Mahatma Gandhi Charitable Trust Managed

Shri Labhubhai Trivedi Institute of Engineering & Technology

CREATING CREATORS - SLTIET

Relational Database Management Systems

RDBMS - Subject Code: 4330702

Prof. Rinkal Umaraniya

CSE Department

SLTIET, Rajkot

CREATING CREATORS - SLTIET

Unit - 4

PL/SQL and Triggers

CREATING CREATORS - SLTIET

Sub Topics:

4.1 Basics of PL/SQL

4.2 Advantages of PL/SQL

4.3 PL/SQL block

4.4 Creating and Executing of PL/SQL block

4.5 Control Structures

=> Conditional control (IF-ELSE / IF-THEN-ELSE etc)

=> Iterative control (LOOP / WHILE / FOR)

=> Sequential control (GOTO statement)

4.6 Exception

4.7 Cursors

4.8 Procedures and Functions

4.9 Packages

4.10 Trigger

4.1 Basics of PL/SQL

- SQL provides various functionalities required to manage a database.
- SQL is so powerful to handling data and various database objects. But it lacks some of the basic functionalities provided by other programming languages.
- For example, SQL does not provide basic procedural capabilities such as conditional, checking, branching and looping.
- In SQL, it is not possible to control execution of SQL statements based on some condition or user inputs.
- Oracle provides **PL/SQL (Procedural Language / Structured Query Language)** to overcome disadvantages of SQL.
- PL/SQL is a super set of SQL.

Continue…

- PL/SQL supports all the functionalities provided by SQL along with its own procedural capabilities.
- Any SQL statements can be used in PL/SQL programs with no change, except SQL's data definition statements such as CREATE TABLE.
- Data definition statements are not allowed because PL/SQL. This is because the PL/SQL code is compiled. And at compile time, it cannot refer to objects that do not yet exist.

Data Types of PL/SQL

- PL/SQL is super set of the SQL. So, it supports all the data types provided by SQL.
- Along with this, in PL/SQL provides subtypes of the data types.
- For example, the data type NUMBER has a subtype called INTEGER.
- These subtypes can be used in PL/SQL block to make the data type compatible with the data types of the other programming languages.

Continue…

Category	Data Type	Subtypes/ Value
Numerical	NUMBER	BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NATURAL, POSITIVE, REAL, SMALLINT
Character	CHAR, VARCHAR, LONG	CHARACTER,VARCHAR, STRING, NCHAR, NVARCHAR2
Date	DATE	YEAR, MONTH, DAY, HOUR
Binary	RAW, LONG RAW	Variable-length is binary
Boolean	BOOLEAN	Can have values like TRUE, FALSE and NULL.
Row ID	ROWID	Stores values of address of each record.

Anchored Data type

Anchored data type means the data type for a variable is determined based on the datatype of another object.

This object can be another variable or column of the table. This provide ability to match the data type of the variables with the data types of the columns. Here, if the data type of column is changed, then data type of variable also changes automatically.

Syntax : variableName object%TYPE [NOT NULL] := initialValue;

Example : no Account.ano%TYPE:
bal Account.balance%TYPE;

4.2 Advantages of PL/SQL

1) Procedural Capabilities:

- PL/SQL provides procedural capabilities such as condition checking, branching and looping.
- This enables programmers to control execution of a program based on some conditions and user inputs.

2) Support to variables:

- PL/SQL supports declaration and use of variables.
- These variables can be used to store intermediate results of a query or some expression.

3) Error Handling:

- When an error occurs, user friendly message can be displayed.
- Also, execution of program can be controlled instead of abruptly terminating the program.

Continue…

4) User Defined Functions:

- Along with a large set of in-build functions, PL/SQL also supports user defined functions and procedures.

5) Portability:

- Programs written in PL/SQL are portable.
- It means, programs can be transferred and executed from any other computer hardware and operating system, where Oracle is operational.

6) Sharing of Code:

- PL/SQL allows users to store compiled code in a database. This code can be accessed and shared by different applications.
- This code can be executed by other programming languages like JAVA.

