Advance Database Management System

Lecture 09:
Introduction to PL/SQL

Learning Objectives

To know about:

- PL/SQL
- PL/SQL Block
- Advantage of PL/SQL
- Example Query

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

PL/SQL Block

- Each PL/SQL program consists of SQL and PL/SQL statements which form a PL/SQL block
- PL/SQL Block consists of three sections:
  - The Declaration section (optional)
  - The Execution section (mandatory)
  - The Exception Handling (or Error) 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 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 Blocks 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

PL/SQL Block

DECLARE

Variable declaration

BEGIN

Program Execution

EXCEPTION

Exception handling

END;

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

Hello World Program


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

THANK YOU

Advance Database Management System

Lecture 10:
PL/SQL Basic Syntax

Learning Objectives

To know about:

- PL/SQL Placeholders
  - PL/SQL Variables
  - PL/SQL Constants
- Scope of PL/SQL Variables
- PL/SQL Literals
- Example Query

PL/SQL Placeholders

- Placeholders are temporary storage area
- PL/SQL 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

PL/SQL Placeholders

- 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 etc.

PL/SQL Variables

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

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.

PL/SQL Variables

- For example, if you want to store the current salary of an employee, you can use a variable.

DECLARE
salary number (6);

* Here "salary" is a variable of datatype number and of length 6.

PL/SQL Variables

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.

PL/SQL Variables

- We can assign values to variables in the two ways given below.
  - We can directly assign values to variables. The General Syntax is:

    variable_name:= value;
  - 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 Query

- 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 forward slash '/' in the above program indicates to execute the above PL/SQL Block.

Scope of PL/SQL 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.

Example Query

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.

```
DECLARE
var_num1 number;
var_num2 number;
BEGIN
var_num1 := 100;
var_num2 := 200;
    DECLARE
    var_mult number;
    BEGIN
    var_mult := var_num1 * var_num2;
    END;
END;
/
```

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.
- 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.

PL/SQL Constants

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.

PL/SQL Constants

- 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 a error. If you execute the below Pl/SQL block you will get error.

```
DECLARE
salary_increase CONSTANT number(3);
BEGIN
salary_increase := 100;
dbms_output.put_line (salary_increase);
END;
```

PL/SQL Literal

- A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier.
- For example, TRUE, 786, NULL, ‘adbms' are all literals of type Boolean, number, or string.
- PL/SQL, literals are case-sensitive.
- PL/SQL supports the following kinds of literals –
  - Numeric Literals
  - Character Literals
  - String Literals
  - BOOLEAN Literals
  - Date and Time Literals

THANK YOU

Advance Database Management System
Lecture 11:
PL/SQL Operators

Learning Objectives

To know about:

- PL/SQL Operators
- Types of PL/SQL Operators
- Example Query
- Operator Precedence

PL/SQL Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators –

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Arithmetic Operators Example


```
BEGIN
   dbms_output.put_line( 10 + 5);
   dbms_output.put_line( 10 - 5);
   dbms_output.put_line( 10 * 5);
   dbms_output.put_line( 10 / 5);
   dbms_output.put_line( 10 ** 5);
END;
/
```

Arithmetic Operators Example (User Input)


```
DECLARE
a number:=:a;
b number:=:b;
c number;
BEGIN
c:=a+b;
dbms_output.put_line(c);
END
```

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, then –

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

Relational Operators Example


```
DECLARE
  a number (2) := 21;
  b number (2) := 10;
BEGIN
  IF (a = b) then
    dbms_output.put_line('Line 1 - a is equal to b');
  ELSE
    dbms_output.put_line('Line 1 - a is not equal to b');
  END IF;
  IF (a < b) then
    dbms_output.put_line('Line 2 - a is less than b');
  ELSE
    dbms_output.put_line('Line 2 - a is not less than b');
  END IF;

  IF ( a > b ) THEN
    dbms_output.put_line('Line 3 - a is greater than b');
  ELSE
    dbms_output.put_line('Line 3 - a is not greater than b');
  END IF;
  -- Lets change value of a and b
  a := 5;
  b := 20;
  IF ( a <= b ) THEN
    dbms_output.put_line('Line 4 - a is either equal or less than b');
  END IF;
  IF ( b >= a ) THEN
    dbms_output.put_line('Line 5 - b is either equal or greater than a');
  END IF;
  IF ( a <> b ) THEN
    dbms_output.put_line('Line 6 - a is not equal to b');
  ELSE
    dbms_output.put_line('Line 6 - a is equal to b');
  END IF;
END;
/
```

