table and a foreign key in order_items table.

If we try to delete a product_id from the product table when it has child records in order_id table an

exception will be thrown with oracle code number -2292.

We can provide a name to this exception and handle it in the exception section as given below.

```

DECLARE
  Child_rec_exception EXCEPTION;
  PRAGMA
    EXCEPTION_INIT (Child_rec_exception, -2292);
BEGIN
  Delete FROM product where product_id= 104;
EXCEPTION
  WHEN Child_rec_exception
  THEN Dbms_output.put_line('Child records are present for this
product_id.');
```

c) User-defined Exceptions

Apart from system exceptions we can explicitly define exceptions based on business rules. These are known as user-defined exceptions.

Steps to be followed to use user-defined exceptions:

- They should be explicitly declared in the declaration section.
- They should be explicitly raised in the Execution Section.
- They should be handled by referencing the user-defined exception name in the exception section.

For Example: Lets consider the product table and order_items table from sql joins to explain user-defined exception.

Lets create a business rule that if the total no of units of any particular product sold is more than 20, then it is a huge quantity and a special discount should be provided.

```

DECLARE
  huge_quantity EXCEPTION;
  CURSOR product_quantity is
  SELECT p.product_name as name, sum(o.total_units) as units
  FROM order_tems o, product p
  WHERE o.product_id = p.product_id;
  quantity order_tems.total_units%type;
  up_limit CONSTANT order_tems.total_units%type := 20;
  message VARCHAR2(50);
BEGIN
  FOR product_rec in product_quantity LOOP
    quantity := product_rec.units;
    IF quantity > up_limit THEN
      message := 'The number of units of product ' || product_rec.name ||
        ' is more than 20. Special discounts should be provided.
        Rest of the records are skipped. '
      RAISE huge_quantity;
    ELSIF quantity < up_limit THEN
      v_message:= 'The number of unit is below the discount limit.';
    END IF;
    dbms_output.put_line (message);
```

```

    END LOOP;
EXCEPTION
    WHEN huge_quantity THEN
        dbms_output.put_line (message);
END;
/

```

RAISE_APPLICATION_ERROR ()

RAISE_APPLICATION_ERROR is a built-in procedure in Oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.

Whenever a message is displayed using **RAISE_APPLICATION_ERROR**, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements).

RAISE_APPLICATION_ERROR raises an exception but does not handle it.

RAISE_APPLICATION_ERROR is used for the following reasons,

- a) to create a unique id for an user-defined exception.
- b) to make the user-defined exception look like an Oracle error.

The General Syntax to use this procedure is:

```
RAISE_APPLICATION_ERROR (error_number, error_message);
```

- The Error number must be between -20000 and -20999
- The Error_message is the message you want to display when the error occurs.

Steps to be followed to use **RAISE_APPLICATION_ERROR** procedure:

1. Declare a user-defined exception in the declaration section.
2. Raise the user-defined exception based on a specific business rule in the execution section.
3. Finally, catch the exception and link the exception to a user-defined error number in **RAISE_APPLICATION_ERROR**.

Using the above example we can display a error message using **RAISE_APPLICATION_ERROR**.

```

DECLARE
    huge_quantity EXCEPTION;
    CURSOR product_quantity IS
    SELECT p.product_name as name, sum(o.total_units) as units
    FROM order_tems o, product p
    WHERE o.product_id = p.product_id;
    quantity order_tems.total_units%type;
    up_limit CONSTANT order_tems.total_units%type := 20;
    message VARCHAR2(50);
BEGIN
    FOR product_rec in product_quantity LOOP
        quantity := product_rec.units;
        IF quantity > up_limit THEN

```

```

        RAISE huge_quantity;
    ELSIF quantity < up_limit THEN
        v_message:= 'The number of unit is below the discount limit.';
    END IF;
    Dbms_output.put_line (message);
END LOOP;
EXCEPTION
    WHEN huge_quantity THEN
        raise_application_error(-2100, 'The number of unit is above the
discount limit.');
```

What is a Trigger?

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax of 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
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
    --- sql statements
END;
```

- **CREATE [OR REPLACE] TRIGGER trigger_name** - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- **{BEFORE | AFTER | INSTEAD OF }** - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- **{INSERT [OR] | UPDATE [OR] | DELETE}** - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- **[OF col_name]** - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- **CREATE [OR REPLACE] TRIGGER trigger_name** - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- **[ON table_name]** - This clause identifies the name of the table or view to which the trigger is associated.
- **[REFERENCING OLD AS o NEW AS n]** - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other

user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Sahaj Computer Solutions