4.3 PL/SQL Block

DECLARE ---Optional

<Declarations Section>

BEGIN ---Mandatory

<Executable Commands>

EXCEPTION ---Optional

<Exception Handling>

END; ---Mandatory

Declaring Variable & Constant

Syntax : variableName datatype [NOT NULL] := initialValue;

Example : no NUMBER (3);
value DECIMAL;
city CHAR (10);
name VARCHAR (14);
counter NUMBER (2) NOT NULL := 0;

Syntax : constantName CONSTANT datatype := initialValue;

Example : pi CONSTANT NUMBER (3, 2) := 3.14;

Assigning Value

1) By using the assignment operator:

Syntax : variableName := value;

Example: no := 10;

2) By reading from the keyboard:

Syntax : variableName := &variableName;

Example : no := &no;

3) Selecting or Fetching table data value into variables:

Syntax : SELECT col1, col2, …, colN INTO var1, var2, …, varN
FROM tableName
WHERE condition ;

Example : SELECT balance INTO bal from Account
where ano = no;

Continue…

Example :

```
DECLARE
        no          Account.ano%TYPE;
        bal         Account.balance%TYPE;
BEGIN
        SELECT ano, balance INTO no, bal
        FROM  Account
        WHERE  ano = 'A01';
END;
```

Displaying Messages

Syntax : dbms_output.put_line (message);

Statement…..	Output…..
dbms_output.put_line (‘Hello World !!’);	Hello World !!
dbms_output.put_line (‘Sum = ‘ || 25);	Sum = 25
dbms_output.put_line (‘PI = ‘ || pi);	PI = 3.14
dbms_output.put_line (‘Square of ‘ || 3 || ‘ is ‘ || 9.);	The Square of 3 is 9.

Comments

1) Single Line Comment

" - - " Treats a is single line as comment.

Example : " This is single line comment"

2) Multiple Line Comment

/* …. */ Treats multiple line as a comments.

Example : /* This statement is sperate over two line, and
both lines are treated as comments. */

Example of PL/SQL Block

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

Output:

```
Hello World!

PL/SQL procedure successfully completed.
```

4.4 Creating and Executing of PL/SQL Block

To create and execute a PL/SQL block, follow the steps given below:

- Open any editor like as notepad. An EDIT command can be used on SQL prompt to open a notepad from the SQL * PLUS environment.

The following syntax creates and opens a file:

Syntax: EDIT filename

Example: EDIT D:/PLSQL/test.sql

Create and opens a file named ‘test.sql’.

Continue…

- Write a program code or statements in a file and save it.
- File should have '.sql' extension and last statement in file should be '/'.

To execute this block, use any of the following commands on prompt.