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either **TRUE, FALSE** or **NULL**.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

Like Comparison Operators Example


```
DECLARE
value varchar2(20):='Zara Ali';
pattern varchar2(20):='Z%A_i';
BEGIN
  IF value LIKE pattern THEN
    dbms_output.put_line ('True');
  ELSE
    dbms_output.put_line ('False');
  END IF;
END;

/
```

Between Comparison Operators Example


```
DECLARE
  x number(2) := 10;
BEGIN
  IF (x between 5 and 20) THEN
    dbms_output.put_line('True');
  ELSE
    dbms_output.put_line('False');
  END IF;

  IF (x BETWEEN 5 AND 10) THEN
    dbms_output.put_line('True');
  ELSE
    dbms_output.put_line('False');
  END IF;

  IF (x BETWEEN 11 AND 20) THEN
    dbms_output.put_line('True');
  ELSE
    dbms_output.put_line('False');
  END IF;
END;
/
```

In and Is Null Comparison Operators Example


```
DECLARE
  letter varchar2(1) := 'm';
BEGIN
  IF (letter in ('a', 'b', 'c')) THEN
    dbms_output.put_line('True');
  ELSE
    dbms_output.put_line('False');
  END IF;

  IF (letter in ('m', 'n', 'o')) THEN
     dbms_output.put_line('True');
  ELSE
    dbms_output.put_line('False');
  END IF;

  IF (letter is null) THEN
   dbms_output.put_line('True');
  ELSE
    dbms_output.put_line('False');
  END IF;
END;
/
```

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and **variable B** holds false, then –

Operator	Description	Examples
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

Logical Operators Example


```
DECLARE
  a boolean := true;
  b boolean := false;
BEGIN
  IF (a AND b) THEN
    dbms_output.put_line('Line 1 - Condition is true');
  END IF;
  IF (a OR b) THEN
    dbms_output.put_line('Line 2 - Condition is true');
  END IF;
  IF (NOT a) THEN
    dbms_output.put_line('Line 3 - a is not true');
  ELSE
    dbms_output.put_line('Line 3 - a is true');
  END IF;
  IF (NOT b) THEN
    dbms_output.put_line('Line 4 - b is not true');
  ELSE
    dbms_output.put_line('Line 4 - b is true');
  END IF;
END;
/
```

PL/SQL Operator Precedence

- Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.
- For example, **x = 7 + 3 * 2**; here, **x** is assigned **13**, not 20 because operator * has higher precedence than +, so it first gets multiplied with **3*2** and then adds into **7**.
- Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.
- The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

PL/SQL Operator Precedence

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

Operator Precedence Example


```
DECLARE
  a number(2) := 20;
  b number(2) := 10;
  c number(2) := 15;
  d number(2) := 5;
  e number(2) ;
BEGIN
  e := (a + b) * c / d;      -- ( 30 * 15 ) / 5
  dbms_output.put_line('Value of (a + b) * c / d is : '|| e );
  e := ((a + b) * c) / d;   -- (30 * 15 ) / 5
  dbms_output.put_line('Value of ((a + b) * c) / d is  : ' ||  e );
  e := (a + b) * (c / d);   -- (30) * (15/5)
  dbms_output.put_line('Value of (a + b) * (c / d) is  : '||  e );
  e := a + (b * c) / d;     --  20 + (150/5)
  dbms_output.put_line('Value of a + (b * c) / d is  : ' ||  e );
END;
/
```

THANK YOU

Advance Database Management System

Lecture 12:
PL/SQL Conditions

Learning Objectives

To know about:

- IF-THEN Statement
- IF-THEN-ELSE Statement
- IF-THEN-ELSIF Statement
- CASE Statement
- SEARCHED CASE Statement
- Nested IF-THEN-ELSE

PL/SQL Conditions

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

PL/SQL programming language provides following types of decision-making statements.

