COMPUTER SCIENCE

B. Sc. II (CBCS)
Semester-IV
2023-2024

2CS2 : RDBMS and Core Java

Unit-III : PL/SQL

PROF. V. V. AGARKAR

Assistant Professor & Head

Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

UNIT – III

***Syllabus*: PL/SQL:** Features and block structure, variables and constant, data types, Identifiers, Operators and expression, Conditional statement, iterative statement. **Cursor:** Concepts of cursor, types of cursor, declaring, opening, using cursors, fetching data, closing a cursor, cursor attributes, Handling Exceptions, Creating Procedures, Creating Function, **Triggers:** Create Triggers, Types of Triggers, Creating BEFORE and AFTER Triggers, INSTEAD-OF triggers, Inserting, Updating and Deleting Triggers.

Introduction

SQL is a very flexible, powerful and easy to learn language. However SQL is a non-procedural language i.e. it does not contain any programming constructs. Hence cannot be used as an application development tool. To use programming constructs with SQL, Oracle provides PL/SQL.

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.

PL/SQL basically stands for "Procedural Language extensions to SQL". This is the extension of Structured Query Language (SQL) that is used in Oracle. Unlike SQL, PL/SQL allows the programmer to write code in procedural format. It combines the data manipulation power of SQL with the processing power of procedural language to create a super powerful SQL queries. It allows the programmers to instruct the compiler 'what to do' through SQL and 'how to do' through its procedural way. Similar to other database languages, it gives more control to the programmers by the use of loops, conditions and object oriented concepts.

Advantages of PL/SQL

1. PL/SQL is development tool that not only supports SQL data manipulation but also provides facilities of conditional checking, branching and looping.
2. PL/SQL sends an entire block of statements to the Oracle engine one at a time. The communication between the program and the Oracle engine reduces. This in turn reduces network traffic.
3. PL/SQL allows declaration and use of variables in blocks of code. These variables can be used to store intermediate results of a query for later processing or calculate values.
4. All PL/SQL codes are portable to any operating system and platform on which Oracle runs. Hence, PL/SQL code blocks written for a DOS version of Oracle will run on its UNIX version, without any modifications at all.
5. Disk I/O is reduced because related functions and procedures are stored together.
6. Errors are easily detected and handled. It also facilitates displaying user friendly messages when errors are encountered.

Features of PL/SQL

Some of the features of PL/SQL are as follows:

1. **Variables and constants:** Variables and constants can be defined in PL/SQL.
2. **Procedural Capabilities:** PL/SQL provides control structures to control the flow of a program. The control structures supported by PL/SQL are IF..THEN, LOOP, FOR..LOOP and Others.
3. **Exception Handling:** PL/SQL allows errors, called exceptions, to be detected and handled. Pre-defined and user-defined error can be handled in PL/SQL. To handle these errors, user can write the code in the form of Exception handlers in PL/SQL.
4. **Cursor:** PL/SQL supports row by row processing using the cursor.
5. **Modularity:** PL/SQL allows process to be divided into different modules, subprograms called procedures, functions and packages.
6. **Better performance:** PL/SQL block is sent as one unit to Oracle server. Without PL/SQL each SQL command is sent to Oracle server, which will increase network traffic heavily. As a collection of SQL statements is passed as a block to Oracle Server, it improves performance.
7. **Portability:** Applications written in PL/SQL are portable to any platform on which Oracle runs without any changes.

PL/SQL Block (*Anonymous Block*)

PL/SQL is a block-structured language. The units that constitute a PL/SQL program are logical blocks. A PL/SQL program may contain one or more blocks (called sub-blocks). Each block may be divided into three sections. The variables are declared before they are used. The following are the different sections of a PL/SQL block:

1. The **Declare** section
2. The Execution **Begin** section
3. The optional **Exception** section, and
4. The **End** section

A PL/SQL block can be diagrammatically represented as follows:

```
[Declare
      … Declarations]

Begin
      … Statements

[Exception
      … Error Handlers]
End;
```

[**Fig. 1:** PL/SQL BLOCK STRUCTURE]

The sections of PL/SQL Block are explained as follows:

- **The Declare Section:**

  The ***declaration section*** contains the declaration of variables and constants used in the executable section of the block. If you don't need to declare a variable, you can omit this section. This section also declares cursors and user-defined exceptions that are referenced in the other block sections.

