PL/SQL

CM Suvarna Varma
Assoc Prof
VVIT

Unit 3 part 2/2

syllabus

- SQL Commands: DDL, DML, TCL, DCL
- Types of Constraints (Primary, Alternate, Not Null, Check, Foreign),
- Basic form of SQL query, joins, outer joins, set operations, group operations,
- various types of queries,

- PL/SQL (Cursor, Procedures, Functions, Packages, Triggers…)

PL/SQL

- PL/SQL stands for Procedural Language/SQL.
- PL/SQL extends SQL by adding control Structures found in other procedural language.
- PL/SQL combines the flexibility of SQL with Powerful feature of 3rd generation Language.
- The procedural construct and database access Are present in PL/SQL.
- PL/SQL can be used in both in database in Oracle Server and in Client side application development tools.
- It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.
- PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java.

notable facts about PL/SQL −

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

Features of PL/SQL

- PL/SQL has the following features −
- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It Contains new libraries of built in packages.
- Allows the calling of external functions and procedures.
- with PL/SQL , an multiple sql statements can be processed in a single command line statement.

Advantages of PL/SQL

- PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block.
- In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

PL/SQL Program units

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

PL/SQL BLOCK

- **PL/SQL is a block-structured** language, means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts −
- It consists of three sections:
- A declaration of variables, constants, cursors and exceptions which is optional.
- A section of executable statements.
- A section of exception handlers, which is optional
- The text of an Oracle Forms trigger is an anonymous PL/SQL block.
- Every PL/SQL statement ends with a semicolon (;).
- PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.

Sections & Description
Declarations This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
Executable Commands This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
Exception Handling This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

Basic structure of PL/sQL

```
DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling>
END;
```

Example

The 'Hello World' Example

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

OUTPUT:

```
Hello World

PL/SQL procedure successfully completed.
```

PRINT

- DBMS_OUTPUT.PUT_LINE ()
- It is a pre-defined package that prints the message inside the parenthesis

The PL/SQL Identifiers

- PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.
- By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator

:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators

PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

```
DECLARE
   -- variable declaration
   message  varchar2(20):= 'Hello, World!';
BEGIN
   /*
   *  PL/SQL executable statement(s)
   */
   dbms_output.put_line(message);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Hello World

PL/SQL procedure successfully completed.
```

PL/SQL Datatypes

- Scalar Types
- BINARY_INTEGER ,DEC,DECIMAL,DOUBLE ,,PRECISION,FLOAT,INT,INTEGER,NATURAL,
- NATURALN,NUMBER, NUMERIC, PLS_INTEGER,POSITIVE,POSITIVEN,REAL,SIGNTYPE,
- SMALLINT,CHAR,CHARACTER,LONG,LONG RAW,NCHAR,NVARCHAR2,RAW,ROWID,STRING,
- VARCHAR,VARCHAR2,
- Composite Types
- TABLE, VARRAY, RECORD
- LOB Types
- BFILE, BLOB, CLOB, NCLOB
- Reference Types
- REF CURSOR
- BOOLEAN, DATE

Subtype – in a datatype

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package **STANDARD**. For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows –

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

PL/SQL User-Defined Subtypes

- A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.
- PL/SQL predefines several subtypes in package **STANDARD**.
- For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows −

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype –

```
DECLARE
   SUBTYPE name IS char(20);
   SUBTYPE message IS varchar2(100);
   salutation name;
   greetings message;
BEGIN
   salutation := 'Reader ';
   greetings := 'Welcome to the World of PL/SQL';
   dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.
```

Variable Declaration in PL/SQL

- PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.
- The syntax for declaring a variable is −

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

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below −

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example −

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

- Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −
- The **DEFAULT** keyword
- The **assignment** operator

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

NULL

DEFAULT

```
DECLARE
   a integer := 10;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
/
```

ASSIGNMENT

When the above code is executed, it produces the following result –

```
Value of c: 30
Value of f: 23.333333333333333

PL/SQL procedure successfully completed.
```

Variable scope

- PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −
- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form –

```
DECLARE
   -- 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;
/
```

