UNIT-2

Que: Basics of PL / SQL

- Basic Syntax of PL/SQL which is a block-structured language;
- This means that the PL/SQL programs are divided and written in logical blocks of code.

Que: PL/SQL blocks Styructure

Declare section (optional)
Executable section (mandatory)
Exception-handling section (optional)

Here

- The executable section is the only mandatory section of the block.
- Both the declaration and exception-handling sections are optional.

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

- Place a semicolon (;) at the end of a SQL statement or PL/SQL control statement.
- Section keywords DECLARE, BEGIN, and EXCEPTION are not followed by semicolons.
- END and all other PL/SQL statements require a semicolon to terminate the statement.

- The different parts/sections of a PL/SQL block are discussed as below.
- The **declare section** contains the definitions of variables and other objects such as constants and cursors etc.
- This section is an **optional part** of a PL/SQL block.
- The **procedure section** contains conditional commands and SQL statements and is where the block is controlled.
- This section is the only **mandatory part** of a PL/SQL block.
- The **exception section** tells the PL/SQL block how to handle specified errors and user-defined exceptions.
- This section is an **optional part** of a PL/SQL block.

Example:

The 'Hello World' Example

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

Difference between PL/SQL AND SQL

SQL	PL/SQL
SQL is a Structured Query Language.	PL-SQL is a procedural Structured Query Language.
SQL is executed one statementat a time.	PL/SQL is executed as a block ofcode.
SQL is used to write Queries, DDL and DML statementsts.	PL/SQL is used to write programblocks, functions, procedures triggers, and packages.
SQL does not support Exception Handling.	PL/SQL support Exception Handling.
SQL does not support variable Declaration.	SQL does not support variable Declaration.

2.3 PL/SQL DATA-TYPES

- When writing PL/SQL blocks, you will be declaring variables, which must be valid data types.
- In PL/SQL Oracle provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use subtypes in your PL/SQL program to make the data types compatible with data types in other programs, such as a COBOL program, particularly if you are embedding PL/SQL code in another program. Subtypes are simply alternative names for Oracle data types and therefore must follow the rules of their associated data type.
- **Character string data types:** Character string data types in PL/SQL, as you might expect, are data types generally defined as having alpha-numeric values.
- Examples of character strings are names, codes, descriptions, and serial numbers that include characters.
- CHAR stores fixed-length character strings. The maximum length of CHAR is 32,767 bytes.

Syntax : char (max_length)

Subtype : character

- **varchar2** stores variable-length character strings. You would normally user varchar2 instead of char to store variable-length data, such as an individual's name. The maximum length of varchar2 is also 32,767 bytes.
- Char is faster than varchar2, sometime it is up to 50%.

Syntax : varchar2 (max_length)

Subtypes : varchar, string

- **long** also stores variable-length character strings, having a maximum length of 32,760 bytes. long is typically used to store lengthy text such as remarks, although varchar2 may be used as well.
- **Numeric data types:** number stores any type of number in an Oracle database.

Syntax : number (max_length)

- You may specify a number's data precision with the following syntax.

  number (precision, scale)

Subtypes : dec, decimal, double precision, integer, int, numeric, real, smallint, float

- **Binary Data Types:** Binary data types store data that is in a binary format, such as graphics or photographs. These data types include raw and longraw.
- **The date data type:** date is the valid Oracle data type in which to store dates. When you define a column as a date, you do not specify a length, as the length of a date field is implied. The format of an Oracle date is, for example, 01-oct-97.
- **boolean:** boolean stores the following values: true, false, and null. Like date, boolean requires no parameters when defining it as a column's or variable's data type.
- **rowid:** rowid is a pseudo-column that exists in every table in an Oracle database. The ROWID is stored in binary format and identifies each row in a table. Indexes use rowids as pointers to data.

Advantages of PL/SQL

1. It is a standard database language and PL/SQL is strongly integrated with SQL.
2. PL/SQL supports both static and also dynamic SQL.
3. Also, it then allows sending an entire block of statements to the database at one time.
4. Applications that are written in PL/SQL languages are portable.
5. This provides high security level.
6. It also provides access to the predefined SQL packages.
7. It also supports for Object-oriented programming.
8. It provides support for developing web applications and server pages.