- **The Executable (Begin) Section:**

  The ***executable section*** is a compulsory section in the PL/SQL block. This section begins with the keyword BEGIN. All the executable statements are given in this section. Actual data manipulation, retrieval, looping and branching constructs are specified in this section.

- **The Exception Section:**

  The ***exception section*** is an optional section, which has executable statements to handle an exception or error that arise during execution of the statements.

- **The End Section:**

  The ***End section*** makes the end of a PL/SQL Block.

➢ *Each statement or declaration in a PL/SQL block is terminated with a semicolon. A statement may be broken down into multiple lines for readability, and the semicolon marks the end of the statement. More than one statement can appear in one line, separated by a semicolon.*

Identifiers

Identifiers are the names given to PL/SQL elements, like tables, cursors or variables.

- Identifiers can be up to 30 characters in length.
- Must start with a letter.
- Then can have any combination of letters, numbers, $, #, _ .

operators

PL/ SQL provide the following types of operators:

Arithmetic operators	:	+, -, *, /, **
Relational operators	:	=, <> 0r != or ~=, >, >=, <, <=
Comparison operators	:	LIKE, BETWEEN, IN, IS NULL
Logical operators	:	AND, OR, NOT
Expression operators	:	:=, .., ||

Variables

Variables are memory locations, which can store data values. As the program runs, the contents of the variables can and do change. Information from the database can be assigned to a variable, or the contents of the variable can be inserted into the database.

Variable are declared in the declarative section of the block. The general syntax for declaring a variable is:

```
VariableName [CONSTANT] datatype [:= value];
```

Where *VariableName* is the name of the variable, *datatype* is the data type and *value* is initial value of the variable.

Example:

The following is an example of variables being defined:

```
1. roll_no number(3);
2. seatnumber number(5) := 25000;
```

- ***Assigning values to variables***

You can assign values to variables by two ways:

1. Using the assignment operator (**:=**)

   For Example,
   ```
   name := 'Ram';
   roll_no := '555';
   total := cps +  mth + etc;
   age := &age;
   ```

2. Selecting or fetching a table data values into variables. For this a SELECT statement is used. The following selects a value into the variable named r_n.

   ```
   r_n number(3);
   select roll_no into r_n from student
   where name := 'Ram';
   ```

There can be only one variable declaration per line in the declaration section. The following section is invalid, since two variables are declared in the same line.

```
v_fname, v_lname varchar2(10);
```

Constant

A ***constant*** is similar to a variable, but its value cannot be changed inside the program. The value to a constant must be assigned when the constant is defined. Its declaration is the same as a variable declaration, but the keyword `CONSTANT` and initial value is included.

Examples :-

```
1.  i_tax constant number(5) := 2000;
2.  counter constant number(3) := 100;
```

Variable and Constant Attributes

Each PL/SQL variable and constant has attributes associated with it. These attributes are properties of the variable or constant that you can reference. Following are the attributes:

1. %type

PL/SQL can use this attribute to declare variables based on definitions on previously defined variables or columns in a table. Hence, if a column's attribute changes, the variable's attribute will change as well.

%type declares a variable or constant to have the same data type as that of a previously defined variable or of a column in a table. When referencing a table, you may name the table and column separated by dot (.).

Examples:

1. ```
   FirstName varchar2(10);
   LastName FirstName%type;
   ```

   In this example, the variable *LastName* is declared with the same data type (size also) as is defined for the *FirstName* variable.

2. ```
   v_name student.name%type
   ```

   In this example, the variable *v_name* is declared with the same data type as is defined for the column *name* in the *student* table.

2. %rowtype

This attributes describes a record type that represents a row in the table. The *%rowtype* attribute can be applied to a database table. It will return the type of a PL/SQL record consisting of all the columns in the table, in the order in which they were specified at table creation.

Example:

1. The following example creates a record named ***stud_rec*** that has fields with the same name and data type the columns that appear in the table *student*.

   ```
   stud_rec student%rowtype;
   ```

Now use the *dot* notation to access any field in the *rowtype* record. The next statement assigns the field *roll_no* to the variable r_no:

```
r_no := stud_rec.roll_no;
```

PL/SQL Data types

To handle various types of information efficiently and conveniently, data types are used to represent data. PL/SQL variables and constants have data types. The datatype specifies a storage format, constraint, and a valid range of values. The PL/SQL provides variety of predefined datatypes, which can be divided into four categories.

