6.1 INTRODUCTION PL/SQL

Introduction : PL/SQL is a combination of SQL along with the procedural features of programming language. It was developed by oracle corporation in early 1980's to enhance the capabilities of SQL.

PL/SQL engine : Oracle uses a PL/SQL engine to process the PL/SQL Statements. A PL/SQL code can be stored in client system or at server side.

Explain PL/SQL Block : Each PL/SQL program consist of SQL and PL/SQL statement which forms a PL/SQL block. PL/SQL block consists of 3 sections.

(i) Declaration section. (ii) Execution section.

(iii) Exception handling section.

Declaration section : It starts with a reserved keyword 'declare'. It is optional and is used to declare any place holders like variables, constant record, cursor etc which are used to manipulate data in execution statement and stores the data temporarily.

Execution Section : It starts with a reserved keyword 'begin' and ends with 'end'. This is a mandatory section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statements and SQL statements form the part of execution section.

Exception handling section : It starts with a reserved keyword 'exception'. This section is optional, any errors in the program can be handled in this section. If the exceptions are not handled the block terminates abruptly with errors.

Note : Every statements in the above 3 sections must be ended with semicolon (;)

Syntax :

```
> declare
variable section/
declaration section;
begin
execution section;
exception
exception handling section;
end;
```

List out PL/SQL Features :

(i) Procedural language capability (ii) Better performance

(iii) Exception handling.

6.2 DATA TYPES

Predefined PL/SQL datatypes are grouped into composite, LOB, reference, and scalar type categories.

- A composite type has internal components that can be manipulated individually, such as the elements of an array, record, or table.
- A LOB type holds values, called lob locators, that specify the location of large objects, such as text blocks or graphic images, that are stored separately from other database data. LOB types include BFILE, BLOB, CLOB, and NCLOB.
- A reference type holds values, called pointers that designate other program items. These types include REF CURSORS and REFs to object types.
- A scalar type has no internal components. It holds a single value, such as a number or character string. The scalar types fall into four families, which store number, character, Boolean, and date/time data.

PL/SQL Datatypes : It is of four types,

(i) Scalar (ii) Large object (LOB)

(iii) Composite (iv) Reference.

Scalar Datatypes : It is divided into four types,

(i) Numeric (ii) Character

(iii) Boolean (iv) Date time.

Numeric Datatype : PL/SQL Predefined numeric datatype are,

(i) int (ii) integer (iii) small int

(iv) float (v) Real (vi) double

(vii) number (viii) decimal.

PL/SQL character datatype :

(i) char (ii) varchar2 (iii) Raw

(iv) nchar (v) nvarchar2 (vi) long

(vii) long raw (viii) rowid.

PL/SQL Boolen datatype :

(i) True (ii) False

PL/SQL date time :

(i) Year (ii) Month (iii) day

(iv) hour (v) minute (vi) second

PL/SQL Large object datatype : Large object datatype refers to data items such as graphical images, video clips and sound waves.

- Predefined PL/SQL LOB datatypes are,

(i) BFILE B - Binary

(ii) BLOB C - Character

(iii) CLOB

(iv) NCLOB

Composite datatype : It is a combination of various scalar datatypes,

Ex : Record

PL/SQL Variable :

Syntax :

- Variablename datatypes; [declaring a variable]

  Ex : Pin varchar2 (30);

- Variablename = value; [Initialization of variable]

(or)

Variablename datatype NOT NULL = value;

Ex : name varchar2 (30) NOT NULL = 'raju';

(or)

Ex : name varchar2 (30);

name = 'raju';

PL/SQL constant :

Syntax : Constant_name constant datatype = value;

Ex :

PI constant Number (3, 2) = 3.14;

Write a PL/SQL program to add 2 numbers?

```
>set serveroutput on;
>declare
a int;          /* a number(3); or a integer:=10; */
b int;          /* b number(3); or b integer:=20; */
c int;          /* c number(3); or c integer; */
begin
a:=10;
b:=20;
c:=a+b;
dbms_output.put_line(c);
end;
```

