PLSQL WITH EXAMPLES

==========

1. IMPORTANCE OF PLSQL

WHAT IS PLSQL

PL/SQL is a block-structured language. The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks. Pl/SQL stands for "Procedural Language extension of SQL" which is used in Oracle. PL/SQL is integrated with the Oracle database (since version 7). The functionalities of PL/SQL are usually extended after each release of the Oracle database. Although PL/SQL is closely integrated with SQL language, it adds some programming constraints that are not available in SQL.

PL/SQL Functionalities

PL/SQL includes procedural language elements like conditions and loops. It allows the declaration of constants and variables, procedures and functions, types and variables of those types, and triggers. It can support Array and handle exceptions (runtime errors). After the implementation of version 8 of the Oracle database has included features associated with object orientation. You can create PL/SQL units like procedures, functions, packages, types, triggers, etc. which are stored in the database for reuse by applications.

With PL/SQL, you can use SQL statements to manipulate Oracle data and flow of control statements to process the data.

PL/SQL is known for its combination of the data manipulating power of SQL with the data processing power of procedural languages. It inherits the robustness, security, and portability of the Oracle Database.

PL/SQL is not case-sensitive so you are free to use lowercase letters or uppercase letters except within string and character literals. A line of PL/SQL text contains groups of characters known as lexical units.

2. PLSQL BASICS

PL/SQL Variables

A variable is a meaningful name which facilitates a programmer to store data temporarily during the execution of code. It helps you to manipulate data in PL/SQL

programs. It is nothing except a name given to a storage area. Each variable in the PL/SQL has a specific data type which defines the size and layout of the variable's memory.

A variable should not exceed 30 characters. Its letter optionally followed by more letters, dollar signs, numerals, underscore etc.

1. It needs to declare the variable first in the declaration section of a PL/SQL block before using it. By default, variable names are not case sensitive. A reserved PL/SQL keyword cannot be used as a variable name.

How to declare variable in PL/SQL

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

Syntax for declaring variable:

Following is the syntax for declaring variable:

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

Here, variable_name is a valid identifier in PL/SQL and datatype must be valid PL/SQL data type. A data type with size, scale or precision limit is called a constrained declaration. The constrained declaration needs less memory than unconstrained declaration.

Example:

```
Radius Number: = 5;
Date_of_birth date;
```

Declaration Restrictions:

- In PL/SQL while declaring the variable some restrictions hold.
- Forward references are not allowed i.e you must declare a constant or variable before referencing it in another statement even if it is a declarative statement
- Val number: =Total-200;

PLSQL WITH EXAMPLES

==========

- Total number: =1000;
- The first declaration is illegal because the TOTAL variable must be declared before using it in an assignment expression.
- Variables belonging to the same datatype cannot be declared in the same statement. N1,N2,N3Number;
  It is an illegal declaration.

Naming rules for PL/SQL variables

The variable in PL/SQL must follow some naming rules like other programming languages.

- The variable_name should not exceed 30 characters.
- Variable name should not be the same as the table table's column of that block.
- The name of the variable must begin with ASCII letter. The PL/SQL is not case sensitive so it could be either lowercase or uppercase. For example: v_data and V_DATA refer to the same variables.
- You should make your variable easy to read and understand, after the first character, it may be any number, underscore (_) or dollar sign ($).
- NOT NULL is an optional specification on the variable.

Initializing Variables in PL/SQL

Evertime you declare a variable, PL/SQL defines a default value NULL to it. If you want to initialize a variable with other value than NULL value, you can do so during the declaration, by using any one of the following methods.

The DEFAULT keyword

The assignment operator

```
1. counter binary_integer: = 0;
2. greetings varchar2(20) DEFAULT 'Hello World';
```

You can also specify a NOT NULL constraint to avoid a NULL value. If you specify the NOT NULL constraint, you must assign an initial value for that variable.

PLSQL WITH EXAMPLES

==========

You must have a good programming skill to initialize variables properly otherwise, sometimes program would produce unexpected result.

Example of initilizing variable

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

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

After the execution, this will produce the following result:

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

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL:

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

Local Variable: Local variables are the inner block variables which are not accessible to outer blocks.