- Scalar — Represents a single value with no internal components.
- Composite — It is a collection of internal components that can be accessed individually.
- Reference — It is a pointer that points to another data item.
- Large Object (LOB) — Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.

The following are the datatypes in various categories:

Scalar	NUMBER, CHAR, VARCHAR2, DATE, BOOLEAN
Composite	RECORD, TABLE, VAARAY
Reference	REF CURSOR, REF Object_type
Large Object (LOB)	BFLE, BLOB, CLOB, and NCLOB

Note :- there may be minor differences between PL/SQL datatypes and SQL datatypes though they have the same name.

1. Scalar Datatypes

Scalar datatypes have no internal components. Scalar types do not have any components within the types and contain a single value. Scalar datatypes are similar to SQL datatypes. Following are some scalar datatypes:

i) CHAR(size)

The CHAR data type is used to store character strings of ***fixed length***. If a value that is inserted in a field of CHAR data type is shorter than the size is defined for it, then it will be padded (fill) with spaces on the right until it reaches the size characters in length. The maximum number of characters this data type can hold is 255 characters (i.e. size is 255 bytes in Oracle 7 and 2000 bytes in Oracle 8, 9i, 10g, and 11g).

ii) VARCHAR2(size)

The VARCHAR2 data type is used to store character strings of a ***variable length***. If a value that is inserted in a field of VARCHAR2 data type is shorter than the size it is defined for, then it will not be padded with spaces. The maximum this data type can hold is 2000 characters (i.e. size is 2000 bytes in Oracle 7 and 4000 bytes in Oracle 8, 9i, 10g, and 11g).

iii) NUMBER(P, S)

This data type is used to store negative, positive, integer, fixed-decimal and floating point numbers. When a NUMBER data type used its precision (P) and scale (S) can be specified. The ***precision P*** is a positive integer that indicates the total number of digits in the number, both to the left and to the right of the decimal point. The ***scale S*** is a total number of digits to the right of the decimal point. The precision P can range from 1 to 38.

iv) INTEGER or INT

The INTEGER data type accepts a 32-bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^{-31} and 2^{31}-1. Attempting to assign values outside this range causes an error.

v) FLOAT(p)

The FLOAT data type accepts a single or double precision floating point number value, for which you may define a precision up to a maximum of 64. If no precision is specified during the declaration, the default precision is 64. Attempting to assign a value lager than the declared precision will cause an error to be raised.

vi) DATE

The DATE data type is used to represent dates. The DATE data type accepts date values, consisting of year, month, and day. No parameters are required when declaring a DATE data type. Date values should be specified in the form: DD-MMM-YY.

vii) BOOLEAN

The BOOLEAN data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in: SQL statements, Built-in SQL functions (such as TO_CHAR).

2. Composite Datatypes

Composite datatypes have internal components you can manipulate. A composite type consists of more than one component within it. A variable of a composite type contains one or more scalar variables. The composite datatypes available in PL/SQL are `RECORD` and `TABLE`.

i) Table

A PL/SQL table is a one-dimensional, unbounded, sparse collection of homogeneous elements, indexed by integers. In technical terms, it is like an array. After tables are defined, they can be reused.

ii) Record

Records in PL/SQL programs are very similar in concept and structure to the rows of a database table. The record as a whole does not have value of its own; instead, each individual component or *field* has a value. The record gives you a way to store and access these values as a group.

3. Large Object (LOB) Data Types

Large object (LOB) data types refer large to data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types:

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

Comments

Comments promote program readability and understanding. Comments are for people, not computers. Oracle ignores comments. They are there simply for your benefit. There are two types of comments:

- **Single Line comment**

Single line comment begins with the double hyphen (--). This comment indicator can be begun anywhere on a line and continued to the end of that line. ***Example:***

```
part_name varchar2(10);      -- define part name
-- begin main processing
```

- **Multi Line comment**

Multi line comment indicator begins with /* and terminates with */. This comment can span multiple lines. A multiline comment is shown in following ***example***:

```
/* This PL/SQL block is used to find the
   factorial of given number.  */
```

Control Structures

A PL/SQL block has a series of SQL and PL/SQL statements. The execution will start from the first statement and will sequentially continue to the last statement. In certain situations you would like to control the flow of execution of statements like:

- execute a set of statements conditionally
- execute a set of statements repetitively
- branch execution to some other statement

PL/SQL has a variety of control structures that allow to control the behavior of the block as it runs. These control structures include *conditional*, *iterative*, *sequential* and *unconditional* control structures.