- IF-THEN Statement
- IF-THEN-ELSE Statement
- IF-THEN-ELSIF Statement
- CASE Statement
- SEARCHED CASE Statement
- Nested IF-THEN-ELSE

IF-THEN Statement

- It is the simplest form of the **IF** control statement, frequently used in decision-making and changing the control flow of the program execution
- The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**
- If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL**, then the **IF** statement does nothing.

Syntax:

IF condition THEN

S;

END IF;

I

If the Boolean expression condition evaluates to true, then the block of code inside the **if statement** will be executed. If the Boolean expression evaluates to false, then the first set of code after the end of the **if statement** (after the closing end if) will be executed.

IF-THEN Statement Example


```
DECLARE
  a number(2) := 10;
BEGIN
  a:= 10;
 -- check the boolean condition using if statement
  IF( a < 20 ) THEN
    -- if condition is true then print the following
    dbms_output.put_line('a is less than 20 ' );
  END IF;
  dbms_output.put_line('value of a is : ' || a);
END;
/
```

Output:

```
a is less than 20
value of a is : 10
```

IF-THEN-ELSE Statement

- A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE.

  Syntax:

  IF condition THEN
  S1;
  ELSE
  S2;
  END IF;

  Where, S1 and S2 are different sequence of statements. In the IF-THEN-ELSE statements, when the test condition is TRUE, the statement S1 is executed and S2 is skipped; when the test condition is FALSE, then S1 is bypassed and statement S2 is executed.

IF-THEN-ELSE Statement Example


```
DECLARE
  a number(3) := 100;
BEGIN
  -- check the boolean condition using if statement
  IF( a < 20 ) THEN
    -- if condition is true then print the following
    dbms_output.put_line('a is less than 20 ' );
  ELSE
    dbms_output.put_line('a is not less than 20 ' );
  END IF;
  dbms_output.put_line('value of a is : ' || a);
END;
/
```

Output:

```
a is not less than 20
value of a is : 100
```

IF-THEN-ELSIF Statement

- IF-THEN-ELSIF statement allows you to choose between several alternatives.
- An IF-THEN statement can be followed by an optional ELSIF...ELSE statement.
- The ELSIF clause lets you add additional conditions.

Syntax:

```
IF(boolean_expression 1)THEN
  S1; -- Executes when the boolean expression 1 is true
ELSIF( boolean_expression 2) THEN
  S2;  -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
  S3; -- Executes when the boolean expression 3 is true
ELSE
  S4; -- executes when the none of the above condition is true
END IF;
```

IF-THEN-ELSIF Statement

When using IF-THEN-ELSIF statements there are a few points to keep in mind.

- *It's ELSIF, not ELSEIF.*
- *An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.*
- *An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.*
- *Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.*

IF-THEN-ELSIF Statement Example


```
DECLARE
  a number(3) := 100;
BEGIN
  IF ( a = 10 ) THEN
    dbms_output.put_line('Value of a is 10' );
  ELSIF ( a = 20 ) THEN
    dbms_output.put_line('Value of a is 20' );
  ELSIF ( a = 30 ) THEN
    dbms_output.put_line('Value of a is 30' );
  ELSE
     dbms_output.put_line('None of the values is matching');
  END IF;
  dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

Output:

None of the values is matching Exact value of a is: 100

CASE STATEMENT

- Like the IF statement, the **CASE statement** selects one sequence of statements to execute.
- However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

CASE STATEMENT EXAMPLE


```
DECLARE
grade char(1) := 'A';
BEGIN
CASE grade
when 'A' then dbms_output.put_line('Excellent');
when 'B' then dbms_output.put_line('Very good');
when 'C' then dbms_output.put_line('Well done');
when 'D' then dbms_output.put_line('You passed');
when 'F' then dbms_output.put_line('Better try again');
else dbms_output.put_line('No such grade');
END CASE;
END;
/
```

Output:

Excellent

SEARCHED CASE STATEMENT

- The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.

SEARCHED CASE STATEMENT Example


```
DECLARE
grade char(1) := 'B';
BEGIN
case
when grade = 'A' then dbms_output.put_line('Excellent');
when grade = 'B' then dbms_output.put_line('Very good');
when grade = 'C' then dbms_output.put_line('Well done');
when grade = 'D' then dbms_output.put_line('You passed');
 when grade = 'F' then dbms_output.put_line('Better try
   again');
else dbms_output.put_line('No such grade');
end case;
 END;
/
```

Output:

Very good

NESTED IF-THEN- ELSE

- You can use one IF-THEN or IF-THEN-ELSIF statement inside another IF-THEN or IF-THEN-ELSIF statement(s).

NESTED IF-THEN-ELSE Example