Global Variable: Global variables are declared in outermost block.

Example of Local and Global variables

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

```
DECLARE
```

PLSQL WITH EXAMPLES

==========

```
-- Global variables
  num1 number := 95;
  num2 number := 85;
BEGIN
  dbms_output.put_line('Outer Variable num1: ' || num1);
  dbms_output.put_line('Outer Variable num2: ' || num2);
  DECLARE
    -- Local variables
    num1 number := 195;
    num2 number := 185;
  BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);
  END;
END;
/
```

After the execution, this will produce the following result:

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

PL/SQL procedure successfully completed.
```

Variable Attributes:

When you declare a PL/SQL variable to hold the column values, it must be of correct data types and precision, otherwise error will occur on execution. Rather than hard coding the data type and precision of a variable. PL/SQL provides the facility to declare a variable without having to specify a particular data type using %TYPE and %ROWTYPE attributes. These two attributes allow us to specify a variable and have that variable data type be defined by a table/view column or a PL/SQL package variable.

A % sign servers as the attribute indicator. This method of declaring variables has an advantage as the user is not concerned with writing and maintaining code.

Following are the types of Variable Attributes in PL/SQL.

- **%TYPE:**

PLSQL WITH EXAMPLES

==========

The %TYPE attribute is used to declare variables according to the already declared variable or database column. It is used when you are declaring an individual variable, not a record. The data type and precision of the variable declared using %TYPE attribute is the same as that of the column that is referred from a given table. This is particularly useful when declaring variables that will hold database values. When using the %TYPE keyword, the name of the columns and the table to which the variable will correspond must be known to the user. These are then prefixed with the variable name. If some previously declared variable is referred then prefix that variable name to the %TYPE attribute.

The syntax for declaring a variable with %TYPE is:

<var_name> <tab_name>.<column_name>%TYPE;

Where <column_name> is the column defined in the <tab_name>.

Consider a declaration.

```
SALARY EMP.SAL % TYPE;
```

This declaration will declare a variable SALARY that has the same data type as column SAL of the EMP table.

Example:

```
DECLARE
SALARY EMP.SAL % TYPE;
ECODE EMP.empno % TYPE;
BEGIN
Ecode :=&Ecode;
Select SAL into SALARY from EMP where EMPNO = ECODE;
dbms_output.put_line('Salary of ' || ECODE || 'is = || salary');
END;
```

After the execution, this will produce the following result:

```
Enter value for ecode: 7499
Salary of 7499 is = 1600
PL/SQL procedure successfully completed.
```

%ROWTYPE:

PLSQL WITH EXAMPLES

==========

The %ROWTYPE attribute is used to declare a record type that represents a row in a table. The record can store an entire row or some specific data selected from the table. A column in a row and corresponding fields in a record have the same name and data types.

The syntax for declaring a variable with %ROWTYPE is:

<var_name> <tab_name>.ROW%TYPE;

Where <variable_name> is the variable defined in the <tab_name>.

Consider a declaration.

EMPLOYEE EMP. % ROW TYPE;

This declaration will declare a record named EMPLOYEE having fields with the same name and data types as that of columns in the EMP table. You can access the elements of EMPLOYEE record as

```
EMPLOYEE.SAL := 10000;
EMPLOYEE.ENAME := 'KIRAN';
```

Example:

```
DECLARE
EMPLOYEE EMP. % ROW TYPE;
BEGIN
EMPLOYEE.EMPNO := 2092;
5   EMPLOYEE.ENAME := 'Sanju';
Insert into EMP where (EMPNO, ENAME) Values (employee.empno, employee.ename);
dbms_output.put_line('Row Inserted');
END;
```

After the execution, this will produce the following result:

```
Row Inserted
PL/SQL procedure successfully completed.
```

PL/SQL Constants

A constant is a value used in a PL/SQL block that remains unchanged throughout the program. It is a user-defined literal value. It can be declared and used instead of actual values.

PLSQL WITH EXAMPLES

==========

Let's take an example to explain it well:

Suppose, you have to write a program which will increase the salary of the employees upto 30%, you can declare a constant and use it throughout the program. Next time if you want to increase the salary again you can change the value of constant than the actual value throughout the program.

Syntax to declare a constant:

constant_name CONSTANT datatype := VALUE;

Constant_name:it is the name of constant just like variable name. The constant word is a reserved word and its value does not change.

VALUE: it is a value which is assigned to a constant when it is declared. It can not be assigned later.

Example of PL/SQL constant

Let's take an example to explain it well:

```
DECLARE
  -- constant declaration
  pi constant number := 3.141592654;
  -- other declarations
  radius number(5,2);
  dia number(5,2);
  circumference number(7, 2);
  area number (10, 2);
BEGIN
  -- processing
  radius := 9.5;
  dia := radius * 2;
  circumference := 2.0 * pi * radius;
  area := pi * radius * radius;
  -- output
  dbms_output.put_line('Radius: ' || radius);
  dbms_output.put_line('Diameter: ' || dia);
  dbms_output.put_line('Circumference: ' || circumference);
  dbms_output.put_line('Area: ' || area);
END;
```

PLSQL WITH EXAMPLES

==========

/

After the execution of the above code at SQL prompt, it will produce the following result:

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53
 Pl/SQL procedure successfully completed.
```

PL/SQL Literals

Literals are the explicit numeric, character, string or boolean values which are not represented by an identifier. For example: TRUE, NULL, etc. are all literals of type boolean. PL/SQL literals are case-sensitive. There are following kinds of literals in PL/SQL:

Numeric Literals

Character Literals

String Literals

BOOLEAN Literals

Date and Time Literals

Example of these different types of Literals:

Literals	Examples
Numeric	75125, 3568, 33.3333333 etc.
Character	'A' '%' '9' ' ' 'z' '('
String	Hello World
Boolean	TRUE, FALSE, NULL etc.
Date and Time	'26-11-2002' , '2012-10-29 12:01:01'

3.PLSQL BLOCKS

Introduction to PL/SQL Block Structure

As we know that PL/SQL is a Block Structure Language because in PL/SQL, we can write multiples block of code or we can arrange multiple codes into different groups of elements.

Before the start let's understand the block, In PL/SQL, Block is a piece of code or some unit of code.

Blocks are divided into two different categories.

1. **Anonymous Block**
2. **Named Block**

Anonymous Block: -

Which block is not given any name is called Anonymous Block.

Anonymous block has been defined into three parts declaration, execution, and exception handling section.

Execution Section is necessary where the declaration and exception handling section is optional.

It is not saved as a database object.

```
DECLARE
(declaration section) ---OPTIONAL---
BEGIN
(execution section)---MANDATORY---
EXCEPTION
(exception handling)--OPTIONAL--
END;
```

We will discuss the details of the Anonymous block in our next PL/SQL Tutorial.

Named Block: -

PLSQL WITH EXAMPLES

==========

Which block is given any name is called Named Block. It is different from the anonymous block.

In the Named Block, Header Section is added instead of declaring the keyword.

Header Section defines the types of Named Block we are creating. Procedure and Function are examples of Named Block.

The named Block is saved as a database object.

Named block is also an example of reusability. The named block create once and runs multiple times.

```
HEADER-------MANDORARY---
IS/AS
(DECLARATION SECTION) ---OPTIONAL---
BEGIN
(EXECUTION SECTION)---MANDORARY---
EXCEPTION
(EXCEPTION HANDLING)OPTIONAL
END;
```

4.CURSOR PLSQL

Context area:

When processing an SQL statement, Oracle creates a temporary work area in the system memory which contains all the information needed for processing the statement known as context area.

Cursor:

A cursor is a pointer to context area i.e. Context area is controlled by the cursor. It is used to fetch and manipulate the data returned by the SQL statement.

Note: The set of rows the cursor holds is known as active set.

Types of cursors:

1. Implicit cursors.

2. Explicit cursors.

Implicit cursors:

Implicit cursors are automatically generated by Oracle while processing an SQL statement when no explicit cursor for the statement is used. They are created by default when DML statements like DELETE, INSERT, UPDATE and SELECT are executed.