• Conditional Controls

Conditional statements check the validity of a condition and accordingly execute a set of statements.

1. IF–THEN Statement

The simplest form of the conditional statement is *if-then* statement. The syntax for the *if-then* statement is:

Syntax:

```
IF condition THEN
     Statement(s);
END IF;
```

First the *condition* is evaluated, if the *condition* evaluates to *true*, *statement(s)* will be executed. If the *condition* evaluates to *false*, the *statement(s)* are not processed and are passed over.

Example :-

```
IF sal >= 25000 THEN
     i_tax := sal*3/100;
END IF;
```

2. IF–THEN–ELSE Statement

The `IF-THEN-ELSE` statement executes a sequence of statements conditionally. The `IF` clause checks a condition, the `THEN` clause defines what to do if the condition is true and the `ELSE` clause defines what to do if the condition is false.

Syntax:

```
IF condition THEN
     Statement1(s);
ELSE
     Statement2(s);
END IF;
```

First the *condition* is evaluated, if the condition evaluates to *true*, *statement1(s)* will be executed and *statement2(s)* will not executed. If the *condition* evaluates to *false*, the *statement2(s)* will be executed and *statement1(s)* will not executed.

Example :-

```
IF age >= 18 THEN
     dbms_output.put_line('Eligible to vote');
ELSE
     dbms_output.put_line('Not Eligible to vote');
END IF;
```

3. IF–THEN–ELSIF Statement

With a normal *If* Statement you can only check one condition, but at times you will want to check for multiple conditions. This can be done with IF-THEN-ELSIF.

It allows you to continually nest statements using *ElsIf*. To use *ElsIf*, you need to separate each new case with the keyword *ElsIf* (one word), closing the condition as normally with *End If*.

Syntax:

```
IF condition-1 THEN
        Statement(s)-1;
ELSIF condition-2 THEN
        Statement(s)-2;
...
ELSIF condition-n THEN
        Statement(s)-n;
ELSE
        Statement-else(s);
END IF;
```

This statement woks like this, "first the *condition-1* is evaluated, if it is true, *statement(s)-1* is executed and the execution jumps down to the ending statement (*End If*) and continues with the next line of code. If *condition-1* is false, then it evaluates *condition-2*. If it is true, *statement(s)-2* is executed and the execution jumps down to the ending statement (*End If*) and continues with the next line of code. If *condition-2* is false, then it evaluates *condition-3* and so on. If all the *ElsIf* conditions evaluate to false, then *else statement(s)* is executed.

Example :-

```
IF per >= 60 THEN
    class := 'First class';
ELSIF per >= 45 THEN
    class := 'Second class';
ELSIF per >= 35 THEN
    class := 'Third class';
ELSE
    class := 'Fail';
END IF;
```

4. CASE Statement

The `CASE` statement is an alternative to the `IF-THEN-ELSIF` statement. The `CASE` statement begins with keyword `CASE` and ends with keywords `END CASE`. The body of the `CASE` statement contains `WHEN` clauses, with values or conditions, and action statements. When a `WHEN` clause's value/condition evaluates to TRUE, its action statements are executed. The `CASE` statement can be either simple or searched.

a) Simple CASE statement

A simple CASE statement evaluates a single expression and compares the result with some values. It has the following syntax:

```
CASE selector
   WHEN selector_value_1 THEN statements_1
   WHEN selector_value_2 THEN statements_2
   ...
   WHEN selector_value_n THEN statements_n
   ELSE else_statements
END CASE;
```

The `selector` is an expression (typically a single variable). Each `selector_value` can be either a literal or an expression. The simple `CASE` statement runs the first statements for which `selector_value` equals `selector`. Remaining conditions are not evaluated. If no `selector_value` equals `selector`, the `CASE` statement runs `else_statements` if they exist.