```
DECLARE
  a number(3) := 100;
b number(3) := 200;
  BEGIN
  -- check the boolean condition
  IF( a = 100 ) THEN
   -- if condition is true then check the following
  IF( b = 200 ) THEN
  -- if condition is true then print the following
  dbms_output.put_line('Value of a is 100 and b is 200' ); END IF;
  END IF;
  dbms_output.put_line('Exact value of a is : ' || a );
  dbms_output.put_line('Exact value of b is : ' || b );
  END;
   /
```

Output:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

THANK YOU

Advance Database Management System

Lecture 13:
PL/SQL Loops

Learning Objectives

To know about:

- PL/SQL Loops
- Types of PL/SQL Loops
- Labeling PL/SQL Loops
- Loop Control Statement

PL/SQL Loops

- Situations may arise when it is needed to execute a block of code several number of times
- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on
- Programming languages provide various control structures that allow for more complicated execution paths
- A loop statement allows to execute a statement or group of statements multiple times

Types of PL/SQL Loops

PL/SQL provides the following types of loop to handle the looping requirements:

- PL/SQL Basic LOOP
- PL/SQL While LOOP
- PL/SQL FOR LOOP
- NESTED LOOPS

PL/SQL Basic LOOP

In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

```
DECLARE
  x number := 10;
BEGIN
  LOOP
    dbms_output.put_line(x);
    x := x + 10;
    IF x > 50 THEN
      exit;
    END IF;
  END LOOP;
  -- after exit, control resumes here
  dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

Output:

```
10
20
30
40
50
After Exit x is: 60
```

PL/SQL While Loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

PL/SQL For Loop

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

```
DECLARE
  a number(2);
BEGIN
  FOR a in 10 .. 20 LOOP
    dbms_output.put_line ('value of a: ' || a);
 END LOOP;
END;
/
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

PL/SQL Nested Loop

You can use one or more loop inside any another basic loop, while, or for loop.

```
DECLARE
   i number(3);
   j number(3);
BEGIN
j:=2;
while j=2 LOOP
i:=2;
  LOOP
    i:=i+1;
    dbms_output.put_line(i);
    exit when i=5 ;
  end LOOP;
j:=j+1;
end LOOP;
END;

/
```

Output:

3

4

5

Labeling a PL/SQL Loop

- PL/SQL loops can be labeled.
- The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement.
- The label name can also appear at the end of the LOOP statement.
- You may use the label in the EXIT statement to exit from the loop.

Labeling a PL/SQL Loop Example


```
DECLARE
  i number(1);
  j number(1);
BEGIN
  << outer_loop >>
  FOR i IN 1..3 LOOP
    << inner_loop >>
    FOR j IN 1..3 LOOP
      dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
    END loop inner_loop;
  END loop outer_loop;
END;
/
```

Output:

```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3
```

Loop Control Statement

Loop control statements change execution from its normal sequence.

PL/SQL supports the following loop control statements.

Loop Control Statement

- **EXIT statement-**The Exit statement completes the loop and control passes to the statement immediately after the END LOOP.

- **CONTINUE statement-**Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Special Note:
As of Oracle Database 11g Release 1, CONTINUE is a PL/SQL keyword.

- **GOTO statement-**Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program.

EXIT Statement Example


```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
      IF a > 15 THEN
         -- terminate the loop using the exit statement
         EXIT;
      END IF;
   END LOOP;
END;
/
```

Output:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15

CONTINUE Statement Example


```
DECLARE
  a number(2) := 10;
BEGIN
  -- while loop execution
  WHILE a < 20 LOOP
    dbms_output.put_line ('value of a: ' || a);
    a := a + 1;
    IF a = 15 THEN
      -- skip the loop using the CONTINUE statement
      a := a + 1;
      CONTINUE;
    END IF;
  END LOOP;
END;
/
```

Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

GOTO Statement Example