Output : cvalue = 30

Scalar Types :

BINARY_DOUBLE
BINARY_FLOAT
BINARY_INTEGER
DEC
DECIMAL
DOUBLE PRECISION
FLOAT
INT
INTEGER
NATURAL
NATURALIN
NUMBER
NUMERIC
PLS_INTEGER
POSITIVE
POSITIVEN
REAL
SIGNTYPE
SMALLINT

CHAR
CHARACTER
LONG
LONGRAW
NCHAR
NVARCHAR2
RAW
ROWID
STRING
UROWID
VARCHAR
VARCHAR2

BOOLEAN

DATE

COMPOSIT TYPES
RECORD
TABLE
VARRAY

COMPOSIT TYPES
REF CURSOR
REF object_type

LOB Types
BFILE
BLOB
CLOB
NCLOB

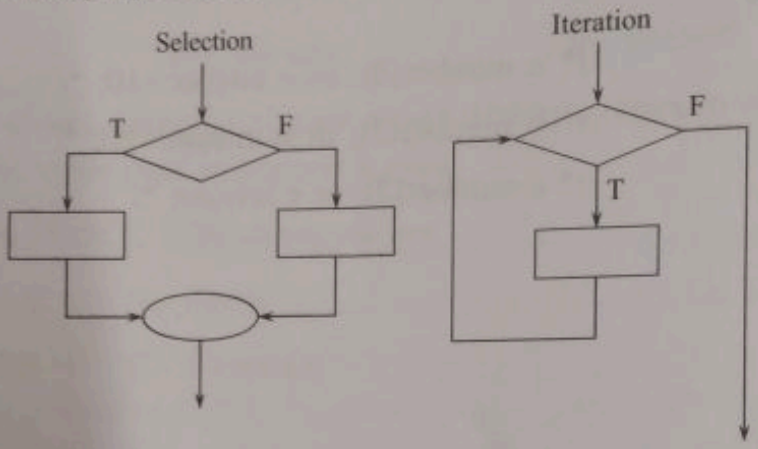
6.3 CONTROL STATEMENTS

Procedural computer programs use the basic control statements.

Control statements in PL/SQL are divided into two types.

(i) Conditional/Selection statements

(ii) Iteration/Looping statements.



- 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.

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 of statements 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.

- **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.
- **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 :

1. EXIT and 2. 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.
- **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.

- **Conditional Statement in PL/SQL :** PL/SQL support programming language features like conditional statments/iterative statment.

Conditional Statment : Oracle Database

(i) If_then (ii) If then else (iii) if then else if

(iv) exit (v) Continue (vi) Goto

(vii) Case (viii) Exit when

Syntax of if :

```
IF condition THEN
Statement;
END IF;
```

Ex :

```
IF (a < = 20)  THEN
C : = C + 1;
END IF;
```

Syntax of if-then else

```
IF condition THEN
Statements;
ELSE
Statements;
END IF;
```

Syntax of if-then else if

```
IF condition1 THEN
Statement 1;
ELSIF condition2 THEN
Statments 2;
ELSIF condition3 THEN
```

```
Statement 3;
ELSE
Statement 4;
END IF;
```

Syntax for CASE :

```
CASE selector
WHEN 'value1' THEN
Statement 1;
WHEN 'value2' THEN
Statement 2;
    :
    :
ELSE statment n;
END CASE;
```

Syntax for exit :

```
EXIT;
```

Syntax for goto;

```
GOTO label;
----------
----------
<<label>>
Statement;
```

Ex :

Write a PL/SQL program usinggoto statement

```
>set serveroutput on;
>declare
a number(2):=30;
BEGIN
<<loopstart>>
```

```
WHILE a < 50 LOOP
dbms_output.put_line('value of a is:' | |a);
a := a+1;
IF a=35 THEN
a := a+14;
GOTO loopstart;
END IF;
END LOOP;
END;
```

Output :

```
value of a : 30
value of a : 31
value of a : 32
value of a : 33
value of a : 34
value of a : 49
```

Iterative Statements :

(i) Loop (ii) While loop (iii) For loop.