Example:

```
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;
```

Result of above program segment is:

```
Very Good
```

b) Searched CASE Statement

The searched `CASE` statement evaluates multiple Boolean expressions and executes the sequence of statements associated with the first condition that evaluates to TRUE. It has the following syntax:

```
CASE
     WHEN condition_1 THEN statements_1
     WHEN condition_2 THEN statements_2
     ...
     WHEN condition_n THEN statements_n
     ELSE else_statements
END CASE;
```

The searched `CASE` statement runs the first statements for which `condition` is true. Remaining conditions are not evaluated. If no `condition` evaluates to TRUE, the `else_statements` in the `ELSE` clause executes.

Example:

```
DECLARE
   grade CHAR(1);
BEGIN
   grade := 'B';
   CASE
      WHEN grade = 'A' THEN
                   DBMS_OUTPUT.PUT_LINE('Excellent');
      WHEN grade = 'B' THEN
                      DBMS_OUTPUT.PUT_LINE('Very Good');
      WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
      WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
      WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
      ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
   END CASE;
END;
```

Result of above program segment is:

```
Very Good
```

- **Functional difference between CASE and SEARCHED CASE statements**
  - The simple CASE performs a simple equality check of "n" against each of the "when" options.
  - The searched CASE evaluates the conditions independently under each of the "when" options. With this structure, far more complex conditions can be implemented with a searched CASE than a simple CASE.
  - A searched CASE can combine multiple tests using several columns, comparisons and AND/OR operators.
  - Note that in both simple and searched CASE constructs, the conditions are evaluated sequentially from top to bottom, and execution exits after the first match are found. So, suppose more than one condition is true, only the first action is considered.

• Iterative Controls

Iterative control statements are used to execute a set of statements repetitively. The iterative control statements supported by PL/SQL are follows:

1. `LOOP`
2. `WHILE LOOP`
3. `FOR LOOP`

1. LOOP Statement

It is the simplest form of iterative statement and has syntax:

Syntax:

```
LOOP
     Statement(s);
END LOOP;
```

The LOOP does not facilitate a checking for a condition and so it is an endless loop, it executes the *statement(s)* in the loop body infinite number of times. To end the iterations, the `EXIT` statement can be used.

Example:

```
LOOP
    total := total + 1;
    IF total >= 100 THEN
        EXIT;
    END IF;
END LOOP;
```

OR

```
LOOP
    total := total + 1;
    EXIT WHEN total >= 100;
END LOOP;
```

The statement total:=total+1 will be executed repeatedly until the condition given in IF..THEN evaluates to TRUE.

2. FOR Loop Statement

The FOR loop uses when the number of iterations is known in advance. The FOR-LOOP statement lets you specify a range of integers, then execute a sequence of statements once for each integer in the range.

Syntax:

```
FOR loop_counter IN [REVERSE] low_bound .. high_bound
LOOP
     Statement(s);
END LOOP;
```

Where, ***loop_counter*** is the implicitly declared variable so do not have to declare it. The ***low_bound*** and ***high_bound*** specify the starting and final values of a loop that decides the number of iterations, and ***statements*** are the contents of the loop. The ***low_bound*** and ***high_bound*** values can be literals, variables, or expressions, but they must evaluate to integers. The ***REVERSE*** keyword is optional. Without this

keyword, the loop counter increases by one with every iteration through the loop, from the lower to the upper bound. With ***REVERSE***, the loop will *decrease* by one instead, going from the upper to the lower bound. The two dots (. .) is a special operator that means "visit all the integers between *lower_bound* and *upper_bound*". *PL/SQL will increment or decrement the loop index only by 1.*

After the each iteration ***loop_counter*** is automatically incremented. The body of the loop is evaluated once. These determine the total number of iterations. *Loop_counter* will take on the values ranging from *low_bound* to *high_bound*, incrementing by 1 each time, until the loop is complete.

Examples:

```
1.   FOR temp IN 1 .. 50
     LOOP
          dbms_output.put_line(temp);
     END LOOP;
```

This loop will increment *temp* from 1 to 50.

```
2.   FOR temp IN REVERSE 1 .. 50
     LOOP
          dbms_output.put_line(temp);
     END LOOP;
```

This loop will decrement *temp* from 50 down to 1.