2.5 COMMENTS

- Programming languages provide commands that allow you to place comments within your code, and PL/SQL is no exception.
- The comments after each line in the preceding sample block structure describe each command.
- The comments given may be for purposes to give idea about code, give information about author, give information about when coding was started etc.
- The accepted comments in PL/SQL are as follows.

```
-- This is a one-line comment.
/* This is a
multiple-line comment.*/
```

2.6 PL/SQL ATTRIBUTES

- There are two types of attributes.

  [1] %type [2] %rowtype

%type:

- When user wants to assign particular data type to a variable then %type may be useful.
- If we do not know data type of rno of student table but still we want to assign data type of rno to some variable then we have to use %type.

- Following is **syntax**.

```
Variable-name table-name. column-name %type;
```

- Following is **example**.

```
declare
        roll student.rno%type := &roll;
        m    student.mark%type;
begin
        select mark into m from student where rno=roll;
        dbms_output.put_line(roll || ' ' || m);
end;
```

Output:

```
Enter value for roll: 9
9 56
PL/SQL procedure successfully completed.
```

%rowtype :

- When user wants to assign every column data type to a single variable then %rowtype may be useful.
- The variable which contains %rowtype attribute can store complete record.
- Following is **syntax**.

```
Variable-name table-name %rowtype;
```

- Following is **example**.

```
declare
        roll student.rno%type := &roll;
        s     student%type;
begin
        select * into s from student where rno=roll;
        dbms_output.put_line(s.rno || ' ' || s.mark);
end;
```

Output:

```
Enter value for roll: 9
9 56
PL/SQL procedure successfully completed.
```

Que:Control Structures :

Conditional

- IF THEN
- IF THEN ELSE
- IF THEN ELSIF
- Nested IF THEN ELSE

Iterative

LOOP – EXIT WHEN – END LOOP
FOR – LOOP – END LOOP
WHILE – LOOP – END LOOP

Sequential

GOTO statement

Conditional

❖ **IF..... END IF**

- if then statement is the most simple decision-making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not.

Syntax:	**Example:**

IF (condition) THEN
statement;
END IF;
statement;
	declare
	a number:= 30;
	b number:= 20;
	
	begin
	if a > b then
	dbms_output.put_line('a is max');
	end if;
	dbms_output.put_line('b is max');
	
	end;

IF..... THEN..... ELSE

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

```
IF ( condition ) THEN
   statement;
ELSE
   statement;
END IF;
```

```
declare
a number:= 10;
b number:= 20;

begin

if a > b then
dbms_output.put_line('a is max');
else
dbms_output.put_line('b is max');
end if;

end;
```

❖ **IF ..... THEN.....ELSIF.....ELSIF.....ELSE**

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

```
IF ( condition-1 ) THEN
   statement-1;
ELSIF ( condition-2 ) THEN
   statement-2;
ELSIF ( condition-3 ) THEN
   statement-3;
ELSE
   statement;
END IF;
```

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

❖ **Nested IF…….. THEN……..ELSE**

IF (condition-1) THEN
IF (condition-2) THEN
statement-1;
ELSE
IF (condition-2) THEN
statements-3;
END IF;
END IF;
END;
	DECLARE
	a number := 10;
	b number := 20;
	c number := 30;
	BEGIN
	if(a >b) THEN
	
	if(a>c) THEN
	dbms_output.put_line('a is max');
	else
	dbms_output.put_line('c is max');
	end if;
	
	else
	
	if(b>c) then
	dbms_output.put_line('b is max');
	else
	dbms_output.put_line('c is max');
	end if;
	end if;
	end;

❖ **Iterative**

LOOP – EXIT WHEN – END LOOP

Syntax

```
LOOP
        Sequence of statements;
END LOOP;
```

Here, the sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop

Example

```
DECLARE
  a number := 1;
BEGIN
  LOOP
    dbms_output.put_line(a);
    a := a + 1;
   exit when a > 5 ;
      exit;
      END LOOP;
End;
/
```

1
2
3
4
5

❖ **FOR – LOOP – END LOOP**

A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

FOR variable in[reverse] start.....end

LOOP

Satements;

END LOOP;

```
DECLARE
  a number;
BEGIN
  FOR a in 10 .. 20
LOOP
    dbms_output.put_line( a);
 END LOOP;
END;
/
```

Reverse FOR LOOP

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

❖ **WHILE – LOOP – END LOOP**

A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

```
WHILE condition
Loop
  statements
END LOOP;
```