Syntax of loop :

```
Loop
Sequence of statements;
END Loop;
```

Syntax of while loop :

```
WHILE condition LOOP
Sequence of statements;
END LOOP;
```

Syntax of FOR LOOP;

```
FOR COUNTER IN initial value..final value LOOP sequence of statements;
END LOOP;
```


Ex :

```
> declare
a number(2);
begin
FOR a in 10..20 Loop (or) FOR a in reverse 10..20 Loop
dbms_output · put_line(a);
end loop;
end;
```

Write a PL/SQL program to find largest among three using Oracle

```
> declare
a integer : = 10;
b integer : = 20;
c integer : = 30;
begin
If ((a>b) and (a>c)) then
dbms_output.put_line ('a is greater' | | a);
else
If ((b>a) and (b>c)) then
dbms_output.put_line ('b is greater' | | b);
else
If ((c>a) and (c>b)) then
dbms_output.put_line ('c is greater' | | c);
end if;
end if;
end if;
end;
```

Using MySQL

```
delimiter@
> Create  Procedure  bigamong3( )
begin
declare a int;
declare b int;
declare c int;
set a = 10;
set b = 20;
set c = 30;
If ((a>b)&&(a>c)) then
select 'a is greater';
select a;
else if ((b>a)&&(b>c)) then
select 'b is greater';
select b;
else if ((c>a)&&(c>b)) then
select 'c is greater';
select c;
end if;
end;
@
```

Output :

```
> call bigamong3 ( );
> @
```

C is greater

c
30

MYSQL : Syntax of conditional statements, according to MYSQL

If then else :

```
If condition1 THEN
Statement1;
[Else IF condition2 THEN
Statement;]
[ELSE]
Statement;
END IF;
```

While :

```
While condition
do
Statements;
END while;
```

CASE Selector :

```
CASE  case_expression
WHEN 'value1' THEN
Statement1;
WHEN 'value2' THEN
Statement2;
      ⋮
ELSE statementn;
END CASE;
```

Syntax for iterative statement :

```
iterate labelname;
```

Syntax for goto :

```
labelname;
Statements;
```

```
Iterate labelname;
Statements;
leave labelname;
```

Write a PL/SQL program to check vowel or consonant using Oracle.

```
> set serveroutput on
> DECLARE
choice char (1) : = 'i';
BEGIN
CASE choice
WHEN 'a' THEN
dbms_output.put_line ('vowel');
WHEN 'e' THEN
dbms_output.put_line ('vowel');
WHEN 'i' THEN
dbms_output.put_line ('vowel');
WHEN 'o' THEN
dbms_output.put_line ('vowel');
WHEN 'U' THEN
dbms_output.put_line('vowel');
ELSE dbms_output.put_line (choice || 'is consonant');
END CASE;
END;
```

Output : Vowel

Write a PL/SQL program on IF-Then along with records

```
select salary into Csal from employee where id = cid;
```


EXAMPLE - 6


```
IF (Csal < = 20000) THEN
UPDATE employee SET
Salary = Salary + 20000 where id = Cid;
dbms_output.put_line ('salary is updated');
END IF;
END;
```

Output : Salary is updated

Write a PL/SQL program to print 1 to 20 values using loop

```
> declare
i integer : = 1;
begin
loop
if (i < = 20) THEN
dbms_output.put_line (i);
i = i + 1;
END If;
END Loop;
END;
/
```

Write a PL/SQL program to find reverse of a number using while loop :

```
> declare
num integer : = 123;
i integer;
```

```
rev integer : = 0;
begin
while (num > 0)
LOOP
i : = num% 10;
rev : = rev * 10 + i;
num : = num/10;
end LOOP;
dbms_output.put_line ('reverse of no =' || rev);
end;
/
```

Output : reverse of no = 321

Write a PL/SQL program using label

```
> set serveroutput on
> 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);
dbms_output.put_lune ('j is =' || j);
end loop inner_loop;
end loop outer_loop;
end;
/
```


