Unit V : PL-SQL

Contents

- Beginning with PL / SQL,
- Identifiers and Keywords
- Operators
- Expressions
- Sequences
- Control Structures
- Cursors and Transaction
- Collections and composite data types
- Procedures and Functions
- Exceptions Handling
- Packages
- With Clause and Hierarchical Retrieval
- Triggers

- **Recommended Books**

Sr.No	Title	Author/s
1.	Database System and Concepts	A Silberschatz, H Korth, S Sudarshan
2.	Database Systems	Rob Coronel
3.	Programming with PL/SQL for Beginners	H. Dand, R. Patil and T. Sambare
4.	Introduction to Database System	C.J.Date

Prerequisites and Linking

Unit V	Pre-Requisites	Sem I & II	Sem. III	Sem . IV	Sem . V	Sem. VI
PL - SQL	-	-	-	-	Project	Project

Unit V **PL-SQL**

Beginning with PL / SQL:

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain 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 supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with 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 blocks can be divide into two broad categories :

1. Anonymous Block:
2. Named Block:

Structure of PL SQL block:

1. Anonymous Block:

PL/SQL program units organize the code into blocks. A block without a name is known as an anonymous block. The anonymous block is the simplest unit in PL/SQL. It is called anonymous block because it is not saved in the Oracle database.

An anonymous block is an only one-time use and useful in certain situations such as creating test units. The following illustrates anonymous block syntax:

```
DECLARE]
  Declaration   statements;
BEGIN
  Execution   statements;
 [EXCEPTION]
    Exception  handling  statements;
END;
/
```

The anonymous block has three basic sections that are the declaration, execution, and exception handling. Only the execution section is mandatory and the others are optional.

- The declaration section allows you to define data types, structures, and variables. You often declare variables in the declaration section by giving them names, data types, and initial values.
- The execution section is required in a block structure and it must have at least one statement. The execution section is the place where you put the

execution code or business logic code. You can use both procedural and SQL statements inside the execution section.

- The exception handling section is starting with the EXCEPTION keyword. The exception section is the place that you put the code to handle exceptions. You can either catch or handle exceptions in the exception section.

Example: To create Pl/SQL block which inserts 2 records in student table

```
Begin
Insert      into      student
Values('A101','Om',50);

Insert      into      student
Values('A102','Ram',55);
    End;
```

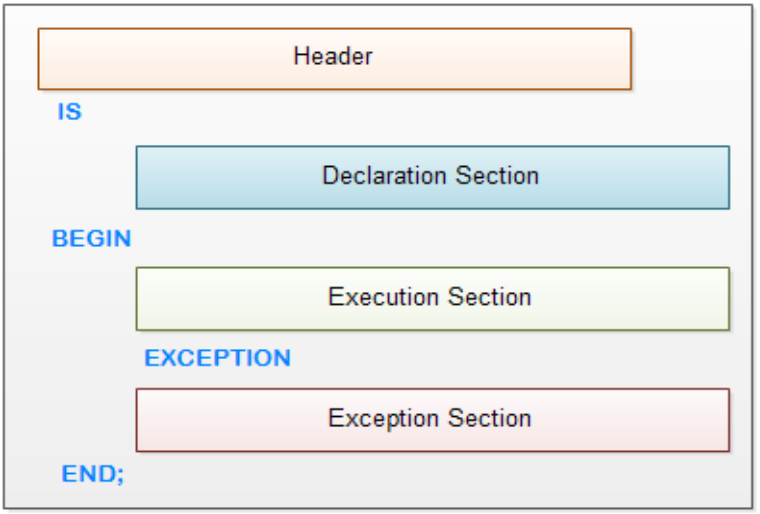
2. Named Block :

Named Block is a type of block which starts with the header section which specifies the name and the type of the block .There are two types of blocks namely :-

a) Procedures

b) Functions

Let's examine the PL/SQL block structure in greater detail.



Header:

Relevant for named blocks only, the header determines the way that the named block or program must be called. The header includes the name , parameter list and return clause(only for function)

Generate output from a PL/SQL block:

DBMS_OUTPUT is a built-in package that enables you to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers

Example:

```
BEGIN
dbms_output.put_line('Hello');
dbms_output.put('          World');
dbms_output.put_line('Welcome');

END;
```

Output: Hello
World
Welcome

Identifiers and Keywords

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.