Oracle provides implicit cursor attributes to check the status of DML operations. When DML statements like INSERT, UPDATE, or DELETE are executed the cursor attributes tell us whether any rows are affected or not and how many have been affected. When a SELECT… INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement or not and returns an error when no data is selected.

Implicit cursor attributes:

%FOUND It returns TRUE if an INSERT, UPDATE or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise returns FALSE.

Example: SQL%FOUND

%NOTFOUND It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise returns FALSE.

Example: SQL%NOTFOUND

%ISOPEN Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

PLSQL WITH EXAMPLES

==========

Example: SQL%ISOPEN

%ROWCOUNT It returns the number of rows affected by an INSERT, UPDATE or DELETE statement or returned by a SELECT INTO statement.

Example: SQL%ROWCOUNT

Example:

```
DECLARE  var_rows number(2);
BEGIN
  UPDATE employees
  SET salary = salary + 2000;
  IF SQL%NOTFOUND THEN
    dbms_output.put_line('No record updated.');
  ELSIF SQL%FOUND THEN
    var_rows := SQL%ROWCOUNT;
    dbms_output.put_line(var_rows || ' records are updated.');
  END IF;
END;
/
```

Output:

```
10 records are updated.
```

Explicit cursors:

Explicit cursors are the user defined cursors to gain more control over the context area. These are defined in the declaration section of the PL/SQL block. An explicit cursor is created on a SELECT Statement which returns more than one row.

Syntax for creating an explicit cursor:

CURSOR cursor_name IS select_statement;

How to use explicit cursor?

1. **DECLARE** the cursor for initialization in the declaration section.

2. **OPEN** the cursor for memory allocation in the execution section.

3. **FETCH** the cursor for retrieving data in the execution section.

4. **CLOSE** the cursor to release allocated memory in the execution section.

Declaring the cursor:

CURSOR cur_students IS

SELECT rollNo, name, address FROM students;

Fetching the cursor:

FETCH cur_students INTO s_rollNo, s_name, s_address;

Closing the cursor:

CLOSE cur_students;

Example:

```
DECLARE
  s_rollNo students.rollNo%type;
  s_name students.name%type;
  s_address students.address%type;
  CURSOR cur_students is
    SELECT rollNo, name, address FROM students;
BEGIN
  OPEN cur_students;
  LOOP
    FETCH cur_students into s_rollNo, s_name, s_address;
    EXIT WHEN cur_students%notfound;
    dbms_output.put_line(s_rollNo || ' ' || s_name || ' ' || s_address);
  END LOOP;
  CLOSE cur_students;
END;
/
```

Output:

```
1 Vivek UK
2 Anil Delhi
3 Mahesh Rajasthan
4 Vishal Delhi
```

5 Binod UP

5.Exception Handling Plsql

Exception:

Exception refers to an exceptional event. Exception is an event that disrupts the normal flow of the program, during program execution.

Pl sql exception handling:

PL/SQL provides a mechanism to handle such exceptions so that normal flow of the program can be maintained.

Types of exceptions:

1. System-defined exceptions.

2. User-defined exceptions.

Syntax for exception handling:

```
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;
/
```

Example:

```
DECLARE
```

```
  s_rollNo students.rollNo%type := 10;
  s_name students.name%type;
  s_address students.address%type;
BEGIN
  SELECT rollNo, name, address FROM students WHERE rollNo = s_rollNo;
  dbms_output.put_line(s_rollNo || ' ' || s_name || ' ' || s_address);
  EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('No such student!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
Output:
No such student!
Pl sql raise exception:
```

Database server automatically raised the exceptions in case of any internal database error. But database exceptions can also be raised explicitly by using RAISE command.