When the above code is executed, it produces the following result –

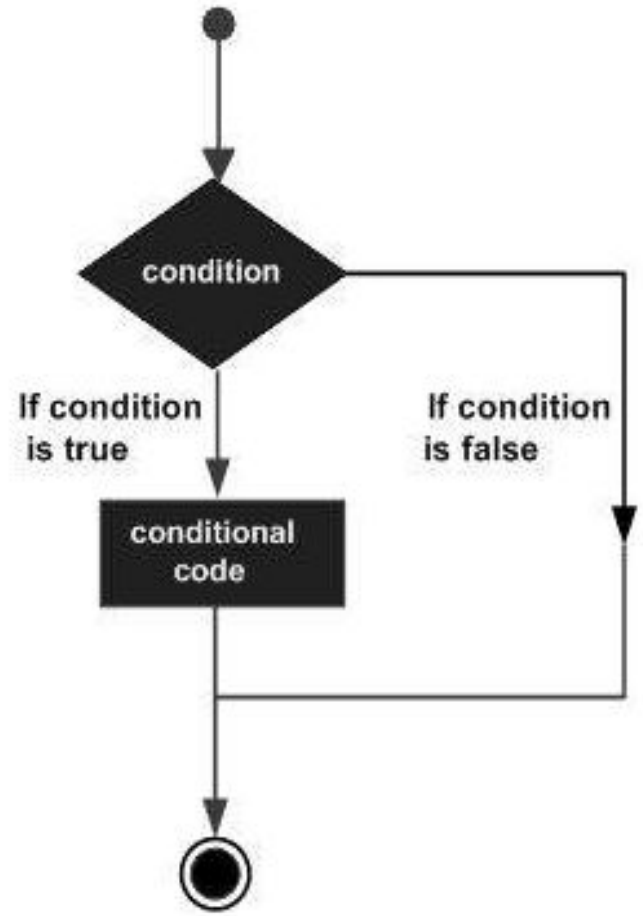
```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

Operator Precedence

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

Conditions

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages −



IF - THEN statement

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.

IF-THEN-ELSE statement

IF statement adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.

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.

Searched CASE statement

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

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).

S.No	Loop Type & Description
1	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.
2	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.
3	PL/SQL FOR LOOP Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	Nested loops in PL/SQL You can use one or more loop inside any another basic loop, while, or for 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.

```
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;
/
```

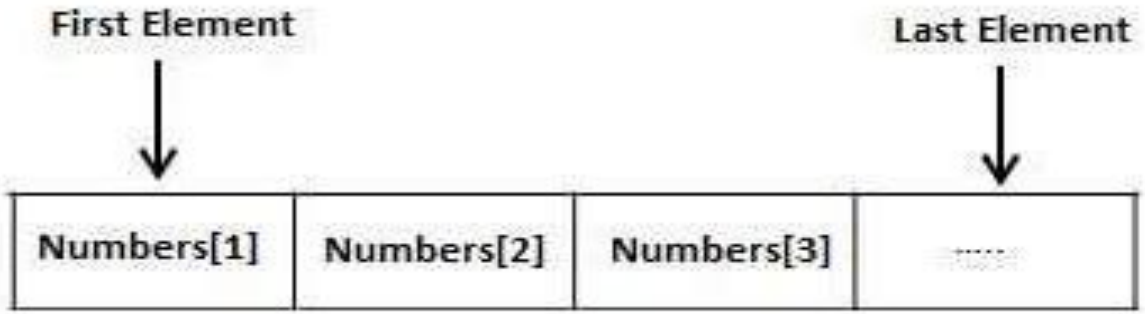
```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3
```

V-Array

The PL/SQL programming language provides a data structure called the **VARRAY**, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter **'PL/SQL Collections'**.

Each element in a **varray** has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the **CREATE TYPE** statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VARRAY type at the schema level is –

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,

- *varray_type_name* is a valid attribute name,
- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/

Type created.
```

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Procedures

- A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs.
- This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.
- A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

Procedures

- At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.
- A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

Procedures & functions

- 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.

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.

Example: Procedure

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

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

When the above code is executed using the SQL prompt, it will produce the following result –

```
Procedure created.
```

To execute a Procedure:

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

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

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement –

```
DROP PROCEDURE greetings;
```

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**.

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**.

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.**

Positional Notation

In positional notation, you can call the procedure as –

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x**, **b** is substituted for **y**, **c** is substituted for **z** and **d** is substituted for **m**.

Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following –

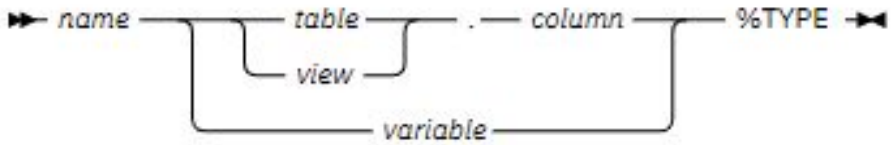
```
findMin(x => a, y => b, z => c, m => d);
```

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

However, this is not legal:

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

Syntax



Description

name

Specifies an identifier for the variable or formal parameter that is being declared.

table

Specifies an identifier for the table whose column is to be referenced.

view

Specifies an identifier for the view whose column is to be referenced.

column

Specifies an identifier for the table or view column that is to be referenced.

variable

Specifies an identifier for a previously declared variable that is to be referenced. The variable does not inherit any other column attributes, such as, for example, the nullability attribute.

ATTRIBUTES

- ATTRIBUTES Allow us to refer to data types and objects from the database.
- PL/SQL variables and Constants can have attributes.
- The main advantage of using Attributes is even if you Change the data definition, you don’t need to change in the application.