3. WHILE Loop Statement

The WHILE loop uses when the number of iterations is not known in advance. The test for the loop execution takes place before the loop takes place so it is possible that the loop will not be executed. The syntax is:

Syntax:

```
WHILE condition
LOOP
        Statement(s);
END LOOP;
```

Before the each iteration of the loop, the *condition* is evaluated. If the condition is *true*, the statements are executed. If the condition is *false* or *null*, the loop is bypassed and control passes to the next statement after the *END LOOP* statement.

Example:

```
WHILE counter <= 50
LOOP
     dbms_output.put_line(counter);
     counter := counter + 1;
END LOOP;
```

• Unconditional Controls

1. GOTO statement

A GOTO statement transfers the control to a label unconditionally. When **GOTO** statement is encountered, the control immediately passes to the statement that follows the label. The syntax of the GOTO statement is:

Syntax:

```
GOTO label;
```

Where, ***label*** is a label defined in the PL/SQL block. Labels are defined by double angle brackets. For example, <<labelname>>.

Example:

In the following example, the GOTO statement is used to transfer the control to the label ***new_part***.

```
IF AGE > 18 THEN
     GOTO new_part;
END IF;
...
<<new_part>>
     …
     …
```

CURSORS

Don't write this in your answer

One of the most important functions of a database is to retrieve the data stored in the tables. For this **SELECT..INTO** query is provided in PL/SQL. But there is problem with the **SELECT..INTO** statement, if it returns more than one row from database. Then it generates an exception (error) of TOO_MANY_ROWS.

The variables in PL/SQL are designed to accommodate one instance of data item. If you need to pull back multiple rows of data from the database and work at a time, you need to use cursors. In cursors, multiple rows that match the query are available to the user, but only one at a time. You define the query first and then scroll through the results and perform whatever processing you need.

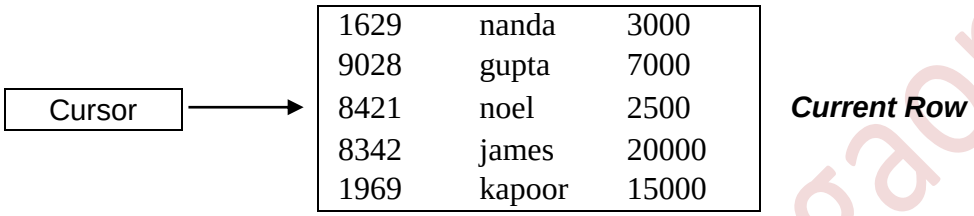
Concept of Cursors

Cursors are the constructs that enable the user to name the private memory area to hold the specific statement for access at a later time. Cursors are used to process query results. The number of rows returned by the query may be zero, one or many depending on query search criteria. Oracle provides the cursors as the mechanism to be used to easily process these multiple row result sets one at time. Without cursors, the Oracle developer will have to explicitly fetch and manage each individual row that is selected by the cursor query.

The multiple rows returned from the query are called the "***Active Set***". PL/SQL defines its size as the number of rows that have met your query search criteria and formed the active set. Following figure-(2) illustrates a cursor that is holding multiple rows. In every cursor there is a ***pointer*** that keeps the track of current row being accessed, which enables your program to process the rows one at a time.

Query:

```
SELECT EMPNO, ENAME, PAY FROM EMP
WHERE DEPTNO = 1;
```



[**Fig.2 :** A Multiple Row Cursor]

Types of Cursors

Cursors are classified depending on the circumstances under which they are opened. There are two types of cursors: *Implicit* and *Explicit*.

1. Implicit Cursors

If the Oracle engine for its internal processing has opened a cursor, it is known as an *'Implicit Cursor'*. Implicit cursors are declared by PL/SQL for all DML statements and for single row queries. A system have its own cursor is called Implicit Cursor, which is executed when SQL statement is used.

2. Explicit Cursors

A user can define a cursor for processing data as required. Such user–defined cursors are known as **'***Explicit Cursors'*. Explicit cursors are declared in PL/SQL program according to our requirement using variables in PL/SQL block. It allows only queries and multiple rows to be processed from the query.

- ***Processing Explicit Cursors***

Following are the steps to create and use the Cursor:

1. Declare the cursor
2. Open the cursor
3. Fetch the data into the cursor
4. Close the cursor

The cursor declaration is the only step that goes in the declarative section of a block. The other three steps are found in the executable or exception section.