The words listed in this appendix are reserved by **PL/SQL**. You should not use them to name program objects such as constants, variables, cursors, schema

objects such as columns, tables, or indexes. These words reserved by **PL/SQL** are classified as **keywords** or reserved words.

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

Here, we will understand the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed in a later chapter − PL/SQL - Strings.

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 −

Show Examples

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

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, the

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

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,

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
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',
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

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.

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.

Operator	Operation

**	Exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	Conjunction
OR	Inclusion

Expression

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function calls, and placeholders) and operators. The simplest expression is a **single** variable.

Expressions are constructed using operands and operators.
An **operand** is a variable, constant, literal, or function call that contributes a value to an expression. An example of a simple arithmetic expression follows:

-X / 2 + 3

Unary operators such as the negation operator (-) operate on one operand; binary operators such as the division operator (/) operate on two operands. PL/SQL has no ternary operators.

The simplest expressions consist of a single variable, which yields a value directly. PL/SQL evaluates an expression by combining the values of the operands in ways specified by the operators. An expression always returns a single value. PL/SQL determines the datatype of this value by examining the expression and the context in which it appears.

Operator Precedence

The operations within an expression are done in a particular order depending on their *precedence* (priority). Table 2-1 shows the default order of operations from first to last (top to bottom).

Table: Order of Operations

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication,

Operator	Operation
	division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	comparison

NOT	negation
AND	conjunction
OR	inclusion

Oracle / PLSQL: Sequences (Autonumber)

This Oracle tutorial explains how to **create and drop sequences** in Oracle with syntax and examples.

Description

In Oracle, you can create an autonumber field by using sequences. A sequence is an object in Oracle that is used to generate a number sequence. This can be useful when you need to create a unique number to act as a primary key.

Create Sequence

You may wish to create a sequence in Oracle to handle an autonumber field.

Syntax

The syntax to create a sequence in Oracle is:

```
CREATE SEQUENCE sequence_name
```

```
MINVALUE value
MAXVALUE value
START WITH value
```

```
INCREMENT BY value
CACHE value;
```

sequence_name
The name of the sequence that you wish to create.

Example

Let's look at an example of how to create a sequence in Oracle.

For example:

```
CREATE SEQUENCE supplier_seq
  MINVALUE 1
  MAXVALUE 9999999999999999999999999999
  START WITH 1
  INCREMENT BY 1
  CACHE 20;
```

This would create a sequence object called *supplier_seq*. The first sequence number that it would use is 1 and each subsequent number would increment by 1 (ie: 2,3,4,...}. It will cache up to 20 values for performance.

If you omit the MAXVALUE option, your sequence will automatically default to:

```
MAXVALUE 9999999999999999999999999999
```

So you can simplify your CREATE SEQUENCE command as follows:

```
CREATE SEQUENCE supplier_seq
```

```
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 20;
```

Now that you've created a sequence object to simulate an autonumber field, we'll cover how to retrieve a value from this sequence object. To retrieve the next value in the sequence order, you need to use *nextval*.

For example:

```
supplier_seq.NEXTVAL;
```

This would retrieve the next value from *supplier_seq*.
The *nextval* statement needs to be used in a SQL statement. For example:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
VALUES
(supplier_seq.NEXTVAL, 'Kraft Foods');
```

This insert statement would insert a new record into the *suppliers* table. The *supplier_id* field would be assigned the next number from the *supplier_seq* sequence. The *supplier_name* field would be set to Kraft Foods.

Drop Sequence

Once you have created your sequence in Oracle, you might find that you need to remove it from the database.

Syntax

The syntax to a drop a sequence in Oracle is:

```
DROP SEQUENCE sequence_name;
```

sequence_name
The name of the sequence that you wish to drop.

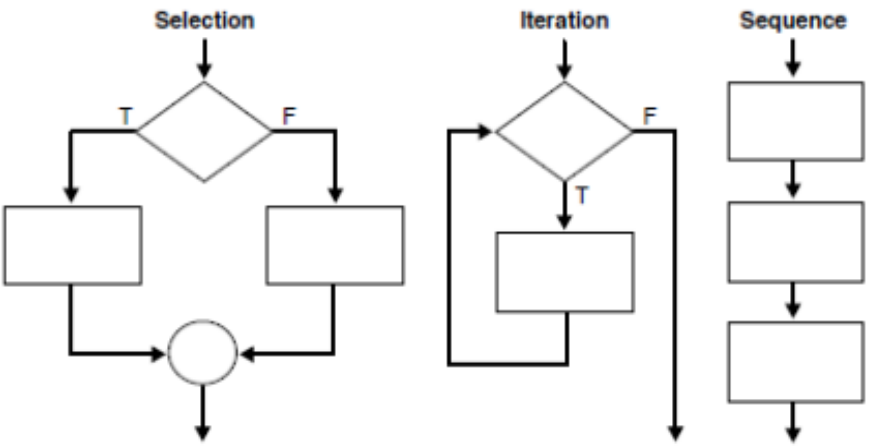
Example

Let's look at an example of how to drop a sequence in Oracle.

For example:

```
DROP SEQUENCE supplier_seq;
```

This example would drop the sequence called *supplier_seq*.



PL/SQL Control Structures

Procedural computer programs use the basic control structures.

- The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a BOOLEAN value (TRUE or FALSE).
- The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.
- The sequence structure simply executes a sequence of statements in the order in which they occur.

Testing Conditions: IF and CASE Statements

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from.

- **Using the IF-THEN Statement**

The simplest form of IF statement associates a condition with a sequence ofstatements enclosed by the keywords THEN and END IF (not ENDIF)

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Example: Using a Simple IF-THEN Statement

```
 DECLARE

sales NUMBER(8,2) := 10100;
```

```
quota NUMBER(8,2) := 10000;

bonus NUMBER(6,2);

emp_id NUMBER(6) := 120;

BEGIN

IF sales > (quota + 200) THEN

bonus := (sales - quota)/4;

UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;

END IF;

END;

/
```

- **Using CASE Statements**

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.

Example: Using the CASE-WHEN Statement

```
DECLARE

grade CHAR(1);
BEGIN

grade := 'B';

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('Good');
```

```
WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');

WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');

ELSE DBMS_OUTPUT.PUT_LINE('No such grade');

END CASE;
```

```
END;
```

Controlling Loop Iterations: LOOP and EXIT Statements

LOOP statements execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

- **Using the LOOP Statement**

The simplest form of LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP

sequence_of_statements
```

END LOOP;

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an EXIT statement to stop looping and prevent an infinite loop. You can place one or more EXIT statements anywhere inside a loop, but not outside a loop. There are two forms of EXIT statements: EXIT and

EXIT-WHEN.

- **Using the EXIT Statement**

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

- **Using the EXIT-WHEN Statement**

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

- **Labeling a PL/SQL Loop**

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

- **Using the WHILE-LOOP Statement**

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP


sequence_of_statements


END LOOP;
```

Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

Example: Using a Simple FOR..LOOP Statement

```
 DECLARE
p NUMBER := 0;
BEGIN
FOR k IN 1..500 LOOP -- calculate pi with 500 terms
p := p + ( ( (-1) ** (k + 1) ) / ((2 * k) - 1) );
END LOOP;
p := 4 * p;
DBMS_OUTPUT.PUT_LINE( 'pi is approximately : ' || p ); -- print result
END;
/
```

Sequential Control: GOTO and NULL Statements

The GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

- **Using the GOTO Statement**

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements.

Example : Using a Simple GOTO Statement

```
 DECLARE
p VARCHAR2(30);
  n PLS_INTEGER := 37; -- test any integer > 2 for prime
  BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
IF n MOD j = 0 THEN -- test for prime
p := ' is not a prime number'; -- not a prime number
GOTO print_now;
END IF;
END LOOP;
p := ' is a prime number';
<<print_now>>
```

```
DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

- **Using the NULL Statement**

The NULL statement does nothing, and passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).

Example: Using the NULL Statement to Show No Action

```
 DECLARE
v_job_id VARCHAR2(10);
v_emp_id NUMBER(6) := 110;
BEGIN
SELECT job_id INTO v_job_id FROM employees WHERE employee_id = v_emp_id;
IF v_job_id = 'SA_REP' THEN
UPDATE employees SET commission_pct = commission_pct * 1.2;
ELSE
NULL; -- do nothing if not a sales representative
END IF;
END;
```

PL/SQL - Cursors

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**.