Example

```
DECLARE
  a number := 1;
BEGIN
  WHILE a < 5
    LOOP
    dbms_output.put_line(a);
    a := a + 1;
  END LOOP;
END;
```

❖ **Sequential**

GOTO statement

- goto statement provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.
- A label can be declare with the << label >>

Syntax

```
GOTO label_name;
        statements
    <<label_name>>
Statement;
```

Example:

```
DECLARE
  A number := 1;
BEGIN
  <<loop1>>

  WHILE a<= 10 LOOP
    dbms_output.put_line (a);
    a:= a+ 1;
    IF a = 5 THEN
      a := a + 1;
      GOTO loop1;
    END IF;
  END LOOP;
END;
/
```

Que: Exceptions: Predefined Exceptions, User defined exceptions

PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition

Syntax

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>

EXCEPTION

  WHEN exception1 THEN
       exception1 statements
  WHEN exception2  THEN
      exception2-statements
  WHEN exception3 THEN
      exception3-statements
  ........
    WHEN others THEN
    Exception -statements
END;
```

There are two types of exceptions:

1. System (pre-defined) Exceptions
2. User-defined Exceptions

1. System (pre-defined) Exceptions

- Such exceptions are the predefined names given by oracle for those exceptions that occur most commonly.

Syntax

```
EXCEPTION
        WHEN <exception_name> THEN
        -- take action
```

There are number of pre-defined named exceptions available by default.

Named Exception	Meaning
LOGIN_DENIED	Occurs when invalid username or invalid password is given while connecting to Oracle.
TOO_MANY_ROWS	Occurs when select statement returns more than one row.
VALUE_ERROR	Occurs when invalid datatype or size is given by the user.
NO_DATA_FOUND	Occurs when no records are found.
DUP_VAL_ON_INDEX	Occurs when a unique constraint is applied on some column and execution of Insert or Update leads to creation of duplicate records for that column.
PROGRAM_ERROR	Occurs when internal error arise in program.
ZERO_DIVIDE	Occurs when the division of any variable value is done by zero.

Example:

```
declare
 a  number := 10;
 b  number := 0;
 c  number;

begin
c := a /b;
    dbms_output.put_line( c);

  exception
      when zero_divide then
        dbms_output.put_line('error: division by zero');
  end;
```

2.User-defined Exceptions

```
declare
roll students.rno%type :=&roll;
m. student.mark%type;
s.exception;

begin
select mark into m from student where rno=roll;
if roll<50 then
 raise s;
end if;

exception
 when no_data_found then
            dbms_output.put_line('no student found');
when value_error then
            dbms_output.put_line('value error');
```

```
when s then
      dbms_output.put_line(m');
end;

output:
Enter value for roll:10
10 70
```

Que: Cursors: Static (Implicit & Explicit), Dynamic Cursors

- **Cursor** is a Temporary Memory or Temporary Work Station
- It is Allocated by Database Server at the Time of Performing DML(Data Manipulation Language) operations on the Table by the User.
- Cursors are used to store Database Tables.

Implicit Cursor?

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.
- 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.

SQL cursor has several useful attributes.

1. %FOUND is true if the most recent SQL operation affected at least one row.
2. %NOTFOUND is true if it didn't affect any rows.
3. %ROWCOUNT is returns the number of rows affected.
4. %ISOPEN checks if the cursor is open.

NAME	CITY	SALARY
AMIT	HIMATNAGAR	35000
JAYESH	SURAT	40000
NISH	AHMEDABAD	25000
NTIN	BARODA	44000

```
DECLARE
  totalrows number;

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

NAME	CITY	SALARY
AMIT	HIMATNAGAR	35500
JAYESH	SURAT	40500
NISH	AHMEDABAD	25500
NTIN	BARODA	44500

Explicit Cursor

Explicit Cursors: Explicit Cursors are created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.

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.

1. Declare

Syntax:

DECLARE cursor_name CURSOR IS SELECT * FROM table_name

Query:

DECLARE s1 CURSOR IS SELECT * FROM studDetails

2. Open Cursor Connection

Syntax:

OPEN cursor_name

Query:

OPEN s1

3. **Fetch Data from the Cursor** There is a total of 6 methods to access data from the cursor. They are as follows:

- **FIRST** is used to fetch only the first row from the cursor table.
- **LAST** is used to fetch only the last row from the cursor table.
- **NEXT** is used to fetch data in a forward direction from the cursor table.
- **PRIOR** is used to fetch data in a backward direction from the cursor table.
- **ABSOLUTE n** is used to fetch the exact n^{th} row from the cursor table.
- **RELATIVE n** is used to fetch the data in an incremental way as well as a decremental way.