1. Declaring a Cursor

A cursor must be declared before it is used by PL/SQL. An explicit cursor is defined in the declaration section of a PL/SQL block. Declaring the cursor accomplishes two goals:

- It names the cursor
- It associates a query with the cursor

The syntax for declaring the cursor is:

```
CURSOR CursorName IS SELECT_Statement;
```

Where, **CursorName** is the name of the cursor. The name assigns to a cursor is an undeclared identifier, not a PL/SQL variable. Values cannot be assigned to a **CursorName** or cannot use it in an expression. **Select_statement** is the query that defines the set of rows to be processed by the cursor.

Examples:

1. In the following example, a cursor named C_EMP is declared.

```
CURSOR C_EMP IS
SELECT * FROM EMP;
```

Above, cursor C_EMP is declared and contains all the columns of table EMP.

2.

```
CURSOR C1_EMP IS
SELECT ENAME, DEPT_NO, PAY FROM EMP
WHERE PAY >= 5000;
```

2. Opening a Cursor

Opening a cursor executes the query and creates the ***active set*** that contains all rows, which meet the query search criteria. When the OPEN command is executed, the cursor identifies only the rows that satisfy the query. The rows are not actually retrieved until the cursor fetch is issued. The syntax for opening a cursor is:

```
OPEN CursorName;
```

Where, CursorName identifies a cursor that has previously been declared. An OPEN statement retrieves records from the database table and places the records in the cursor. When a cursor is opened, the following things happen:

- The active set is determined.
- The active set pointer is set to the first row.

Example:

```
OPEN C_EMP;
```

The above statement opens the cursor C_EMP and sets a pointer to first record in the active set.

3. Fetching the data from the cursor

The FETCH statement is used to retrieve a row (current row) from the active set, one at a time, into the PL/SQL variables. Each time `FETCH` is executed, it moves one row from active set into the memory variables and also advances the cursor pointer to the next row in the Active set. The `FETCH` statement's has syntax:

Syntax:

```
FETCH CursorName INTO List_Of_Variables;
```

OR

```
FETCH CursorName INTO PL/SQL_Record;
```

Where, `CursorName` identifies a previously declared and opened cursor, `List_of_Variables` is a comma separated list of previously declared PL/SQL variables and `PL/SQL_record` is a previously declared PL/SQL record variable.

For each column value returned by the cursor query, there must be a corresponding variable in the `INTO` list of `FETCH` statement. In either cases, the variables in the `INTO` clause must be type compatible with the `SELECT` list of query.

The `FETCH` statement is placed inside a loop, which causes the data to be fetched into the memory variables and processed until all the rows in the active data set are processed. The fetch then exits. The exiting of the `FETCH` loop is user controlled.

Examples:

1. `FETCH C_EMP INTO ENO, ENAM, SAL;`
2.
```
BEGIN
    OPEN C1_EMP;
    LOOP
        FETCH C1_EMP INTO ENO, ENAM, SAL;
        EXIT WHEN C1_EMP%NOTFOUND;
    END LOOP;
END;
```

In the above examples `ENO, ENAM, SAL` are previously defined variables.

4. Closing the Cursor

When all records of the active set have been retrieved, the cursor should be closed. When the cursor is closed, it release all the resources associated to the cursor. These resources include the memory occupied by the cursor and the active set. Once a cursor is closed, fetching from it will yield the error. The **`CLOSE`** statement is used to close the cursor. The syntax for closing cursor is:

Syntax:

```
CLOSE CursorName;
```

Where, **CursorName** is a previously opened cursor's name.