1. RUN filename
2. START filename
3. @ filename

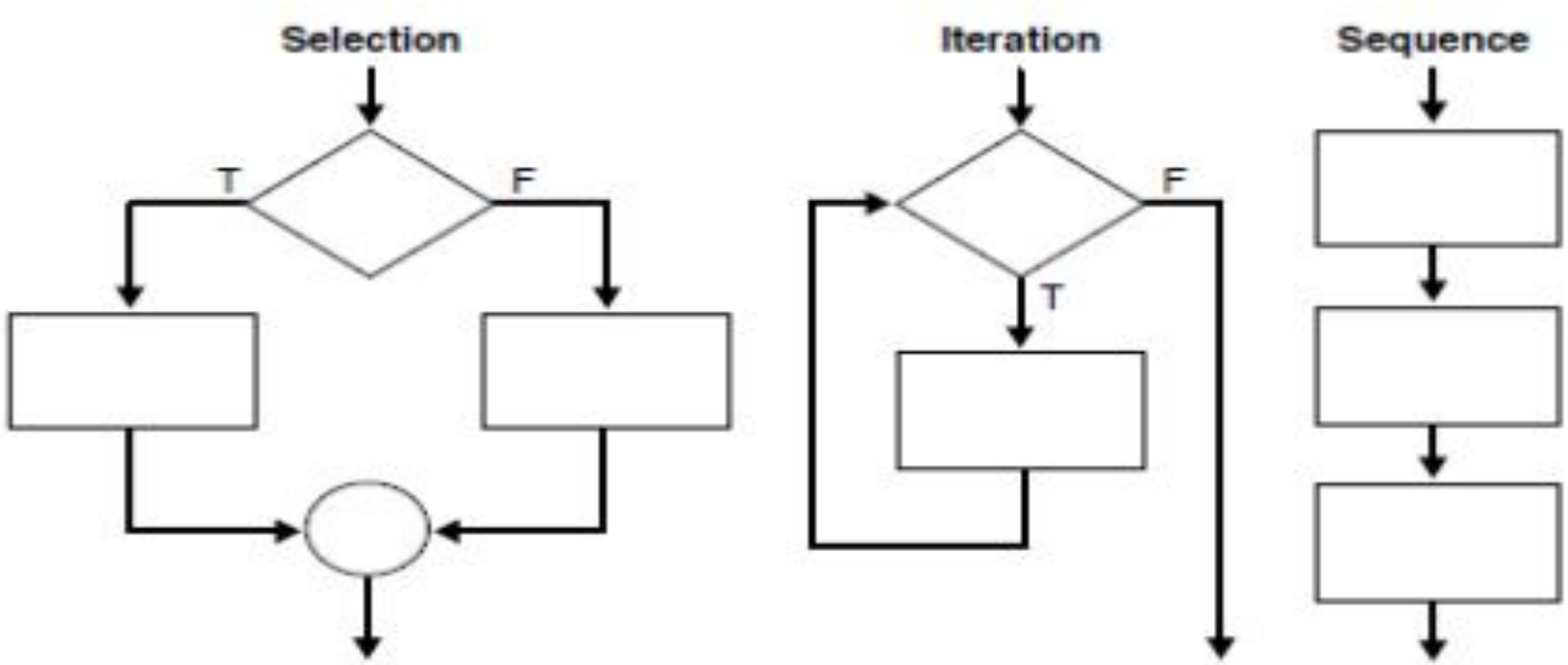
Example: @ D:/PLSQL/test.sql

Above command executes a block saved in a file, 'test.sql', created earlier.

4.5 Control Structures

In PL/SQL , the flow of execution can be controlled in three different manners as given below:

- Conditional Control
- Iterative Control
- Sequential Control



Continue…

Describe various example using the following employee table.

Table : employee

emp_id	emp_name	salary	city
E01	Abir Varma	40000	Rajkot
E02	Ruhi Sharma	53000	Pune
E03	Yug Modi	60000	Jamnagar
E04	Ved Parmar	48000	Pune
E05	Diya Mehta	53000	Rajkot

A) Conditional Control

1. IF-THEN Statement

Syntax:

IF condition
THEN
Statement;
END IF;

=> This syntax is used when user needs to execute statements when condition is true.

Continue…

2. IF-THEN-ELSE Statement

Syntax:

IF condition
THEN
[Statements to execute when condition is TRUE]
ELSE
[Statements to execute when condition is FALSE]
END IF;

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

Continue…

3. IF-THEN-ELSIF-ELSE Statement

Syntax:

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;

=> This syntax is used to execute one set of statements if condition1 is TRUE, a different set of statements when condition2 is TRUE or a third set of statements when both condition1 and condition2 are false.

Continue…

Example :

Write a program to read number from the user and determine whether it is odd or even.

Input :

```
DECLARE
      no NUMBER;                    - - declare a variable to store number.
BEGIN
      - - read a number from the user….
      no := &no;
      - - check the result of the MOD function….
      IF MOD (no, 2) = 0 THEN
            dbms_output.put_line ('Given number '|| no || ' is EVEN.');
      ELSE
             dbms_output.put_line ('Given number '|| no || ' is ODD.');
      END IF;
END;
/
```

Continue…

Output :

Enter value for no: 7

Given number 7 is ODD.

B) Iterative Control

- Iterative control allows a group of statements to execute repeatedly in a program. It is called Looping.
- Loops are iterative control statements.
- They are used to repeat execution of one or more statements for a define number of times.

1) LOOP
2) WHILE
3) FOR

1) LOOP

Syntax :

LOOP

- - Execute commands …..

END LOOP

- LOOP is an infinite loop. It executes commands in its body infinite times.
- So, it requires EXIT statement within its body to terminate the loop after executing specific iterations.

Example : Display number from 1 to 5 and their square using loop

Continue…

Input :

```
DECLARE
        - - declare required variable….
        counter   NUMBER(3) := 1;
BEGIN
        - - display headers for output….
        dbms_output.put_line(‘ value ‘ || ‘ square ‘ );
        - - traverse loop…
LOOP
        EXIT WHEN counter > 5;
        dbms_output.put_line(‘ ‘ || counter ||’ ‘||counter*counter);
        counter := counter + 1;
END LOOP;
END;
/
```

Continue…

Output :

value	square
1	1
2	4
3	9
4	16
5	25

PL/SQL procedure successfully completed.

2) WHILE

Syntax :

WHILE condition

LOOP

- - Execute commands …..

END LOOP

- WHILE loop execute commands in its body as long as the condition remains TRUE.
- The loop terminates when the condition evaluates to FALSE or NULL
- The EXIT statement can also be used to exit the loop.

Continue…

Example : Display number from 1 to 5 and their square using while.

Input :

```
DECLARE
      - - declare required variable….
      counter   NUMBER(3) := 1;
BEGIN
      - - display headers for output….
      dbms_output.put_line(‘ value ‘ || ‘ square ‘ );

      - - traverse loop…
      WHILE  counter <= 5
      LOOP
            dbms_output.put_line(‘ ‘ || counter ||’ ‘||counter*counter);
            counter := counter + 1;
      END LOOP;
END;
/
```

Continue…

Output :

value	square
1	1
2	4
3	9
4	16
5	25

PL/SQL procedure successfully completed.