(nesting of loop) output

```
i is = 1
j is = 1
i is = 1
j is = 2
i is = 1
j is = 3
i is = 2
j is = 1
i is = 2
j is = 2
i is = 2
j is = 3
i is = 3
j is = 1
i is = 3
j is = 2
j is = 3
j is = 3
```

Write a PL/SQL program on exit when statement

```
> Set serveroutput on
> declare
a number(2) : = 10;
begin
While a < 20 loop
dbms_output.put_line (a);
a : = a  + 1;
exit when a > 15;
```

```
end loop;
end;
```

Output :

10
11
12
13
14
15

Write a PL/SQL program using continue statement

```
> Set serveroutput on
> declare
  a number(2) : = 10;
  begin
  while a < 20 loop
  dbms_output.put_line(a);
  a : = a + 1;
  If a = 15
  Then a : = a + 1;
  Continue;
  end if;
  end loop;
  end;
  /
```


Output :

10
11
12
13
14
16
17
18
19

Write a PL/SQL program using IF_then statement

```
Declare a number(2) : = 10;
begin
a : = 10;
if (a < 20) then
dbms_output.put_line ('a is less than 20');
end if ;
dbms_output.put_line ('value of a is :' || a);
end;
/
```

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

6.4 SEQUENTIAL CONTROL GOTO AND NULL STATEMENTS

- **GOTO and NULL Statements :** The GOTO statement is rarely used. 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 rarely. 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 - 1

Using a Simple GOTO Statement

Write a PL/SQL program using goto statement

```
> set serveroutput ON
> declare
a number (2) : = 10;
begin
<< loopstart >>
While a < 20 loop
dbms_output.put_line (a);
a : = a + 1;
If a = 15 then
a : a + 1;
GOTO loopstart;
end if;
end loop;
end;
```

Output :

```
10
11
12
13
14
```



```
16
17
18
19
```

- **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 - 2

Using the NULL Statement to Show No Action.

```
DECLARE
vjobID VARCHAR2(10);
vempID NUMBER(6) := 110;
BEGIN
SELECT job-id INTO vjobID FROM employees WHERE employee-id = vempID;
IF vjobID = 'SALES-REP' THEN
UPDATE employees SET commission= commission * 1.2;
ELSE
NULL; -- do nothing if not a sales representative
END IF;
END;
/
```

6.5 SUBPROGRAMS

Subprograms are named PL/SQL blocks that can be called with a set of parameters. PL/SQL has two types of subprograms, procedures and functions.

Subprograms have :

- A declarative part, with declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local and cease to exist when the subprogram ends.

- An executable part, with statements that assign values, control execution, and manipulate Oracle data.
- An optional exception-handling part, which deals with runtime error conditions.

Advantages of PL/SQL Subprograms :

- Subprograms let you extend the PL/SQL language. Procedures act like new statements. Functions act like new expressions and operators.
- Subprograms break a program down into manageable, well-defined modules.
- Subprograms promote reusability.
- Subprograms promote maintainability.
- Dummy subprograms (stubs) let you defer the definition of procedures and functions until after testing the main program.
- The application makes a single call to the database to run a block of statements which improves performance against running SQL multiple times. This will reduce the number of calls between the database and the application.
- PL/SQL is secure since the code resides inside the database thus hiding internal database details from the application. The application will only make a call to the PL/SQL sub program
- PL/SQL and SQL go hand in hand so there would be no need of any translation required between PL/SQL and SQL.

6.6 PROCEDURES

PL/SQL Procedures : A procedure is a subprogram that performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the BEGIN-END block that contains its code and handles any exceptions.

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

- These parameters should be defined along with the subprograms at the time of creation.
- These parameters are included in the calling statement of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and the calling statement should be same.