Syntax:

FETCH fetch_data FROM cursor_name

FETCH FIRST FROM s1

4. Close cursor connection

Syntax:

CLOSE cursor_name

Query:

CLOSE s1

Example:

Customers

NAME	CITY	SALARY
AMIT	HIMATNAGAR	35000
JAYESH	SURAT	40000
NISH	AHMEDABAD	25000
NTIN	BARODA	44000

```
DECLARE

  c_name customer.name%type;
  c_city customer.city%type;

  CURSOR c_customer is  SELECT name, city FROM customer;

BEGIN
  OPEN c_customer;
  LOOP
    FETCH c_customer into c_name, c_city;
    EXIT WHEN c_customer%notfound;
    dbms_output.put_line(c_name || ' ' || c_city );
  END LOOP;

  CLOSE c_customer;
END;
```

Output:

```
AMIT   HIMATNAGAR
JAYESH   SURAT
NISH   AHMEDABAD
NTIN   BARODA

Statement processed.
```

Que: Procedures & Functions

❖ **Stored Procedures**

➢ A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [ (parameter [IN | OUT | INOUT.....]) ]
IS
  [declaration_section]
BEGIN
  executable_section
[EXCEPTION   exception_section]
END [procedure_name];
```

Example:

Table creation:

```
Create table s2(id number(10), name varchar2(100));
```

Procedure Code:

```
create or replace procedure "INSERTUSER"
(id IN NUMBER,   name IN VARCHAR2)
is
begin
insert into s2 values(id,name);
end;
```

```
BEGIN
  insertuser(101,'Amit');
  dbms_output.put_line('record inserted successfully');
END;
```

ID	Name
101	Amit

❖ **Function**

- The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.
- A stored function is a set of SQL statements that perform some operation and return a single value.

Syntax:

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

Example:1

Create a function.

```
create or replace function adder
    (n1 in number, n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
```

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
```

Difference between Procedures and Functions are given below:

Functions	Procedures
○ A function **must** return a value.	○ A procedure **may or may** not return a value.
○ A function can return **only one value**.	○ A procedure can return **multiple values**.
○ A function can be used with **SELECT** statement, like in-built SQL functions.	○ A procedure **cannot** be used with SELECT statement.
○ A function cannot directly execute using **EXEC** command.	○ A procedure can directly execute using **EXEC** command.

Que: Packages: Package specification, Package body, Advantages of package

- A package is one kind of database object.
- It is used to group together logically related objects like variables, constants, cursors, exceptions, procedures and functions.
- A successfully compiled package is stored in oracle database like procedures and functions.
- Unlike procedure and functions, package itself cannot be called.

Structure of a Package ¬ A package contains **two sections**:
1) Package Specification
2) Package Body

While creating packages, package specification and package body are created separately.

Package Specification:

- Various objects (like variables, constants etc..) to be held by package are Declared in this section.
- This declaration is global to the package, means accessible from anywhere in the package.

Syntax:

```
CREATE OR REPLACE PACKAGE packageName
IS /AS
        Package Specification
END   packageName;
```

Package specification consists of list of variables, constants, functions, procedures and cursors.

Example

```
CREATE OR REPLACE PACKAGE MathOperations AS
  PROCEDURE AddNumbers(a IN NUMBER, b IN NUMBER, result OUT NUMBER);
END MathOperations;
/
```

Package Body:

- It contains the formal definition of all the objects declared in the specification section.

Syntax:

```
CREATE OR REPLACE PACKAGE BODY packageName
IS/AS
 package body
END  packageName;
```

If a package contains only variables, constants and exceptions then package body is optional.