Syntax of 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 exception:
```

The PL/SQL provides the facility to define the custom or user-defined exceptions according to the need of program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

PLSQL WITH EXAMPLES

==========

Syntax for declaring a user defined exception:

DECLARE custom-exception EXCEPTION;

Example:

```
DECLARE
  s_rollNo students.rollNo%type := &ss_rollNo;
  s_name students.name%type;
  s_address students.address%type;
   -- user defined exception
  ex_invalid_rollNo  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
     RAISE ex_invalid_rollNo;
  ELSE
   SELECT rollNo, name, address FROM students WHERE rollNo = s_rollNo;
   dbms_output.put_line(s_rollNo || ' ' || s_name || ' ' || s_address);
  END IF;
  EXCEPTION
    WHEN ex_invalid_rollNo THEN
     dbms_output.put_line('rollNo must be greater than zero!');
    WHEN no_data_found THEN
     dbms_output.put_line('No such student!');
    WHEN others THEN
     dbms_output.put_line('Error!');
END;
/
Output:
(Enter a value less than 0 for rollNo)
rollNo must be greater than zero!
Pl sql predefined exceptions list:
```

OUTPUT

(Enter a value less than 0 for rollNo)

```
rollNo must be greater than zero!
PLSQL PREDEFINED EXCEPTIONS LIST
```

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.

PLSQL WITH EXAMPLES

==========

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 s 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 s 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.

6 . PL/SQL Loop

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

The syntax for a basic loop:

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

There are 4 types of PL/SQL Loops.

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

PL/SQL Exit Loop (Basic Loop)

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

Syntax of basic loop:

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

Syntax of exit loop:

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

PLSQL WITH EXAMPLES

==========

Example of PL/SQL EXIT Loop

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

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

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

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

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

Initialize a variable before the loop body

Increment the variable in the loop.

You should use EXIT WHEN statement to exit from the Loop. Otherwise the EXIT statement without WHEN condition, the statements in the Loop is executed only once.

PL/SQL EXIT Loop Example 2

```
DECLARE
VAR1 NUMBER;
VAR2 NUMBER;
BEGIN
VAR1:=100;
VAR2:=1;
LOOP
DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
```

```
IF (VAR2=10) THEN
EXIT;
END IF;
VAR2:=VAR2+1;
END LOOP;
END;
Output:

100
200
300
400
500
600
700
800
900
1000
```

PL/SQL While Loop

PL/SQL while loop is used when a set of statements has to be executed as long as a condition is true, the While loop is used. The condition is decided at the beginning of each iteration and continues until the condition becomes false.

Syntax of while loop:

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

Example of PL/SQL While Loop

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

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

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

```
1
2
```

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

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

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

PL/SQL WHILE Loop Example 2

```
DECLARE
VAR1 NUMBER;
VAR2 NUMBER;
BEGIN
VAR1: =200;
VAR2: =1;
WHILE (VAR2<=10)
LOOP
DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
VAR2: =VAR2+1;
END LOOP;
END;
```

Output:

```
200
400
600
800
1000
1200
1400
1600
1800
2000
```

PL/SQL Continue Statement

PLSQL WITH EXAMPLES

==========

The continue statement is used to exit the loop from the reminder if its body either conditionally or unconditionally and forces the next iteration of the loop to take place, skipping any codes in between.

The continue statement is not a keyword in Oracle 10g. It is a new feature encorporated in oracle 11g.

For example: If a continue statement exits a cursor FOR LOOP prematurely then it exits an inner loop and transfer control to the next iteration of an outer loop, the cursor closes (in this context, CONTINUE works like GOTO).

Syntax:

continue;

Example of PL/SQL continue statement

Let's take an example of PL/SQL continue statement.

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

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

```
Inside loop:  x = 0
Inside loop:  x = 1
```

```
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

Note: The continue statement is not supported in Oracle 10g. Oracle 11g supports this as a new feature.

PL/SQL GOTO Statement

In PL/SQL, GOTO statement makes you able to get an unconditional jump from the GOTO to a specific executable statement label in the same subprogram of the PL/SQL block.

Here the label declaration which contains the label_name encapsulated within the << >> symbol and must be followed by at least one statement to execute.

Syntax:

GOTO label_name;

Here the label declaration which contains the label_name encapsulated within the << >> symbol and must be followed by at least one statement to execute.

GOTO label_name;

..

..

