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.

The PL/SQL Engine:

- Oracle uses a PL/SQL engine to processes the PL/SQL statements.
- A PL/SQL code can be stored in the client system (client-side) or in the database (server-side).

A PL/SQL Block:

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

Declaration Section

- The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE.
- This section is optional and is used to declare any placeholders like variables, constants, records and cursors,

Declaration Section

- 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.

Exception Section

- 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

- This is how a sample PL/SQL Block looks.

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

PL/SQL Placeholders

- Placeholders are temporary storage areas.
- 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 , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

PL/SQL Variables

- These are placeholders that store the values that can change through the PL/SQL Block.
- The General Syntax to declare a variable is:
  - variable_name datatype [NOT NULL := value];
- *variable_name* is the name of the variable.
- *datatype* is a valid PL/SQL datatype.

PL/SQL Variables

- 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);
- * "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";

PL/SQL Variables

- The value of a variable can change in the execution or exception section of the PL/SQL Block.
- We can assign values to variables in two ways.
- We can directly assign values to variables.
  - The General Syntax is:
  - variable_name:= value;

PL/SQL Variables

- We can assign values to variables directly from the database columns by using a SELECT.. INTO statement.
- The General Syntax is:
  - SELECT column_name INTO variable_name FROM table_name [WHERE condition];

Example

- The below program will get the salary of an employee with id '1116' and display it on the screen.
- DECLARE
  - var_salary number(6);
  - var_emp_id number(6) = 1116;
- BEGIN SELECT salary INTO var_salary
- FROM employee

Example

- 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;

Scope of Variables

- PL/SQL allows the nesting of Blocks within Blocks
- i.e, the Execution section of an outer block can contain inner blocks.
- Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks.

Scope of Variables

- 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.

Scope of Variables

- *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.
- For Example:
- creating two variables in the outer block and assigning their product to the third variable created in the inner block.

Scope of Variables

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

Scope of Variables

- 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.

PL/SQL Constants

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

PL/SQL Constants

- 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

- The General Syntax to declare a constant is:
- constant_name CONSTANT datatype := VALUE;
- *constant_name* is the name of the constant i.e. similar to a variable name.

PL/SQL Constants

- The word *CONSTANT* is a reserved word and ensures that the value does not change.
- *VALUE* - It is a value which must be assigned to a constant when it is declared.
- You cannot assign a value later.
- For example, to declare salary_increase, you can write code as follows:

PL/SQL Constants

- 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.

PL/SQL Constants

- If you execute the below PI/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 Records

- Records are another type of datatypes
- which oracle allows to be defined as a placeholder.
- Records are composite datatypes,
- which means it is a combination of different scalar datatypes like char, varchar, number etc.

PL/SQL Records

- Each scalar data types in the record holds a value.
- A record can be visualized as a row of data.
- It can contain all the contents of a row.

Declaring a record

- To declare a record,
- you must first define a composite datatype;
- then declare a record for that type.
- The General Syntax to define a composite datatype is:

Declaring a record

- TYPE record_type_name IS RECORD (first_col_name column_datatype, second_col_name column_datatype, ...);
- *record_type_name* – it is the name of the composite type you want to define.

Declaring a record

- *first_col_name, second_col_name, etc.,*
- *- it is the* names of the fields/columns within the record.
- *column_datatype* defines the scalar datatype of the fields.

Declaring a record

- There are different ways you can declare the datatype of the fields.
- 1) You can declare the field in the same way you declare the fields when creating a table.
- 2) If a field is based on a column from database table, you can define the field_type as follows:
  - *col_name* table_name.column_name%type;

Declaring a record

- By declaring the field datatype in the above method,
- the datatype of the column is dynamically applied to the field.
- This method is useful when you are altering the column specification of the table, because you do not need to change the code again.
- **NOTE:** You can use also *%type* to declare variables and constants.

Declaring a record

- The General Syntax to declare a record of a user-defined datatype is:
  - record_name record_type_name;
- The following code shows how to declare a record called *employee_rec* based on a user-defined type

Declaring a record

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

Declaring a record

- If all the fields of a record are based on the columns of a table,
- we can declare the record as follows:
  - record_name table_name%ROWTYPE;
- For example, the above declaration of employee_rec can be as follows:
- DECLARE employee_rec employee%ROWTYPE;

Declaring a record

- The advantages of declaring the record as a ROWTYPE are:
- You do not need to explicitly declare variables for all the columns in a table.
- If you alter the column specification in the database table, you do not need to update the code.