- The size of the datatype should not mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as,

1. IN parameter.
2. OUT parameter.
3. IN OUT parameter.

1. **IN Parameter :**
   - This parameter is used for giving input to the subprograms.
   - It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprograms.
   - In the calling statement, these parameters can be a variable or or literal value or an expression, for example, it could be the arithmetic expression like '5×8' or 'a/b' where 'a' and 'b' are variables.
   - By default, the parameters are of IN type.
2. **OUT Parameter :**
   - This parameter is used for getting output from the subprograms.
   - It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
   - In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.
3. **IN OUT Parameter :**
   - This parameter is used for both giving input and for getting output from the subprograms.
   - It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
   - In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter types should be mentioned at the time of creating the subprograms.

IN Parameter : It is used to send values to procedure and it is a read only parameter.

```
CREATE [OR REPLACE] PROCEDURE
    procedure_name(
    Parametername1 IN datatype,
    Parametername2 IN datatype,
    .....)
    [IS/AS]
    declaration section;
    BEGIN
    execution section;
    EXCEPTION
    exception handing section;
    END;
    (or)
CREATE [OR REPLACE] PROCEDURE
    procedurename(
    IN parametername1 datatype,
    IN parametername2 datatype,
    .....)
    [IS/AS]
    declaration section;
    BEGIN
    execution section;
    EXCEPTION
    exception handling section;
    END;
CREATE PROCEDURE addition(
    a IN integer,
    b IN integer)
```



```
> CREATE PROCEDURE addition(
    IN a integer,
    IN b integer)
```

Out/Parameter : These are used to get values from procedure, this is a write only parameter and are used to send the output from a procedure or function i.e., we cannot pass values to outparameter while executing.

SQL Syntax :

```
CREATE PROCEDURE procedurename(
    parametername1 OUT datatype
    parametername2 OUT datatype
        ⋮
    )
    AS
    declaration section;
    begin
    execution section;
    exception handling section;
    end;
```

Syntax : MYSQL

```
> CREATE PROCEDURE procedurename(
  OUT parametername1 datatype,
  OUT parametername2 datatype,
  .....)
  BEGIN
  excution section;
  EXCEPTION
  exception handling section;
  END;
```

```
> CREATE PROCEDURE addition(
  a  OUT  integet,
  b  OUT  integer)
> CREATE  PROCEDURE  addition(
  OUT  a  integer,
  OUT  b  integer)
> CREATE  PROCEDURE  procedurename(
  parametername1  IN  datatype,
  parametername2  OUT  datatype,
        ⋮
  )
  begin
Execution  section;
Exception  handling  section;
end;
```

Ex :

```
  > CREATE  PROCEDURE  add(
  IN  a  int,
  OUT  b  int)
  /
```

IN OUT Parameter :

```
  > CREATE  PROCEDURE  procedurename(
    parametrename1  IN OUT  datatype,
    parametername2  IN OUT  datatype)
        .....)
```


Ex :

```
  > CREATE  OR  REPLACE  PROCEDURE name(  )
    AS
    BEGIN
    dbms_output.put_line('Bahubali');
    END;
  /
```

Execute procedure :

```
  EXECUTE  name(  );
              (or)
  EXEC  name (  );
```

Deleting procedure :

```
  > DROP  PROCEDURE  procedurename;
```

Write a procedure to find minimum number

```
DECLARE
a  number;
b  number;
c  number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
IF x < y THEN
z : = x;
ELSE
z : = y;
END IF;
END;
BEGIN
```

```
a : = 23;
b : = 45;
findMin(a, b, c);
dbms_output.put_line('Minimum of (23, 45) :' ||c);
END;
/
```

Output : Minimum of (23, 45) : 23

Write a PL/SQL procedure to find square of number using IN & OUT mode

```
DECLARE
 a number (5);
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
 x : = x * x;
END;
BEGIN
 a : = 23;
 squareNum(a);
 dbms_output.put_line('Square of (23) :' ||a);
END;
/
```

Output : Square of (23) : 529

Write a PL/SQL procedure in MYSQL to add two numbers

```
> DELIMETER @
> Create PROCEDURE addition ( )
BEGIN
DECLARE a int;
DECLARE b int;
DECLARE c int;
```



```
SET a = 10;
SET b = 20;
SET c = a + b;
SELECT C;
END;
@
```