<<label_name>>

Statement;

Example of PL/SQL GOTO statement

Let's take an example of PL/SQL GOTO statement.

```
DECLARE
   a number(2) := 30;
BEGIN
   <<loopstart>>
```

```
  -- while loop execution
  WHILE a < 50 LOOP
    dbms_output.put_line ('value of a: ' || a);
    a := a + 1;
    IF a = 35 THEN
      a := a + 1;
      GOTO loopstart;
    END IF;
  END LOOP;
END;
/
```

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

```
TEFvalue of a: 30
value of a: 31
value of a: 32
value of a: 33
value of a: 34
value of a: 36
value of a: 37
value of a: 38
value of a: 39
value of a: 40
value of a: 41
value of a: 42
value of a: 43
value of a: 44
value of a: 45
value of a: 46
value of a: 47
value of a: 48
value of a: 49
```

Statement processed.

Restriction on GOTO statement

Following is a list of some restrictions imposed on GOTO statement.

Cannot transfer control into an IF statement, CASE statement, LOOP statement or sub-block.

Cannot transfer control from one IF statement clause to another or from one CASE statement WHEN clause to another.

Cannot transfer control from an outer block into a sub-block.

Cannot transfer control out of a subprogram.

Cannot transfer control into an exception handler.

7.IF ELSE PLSQL

PL/SQL If statement:

If statement is used to execute a block of statements if specified condition is true.

Commonly used PL/SQL If statement:

IF-THEN statement:

Syntax:

```
IF condition
THEN
  //Block of statements1
END IF;
```

Block of statements1 executes when the specified condition is true.

IF-THEN-ELSE statement:

Syntax:

```
IF condition
THEN
    //Block of statements1
ELSE
    //Block of statements2
END IF;
```

Block of statements1 executes when the specified condition is true otherwise Block of statements2 executes.

IF-THEN-ELSIF statement:

Syntax:

```
IF condition1
THEN
    //Block of statements1
ELSIF condition2
    //Block of statements2
```

```
ELSE
    //Block of statements3
END IF;
```

Block of statements1 executes when condition1 is true if false codition2 is checked and Block of statements2 executes if condition2 is true and so on. Block of statements in ELSE block executes when no condition is true.

Example:

```
DECLARE
  var number(3) := 50;
BEGIN
  IF (var = 10) THEN
    dbms_output.put_line('Value of var is 10');
  ELSIF (var = 20) THEN
    dbms_output.put_line('Value of var is 20');
  ELSIF (var = 30) THEN
    dbms_output.put_line('Value of var is 30');
  ELSE
     dbms_output.put_line('None of the above condition is true.');
  END IF;
  dbms_output.put_line('Exact value of var is: '|| var);
END;
/
```

Output:

None of the above condition is true.

Exact value of var is: 50

8. PL/SQL PROCEDURE

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

The procedure contains a header and a body.

PLSQL WITH EXAMPLES

==========

Header: The header contains the name of the procedure and the parameters or variables passed to the procedure.

Body: The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

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

IN parameters: The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.

INOUT parameters: The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

A procedure may or may not return any value.

PL/SQL Create Procedure

Syntax for creating procedure:

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

In this example, we are going to insert, update, or delete a record in the user table

How to create a procedure?

Procedure example without parameters:

```
create or replace procedure "INSERTUSER"
(id IN NUMBER,
name IN VARCHAR2)
is
```

PLSQL WITH EXAMPLES

==========

```
begin
insert into user values(id,name);
end;
/
```

Output:

Procedure created.

PL/SQL program to call procedure

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

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

Procedure example with parameters:

```
CREATE OR REPLACE PROCEDURE add_student(rollNo IN NUMBER, name IN VARCHAR2)
IS
  BEGIN
   insert into students values(rollNo,name);
  END;
/
```

How to execute stored procedure?

A procedure can be executed by using EXEC or EXECUTE statement.

EXEC procedure_name();

EXEC procedure_name;

Note: Execute procedure with parameters:

EXEC procedure_name(param1,param2…paramN);

A procedure can also be invoked from other PL SQL block.

```
BEGIN
  procedure_name;
```

END;

/

How to drop stored procedure?