Declaring a record

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

Declaring a record

- **NOTE:** When you are creating a record,
- you are just creating a datatype,
- similar to creating a variable.
- You need to assign values to the record to use them.
- The following table consolidates the different ways in which you can define and declare a pl/sql record

Declaring a record

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

Passing Values To and From a Record

- When you assign values to a record,
- you actually assign values to the fields within it.
- The General Syntax to assign a value to a column within a record direclty is:
- record_name.col_name := value;

Passing Values To and From a Record

- If you used %ROWTYPE to declare a record, you can assign values as shown:
  - record_name.column_name := value;
- We can assign values to records using SELECT Statements as shown:

Passing Values To and From a Record

  - SELECT col1, col2
  - INTO record_name.col_name1, record_name.col_name2 FROM table_name [WHERE clause];
- If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each columns separately.

Passing Values To and From a Record

- In this case, you must SELECT all the columns from the table into the record as shown:
- SELECT *
- INTO record_name
- FROM table_name
- [WHERE clause];

Passing Values To and From a Record

- The General Syntax to retrieve a value from a specific field into another variable is:
  - var_name := record_name.col_name;
- The following table consolidates the different ways you can assign values to and from a record:

Passing Values To and From a Record

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

PL/SQL function

- ***PL/SQL function*** is a named block that returns a value.
- PL/SQL functions are also known as subroutines or subprograms.
- To create a PL/SQL function, you use the following syntax

```
CREATE [OR REPLACE] FUNCTION {function_name}
[(
    {parameter_1} [IN] [OUT] {parameter_data_type_1},
    {parameter_2} [IN] [OUT] {parameter_data_type_2},...
    {parameter_N} [IN] [OUT] {parameter_data_type_N} )]

RETURN {return_datatype} IS
    --the declaration statements
BEGIN
    -- the executable statements
RETURN {return_data_type};
EXCEPTION
    -- the exception-handling statements
END;
```

PL/SQL function

- The {function_name} is the name of the function.
- Function name should start with a verb for example function convert_to_number.
- {parameter_name} is the name of parameter being passed to function along with parameter's data type {parameter_data_type}.
- There are three modes for parameters: IN,OUT and IN OUT.

PL/SQL function

- The IN mode is the default mode.
- You use the IN mode when you want the formal parameter is read-only.
- It means you cannot alter its value in the function.
- The IN parameter behaves like a constant inside the function.
- You can assign default value to the IN parameter or make it optional.

PL/SQL function

- The OUT parameters return values to the caller of a subprogram.
- An OUT parameter cannot be assigned a default value therefore you cannot make it optional.
- You need to assign values to the OUT parameter before exiting the function or its value will be NULL.
- From the caller subprogram, you must pass a variable to the OUT parameter.

PL/SQL function

- In the IN OUT mode, the actual parameter is passed to the function with initial values.
- And then inside the function, the new value is set for the IN OUT parameter and returned to the caller.
- The actual parameter must be a variable.

PL/SQL function

- The function must have at least one RETURN statement in the execution part.
- The RETURN clause in the function header specifies the data type of returned value.
- The block structure of a function is the same as an PL/SQL block except for the addition CREATE [OR REPLACE] FUNCTION, the parameters section, and a RETURN clause.

Examples of PL/SQL Function

- We are going to create a function that parses a string and returns a number if the string being passed is a number otherwise it returns NULL.

```
CREATE OR REPLACE FUNCTION try_parse
(
   iv_number IN VARCHAR2)
RETURN NUMBER IS
BEGIN
      RETURN TO_NUMBER(iv_number);
EXCEPTION
      WHEN OTHERS THEN
      RETURN NULL;
END;
```

PL/SQL function

- The input parameter is iv_number that is a *varchar2* type.
- We can pass any string to the function *try_parse()*.
- We use built-in function *to_number* to convert a string into a number.
- If any exception occurs, the function will return *NULL* in the exception section of the function block.