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.

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**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

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.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

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 will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
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;
/
```

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated –

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   | +----
+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 | |
2 | Khilan   |  25 | Delhi          |  2000.00 |
|  3 | kaushik  |  23 | Kota       |  2500.00 |
|  4 | Chaitali |  25 |  Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal     |   9000.00  |
|  6 | Komal    |  22 | MP         |  5000.00 |  +-
---+----------+-----+-----------+----------+
```

Explicit Cursors

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.

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;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customerS.No.ame%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;
/
```

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

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

PL/SQL procedure successfully completed.
```

Source: https://www.youtube.com/watch?v=Y1dcZF4svuk

PL/SQL - Transactions

A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Transaction:

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place −

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place −

- A **COMMIT** or a **ROLLBACK** statement is issued.

- A **DDL** statement, such as **CREATE TABLE** statement, is issued; because in that case a COMMIT is automatically performed.
- A **DCL** statement, such as a **GRANT** statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from **SQL*PLUS** by issuing the **EXIT** command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a **ROLLBACK** is automatically performed.
- A **DML** statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is −

```
COMMIT;
```

For example,

```
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 );

COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is −

```
ROLLBACK [TO SAVEPOINT < savepoint_name>];
```

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint**, then simply use the following statement to rollback all the changes −

```
ROLLBACK;
```

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the **SAVEPOINT** command.

The general syntax for the SAVEPOINT command is −

```
SAVEPOINT < savepoint_name >;
```

For example

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );
SAVEPOINT sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000;
ROLLBACK TO sav1;
```

```
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;

COMMIT;
```

ROLLBACK TO sav1 − This statement rolls back all the changes up to the point, where you had marked savepoint sav1.

After that, the new changes that you make will start.

Automatic Transaction Control

To execute a **COMMIT** automatically whenever an **INSERT, UPDATE** or **DELETE** command is executed, you can set the **AUTOCOMMIT** environment variable as −

```
SET AUTOCOMMIT ON;
```

You can turn-off the auto commit mode using the following command −

```
SET AUTOCOMMIT OFF;
```

Collections and composite data types

PL/SQL - Collections

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types −

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections −

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level

Variablesize array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level

We have already discussed varray in the chapter **'PL/SQL arrays'**. In this chapter, we will discuss the PL/SQL tables.

Both types of PL/SQL tables, i.e., the index-by tables and the nested tables have the same structure and their rows are accessed using the subscript notation. However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-By Table

An **index-by** table (also called an **associative array**) is a set of **key-value**pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table_name**, the keys of which will be of the subscript_type and associated values will be of the *element_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;
```

Example

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
   TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
   salary_list salary;
   name   VARCHAR2(20);
BEGIN
   -- adding elements to the table
   salary_list('Rajnish') := 62000;
   salary_list('Minakshi') := 75000;
   salary_list('Martin') := 100000;
   salary_list('James') := 78000;

   -- printing the table
   name := salary_list.FIRST;
   WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
   END LOOP;
END;
/
```

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

```
Salary of James is 78000 Salary of Martin
is 100000 Salary of Minakshi is 75000
Salary of Rajnish is 62000

PL/SQL procedure successfully completed.
```

Example

Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following

example illustrates the concept. We will use the **CUSTOMERS** table stored in our database as −

```
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   |
```

```
DECLARE
   CURSOR c_customers is
      select name from customers;

   TYPE c_list IS TABLE of customers.Name%type INDEX BY
binary_integer;
   name_list c_list;
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list(counter) := n.name;
      dbms_output.put_line('Customer('||counter||'):'||name_lis t(counter));
   END LOOP;
END;
/
```

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

```
Customer(1): Ramesh Customer(2):
Khilan Customer(3): kaushik
Customer(4): Chaitali Customer(5):
Hardik Customer(6): Komal

PL/SQL procedure successfully completed
```

Nested Tables

nested table is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects −

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax −

```
TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;
```

This declaration is similar to the declaration of an **index-by** table, but there is no **INDEX BY** clause.