3) FOR

Syntax :

FOR variable **IN [REVERCE]** start .. end

LOOP

- - Execute commands …..

END LOOP

- The for loop can be used when the number of iterations to be executed is known.
- Variable is a loop control variable. It is declare implicitly by PL/SQL. So, it should not be declared explicitly.
- For loop always increment by 1 and any other increment value can not be specified.
- A start and end specify lower and upper bound limit
- It REVERCE keyword is provided, the loop executed in reverse order, i.e. from end to start.

Continue…

Example : Display number from 1 to 5 and their square using for.

Input :

```
BEGIN
      - - display headers for output….
      dbms_output.put_line(‘ value ‘ || ‘ square ‘ );

      - - traverse loop…
      FOR  counter IN 1 .. 5
      LOOP
            dbms_output.put_line(‘ ‘ || counter ||’ ‘||counter*counter);
      END LOOP;
END;
/
```

Continue…

Output :

value	square
1	1
2	4
3	9
4	16
5	25

PL/SQL procedure successfully completed.

C) Sequential Control

Typically, execution proceeds sequentially within the block of the code. But, this sequence can be changed conditionally, as shown above, as well as unconditionally.

The GOTO statement can be used to alter the sequence unconditionally.

GOTO Statement

Syntax :

```
GOTO  jumpHere;
      :
      :
      :
<< jump >>
```

Continue…

Input : **BEGIN**

```
dbms_output.put_line('Code Start …..');
 dbms_output.put_line('Before GOTO statement');
GOTO jump;
 dbms_output.put_line('This statement will not get executed');
<<jump>> dbms_output.put_line('The flow of execution jumped here…');
END;
```

Output : Code Start …..

Before GOTO statement

The flow of execution jumped here…

4.6 Exception

- Exceptions are the method of handling the errors that occur during the execution of the programs.
- These errors are the results of data values that occur as a result of program execution.
- The developer will not know in prior where and when the error can occur. But he will have an idea that where errors might occur.
- In such cases, he adds exceptions to handle the program so that it does not fail, but it should display a proper message or alternative method to execute the program without failing it.
- Exception may contain a message to the user or block of code which will be executed in case of failure.
- Exception has two types :