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
    n_x NUMBER;
    n_y NUMBER;
    n_z NUMBER;
BEGIN
    n_x := try_parse('574');
    n_y := try_parse('12.21');
    n_z := try_parse('abcd');
    DBMS_OUTPUT.PUT_LINE(n_x);
    DBMS_OUTPUT.PUT_LINE(n_y);
    DBMS_OUTPUT.PUT_LINE(n_z);
END;
```

PL/SQL procedure

- Like a PL/SQL function, a ***PL/SQL procedure*** is a named block that performs one or more actions.
- PL/SQL procedure allows you to wrap complex business logic and reuse it.
- The following illustrates the PL/SQL procedure's syntax:

```
1. PROCEDURE [schema.]name[( parameter[, parameter...] )]
2. [AUTHID DEFINER | CURRENT_USER]
3. IS
4. [--declarations statements]
5. BEGIN
6. --executable statements
7. [EXCEPTION
8. ---exception handlers]
9. END [name];
```

- We can divide the PL/SQL procedure into two sections: header and body.
- **PL/SQL Procedure's Header**
- The section before the keyword IS is called procedures' header or procedure's signature.
- The elements in the procedure's header are listed as follows:

- Schema:
  - The optional name of the schema that own this procedure.
  - The default is the current user.
  - If you specify a different user, the current user must have privileges to create a procedure in that schema.

- Name:
  - The name of the procedure.
  - The name of the procedure like a function should be always meaningful and starting by a verb.

- Parameters:
  - The optional list of parameters.
  - Refer to the PL/SQL function for more information on parameter with different modes IN, OUT and IN OUT.

- AUTHID:
  - The optional AUHTID determines whether the procedure will execute with the privileges of the owner (DEFINER) of the procedure or the current user (CURRENT_USER).

PL/SQL Procedure's Body

- Everything after the keyword IS is known as procedure's body.
- The procedure's body consists of declaration, execution and exception sections.
- The declaration and exception sections are optional.
- You must have at least one executable statement in the execution section.

- In PL/SQL procedure you still have RETURN statement.
- However unlike the RETURN statement in function that returns a value to calling program,
- RETURN statement in procedure is used only to halt the execution of procedure and return control to the caller.
- RETURN statement in procedure does not take any expression or constant.

Example of PL/SQL Procedures

- We're going to develop a procedure called *adjust_salary()*.
- We'll update the salary information of employees in the table *employees* by using SQL UPDATE statement.
- Here is the PL/SQL procedure *adjust_salary()* code sample:

```
1.  CREATE OR REPLACE PROCEDURE adjust_salary(
2.     in_employee_id IN EMPLOYEES.EMPLOYEE_ID%TYPE,
3.     in_percent IN NUMBER
4.  ) IS
5.  BEGIN
6.     -- update employee's salary
7.     UPDATE employees
8.     SET salary = salary + salary * in_percent / 100
9.     WHERE employee_id = in_employee_id;
10. END;
```

- There are two parameters of the procedure IN_EMPLOYEE_ID and IN_PERCENT.
- This procedure will update salary information by a given percentage (IN_PERCENT) for a given employee specified by IN_EMPLOYEE_ID.
- In the procedure's body, we use SQL UPDATE statement to update salary information.
- Let's take a look how to call this procedure.

Calling PL/SQL Procedures

- A procedure can call other procedures.
- A procedure without parameters can be called directly by using keyword EXEC or EXECUTE followed by procedure's name as below:
  - EXEC procedure_name();
  - EXEC procedure_name;

- Procedure with parameters can be called by using keyword EXEC or EXECUTE followed by procedure's name and parameter list in the order corresponding to the parameters list in procedure's signature.
  - EXEC procedure_name(param1,param2…paramN);