Output : > call addition ( )

@ C / 30

Write a PL/SQL program in MYSQL for parameter modes (IN)

```
> DELIMITER //
> CREATE PROCEDURE getpin (IN rollno varchar(30))
BEGIN
SELECT * FROM student WHERE
pin = rollno;
END;
//
> call getpin ('17001_CM_230');
```

Pin	Address
17001 - CM - 230	Mumbai

Write a PL/SQL program in MYSQL for parameter modes (out)

```
> DELIMITER //
> CREATE PROCEDURE outdemo (IN sbranch varchar(30), OUT total int)
BEGIN
SELECT count (*) INTO total FROM
Student where branch = sbranch;
END;
//
```

> call outdemo ('CME', @ Total);

> Select @total;

Output :

@ total/6

6.7 FUNCTIONS

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause.

Like a procedure, a function has two parts: the spec and the body. The function spec begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the datatype of the return value. Parameter declarations are optional. Functions that take no parameters are written without parentheses.

The function body begins with the keyword BEGIN and ends with the keyword END followed by an optional function name. The function body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The RETURN statement immediately ends the execution of a subprogram and returns control to the caller. A subprogram can contain several RETURN statements.

PL/SQL functions : A function is a named PL/SQL block.

- It is similar to a procedure, the major difference between a procedure and function must return a value but procedure may not return avalue

SQL Syntax :

```
> CREATE OR REPLACE FUNCTION FUNCTIONNAME(
[list of parameters]
RETURN return_datatype;
AS
declaration section;
BEGIN
execution section;
EXCEPTION
```



```
exception section;
END;
```

Let's see a simple example to create a function.

```
CREATE OR REPLACE FUNCTION adder(n1 IN number, n2 IN number)
RETURN number
AS
n3 number(8);
BEGIN
n3 :=n1+n2;
RETURN n3;
END;
/
```

Now write another program to call the function.

```
DECLARE
  n3 number(2);
BEGIN
  n3 := adder(11,22);
  dbms_output.put_line('Addition is: ' || n3);
END;
/
```

Output :

```
Addition is: 33
Statement processed.
0.05 seconds
```

Example programs using functions

Write a PL/SQL function program using IN parameter

> DELIMITER //

```
> CREATE FUNCTION calcincome (value INT)
    RETURNS INT
    BEGIN
      DECLARE income INT;
      SET income = 0;
      WHILE (income < = 20000)DO
      SET income = income + value;
      END WHILE;
      RETURN income;
    END; //
    DELIMITER;
    > select calcincome (8000);
```

Output : 16000

Write a PL/SQL function program in MYSQL without parameters

```
> CREATE FUNCTION fdemo( )
RETURN int
AS
BEGIN
RETURN (SELECT SUM(marks) FROM student);
END;
/
```

Write a PL/SQL function program SQL without parameters

```
> CREATE FUNCTION totalemp()
   RETURN number IS
   total number(2) : = 0;
   BEGIN
   SELECT COUNT (*) INTO total
```



```
   FROM employee;
   RETURN total;
   END;
   /
```

Output :

Function created

Calling a function :

```
> declare
 c number(2);
begin
 c : = totalemp ( ); [Calling function]
 dbms_output.put_line ('total employees =' || c);
end;
/
```

6.8 RECURSION

PL/SQL does support recursion via function calls. Recursion is the act of a function calling itself, and a recursive call requires PL/SQL to create local copies of its memory structures for each call.

Recursion is a powerful technique for simplifying the design of algorithms. Basically, recursion means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ...), is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined as simpler versions of itself. Consider the definition of n factorial (n!), the product of all integers from 1 to n :

n! = n * (n - 1)!

A recursive subprogram is one that calls itself. Each recursive call creates a new instance of any items declared in the subprogram, including parameters, variables, cursors, and exceptions.