```
DECLARE
  a number(2) := 10;
BEGIN
  <<loopstart>>
  -- while loop execution
  WHILE a < 20 LOOP
  dbms_output.put_line ('value of a: ' || a);
    a := a + 1;
    IF a = 15 THEN
      a := a + 1;
      GOTO loopstart;
    END IF;
  END LOOP;
END;
/
```

Output:

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19

THANK YOU

Advance Database Management System

Lecture 14:
PL/SQL Subprogram Part 01:
Procedure

Learning Objectives

To know about:

- Subprogram
- Types of Subprogram
- Parts of Subprogram
- Procedure
- Parameter Models

Subprograms and its Types

- A **subprogram** is a program unit that performs a particular task
- Subprograms are combined to form larger programs
- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters
- PL/SQL provides two kinds of subprograms –
  - **Functions** – These subprograms return a single value; mainly used to compute and return a value.
  - **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.

  This lecture is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next lecture.

Parts of a PL/SQL Subprograms

- Each PL/SQL subprogram has a name, and may also have a parameter list
- Like anonymous PL/SQL blocks, the named blocks will also have the following three parts –
  - **Declarative Part**

  It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
  - **Executable Part**

  This is a mandatory part and contains statements that perform the designated action.
  - **Exception-handling**

  This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

- A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

  CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter_name [IN | OUT | IN OUT] type [, ...])] {IS | AS} BEGIN < procedure_body > END procedure_name;

  Where,

  - *procedure-name* specifies the name of the procedure.
  - [OR REPLACE] option allows the modification of an existing procedure.
  - The optional parameter list contains name, mode and types of the parameters.
  - **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
  - *procedure-body* contains the executable part.
  - The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Procedure Example


```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

Executing and Deleting a Standalone procedure

- A standalone procedure can be called by –
  - the name of the procedure from a PL/SQL block
- The procedure named **'greetings'** can be called from another PL/SQL block –

```
BEGIN
   greetings;
END;
/
```

- A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is –

```
DROP PROCEDURE  greetings;
```

Parameter Modes in PL/SQL Subprograms

IN

- An IN parameter lets you pass a value to the subprogram
- **It is a read-only parameter**
- Inside the subprogram, an IN parameter acts like a constant
- It cannot be assigned a value
- You can pass a constant, literal, initialized variable, or expression as an IN parameter
- You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call
- **It is the default mode of parameter passing**
- **Parameters are passed by reference**

Parameter Modes in PL/SQL Subprograms

OUT

- An OUT parameter returns a value to the calling program
- Inside the subprogram, an OUT parameter acts like a variable
- You can change its value and reference the value after assigning it
- **The actual parameter must be variable and it is passed by value**

Parameter Modes in PL/SQL Subprograms

IN OUT

- An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller
- It can be assigned a value and the value can be read
- The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression
- Formal parameter must be assigned a value
- **Actual parameter is passed by value**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
  a number;
  b number;
  c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
  IF x < y THEN
    z:= x;
  ELSE
    z:= y;
  END IF;
END;
BEGIN
  a:= 23;
  b:= 45;
  findMin(a, b, c);
  dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

Output:

Minimum of (23, 45) : 23

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

Output:

Square of (23): 529

Methods for Passing Parameters

- Actual parameters can be passed in three ways –
  - Positional notation->findMin(a, b, c, d);
  - Named notation->findMin(x => a, y => b, z => c, m => d);
  - Mixed notation->the positional notation should precede the named notation.

    Legal->findMin(a, b, c, m => d);

    Illegal->findMin(x => a, b, c, d);

THANK YOU

Advance Database Management System

Lecture 15:
PL/SQL Subprogram Part 02:
Functions

Learning Objectives

To know about:

- PL/SQL Functions
- Creating and Calling PL/SQL Functions
- Recursive Functions

PL/SQL Functions

- A function is same as a procedure except that it returns a value
- Therefore, all the discussions of the previous lecture are true for functions too

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE FUNCTION** statement is as follows –

CREATE [OR REPLACE] FUNCTION function_name [(parameter_name [IN | OUT | IN OUT] type [, ...])] RETURN return_datatype
{IS | AS}
BEGIN
< function_body >
END [function_name];

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Function Example


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

Calling a Function


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

Function Example


```
DECLARE
  a number;
  b number;
  c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    z:= y;
  END IF;
  RETURN z;
END;
BEGIN
  a:= 23;
  b:= 45;
  c := findMax(a, b);
  dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

Output:

Maximum of (23,45): 45

PL/SQL Recursive Functions


```
DECLARE
  num number;
  factorial number;

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

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

Output:

Factorial 6 is 720

THANK YOU

Advance Database Management System

Lecture 16:
PL/SQL Cursor

Learning Objectives

To know about:

- PL/SQL Cursor
- Types of PL/SQL Cursor

PL/SQL Cursor

- Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.
- A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

Types of Cursor

A cursor can be named such that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors –

- Implicit cursors
- Explicit cursors

Implicit Cursor

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

- In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**.