A nested table can be stored in a database column. It can further be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example

The following examples illustrate the use of nested table −

```
DECLARE
   TYPE names_table IS TABLE OF VARCHAR2(10);
   TYPE grades IS TABLE OF INTEGER;
   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i IN 1 .. total LOOP
```

```
      dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
   end loop;
END;
/
```

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

```
Total          5          Students
Student:Kavita,      Marks:98
Student:Pritam,      Marks:97
Student:Ayan,        Marks:78
Student:Rishav,      Marks:87
Student:Aziz, Marks:92
```

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

Example

Elements of a **nested table** can also be a **%ROWTYPE** of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as −

```
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  |
```

```
DECLARE
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list IS TABLE of customerS.No.ame%type;
   name_list c_list := c_list();
   counter integer :=0;
BEGIN
   FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter) := n.name;
      dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
   END LOOP;
END;
/
```

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

```
Customer(1):      Ramesh
Customer(2):        Khilan
Customer(3):       kaushik
Customer(4):       Chaitali
Customer(5):        Hardik
Customer(6): Komal

PL/SQL procedure successfully completed.
```

Collection Methods

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose −

S.No	Method Name & Purpose
1	**EXISTS(n)** Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.
2	**COUNT**
	Returns the number of elements that a collection currently contains.
3	**LIMIT** Checks the maximum size of a collection.
4	**FIRST** Returns the first (smallest) index numbers in a collection that uses the integer subscripts.
5	**LAST** Returns the last (largest) index numbers in a collection that uses the integer subscripts.

6	**PRIOR(n)** Returns the index number that precedes index n in a collection.
7	**NEXT(n)** Returns the index number that succeeds index n.
8	**EXTEND** Appends one null element to a collection.
9	**EXTEND(n)** Appends n null elements to a collection.
10	**EXTEND(n,i)** Appends **n** copies of the i^{th} element to a collection.
11	**TRIM** Removes one element from the end of a collection.
12	**TRIM(n)** Removes **n** elements from the end of a collection.
13	**DELETE** Removes all elements from a collection, setting COUNT to 0.

14	**DELETE(n)** Removes the **n**th element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If **n** is null, **DELETE(n)** does nothing.
15	**DELETE(m,n)** Removes all elements in the range **m..n** from an associative array or nested table. If **m** is larger than **n** or if **m** or **n** is null, **DELETE(m,n)**does nothing.

Collection Exceptions

The following table provides the collection exceptions and when they are raised −

Collection Exception	Raised in Situations
COLLECTION_IS_NULL	You try to operate on an atomically null collection.
NO_DATA_FOUND	A subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	A subscript exceeds the number of elements in a collection.

SUBSCRIPT_OUTSIDE_LIMIT	A subscript is outside the allowed range.
VALUE_ERROR	A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a **PLS_INTEGER** range, and the subscript is outside this range.

Composite Data Types

A **composite data type** stores values that have internal components. You can pass entire composite variables to subprograms as parameters, and you can access internal components of composite variables individually. Internal components can be either scalar or composite. You can use scalar components wherever you can use scalar variables. PL/SQL lets you define two kinds of composite data types, collection and record. You can use composite components wherever you can use composite variables of the same type.

Composite data types falls in two categories:

1) PL/SQL Records
2) PL/SQL Collections

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. 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.

The General Syntax to define a composite datatype is:

```
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.
- *first_col_name, second_col_name, etc.,- it is the* names the fields/columns within the record.

- *column_datatype* defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.

1) You can declare the field in the same way as you declare the fieds while creating the 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;

The General Syntax to declare a record of a user-defined datatype is: record_name record_type_name;

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;

Select using %RowType attribute

1) To get the names of the products and price having unit price than 1000

```
Create table product select *
from product

set serveroutput on; Declare
        cursor cur_prod is
        Select pname,unitprice from product; prec
        product%rowtype;
Begin
        for prec in cur_prod loop dbms_output.put_line(prec.pname||'
                '||prec.unitprice);
        End loop; End;
```

PL/SQL Collection:

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types −

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Index-By Table

- An **index-by** table (also called an **associative array**) is a set of **key-value**pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.
- An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table_name**, the keys of which will be of the subscript_type and associated values will be of the *element_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;
```

- Example
- Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
  TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20); •  salary_list salary;
  name                  VARCHAR2(20);
□□□□□□□□□□□BEGIN
  -- adding elements to the table
  salary_list('Rajnish')  := 62000;
  salary_list('Minakshi')  := 75000;
  salary_list('Martin')  := 100000;
  salary_list('James') := 78000;