Recursive function :

```
> CREATE OR REPLACE FUNCTION factorial (A number(2))
RETURN number
IS
BEGIN
IF A = 1 THEN
RETURN A;
ELSE
RETURN A*factorial (A – 1);
END IF;
END;
/
```

Calling a function :

```
> DECLARE
c Number(3);
BEGIN
c : = factorial (5);
dbms_output.put_line ('factorial =' || c);
END;
/
```

Output :

```
Factorial = 120
> select factorial (5)
from dual;
> /
```

factorial(5)
120

6.9 STORED PROCEDURES

Stored Procedures : A stored procedure is one which perform one or more specific task in PL/SQL block. This is similar to a procedure in other programming languages.

- A procedure consists of header and a body.
- Header consists of name of the procedure, parameters (or) variables passed to the procedure.
- Body consist of declaration, execution and exception section.

Syntax : (SQL)

```
> CREATE PROCEDURE PROCEDURENAME(
[List of parameters])
AS
declaration section;
BEGIN
execution section;
EXCEPTION
exception section;
END;
```

6.10 EXCEPTIONS

What is Exception : An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two type of exceptions :

- System-defined Exceptions.
- User-defined Exceptions.

PL/SQL Exception Handling :

Syntax for exception handling : Following is a general syntax for exception handling:

```
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 of exception handling :

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT * FROM CUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000


```
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;
/
```

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

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

Note : You get the result "No such customer" because the customer_id used in the above example is 8 and there is no cutomer having id value 8 in that table.

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. For a demo example : here, we are using the id 5.

```
DECLARE
  c_id customers.id%type := 5;
  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;
/
```

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

```
Name: alex
Address: paris
PL/SQL procedure successfully completed.
```

Raising Exceptions : In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

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;
```


PL/SQL User-defined Exceptions :

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

Syntax for user define exceptions

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

PL/SQL Pre-defined Exceptions

There are many pre-defined exception in PL/SQL which are executed when any database rule is violated by the programs.

For example : NO_DATA_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.

Following is a list of some important pre-defined exceptions :

Exception	Oracle Error	Description
ACCESS_INTO_NULL	06530	It is raised when a NULL object is automatically assigned a value.
CASE_NOT_FOUND	06592	It is raised when none of the choices in the ?WHEN? clauses of a CASE statement is selected, and there is no else clause.
COLLECTION_IS_NULL	06531	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	It is raised when duplicate values are attempted to be stored in a column with unique index.

INVALID_CURSOR	01001	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	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	It is raised when s program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	It is raised when a select into statement returns no rows.
NOT_LOGGED_ON	01012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	01476	It is raised when an attempt is made to divide a number by zero.

6.11 CURSORS

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to 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.

1. **PL/SQL Implicit Cursors :** The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Oracle provides some attributes known as implicit cursors attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

For example : When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

The following table specifies the status of the cursor with each of its attribute.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

PL/SQL Implicit Cursor Example :

Create customers table and have records :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected :

Create procedure :

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
     dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output :

```
6 customers updated
PL/SQL procedure successfully completed.
```

Now, if you check the records in customer table, you will find that the rows are updated.

```
select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	Chandan	25	Noida	31000
5	Alex	21	Paris	33000
6	Sunita	20	Delhi	35000

2. **PL/SQL Explicit Cursors :** The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors 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.

Following is the syntax to create an explicit cursor :

Syntax of explicit cursor

Following is the syntax to create an explicit cursor:

CURSOR cursor_name IS select_statement;

Steps : You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

(i) **Declare the cursor :** It defines the cursor with a name and the associated SELECT statement.

Syntax for explicit cursor declaration

CURSOR name IS

SELECT statement;

(ii) **Open the cursor :** It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax for cursor open :

OPEN cursor_name;

(iii) **Fetch the cursor :** It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows :

Syntax for cursor fetch :

FETCH cursor_name INTO variable_list;

(iv) **Close the cursor :** It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

Syntax for cursor close :

Close cursor_name;

PL/SQL Explicit Cursor Example : Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.

Create customers table and have records :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create procedure : Execute the following program to retrieve the customer name and address.