Implicit Cursor

S/no:	Attribute & Description
1.	**%FOUND** 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, it returns FALSE.
2.	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3.	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4.	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Implicit Cursor

Any SQL cursor attribute can be accessed as **sql%attribute_name** as shown below:

```
DECLARE
total_rows number(2);
BEGIN
UPDATE emp
SET sal = sal + 500;
IF sql%notfound THEN
dbms_output.put_line('no sal updated');
 ELSIF sql%found THEN
total_rows := sql%rowcount;
dbms_output.put_line( total_rows || ' sal updated ');
 END IF;
END; /

rollback;

select * from emp;
```

Explicit Cursor

- Explicit cursors are programmer-defined cursors for gaining more control over the **context area**
- An explicit cursor 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

Explicit Cursor

The syntax for creating an explicit cursor is

- CURSOR cursor_name IS select_statement;

- Working with an explicit cursor includes the following steps –
  - Declaring the cursor for initializing the memory
  - Opening the cursor for allocating the memory
  - Fetching the cursor for retrieving the data
  - Closing the cursor to release the allocated memory

Explicit Cursor

S/no:	Attribute & Description
1.	**%FOUND** This evaluates TRUE if last fetch succeeded.
2.	**%NOTFOUND** Evaluates TRUE if last fetch failed.
3.	**%ISOPEN** This evaluates TRUE when cursor is open else FALSE.
4.	**%ROWCOUNT** This returns number of record fetched from active set.

Explicit Cursor


```
declare
d_name dept.dname%type;
d_loc dept.loc%type;
cursor c_dept is
select dname,loc from dept;
begin
open c_dept;
fetch  c_dept into d_name,d_loc;
dbms_output.put_line(d_name||' '||d_loc);
close c_dept;
end
/
```

THANK YOU

Advance Database Management System

Lecture 17:
PL/SQL Record

Learning Objectives

To know about:

- PL/SQL Record
- Types of Record

PL/SQL Record

- A record is a data structure that can hold data items of different kinds
- Records consist of different fields, similar to a row of a database table

PL/SQL?

PL/SQL can handle the following types of records –

- Table-Based Records
- Cursor-Based Records
- User-Defined Records

Table-Base Record


```
declare
dept_rec dept%rowtype;
begin
select * into dept_rec from dept
where dname='ACCOUNTING';
dbms_output.put_line(dept_rec.dname||'
  '||dept_rec.loc||' '||dept_rec.deptno);
end
/
```

Cursor-Based Record


```
declare
cursor c_emp is
select * from emp where ename='ALLEN';
rec_emp emp%rowtype;
begin
open c_emp;
fetch c_emp into rec_emp;
dbms_output.put_line(rec_emp.empno||'
   '||rec_emp.ename||' '||rec_emp.job||' '||rec_emp.mgr||'
   '||rec_emp.hiredate||' '||rec_emp.sal||' '||rec_emp.comm||'
   '||rec_emp.deptno);
close c_emp;
end;
/
```

User-Defined Record


```
DECLARE
  type books is record
    (title varchar2(50),
     book_id number);
  book1 books;
  book2 books;
BEGIN
  -- Book 1 specification
  book1.title  := 'Database System Concepts';
  book1.book_id := 6495407;
  -- Book 2 specification
  book2.title := 'Introduction to PL/SQL';
  book2.book_id := 6495700;

 -- Print book 1 record
  dbms_output.put_line('Book 1 title : '|| book1.title);
  dbms_output.put_line('Book 1 book_id : ' || book1.book_id);

  -- Print book 2 record
  dbms_output.put_line('Book 2 title : '|| book2.title);
  dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

Records as Subprogram Parameters

You can pass a record as a subprogram parameter just as you pass any other variable.