  -- printing the table
  name := salary_list.FIRST;
  WHILE name IS NOT null LOOP
□□□□□□□□□□□□□□□□□□□dbms_output.put_line
```

```
•          ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name))); •  name      :=
salary_list.NEXT(name);
•        END                LOOP;
•□□□□□□□□□□□END;
•      /
```

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

```
•      Salary of James is 78000 •    Salary
of Martin is 100000
•      Salary of Minakshi is 75000 • Salary   of
Rajnish is 62000   •
•      PL/SQL procedure successfully completed.
```

- Example
- Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the **CUSTOMERS** table
stored in our database as –

```
•      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 |
•      +----+----------+-----+-----------+----------+
```

```
DECLARE
  CURSOR c_customers is
    select name from customers;
  TYPE c_list IS TABLE of customers.Name%type INDEX BY binary_integer;
  name_list c_list;
  counter integer :=0;
BEGIN
  FOR n IN c_customers LOOP
    counter := counter +1;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer('||counter||'):'||name_lis t(counter));
  END LOOP;
END;
```

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

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
PL/SQL procedure successfully completed
```

Procedures and Functions:

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**.

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.

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.
```

The procedure can also be called from another PL/SQL block −

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

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms –

S.No	Parameter Mode & Description
1	**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**.
2	**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**.
3	**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.**

Creating a Function:

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.

Example

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;
END;
/
```

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

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the
function **totalCustomers** from an anonymous block −

```
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.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
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;
/
```

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

```
Maximum of (23,45): 45
```

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

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

```
n! = n*(n-1)!
  = n*(n-1)*(n-2)! ...
  = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively −

```
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;
/
```

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

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

Exceptions Handling:

An exception is an error condition during a program execution. 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
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

```
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;
```

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters −

```
DECLARE
  c_id customers.id%type := 8;
  c_name  customerS.No.ame%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;
/
```

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

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the**EXCEPTION block**.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

```
DECLARE
   exception_name EXCEPTION;
BEGIN
```

```
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure**DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
   my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the
exception **invalid_id** is raised.

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name  customerS.No.ame%type;
   c_addr customers.address%type;
   -- user defined exception
   ex_invalid_id  EXCEPTION;
BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      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);
   END IF;

EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line('ID must be greater than zero!');
   WHEN no_data_found THEN
```

```
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

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

```
Enter value for cc_id: -6 (let's enter a value -6) old
2: c_id customers.id%type := &cc_id;
new 2: c_id customers.id%type := -6; ID
must be greater than zero!

PL/SQL procedure successfully completed.
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions −

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is
			and there is no ELSE clause.

COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
O_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.

NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.

ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

Source: https://www.youtube.com/watch?

Packages:

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

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 - Variableschapter.

```
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.
```

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;
/
```

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows −

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records −

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   | +----
+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  3000.00 | |
2 | Khilan   |  25 | Delhi          |  3000.00 |
|  3 | kaushik  |  23 | Kota       |  3000.00 |
|  4 | Chaitali |  25 | Mumbai     |  7500.00 |
|  5 | Hardik   |  27 | Bhopal     |  9500.00 |
|  6 | Komal    |  22 | MP         |  5500.00 |
```

```
+----+----------+-----+-----------+----------+
```

The Package Specification

```
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;
/
```

When the above code is executed at the SQL prompt, it creates the above package and displays the following result –

```
Package created.
```

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
   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)
  IS
  BEGIN
    INSERT INTO customers (id,name,age,address,salary)
      VALUES(c_id, c_name, c_age, c_addr, c_sal);
  END addCustomer;

  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
    DELETE FROM customers
    WHERE id = c_id;
  END delCustomer;

  PROCEDURE listCustomer IS
  CURSOR c_customers is
    SELECT  name FROM customers;
```

```
TYPE c_list is TABLE OF customerS.No.ame%type;
  name_list c_list := c_list();
  counter integer :=0;
  BEGIN
    FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
    END LOOP;
  END listCustomer;
```

```
END c_package;
/
```

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

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

```
Package body created.
```

Using The Package

The following program uses the methods declared and defined in the package*c_package*.

```
DECLARE
   Code customers.id%type:= 8;
```

```
   BEGIN
   c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
   c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
   c_package.listcustomer;
   c_package.delcustomer(code);
   c_package.listcustomer;
END;
/
```