```
1. -- before adjustment
2. SELECT salary FROM employees WHERE employee_id = 200;
3. -- call procedure
4. exec adjust_salary(200,5);
5. -- after adjustment
6. SELECT salary FROM employees WHERE employee_id = 200;
```

Conditional Statements in PL/SQL

- PL/SQL supports programming language features like conditional statements, iterative statements.
- The programming constructs are similar to how you use in programming languages like Java and C++.

PL/SQL IF Statement

- The ***PL/SQL IF statement*** allows you to execute a sequence of statements conditionally.
- The IF statements evaluate a condition.
- The condition can be anything that evaluates to a logical true or false
- such as comparison expression or combination of multiple comparison expressions.

PL/SQL IF Statement

- You can compare two variables of the same type or different types but they are convertible to each other.
- You can compare two literals.
- In addition, a Boolean variable can be used as a condition.
- The PL/SQL IF statement has three forms:
- IF-THEN, IF-THEN-ELSE and IF-THEN-ELSIF

PL/SQL IF-THEN Statement

- The following is the syntax of the IF-THEN statement:
- IF condition THEN
  - sequence_of_statements;
- END IF;

PL/SQL IF-THEN Statement

- If the condition evaluates to true, the sequence of statements will execute.
- If the condition is false or NULL,
- the IF statement does nothing.
- Note that END IF is used to close the IF statement, not ENDIF.

PL/SQL IF-THEN Statement

```
1.    DECLARE
2.      n_min_salary NUMBER(6,0);
3.      n_max_salary NUMBER(6,0);
4.      n_mid_salary NUMBER(6,2);
5.      n_salary     EMPLOYEES.SALARY%TYPE;
6.      n_emp_id     EMPLOYEES.EMPLOYEE_ID%TYPE := 200;
```

PL/SQL IF-THEN Statement

```
1.    BEGIN
2.      -- get salary range of the employee
3.      -- based on job
4.      SELECT min_salary,
5.             max_salary
6.      INTO n_min_salary,
7.           n_max_salary
8.      FROM JOBS
9.      WHERE JOB_ID = (SELECT JOB_ID
10.                     FROM EMPLOYEES
11.                     WHERE EMPLOYEE_ID = n_emp_id);
```

PL/SQL IF-THEN Statement

```
1.    -- calculate mid-range
2.      n_mid_salary := (n_min_salary + n_max_salary) / 2;
3.      -- get salary of the given employee
4.      SELECT salary
5.      INTO n_salary
6.      FROM employees
7.      WHERE employee_id = n_emp_id;
8.
```

PL/SQL IF-THEN Statement

```
1.    -- update employee's salary if it is lower than
2.      -- the mid range
3.      IF n_salary < n_mid_salary THEN
4.        UPDATE employees
5.        SET salary = n_mid_salary
6.        WHERE employee_id = n_emp_id;
7.      END IF;
8.    END;
```

PL/SQL IF-THEN-ELSE Statement

- This is the second form of the IF statement.
- The ELSE keyword is added with the alternative sequence of statements.
- Below is the syntax of the IF-ELSE statement.

PL/SQL IF-THEN-ELSE Statement

- IF condition THEN
  - sequence_of_if_statements;
- ELSE
  - sequence_of_else_statements;
- END IF;
- If the condition is NULL or false, the sequence of else statements will execute.

PL/SQL IF-THEN-ELSE Statement

```
1.    -- update employee's salary if it is lower than
2.      -- the mid range, otherwise increase 5%
3.      IF n_salary < n_mid_salary THEN
4.        UPDATE employees
5.        SET salary = n_mid_salary
6.        WHERE employee_id = n_emp_id;
7.      ELSE
8.        UPDATE employees
9.        SET salary = salary + salary * 5 /100
10.       WHERE employee_id = n_emp_id;
11.     END IF;
```

PL/SQL IF-THEN-ELSIF Statement

- PL/SQL supports IF-THEN-ELSIF statement to allow you to execute a sequence of statements based on multiple conditions.
- The syntax of PL/SQL IF-THEN-ELSIF is as follows:

PL/SQL IF-THEN-ELSIF Statement

- IF condition1 THEN
  - sequence_of_statements1
- ELSIF condition2 THEN
  - sequence_of_statements2
- ELSE
  - sequence_of_statements3
- END IF;

PL/SQL IF-THEN-ELSIF Statement

- Note that an IF statement can have any number of ELSIF clauses.
- IF the first condition is false or NULL, the ELSIF clause checks second condition and so on.
- If all conditions are NULL or false, the sequence of statements in the ELSE clause will execute.

PL/SQL IF-THEN-ELSIF Statement

- Note that the final ELSE clause is optional so you can omit it.
- If any condition from top to bottom is true, the corresponding sequence of statements will execute.

PL/SQL IF-THEN-ELSIF Statement