```
DECLARE
  type books is record
  (title varchar2(50),
   book_id number);
  book1 books;
  book2 books;
  PROCEDURE printbook (book books) IS
BEGIN
  dbms_output.put_line ('Book  title :  ' || book.title);
  dbms_output.put_line( 'Book book_id : ' || book.book_id);
END;
BEGIN
  -- Book 1 specification
  book1.title  := 'Database System Concepts';
  book1.book_id := 6495407;
  -- Book 2 specification
  book2.title := 'Introduction to PL/SQL';
  book2.book_id := 6495700;
 -- Use procedure to print book info
  printbook(book1);
  printbook(book2);
END;
/
```

THANK YOU

Advance Database Management System
Lecture 18:
PL/SQL Trigger

Learning Objectives

To know about:

- PL/SQL Trigger

Trigger

To know about:

- 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

Trigger Syntax


```
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;
```

Example 1


```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
WHEN (NEW.EMPNO > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.sal  - :OLD.sal;
  dbms_output.put_line('Old salary: ' || :OLD.sal);
  dbms_output.put_line('New salary: ' || :NEW.sal);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
update emp set sal='3975' where empno='7566'
insert into emp values ('1','JUENA','TEACHER','7698','12-JAN-18','2300','0','10')
select * from emp;
drop trigger display_salary_changes
```

Example 2


```
CREATE OR REPLACE TRIGGER dept_added
after INSERT ON dept
FOR EACH ROW
WHEN (NEW.deptno > 0)

BEGIN

  dbms_output.put_line('New Department Added');

END;
/
select * from dept;
insert into dept values ('50','TEACHING','KURIL');
```

Difference between Row Level & Statement Level Trigger

- Statement level trigger :
  - It works if any statement is executed
  - Does not depends on how many rows or any rows affected
  - It executes only once.

Difference between Row Level & Statement Level Trigger

- Row level trigger :
  - Executes each time when an row is affected
  - If zero rows affected no row level trigger will execute

THANK YOU

Advance Database Management System

Lecture 19:
PL/SQL Package

Learning Objectives

To know about:

- PL/SQL Package

PL/SQL Package

- Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms
- A package will have two mandatory parts –
  - Package specification
  - Package body or definition
- You can have many global variables defined and multiple procedures or functions inside a package.

Package Specification


```
CREATE OR REPLACE PACKAGE emp_pack AS
  PROCEDURE display_ename(e__id emp.empno%type);
  PROCEDURE display_sal(e__id emp.empno%type);
END emp_pack;
```

Package Body


```
CREATE OR REPLACE PACKAGE BODY emp_pack AS

  PROCEDURE display_ename(e__id emp.empno%TYPE) IS
  e_nam emp.ename%TYPE;
  BEGIN
    SELECT ename INTO e_nam
    FROM emp
    WHERE empno = e__id;
    dbms_output.put_line('Employee Name: '|| e_nam);
  END display_ename;

PROCEDURE display_sal(e__id emp.empno%TYPE) IS
  e_sal emp.sal%TYPE;
  BEGIN
    SELECT sal INTO e_sal
    FROM emp
    WHERE empno = e__id;
    dbms_output.put_line('Employee Salary '|| e_sal);
  END display_sal;
END emp_pack;
/
```

Using the Package


```
begin
emp_pack.display_ename('7369');
emp_pack.display_sal('7369');
end
```

THANK YOU

Advance Database Management System
PL/SQL Tutorial

Learning Objectives

To know about:

- Procedure
- Function
- Cursor
- Record
- Trigger
- Package

Procedure

Procedure

- Named PL/SQL block which performs one or more specific task
- Similar to a procedure in other programming languages

Further Information:

- http://plsql-tutorial.com/plsql-procedures.htm
- https://www.tutorialspoint.com/plsql/plsql_procedures.htm

Procedure Example


```
CREATE OR REPLACE PROCEDURE adjust_hisal(
   in_hisal IN salgrade.hisal%TYPE
)
IS
BEGIN

  UPDATE salgrade
  SET hisal ='8888'
  WHERE hisal= in_hisal;
END;


begin
adjust_hisal('9999');
end

select * from salgrade;
rollback
```

Function

Function

- 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

Further Information:

- http://plsql-tutorial.com/plsql-functions.htm
- https://www.tutorialspoint.com/plsql/plsql_functions.htm

Function Example