```
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;
/
```

Output :

```
1. Ramesh Allahabad        2. Suresh Kanpur
3. Mahesh Ghaziabad        4. Chandan Noida
5. Alex Paris              6. Sunita Delhi.
PL/SQL procedure successfully completed.
```

6.12 TRIGGERS

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- Database definition (DDL) statements (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers : These are the following advantages of Triggers,

- Trigger generates some derived column values automatically.
- Enforces referential integrity.
- Event logging and storing information on table access.
- Auditing.
- Synchronous replication of tables.
- Imposing security authorizations.
- Preventing invalid transactions.

Creating a trigger :

Syntax for creating trigger :

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Here :

- CREATE [OR REPLACE] TRIGGER trigger_name : It creates or replaces an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would 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 would 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, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] : This specifies a row level trigger, i.e., the trigger would 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.

PL/SQL Trigger Example : Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table :

Create table and have records :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create Trigger : Let's take a program to create 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;
/
```

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

Check the salary difference by procedure :

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output :

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
```



```
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
Old salary: 30000
New salary: 35000
Salary difference: 5000
6 customers updated
```

Note : As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

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

```
Old salary: 25000
New salary: 30000
Salary difference: 5000
Old salary: 27000
New salary: 32000
Salary difference: 5000
Old salary: 29000
New salary: 34000
Salary difference: 5000
Old salary: 31000
New salary: 36000
Salary difference: 5000
Old salary: 33000
New salary: 38000
Salary difference: 5000
Old salary: 35000
```

```
New salary: 40000
Salary difference: 5000
6 customers updated
```

Important Points : Following are the two very important point and should be noted carefully.

- OLD and NEW references are used for record level triggers these are not available for table 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.

6.13 PACKAGES

Packages : Packages are schema objects that groups logically related PL/SQL types, variables and subprograms. A package will have two mandatory parts,

(i) Package specification.

(ii) Package body or definition.

Package Specification : It is the interface to the package. It just declares the types, variables, constant, exceptions, cursors and sub-programs that can be referenced from outside the package. In otherwords, it contains all information about the content of the package. All objects placed in specification are public objects.

Syntax :

```
{Creating a package specification}
> CREATE [OR REPLACE] PACKAGE package_name
[IS/AS]
SUB program and public element declaration
.....
END package_name;
```

Ex :

```
> CREATE [OR REPLACE] PACKAGE emp–sal
AS
```



```
PROCEDURE final_sal (sid employee,id%type);
.....
END emp_sal;
```

Advantages of Packages :

(i) Modularity
(ii) Easier application design
(iii) Information hiding
(iv) Added functionality
(v) Better performance.

Package body : The package body has the codes for various methods declared in the package specification.

The characteristics of a package body are :

(i) It should contain definition for all the subprogram or cursors that have been declared in the specification.

(ii) It can also have more subprograms or other element that are not declared in specification. These are called private elements.

(iii) The first part of the package is global dedaration part. The last part of package is package initialization part that execute one time whenever a package is reffered first time in the session

Syntax :

```
> CREATE [OR REPLACE] PACKAGE BODY package_name
  [IS/AS]
< global_declaration part>
< private element definition>
< sub program and public element definition>
  .....
  EXCEPTION
  .....
< package Initialization>
  END package_name;
```

Ex :

```
> CREATE OR REPLACE PACKAGE BODY emp_sal
AS
PROCEDURE find_sal (sid employee.id%type)
IS
esal  employee.salary%type;
BEGIN
SELECT  salary  INTO  esal
FROM  employee  WHERE  id = sid;
dbms_output.put_line('salary =' || esal);
END  find_sal;
END  emp_sal;
```

Output : Package body created

Using the package element : The package element (variables, procedures on functions) are accussed with the following syntax

```
> PACKAGE  name.elementname;
```

Ex :

```
> DECLARE
Code  employee.id % type : = &sid;
BEGIN
emp_sal.find_sal(code);
END;
/
```

Output :

```
Enter value for sid : 1        Old 2 : code employee, id% type = & sid;
Salary = 20000                 new 2 : code employee id% type : = 1;
PL/SQL Procedure successfully completed.
```