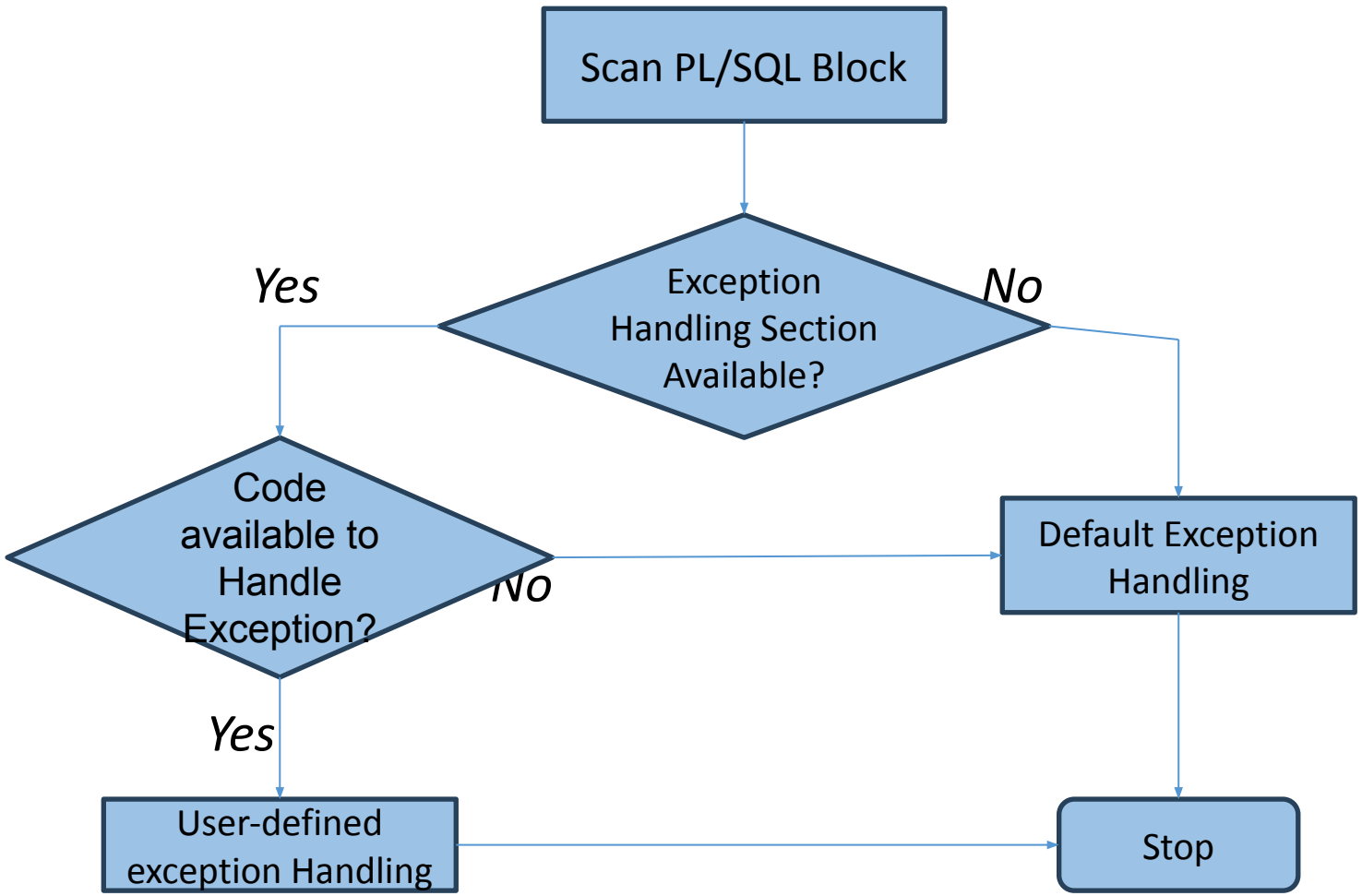
  1) Pre-defined Exception

  2) User-defined Exception

Continue…

Syntax: **WHEN** exception **THEN** statement;

```
DECLARE
    declarations section;
BEGIN
    executable command(s);
EXCEPTION
    WHEN exception1 THEN
    statement1;
    WHEN exception2 THEN
    statement2;
    [WHEN others THEN]
    /* default exception handling code */
END;
```



A) Pre-defined/System Defined Exception

Named System exceptions –

These are the predefined exceptions created by the SQL to handle the known types of errors in the code. They are also known as named exceptions. They are defined by the SQL and need not be redefined by the user. These exceptions need to be handled in the exception block. But they will be raised automatically by the SQL when such error occurs in the code.

For example, 'ZERO_DIVIDE' is the predefined exception to handle division by zero.

Few of the pre-defined named exceptions are here:

Continue…

Exception Name	Reason
CURSOR_ALREADY_OPEN	When trying to open a cursor that is already open.
INVALID_CURSOR	While performing an invalid operation on a cursor like closing a cursor or fetch data from a cursor that is not opened.
NO_DATA_FOUND	When a SELECT…INTO clause does not return any row from a table.
TOO_MANY_ROWS	When trying to SELECT or fetch more than one row into a record or variable.
ZERO_DIVIDE	While dividing a number by zero.
DUP_VAL_ON_INDEX	It is raised when duplicate values are attempted to be stored in a column with a unique index.
INVALID_NUMBER	It is raised when the conversion of a character string into a number fails since the string is not a valid number

Continue…

Input :

DECLARE

- - declared required variables…

no Account.ano%TYPE;

bal Account.bal%TYPE;

branch Account.branch%TYPE;

Continue…

```
BEGIN
    - - read ano, balance and branch name for a new record…
        no  := &no;
        bal  := &bal;
        branch := &branch;
    - - insert record into the Account table…
        INSERT INTO Account VALUES (no, bal, branch);
    - - commit and display a message confirming insertion…
        COMMIT;
        dbms_output.put_line('Record inserted successfully…');
EXCEPTION
    - - handle named exception…
        WHEN DUP_VAL_ON_INDEX    THEN
        dbms_output.put_line('Duplicate value found in Primary Key');
END;
/
```

Continue…

Output :

Enter value for no: ‘A01’
Enter value for bal: 5000
Enter value for branch: ‘vvn’
Record inserted successfully…

Enter value for no: ‘A01’
Enter value for bal: 10000
Enter value for branch: ‘ksad’
Duplicate value found in Primary Key

B) User Defined Exception

1) Numbered exceptions: –

- This exception declare in declaration section.
- This exception used PRAGMA keyword to bind exception to some name
- Function EXCEPTION_INIT takes two parameters :
  - one is the exception name
  - and the other is the number of the exception to be handled.