```
CREATE OR REPLACE FUNCTION totalemp
RETURN number IS
  total number(12) := 0;
BEGIN
  SELECT count(*) into total
  FROM emp;
  RETURN total;
END;
/
DECLARE
  c number(12);
BEGIN
  c := totalemp();
  dbms_output.put_line('Total No of Employees: ' || c);
END;
```

Cursor

Cursor

- A cursor is a temporary work area created in the system memory when a SQL statement is executed
- This temporary work area is used to store the data retrieved from the database, and manipulate this data

Further Information:

- http://plsql-tutorial.com/plsql-cursors.htm
- https://www.tutorialspoint.com/plsql/plsql_cursors.htm

Cursor Example One Row Print


```
declare
d_name dept.dname%type;
d_loc dept.loc%type;
cursor c_dept is
select dname,loc from dept;
begin
open c_dept;
fetch  c_dept into d_name,d_loc;
dbms_output.put_line(d_name||' '||d_loc);
close c_dept;
end
/
```

Cursor Example Multiple Row Print


```
declare
d_name dept.dname%type;
d_loc dept.loc%type;
cursor c_dept is
select dname,loc from dept;
begin
open c_dept;
loop
fetch  c_dept into d_name,d_loc;
exit when c_dept%notfound;
dbms_output.put_line(d_name||' '||d_loc);
end loop;
close c_dept;
end
/
```

Cursor Example Multiple Row Print


```
declare
lo_sal salgrade.losal%type;
cursor c_salgrade is
select losal from salgrade;
begin
open c_salgrade;
loop
fetch c_salgrade into lo_sal;
exit when c_salgrade%notfound;
dbms_output.put_line(lo_sal);
end loop;
close c_salgrade;
end
```

Record

Record

- A record is a data structure that can hold data items of different kinds
- Records consist of different fields, similar to a row of a database table

Further Information:

- http://plsql-tutorial.com/plsql-records.htm
- https://www.tutorialspoint.com/plsql/plsql_records.htm

Record Example One row print


```
declare
dept_rec dept%rowtype;
begin
select * into dept_rec from dept
where dname='ACCOUNTING';
dbms_output.put_line(dept_rec.loc)
  ;
end
/
```

Record Example multiple row print


```
declare
dept_rec dept%rowtype;
begin
for dept_rec
in(select * from dept)
loop
dbms_output.put_line(dept_rec.loc||'
   '||dept_rec.dname);
end loop;
end
/
```

Cursor Based record multiple/single row print


```
declare
cursor c_emp is
select * from emp;
rec_emp emp%rowtype;
begin
open c_emp;
--loop
fetch c_emp into rec_emp;
--exit when c_emp%notfound;
dbms_output.put_line(rec_emp.ename);
--end loop;
close c_emp;
end
/
```

Trigger

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

Further Information:

- http://plsql-tutorial.com/plsql-triggers.htm
- https://www.tutorialspoint.com/plsql/plsql_triggers.htm

Trigger Example


```
CREATE OR REPLACE TRIGGER salgrade_added
after INSERT ON salgrade
FOR EACH ROW
BEGIN
  dbms_output.put_line('New Salgrade Added');
END;
/
select * from salgrade;
insert into salgrade values ('6','1234','9999');
rollback
```

Package

Package

- Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms
- A package will have two mandatory parts –
  - Package specification
  - Package body or definition

Further Information:

- https://www.tutorialspoint.com/plsql/plsql_packages.htm

Package Example


```
CREATE PACKAGE emp_pack AS
   PROCEDURE display_ename(e__id
   emp.empno%type);
END emp_pack;
/
```

Package Example


```
CREATE OR REPLACE PACKAGE BODY emp_pack AS

  PROCEDURE display_ename(e__id emp.empno%TYPE) IS
  e_nam emp.ename%TYPE;
  BEGIN
    SELECT ename INTO e_nam
    FROM emp
    WHERE empno = e__id;
    dbms_output.put_line('Employee Name: '|| e_nam);
  END display_ename;
END emp_pack;
/
```

Package Example


```
begin
emp_pack.display_ename('7369');
end

select * from emp;
```

THANK YOU