Example

```
CREATE OR REPLACE  PACKAGE BODY  MathOperations  AS
  PROCEDURE AddNumbers(a IN NUMBER, b IN NUMBER, result OUT NUMBER) IS
  BEGIN
result := a + b;
  END AddNumbers;
END MathOperations;
/
```

Now, you can use the MathOperations package in a PL/SQL block:

```
DECLARE
sum_result NUMBER;
BEGIN
MathOperations.AddNumbers(10, 5, sum_result);
  DBMS_OUTPUT.PUT_LINE('Sum Result: ' || sum_result);

END;
/
```

Advantages

- Advantages of package are given below:
  1) **Modularity:**
     - Package provides modular approach to programming.
     - It is always to better to write more than **one smaller programs** instead of **one large program**.
  2) **Security:**
     - Programs can be created to provide various functionalities and can be group together into packages.
     - Privileges can be granted to these packages rather than entire tables. So, **privileges can be granted efficiently**.
  3) **Improved Performance:**
     - An entire package, including all objects within it, is loaded into memory when the first component is accessed.
     - This eliminates additional calls to other related objects which results in reduced disk I/O.
     - So, performance can be improved.
  4) **Sharing of Code:**
     - Once a package is created, objects in that package can be shared among multiple users.
     - This reduces the **redundant coding**.
  5) **Overloading of procedures and functions:**
     - Procedures and functions can be overloaded using packages.

Que: Fundamentals of Database Triggers / Creating Triggers/ Types of Triggers: Before, after for each row, for each statement

- Triggers are the SQL statements that are **automatically executed** when there is any change in the database.
- The triggers are executed **in response to certain events** (INSERT, UPDATE or DELETE) in a particular table.

The advantages of triggers are as given below:

- To prevent **misuse** of database.
- To implement **automatic backup** of the database.
- To implement **business rule constraints**, such as balance should not be **negative**.
- Based on change in one table, we want to update other table.

Syntax

```
CREATE TRIGGER   Trigger_name
        (BEFORE | AFTER)
ON       [table_name]
        [for each row]
        [trigger_body]
DECLARE
         Declaration section
BEGIN
        Executable statements
EXCEPTION
        Exception handling
END ;
```

1. **CREATE TRIGGER:** specify that a triggered block is going to be declared.
2. **TRIGGER_NAME:** It creates or replaces an existing trigger with the Trigger_name.
3. **BEFORE | AFTER:** It specifies when the trigger will be initiated i.e. before the ongoing event or after the ongoing event.
4. **INSERT | UPDATE | DELETE** : These are the DML operations and we can use either of them in a given trigger.
5. **ON[TABLE_NAME]:** It specifies the name of the table on which the trigger is going to be applied.
6. **FOR EACH ROW:** Row-level trigger gets executed when any row value of any column changes.
7. **TRIGGER BODY:** It consists of queries that need to be executed when the trigger is called.

❖ **Types of Triggers**

BEFORE Triggers:

- Executed before the triggering event (e.g., INSERT, UPDATE, DELETE).
- Commonly used for validation or modification of data before the change occurs.
- Specified using the BEFORE keyword.

AFTER Triggers:

- Executed after the triggering event has occurred and the changes have been made.
- Often used for tasks that need to be performed after the data has been modified.
- Specified using the AFTER keyword.

Row-level Triggers:

- Operate on each row affected by the triggering event.
- Specified using the FOR EACH ROW clause.

Statement-level Triggers:

- Operate once for each triggering event, regardless of the number of rows affected.
- Do not use the FOR EACH ROW clause.

Example

STUDENT

ROLLNO	SNAME	AGE	COURSE
11	Anu	20	BSC
12	Asha	21	BCOM
13	Arpit	18	BCA
14	Chetan	20	BCA
15	Nihal	19	BBA

```
CREATE OR REPLACE TRIGGER CheckAge
BEFORE
INSERT OR UPDATE ON student
FOR EACH ROW
BEGIN
        IF :new.Age>30 THEN
                raise_application_error(-20001, 'Age should not be greater than 30');
        END IF;
END;
```

After initializing the trigger CheckAge, whenever we will insert any new values or update the existing values in the above table STUDENT our trigger will check the age before executing INSERT or UPDATE statements and according to the result of triggering restriction or condition it will execute the statement.

INSERT into STUDENT values(16, 'Saina', 32, 'BCOM');

Output:
Age should not be greater than 30

Advantages of Triggers in SQL

1. Helps us to automate the data alterations.
2. Allows us to reuse the queries once written.
3. Provides a method to check the data integrity of the database.
4. Helps us to detect errors on the database level.
5. Allows easy auditing of data.

Disadvantages of Triggers in SQL

1. Increases the overhead costs of the server.

2. Provides only extended validations i.e. not all validations are accessible in SQL triggers.

3. Troubleshooting errors due to triggers is a tedious job.

4. Can cause logical errors in the application even if a slight mistake in query exists.

5. We could lose the original data if we set a wrong trigger by mistake.