```
1.     -- update employee's salary if it is lower than
2.       -- the mid range, otherwise increase 5%
3.       IF n_salary > n_mid_salary THEN
4.         DBMS_OUTPUT.PUT_LINE('Employee ' || TO_CHAR(n_emp_id) ||
5.                              ' has salary $' || TO_CHAR(n_salary) ||
6.                              ' higher than mid-range $' ||
   TO_CHAR(n_mid_salary));
```

PL/SQL IF-THEN-ELSIF Statement

```
1.     ELSIF n_salary < n_mid_salary THEN
2.         DBMS_OUTPUT.PUT_LINE('Employee ' || TO_CHAR(n_emp_id) ||
3.                              ' has salary $' || TO_CHAR(n_salary) ||
4.                              ' lower than mid-range $' ||
   TO_CHAR(n_mid_salary));
```

PL/SQL IF-THEN-ELSIF Statement

```
1.     ELSE
2.         DBMS_OUTPUT.PUT_LINE('Employee ' || TO_CHAR(n_emp_id) ||
3.                              ' has salary $' || TO_CHAR(n_salary) ||
4.                              ' equal to mid-range $' ||
   TO_CHAR(n_mid_salary));
5.       END IF;
6.     END;
```

PL/SQL FOR Loop

- PL/SQL FOR loop is an iterative statement that allows you to execute a sequence of statements a fixed number of times.
- Unlike the PL/SQL WHILE loop, the number of iterations of the PL/SQL FOR loop is known before the loop starts.
- The following illustrates the PL/SQL FOR loop statement syntax:

PL/SQL FOR Loop

- FOR loop_counter IN [REVERSE] lower_bound .. higher_bound
- LOOP
  - sequence_of_statements;
- END LOOP;

PL/SQL FOR Loop

- SET SERVEROUTPUT ON SIZE 1000000;
- DECLARE
  - n_times NUMBER := 10;
- BEGIN
- FOR n_i IN 1..n_times LOOP
  - DBMS_OUTPUT.PUT_LINE(n_i);
- END LOOP;
- END;
- /

PL/SQL FOR Loop

- SET SERVEROUTPUT ON SIZE 1000000;
- DECLARE
  - n_times NUMBER := 10;
- BEGIN
- FOR n_i IN REVERSE 1..n_times LOOP
  - DBMS_OUTPUT.PUT_LINE(n_i);
- END LOOP;
- END;

PL/SQL WHILE Loop

- If you don't know in advance how many times to execute a sequence of statements because the execution depends on a condition that is not fixed.
- In such cases, you should use PL/SQL WHILE loop statement.
- The following illustrates the PL/SQL WHILE LOOP syntax:

PL/SQL WHILE Loop

- WHILE condition
- LOOP
  - sequence_of_statements;
- END LOOP;

PL/SQL CASE Statement

- The PL/SQL CASE statement allows you to execute a sequence of statements based on a selector.
- A selector can be anything such as variable, function, or expression that the CASE statement evaluates to a Boolean value.
- You can use almost any PL/SQL data types as a selector except BLOB, BFILE and composite types.
- syntax:

- Unlike the PL/SQL IF statement, PL/SQL CASE statement uses a selector instead of combination of multiple Boolean expressions.
- The following illustrates the PL/SQL CASE statement

```
1.    [<<label_name>>]
2.    CASE [TRUE | selector]
3.      WHEN expression1 THEN
4.           sequence_of_statements1;
5.      WHEN expression2 THEN
6.           sequence_of_statements2;
7.      ...
8.      WHEN expressionN THEN
9.           sequence_of_statementsN;
10.     [ELSE sequence_of_statementsN+1;]
11.   END CASE [label_name];
```

- Followed by the keyword CASE is a selector.
- The PL/SQL CASE statement evaluates the selector only once to decide which sequence of statements to execute.
- Followed by the selector is any number of the WHEN clause.
- If the selector value is equal to *expression* in the WHEN clause,
- the corresponding sequence of statement after the THEN keyword will be executed.

- If the selector's value is not one of the choices covered by WHEN clause,
- the sequence of statements in the ELSE clause is executed.
- The ELSE clause is optional so if you omit the ESLE clause, PL/SQL will add the following implicit ELSE clause:
  - ELSE RAISE CASE_NOT_FOUND;
- The keywords END CASE are used to terminate the CASE statement.

Example of Using PL/SQL CASE Statement

- The following code snippet demonstrates the PL/SQL CASE statement. We'll use table *employees* for demonstration