%TYPE

It is used when declaring variables that refer to the database columns.
Using %TYPE to declare variable has two advantages.

- First, you need not know the exact datatype of variable.
- Second, if the database definition of variable changes, the datatype of variable changes accordingly at run time.

- %ROWTYPE
- The %ROWTYPE attribute provides a record type that represents a row in a table (or view).
- The record can store an entire row of data selected from the table or fetched from a cursor or strongly typed cursor variable.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_deptno        NUMBER(2);
    v_avgsal        NUMBER(7,2);
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

FUNCTIONS

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

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter –

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

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

   RETURN total;
```

Calling function

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

   RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the fc
–

```
Function created.
```

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

When the above code is executed at the SQL prompt, it produces the following result –

```
Total no. of Customers: 6


PL/SQL procedure successfully completed.
```

Package

A package will have two mandatory parts –

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Package body created.
```

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name  customerS.No.ame%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);

   -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);
   --Lists all customers
   PROCEDURE listCustomer;

END c_package;
/
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax −

```
package_name.element_name;
```

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package −

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

Cursors

- 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**.
- You can name a cursor so 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 Cursors

- 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.

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.

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

Explicit cursors

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

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

Triggers

- Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −
- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).
- Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

- Triggers can be written for the following purposes −
- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

Ref table: Customer table

```
CREATE TABLE CUSTOMERS(   ID   INT NOT NULL,   NAME VARCHAR (20) NOT NULL,
 AGE INT NOT NULL,   ADDRESS CHAR (25),   SALARY   DECIMAL (18, 2),         PRIMARY KEY (ID) );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)  VALUES (1, 'Ramesh', 32,
'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, 'Khilan', 25,
'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3, 'kaushik', 23,
'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4, 'Chaitali', 25,
'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (5, 'Hardik', 27,
'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (6, 'Komal', 22, 'MP',
4500.00 );
```

```
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)


 DECLARE Declaration-statements
 BEGIN Executable-statements
 EXCEPTION Exception-handling-statements
 END;
```

- Where,
- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Trigger program

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

```
CREATE OR REPLACE TRIGGER display_salary_afterchanges
AFTER DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

The trigger “Display_salary_change” triggers for update, insert or delete operations of customer table as below:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
 VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

```
UPDATE customers SET salary = salary + 500 WHERE id = 2;
```

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

```
DELETE FROM customers  WHERE id = 2;
```

EXCEPTION HANDLING

- PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −
- System-defined exceptions (Pre-defined)
- User-defined exceptions

Exception Types

- 1. Predefined Exceptions
- An internal exception is raised implicitly whenever your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. Every Oracle error has a number, but exceptions must be handled by name.
- So, PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception NO_DATA_FOUND if a SELECT INTO statement returns no rows.

- 2. User – Defined exceptions
- User – defined exception must be defined and explicitly raised by the user

- EXCEPTION_INIT
- A named exception can be associated with a particular oracle error. This can be used to trap the error specifically.
- PRAGMA EXCEPTION_INIT(exception name, Oracle_error_number);
- The pragma EXCEPTION_INIT associates an exception name with an Oracle, error number. That allows you to refer to any internal exception by name and to write a specific handler

- RAISE_APPLICATION_ERROR
- The procedure raise_application_error lets you issue user-defined error messages from stored subprograms.
- That way, you can report errors to your application and avoid returning unhandled exceptions.
- To call raise_application_error, you use the syntax
- raise_application_error(error_number, message[, {TRUE | FALSE}]);
- where error_number is a negative integer in the range -20000 .. -20999 and message is a character string up to 2048 bytes long

Using SQLCODE and SQLERRM

- For internal exceptions, SQLCODE returns the number of the Oracle error. The number that SQLCODE returns is negative unless the Oracle error is no data found, in which case SQLCODE returns +100.
- SQLERRM returns the corresponding error message. The message begins with the Oracle error code.

- Unhandled Exceptions
- PL/SQL returns an unhandled exception error to the host environment, which determines the outcome.
- When Others
- It is used when all exception are to be trapped.

Exception syntax

```
DECLARE <declarations section>
BEGIN <executable command(s)>
EXCEPTION <exception handling goes here > WHEN exception1 THEN
exception1-handling-statements WHEN exception2 THEN
exception2-handling-statements WHEN exception3 THEN
 exception3-handling-statements ........ WHEN others THEN
 exception3-handling-statements
END;
```

Program

```
DECLARE c_id customers.id%type := 8;
 c_name customerS.Name%type;
 c_addr customers.address%type;
 BEGIN
 SELECT name, address INTO c_name, c_addr
 FROM customers WHERE id = c_id;
 DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION WHEN no_data_found THEN
 dbms_output.put_line('No such customer!’);
 WHEN others THEN dbms_output.put_line('Error!’);
 END;
/
```