Input : **DECLARE**

```
- - declared exception and bind it …
    exNULL  EXCEPTION;
    PRAGMA EXCEPTION_INIT(exNULL, -1400);
- - declared required variables…
    no      Account.ano%TYPE;
    bal     Account.bal%TYPE;
    branch  Account.branch%TYPE;
```

Continue…

```
BEGIN
    - - read ano, balance and branch name for a new record…
        no  := &no;
        bal  := &bal;
        branch := &branch;
    - - insert record into the Account table…
        INSERT INTO Account VALUES (no, bal, branch);
    - - commit and display a message confirming insertion…
        COMMIT;
        dbms_output.put_line('Record inserted successfully…');
EXCEPTION
    - - handle named exception…
        WHEN DUP_VAL_ON_INDEX    THEN
         dbms_output.put_line('Duplicate value found in Primary Key');
```

Continue…

```
    - - handle numbered exception…
        WHEN  exNull    THEN
         dbms_output.put_line('Null value found in Primary Key');
END;
/
```

Output :

Enter value for no: 'A052'
Enter value for bal: 6000
Enter value for branch: 'ksad'
Record inserted successfully…

Enter value for no: null
Enter value for bal: 7000
Enter value for branch: 'anand'
Null value found in Primary Key

Continue…

2) User defined exceptions: –

Input :

```
DECLARE
    - - declared exception and bind it …
        exNULL  EXCEPTION;
        PRAGMA EXCEPTION_INIT(exNULL, -1400);
    - - declared user defined exception …
        myEx EXCEPTION;
    - - declared required variables…
        no        Account.ano%TYPE;
        bal       Account.bal%TYPE;
        branch  Account.branch%TYPE;
```

Continue…

```
BEGIN
    - - read ano, balance and branch name for a new record…
        no  := &no;
        bal  := &bal;
        branch := &branch;
    - - check balance; if negative, raise 'myEx' exception …
        IF bal > 0 THEN
            RAISE myEx;
        END IF;
    - - insert record into the Account table…
        INSERT INTO Account VALUES (no, bal, branch);
    - - commit and display a message confirming insertion…
        COMMIT;
        dbms_output.put_line('Record inserted successfully…');
```

Continue…

```
EXCEPTION
    - - handle named exception…
        WHEN DUP_VAL_ON_INDEX    THEN
         dbms_output.put_line('Duplicate value found in Primary Key');
    - - handle numbered exception…
        WHEN  exNull    THEN
         dbms_output.put_line('Null value found in Primary Key');
    - - handle  user-defined exception…
        WHEN myEx    THEN
         dbms_output.put_line('Balance can not be negative value');
END;
/
```

Continue…

Output :

Enter value for no: ‘A03’
Enter value for bal: 7000
Enter value for branch: ‘anand’
Record inserted successfully…

Enter value for no: ‘A04’
Enter value for bal: -2000
Enter value for branch: ‘anand’
Balance can not be negative value

4.7 Cursors

- When an SQL statement is processed, A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

- There are two types of cursors:
  Implicit Cursors
  Explicit Cursors

1) 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.

For Example,

- When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

Continue…

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

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

Continue…

Example: Create customers table and have records:

ID	Name	Age	Address	Salary
1	Ramesh	20	Rajkot	20000
2	Suresh	25	Raipur	25000
3	Neha	22	Jetput	15000
4	Mansi	23	Gondal	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:

Continue…

Input :

```
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;
/
```

Continue…

Output :

4 customers updated

PL/SQL procedure successfully completed.

=> Now, if you check the records in the customer table, you will find that the rows are updated.

ID	Name	Age	Address	Salary
1	Ramesh	20	Rajkot	25000
2	Suresh	25	Raipur	30000
3	Neha	22	Jetput	20000
4	Mansi	23	Gondal	35000

2) 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 :

CURSOR cursor_name IS select_statement;

Where, cursor_name – A suitable name for the cursor.

select_statement – A select query which returns multiple rows

Continue…

You must follow these steps while working with an explicit cursor.

1) Declare the cursor:

- **Declare** the cursor to initialize in the memory.
- It defines the cursor with a name and the associated SELECT statement.
- **Syntax**: CURSOR name IS
  SELECT statement;

2) Open the cursor:

- **Open** the cursor to allocate memory.
- 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:** OPEN cursor_name;

Continue…

3) Fetch the cursor:

- **Fetch** the cursor to retrieve data.
- It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:
- **Syntax**: FETCH cursor_name INTO variable_list;

4) Close the cursor:

- **Close** the cursor to release allocated memory.
- It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.
- **Syntax**: CLOSE cursor_name;

Continue…

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

Attribute	Description
SQL%FOUND	If record was fetched successfully in the last FETCH statement, returns TRUE. Else returns FALSE. It indicating no more record available in an active data set.
SQL%NOTFOUND	If record was not fetched successfully in the last FETCH statement, returns TRUE. Else returns FALSE.
SQL%ISOPEN	If the explicit cursor is open, return TRUE; Else return FALSE.
SQL%ROWCOUNT	It returns the number of records fetched from active data set. It is set to zero when the cursor is opened.