```
1.    SET SERVEROUTPUT ON SIZE 1000000;
2.    DECLARE
3.      n_pct    employees.commission_pct%TYPE;
4.      v_eval   VARCHAR2(10);
5.      n_emp_id employees.employee_id%TYPE := 145;
6.    BEGIN
7.      -- get commission percentage
8.      SELECT commission_pct
9.      INTO n_pct
10.     FROM employees
11.     WHERE employee_id = n_emp_id;
```

```
1.      -- evalutate commission percentage
2.        CASE n_pct
3.          WHEN 0 THEN
4.            v_eval := 'N/A';
5.          WHEN 0.1 THEN
6.            v_eval := 'Low';
7.          WHEN 0.4 THEN
8.            v_eval := 'High';
9.          ELSE
10.           v_eval := 'Fair';
11.       END CASE;
12.       -- print commission evaluation
13.       DBMS_OUTPUT.PUT_LINE('Employee ' || n_emp_id ||
14.                            ' commission ' || TO_CHAR(n_pct) ||
15.                            ' which is '   || v_eval);
16.   END;
```

PL/SQL LOOP Statement

- PL/SQL LOOP is an iterative control structure that allows you to execute a sequence of statements repeatedly.
- The simplest of LOOP consists of
  - the LOOP keyword,
  - the sequence of statements and
  - the END LOOP keywords

```
1.    LOOP
2.      sequence_of_statements;
3.    END LOOP;
```

- Note that there must be at least one executable statement between LOOP and END LOOP keywords.
- The sequence of statements is executed repeatedly until it reaches a loop exits.
- PL/SQL provides you EXIT and EXIT-WHEN statements to allow you to terminate a loop.

- The EXIT forces the loop halt execution unconditionally and passes control to the next statement after keyword END LOOP.
- The EXIT-WHEN statement allows the loop complete conditionally.
- When the EXIT-WHEN statement is reached, the condition in the WHEN clause is checked.

- If the condition is true, the loop is terminated and pass control to the next statement after keyword END LOOP.
- If condition is false, the loop will continue repeatedly until the condition is evaluated to true.
- Therefore if you don't want to have a infinite loop you must change variable's value inside loop to make condition true.

- The following illustrates PL/SQL LOOP with EXIT and EXIT-WHEN statements:

```
1.    LOOP
2.      ...
3.      EXIT;
4.    END LOOP;
```

```
1.    LOOP
2.      ...
3.      EXIT WHEN condition;
4.    END LOOP;
```

Example of PL/SQL LOOP with EXIT Statement

- In this example, we declare a counter. Inside the loop we add 1 to the counter and print it out.
- If the counter is 5, we use EXIT statement to terminate the loop.
- Below is the code example of PL/SQL LOOP statement with EXIT:

```
1.    SET SERVEROUTPUT ON SIZE 1000000;
2.    DECLARE n_counter NUMBER := 0;
3.    BEGIN
4.      LOOP
5.        n_counter := n_counter + 1;
6.        DBMS_OUTPUT.PUT_LINE(n_counter);
7.        IF n_counter = 5 THEN
8.          EXIT;
9.        END IF;
10.     END LOOP;
11.   END;
12.   /
```

Example of PL/SQL LOOP with EXIT-WHEN Statement

- We'll use the same counter example above. However instead of using the IF-THEN and EXIT statements,
- we use EXIT-WHEN to terminate the loop.
- The code example is as follows:

```
1.    SET SERVEROUTPUT ON SIZE 1000000;
2.    DECLARE n_counter NUMBER := 0;
3.    BEGIN
4.      LOOP
5.        n_counter := n_counter + 1;
6.        DBMS_OUTPUT.PUT_LINE(n_counter);
7.        EXIT WHEN n_counter = 5;
8.      END LOOP;
9.    END;
10.   /
```