DROP PROCEDURE procedure_name;

Procedure example out parameters:

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.

```
create or replace procedure emp_5(P_DEPT OUT number,P_MGR IN NUMBER)
AS
V_EMPNO           NUMBER(4) ;
V_ENAME           VARCHAR2(10);
V_JOB             VARCHAR2(9) ;
V_MGR             NUMBER(4)  ;
V_HIREDATE        DATE  ;
V_SAL             NUMBER(7,2);
V_COMM            NUMBER(7,2) ;
V_DEPTNO          NUMBER(2);
--P_DEPT NUMBER;
CURSOR C_EMP123 IS
 select *  from EMP WHERE  DEPT=P_DEPT;
BEGIN
 BEGIN
SELECT DEPT INTO P_DEPT FROM EMP WHERE EMPNO=P_MGR;
DBMS_OUTPUT.PUT_LINE('DEPT='||P_DEPT);
END;
OPEN C_EMP123;
LOOP
FETCH                                C_EMP123                                INTO
V_EMPNO,V_ENAME,V_JOB,V_MGR,V_HIREDATE,V_SAL,V_COMM,V_DEPTNO;
EXIT WHEN C_EMP123%notfound;
--INSERT INTO EMP_10(EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPT)
--
VALUES(V_EMPNO,V_ENAME,v_JOB,V_MGR,V_HIREDATE,V_SAL,V_COMM,V_DEPTN
O);
--UPDATE  EMP_10 SET SAL=(V_SAL+100) WHERE EMPNO=V_EMPNO;
DBMS_OUTPUT.PUT_LINE('EMPNO='||v_EMPNO);
DBMS_OUTPUT.PUT_LINE('ENAME='||V_ENAME);
DBMS_OUTPUT.PUT_LINE('JOB='||v_JOB);
```

PLSQL WITH EXAMPLES

==========

```
DBMS_OUTPUT.PUT_LINE('MGR='||V_MGR);
DBMS_OUTPUT.PUT_LINE('SAL='||V_SAL);
DBMS_OUTPUT.PUT_LINE('COMM='||V_COMM);
DBMS_OUTPUT.PUT_LINE('DEPTNO='||v_DEPTNO);
DBMS_OUTPUT.PUT_LINE('END');
 END LOOP;
    CLOSE C_EMP123;
EXCEPTION
WHEN no_data_found THEN
    dbms_output.put_line('No such customer!'||SQLERRM);
    WHEN too_many_rows THEN
    dbms_output.put_line('error trying to SELECT too many rows'||SQLERRM);
  WHEN others THEN
    dbms_output.put_line('Error!'||SQLERRM);
END;
```

A procedure can also be invoked from other PL SQL blocks.

```
declare
w varchar(50);
begin
emp_5( P_DEPT => W,P_MGR =>7369);
end;
/
```

9.PLSQL FUNCTION:

The pl SQL function is a named PL/SQL block that performs one or more specific tasks and must return a value.

How do pass parameters in a function?

We can use the below modes to pass the parameters in a function:

IN-PARAMETERS: These parameters are the read-only parameters. Function cannot change the value of IN parameters.

IN OUT-PARAMETERS: These parameters are read and write parameters i.e. a function can reads and change the IN OUT parameter value and return it back to the calling program.

PLSQL WITH EXAMPLES

==========

Syntax of pl sql function:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS|AS
    //Declaration block
BEGIN
    //Execution_block
    Return return_variable;
EXCEPTION
    //Exception block
    Return return_variable;
END;
/
```

How to create a function?

```
create or replace function getMultiple(num1 in number, num2 in number)
return number
is
  num3 number(8);
begin
  num3 :=num1*num2;
  return num3;
end;
/
```

How to execute a function?

A function's return value can be assigned to a variable.

```
result :=  getMultiple(4, 5);
```

As a part of a SELECT statement:

```
SELECT getMultiple(4, 5) FROM dual;
```

In a PL/SQL Statement:

```
dbms_output.put_line(getMultiple(4, 5));
```

PLSQL WITH EXAMPLES

==========

How to drop a function?

DROP FUNCTION function_name;

OUT-parameters: These parameters are the write-only parameters used to return values to the calling program. The function can change the value of OUT parameters.