Continue…

Example: Create customers table and have records:

id	name	age	address	salary
1	Ramesh	20	Rajkot	20000
2	Suresh	25	Raipur	25000
3	Neha	22	Jetput	15000
4	Mansi	23	Gondal	30000

Input :

```
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;
```

Continue…

```
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;
/
```

Continue…

Output:

1 Ramesh Rajkot

2 Suresh Raipur

3 Neha Jetpur

4 Mansi Gondal

PL/SQL procedure successfully completed.

4.8 Procedures & Function

- A procedure and function is a group of PL/SQL statements that performs specific task.
- A procedure and function is a named PL/SQL block of code. This block can be compiled and a successfully compiled block can be stored in the Oracle database. This procedure and function is called Stores Procedure and Function.
- We can pass parameters to procedures and functions. So that their execution can be changed dynamically.

Procedures:

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:

Continue…

- **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.

Continue…

1) Create Procedure

Syntax :

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

Continue…

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 VARCHAR (100));

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

=> Procedure Code:

```
CREATE or REPLACE PROCEDURE “insertuser”

    (id in NUMBER, name IN VARCHAR)
IS
BEGIN
    INSERT INTO user VALUES (id, name);
END;
```

Continue…

Output:

Procedure created

=> 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

record inserted successfully.

Continue…

2) Drop Procedure

Syntax:

DROP PROCEDURE procedure_name;

Example:

Drop Procedure “insertuser”;

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

Syntax :

```
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];
```

Continue…

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.

Continue…

Example:

```
CREATE  or REPLACE function adder(n1 in number, n2 in number)
    return number
IS
    - - declare variable…
    n3 number(8);
BEGIN
    n3 :=n1+n2;
    return n3;
END;
/
```

Continue…

=> Program to call Function

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

Output:

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

Continue…

Example: Function example using table:

Table : customers

id	name	age	address	salary
1	Ramesh	20	Rajkot	20000
2	Suresh	25	Raipur	25000
3	Neha	22	Jetput	15000
4	Mansi	23	Gondal	30000

Continue…

=> Create Function

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

=> Calling Function

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

Output :

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

4.9 Packages

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

A package is compiled and then stored in the database, which then can be shared with many applications.

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

1. Package Specification
2. Package Body

Continue…

1) Package Specification

Syntax :

```
CREATE OR REPLACE PACKAGE packageName
IS
    - - package specification…
END packageName;
```

2) Package Body

Syntax :

```
CREATE OR REPLACE PACKAGE BODY packageName
IS
    - - package specification…
END packageName;
```

Example

```
-- Create a PL/SQL package specification

CREATE OR REPLACE PACKAGE math_operations IS


 -- Procedure to add two numbers with an output parameter

PROCEDURE add_numbers(x NUMBER, y NUMBER, result OUT NUMBER);


 -- Function to multiply two numbers

FUNCTION multiply_numbers(x NUMBER, y NUMBER) RETURN NUMBER;


END math_operations;
  /
```

Continue…

```
-- Create the body of the math_operations package
CREATE OR REPLACE PACKAGE BODY math_operations IS
 -- Implementation of the add_numbers procedure
 PROCEDURE add_numbers(x NUMBER, y NUMBER, result OUT NUMBER) IS
  BEGIN
     result := x + y;
  END add_numbers;
```

Continue…

```
-- Implementation of the multiply_numbers function
  FUNCTION multiply_numbers(x NUMBER, y NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN x * y;
  END multiply_numbers;
END math_operations;
/
-- PL/SQL block to test the math_operations package
DECLARE
  -- Declare variables to store results
  sum_result NUMBER;
  product_result NUMBER;
```

Continue…

```
BEGIN
  -- Call the procedure and pass output parameter
  math_operations.add_numbers(5, 7, sum_result);
  -- Display the result of the add_numbers procedure
  DBMS_OUTPUT.PUT_LINE('Sum Result: ' || sum_result);


  -- Call the function and retrieve the result
  product_result := math_operations.multiply_numbers(3, 4);
  -- Display the result of the multiply_numbers function
  DBMS_OUTPUT.PUT_LINE('Product Result: ' || product_result);
END;
  /
```

Continue…

OUTPUT :

PACKAGE CREATED.

PACKAGE BODY CREATED.

STATEMENT PROCESSED.

Sum Result : 12

Product Result : 12

Advantages of Package

The advantages of the Packages are described below:

- **Modularity**: Packages provide a modular structure, allowing developers to organize and manage code efficiently.
- **Code Reusability**: Procedures and functions within a package can be reused across multiple programs, reducing redundancy.
- **Private Elements**: Packages support private procedures and functions, limiting access to certain code components.
- **Encapsulation**: Packages encapsulate related logic, protecting internal details and promoting a clear interface to other parts of the code.