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

```
Customer(1):      Ramesh
Customer(2):       Khilan
Customer(3):      kaushik
Customer(4):      Chaitali
Customer(5):       Hardik
Customer(6):       Komal
Customer(7):      Rajnish
Customer(8):     Subham
Customer(1):      Ramesh
Customer(2):       Khilan
Customer(3):      kaushik
Customer(4):      Chaitali
Customer(5):       Hardik
Customer(6):       Komal
Customer(7): Rajnish
PL/SQL procedure successfully completed
```

Source: https://www.youtube.com/watch?v=Lqz78kOCx9k

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

Creating Triggers

The syntax for creating a trigger is −

```
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.

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 −

```
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;
/
```

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

```
Trigger created.
```

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
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
differenc
e:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

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

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

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

Source: https://www.youtube.com/watch?v=0lYmB6rpaPY

MCQs:

1. Which statements are used to control a cursor variable?

a. OPEN-FOR

b. FETCH

c. CLOSE

d. All mentioned above

2) Which of the following is used to declare a record?

a. %ROWTYPE
b. %TYPE
c. Both A & B
d. None of the above

3)Which of the following has a return type in its specification and must return a value specified in that type?

a. Function
b. Procedure
c. Package
d. None of the above

4)Which collection exception is raised when a subscript designates an element that was deleted, or a nonexistent element of an associative array?

a. NO_DATA_FOUND
b. COLLECTION_IS_NULL
c. SUBSCRIPT_BEYOND_COUNT
d. SUBSCRIPT_OUTSIDE_LIMIT

5. Observe the following code and fill in the blanks −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE employees
  SET salary = salary + 500;
  IF _____________ THEN
    dbms_output.put_line('no employees selected');
  ELSIF _____________ THEN
    total_rows := _____________;
    dbms_output.put_line( total_rows || ' employees selected ');
  END IF;
END;
```

A - %notfound, %found, %rowcount.

B - sql%notfound, sql%found, sql%rowcount.

C - sql%found, sql%notfound, sql%rowcount.

D - %found, %notfound, %rowcount.

6. Which of the following is true about PL/SQL index-by tables?

A - It is a set of key-value pairs.

B - Each key is unique and is used to locate the corresponding value.

C - The key can be either an integer or a string.

D - All of the above.

7. Which keyword is used instead of the assignment operator to initialize variables?

a. NOT

b.

Defaul

t

c.

%TYPE

d. %ROWTYPE

8. Which of the following returns all distinct rows selected by either query?

a.

INTERSE

CT **b.**

MINUS

c. UNION

d. UNION ALL

9. For which Exception, if a SELECT statement attempts to retrieve data based on its conditions, this exception is raised when no rows satisfy the SELECT criteria?

a. TOO_MANY_ROWS
b. NO_DATA_FOUND
c. VALUE_ERROR
d. DUP_VAL_ON_INDEX

10.Which keyword and parameter used for declaring an explicit cursor?

a. constraint
b. cursor_variable_declaration
c. collection_declaration
d. cursor_declaration

Questions:

1. .Explain advantages of PL/SQL
2. Explain PL/SQL block structure.
3. Explain scalar data types
4. Explain the following:

i)%Type ii) %Rowtype iii) Sequences in PL/SQL iv) Bind Variables

5. Explain various data types conversion functions with examples.
6. Write a PL/SQL block to demonstrate cube of an input number
7. Write a PL/SQL program to demonstrate the use of basic arithmetic operators on the numbers input by user.
8. Explain various looping/iterative constructs in PL/SQL.
9. Explain various conditional statements in PL/SQL.
10. Explain PL/SQL Records with example.
11. Explain Explicit Cursors with example.
12. Explain Attributes of Explicit Cursors with example.
13. Explain the concept of Cursor For Loop with example.
14. Explain For Update clause and where current clause with example.
15. Explain Exception Handling in PL/SQL with example.
16. Differentiate Anonymous blocks and subprograms
17. Differentiate Procedures and Functions
18. Create a PL/SQL function to find out greatest two numbers. Call the function to display output.
19. Explain parts of Triggers with the help of example.
20. Give the syntax of AFTER INSERT TRIGGER and explain with the help of example.