```
WITH
 FUNCTION SURI_123(A IN NUMBER,C OUT NUMBER)
 RETURN NUMBER
AS
BEGIN
C:=A+A;
RETURN C*C;
dbms_output.put_line(' Minimum of (23, 45) : ' ||C );
END;

 FUNCTION SURI_1234( C IN NUMBER)RETURN NUMBER
AS
IV_A NUMBER;
IV_B NUMBER;
BEGIN
IV_A:=SURI_123(C,IV_B);
RETURN IV_A;
END;

SELECT SURI_1234(2) FROM DUAL;
```

10.TRIGGERS PLSQL

Oracle pl sql triggers:

A database trigger is a stored program which is automatically fired or executed when some events occur. A trigger can execute in response to any of the following events:

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

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

3. A database operation like SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN.

PLSQL WITH EXAMPLES

==========

Note: A trigger can be defined on the table, view, schema or database with which the event is associated.

Types of PL SQL triggers:

1. Row level trigger – An event is triggered at row level i.e. for each row updated, inserted or deleted.

2. Statement level trigger – An event is triggered at table level i.e. for each sql statement executed.

Syntax for creating a 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)
 BEGIN
   --- sql statements
 END;
/
```

Where:

CREATE [OR REPLACE] TRIGGER trigger_name – It creates a trigger with the given name or overwrites an existing trigger with the same name.

{BEFORE | AFTER | INSTEAD OF } – It specifies the trigger get fired. i.e before or after updating a table. INSTEAD OF is used to create a trigger on a view.

{INSERT [OR] | UPDATE [OR] | DELETE} – It specifies the triggering event. The trigger gets fired at all the specified triggering event.

[OF col_name] – It is used with update triggers. It is used when we want to trigger an event only when a specific column is updated.

PLSQL WITH EXAMPLES

==========

[ON table_name] – It specifies the name of the table or view to which the trigger is associated.

[REFERENCING OLD AS o NEW AS n] – It 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 old values cannot be referenced when inserting a record and new values cannot be referenced when deleting a record, because they do not exist.

[FOR EACH ROW] – It is used to specify whether a trigger must fire when each row being affected (Row Level Trigger) or just once when the sql statement is executed (Table level Trigger).

WHEN (condition) – It is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Existing data:

Select * from employees;

EMP_ID	NAME	AGE	ADDRESS	SALARY
1	Shveta	23	Delhi	50000
2	Bharti	22	Karnal	52000
3	Deepika	24	UP	54000
4	Richi	25	US	56000
5	Bharat	21	Paris	58000
6	Sahdev	26	Delhi	60000

TRIGGER:

```
CREATE OR REPLACE TRIGGER show_salary_difference
BEFORE DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
WHEN (NEW.EMP_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;
/
```

Note: The above trigger will execute for every INSERT, UPDATE or DELETE operations performed on the EMPLOYEES table.

Drop a trigger:

DROP TRIGGER trigger_name;

11. PACKAGE PLSQL

Oracle pl sql package:

A package is a schema object that groups logically related PL/SQL types, variables and subprograms.

PARTS OF A PACKAGE:

1. Package specification

2. Package body or definition

Package specification:

The package specification is the package interface that declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.

Note: All objects in the package specification are known as public objects.

Syntax of package specification:

```
CREATE PACKAGE package_name AS
  PROCEDURE procedure_name;
```

```
END cust_sal;
/
```

Example:

```
CREATE PACKAGE emp_sal AS
  PROCEDURE find_sal(e_id employees.id%type);
END emp_sal;
/
```

Package body or definition:

The package body or definition defines the queries for the cursors and the code for the subprograms.

Note: All objects in the package body or definition are known as private objects.

Syntax of body or definition:

```
CREATE OR REPLACE PACKAGE BODY package_name AS
  PROCEDURE procedure_name IS
    //procedure body
  END procedure_name;
END package_name;
/
```

Example:

```
CREATE OR REPLACE PACKAGE BODY emp_sal AS
  PROCEDURE find_sal(e_id employees.id%TYPE) IS
  e_sal employees.salary%TYPE;
  BEGIN
    SELECT salary INTO e_sal
    FROM employees
    WHERE id = e_id;
    dbms_output.put_line('Salary: '|| e_sal);
  END find_sal;
END emp_sal;
/
```