4.10 Triggers

- Triggers are stored programs, which are automatically executed or fired when some events occur.
- "A trigger is a group or set of SQL and PL/SQL statements executed by the software itself."
- Trigger are like procedures and functions, through not exactly the same.
- The main characteristic of the trigger is executed automatically based on database events.

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).

Continue…

Benefits of Triggers

Triggers can be written for the following purposes –

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Types of Trigger

The above diagram clearly indicated that Triggers can be classified into three categories:

1. Level Triggers
2. Event Triggers
3. Timing Triggers

Level Triggers

1. **ROW LEVEL TRIGGERS**
   - It fires for every record that got affected with the execution of DML statements like INSERT, UPDATE, DELETE etc.
   - It always use a FOR EACH ROW clause in a triggering statement.
2. **STATEMENT LEVEL TRIGGERS**
   - It fires once for each statement that is executed.

Continue…

Event Triggers

1. **DDL EVENT TRIGGER**
   - It fires with the execution of every DDL statement(CREATE, ALTER, DROP, TRUNCATE).
2. **DML EVENT TRIGGER**
   - It fires with the execution of every DML statement(INSERT, UPDATE, DELETE).
3. **DATABASE EVENT TRIGGER**
   - It fires with the execution of every database operation which can be LOGON, LOGOFF, SHUTDOWN, SERVERERROR etc.

Continue…

Timing Triggers

1. **BEFORE TRIGGER**
   - It fires before executing DML statement.
   - Triggering statement may or may not executed depending upon the before condition block.
2. **AFTER TRIGGER**
   - It fires after executing DML statement.

1) Create Trigger

CREATE [OR REPLACE] TRIGGER trigger_name

{BEFORE | AFTER | INSTEAD OF } {INSERT [OR] | UPDATE [OR] | DELETE}

ON table_name

[FOR EACH ROW]

WHEN (condition)

DECLARE

  Declaration-statements

BEGIN

  Executable-statements

EXCEPTION

  Exception-handling-statements

END;

Continue…

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.
- **[ON table_name]:** This specifies the name of the table associated with the trigger.
- **[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.

Continue…

=> Create Triggers

```
CREATE 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;
/
```

After executed this code OUTPUT will be…..

Trigger Created.

Continue…

Example:

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

ID	Name	Age	Address	Salary
1	Ramesh	20	Rajkot	20000
2	Suresh	25	Raipur	25000
3	Neha	22	Jetput	15000
4	Mansi	23	Gondal	30000
5	Komal	30	Junagadh	17000

Continue…

=> Check the salary difference by procedure:

```
DECLARE
        total_rows number(2);
BEGIN
        UPDATE  customers
        SET salary = salary + 5000;
            IF sql%notfound THEN
                dbms_output.put_line('no customers updated');
            ELSE IF sql%found THEN
                total_rows := sql%rowcount;
                dbms_output.put_line( total_rows || ' customers updated ');
            END IF;
END;
/
```

Continue…

OUTPUT:

Old salary: 20000
New salary: 25000
Salary difference: 5000

Old salary: 25000
New salary: 30000
Salary difference: 5000

Old salary: 15000
New salary: 20000
Salary difference: 5000

Old salary: 30000
New salary: 35000
Salary difference: 5000

Old salary: 17000
New salary: 23000
Salary difference: 5000

5 customers updated

Continue…

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.

OUTPUT:

Old salary: 25000
New salary: 30000
Salary difference: 5000

Old salary: 35000
New salary: 40000
Salary difference: 5000

Old salary: 30000
New salary: 35000
Salary difference: 5000

Old salary: 23000
New salary: 28000
Salary difference: 5000

Old salary: 20000
New salary: 25000
Salary difference: 5000

5 customers updated

Questions

Q-1) What is PL/SQL block? Also write advantages of PL/SQL block.

Q-2) How to declare variable, assign value to the variable and display message in PL/SQL block.

Q-3) Explain comments in PL/SQL block.

Q-4) Write about conditional control with example in PL/SQL block.

Q-5) Explain iterative control (LOOP, WHILE and FOR statement) with example.

Q-6) Write GOTO statement with example.

Q-7) What is Exception in PL/SQL block? Give example of user-defined exception.

Q-8) Explain cursor in PL/SQL block.

Q-9) How to create function in PL/SQL block? Give one example.

Q-10) What is trigger in PL/SQL ?

SLTIET
आत्मदीपोभव

Thank You

CREATING CREATORS - SLTIET