Example:

1. `CLOSE C_EMP;`

Cursor Attributes

Each defined cursor has four attributes. These attributes can be accessed to obtain useful information about the cursor. These cursor attributes are as follows:

1. %ISOPEN

This attribute evaluate to TRUE, if an explicit cursor is open; or to FALSE, if it is closed. The syntax for accessing this attribute is:

```
CursorName%ISOPEN
```

Example:

The following example checks whether the cursor named C_CURSOR is open or not. If it's already open, the fetch is executed. If the cursor is closed, the OPEN cursor command is used.

```
IF C_CURSOR%ISOPEN THEN
     FETCH C_CURSOR INTO A, B;
ELSE
     OPEN C_CURSOR;
END IF;
```

2. %FOUND

This attribute evaluate TRUE, if the last fetch succeeded because a row was available to fetch; or to FALSE, if the last fetch failed because no more rows are available. The syntax for accessing this attribute is:

```
CursorName%FOUND
```

Example:

The following example uses the %FOUND attribute to control the execution of the INSERT command.

```
LOOP
     FETCH C_CURSOR INTO A, B;
     IF C_CURSOR%FOUND THEN
          INSERT INTO MASTER VALUES(A, B);
     ELSE
          EXIT;
     END IF;
END LOOP;
```

3. %NOTFOUND

It evaluates to TRUE, if the last fetch is failed because no more rows were available; or to FALSE, if the last fetch returned a row. (It is the logical opposite of %FOUND.) The syntax for accessing this attribute is:

```
CursorName%NOTFOUND
```

Example:

The following example uses the %NOTFOUND attribute to exit a loop when there are no more rows to process.

```
LOOP
     FETCH C_CURSOR INTO A, B;
     EXIT WHEN C_CURSOR%NOTFOUND;
END LOOP;
```

4. %ROWCOUNT

This attribute returns the number of rows fetched from the active set. It is set to zero when the cursor is opened. The syntax for accessing this attribute is:

```
CursorName%ROWCOUNT
```

Example:

The following example uses %ROWCOUNT to halt execution after the first 100 rows have been processed.

```
LOOP
     FETCH C_CURSOR INTO A, B;
     EXIT WHEN C_CURSOR%ROWCOUNT > 100;
END LOOP;
```

Cursor FOR Loop

To use explicit cursor requires explicit processing of the cursor. This is done via the OPEN, FETCH, and CLOSE cursor statements. PL/SQL provides a short cut to this, via the cursor FOR loop, which implicitly handles the cursor processing.

The cursor FOR loop automatically does the following:

1. Implicitly declares its loop index as a ***%rowtype*** record.
2. Opens a cursor.
3. Fetches the row from the cursor for the each loop iteration.
4. Closes the cursor when all the rows have been processed.

The following is syntax of a cursor FOR LOOP:

```
FOR memory_variable IN cursor_name
LOOP
     Statement(s);
END LOOP;
```

Where, *memory_variable* is an undeclared identifier. It is automatically created and defined as the ***%rowtype*** of the cursor. Each record in the opened cursor becomes a value for the *memory_variable* of the *%rowtype*. *Cursor_name* is a name of an opened cursor.

The FOR ensures that a row from the cursor is loaded in the declared *memory_variable* and the loop executes once. This goes on until all the rows of the cursor have been loaded into the memory variable. When all the rows from the active set are completely fetched, the cursor is closed automatically at the end of the loop.

Example:

```
DECLARE
     CURSOR emp_cur IS
     SELECT * FROM emp WHERE deptno = 1;
BEGIN
     ...
     FOR v_emp_cur IN emp_cur
     LOOP
          dbms_output.put_line(v_emp_cur);
     END LOOP;
     ...
END;
```

Handling Exceptions

Exceptions are runtime errors or unexpected events that occur during the execution of a PL/SQL code block and that disrupts the normal flow of program instructions. PL/SQL provides the `Exception` block to handle the exceptions. The basic structure of how exception handling works in PL/SQL is as follows:

```
DECLARE
    -- Declare variables
BEGIN
    -- Main block of PL/SQL code
EXCEPTION
    -- Exception handling block
    WHEN <exception_name> THEN
        -- Code to handle the specific exception
    WHEN OTHERS THEN
        -- Code to handle any other exception
END;
```

In the above syntax:

- **`EXCEPTION`**: This marks the beginning of the exception handling block. Inside this block, you can handle specific exceptions or catch any other unhandled exceptions.
- **`WHEN <exception_name> THEN`**: This is a specific exception handler where you can catch a particular exception by name. You can have multiple `WHEN` clauses to handle different exceptions.
- **`WHEN OTHERS THEN`**: This is a catch-all handler that catches any exception not caught by the specific exception handlers.

- ***Types of exceptions in PL/SQL:***

In PL/SQL, exceptions can be broadly categorized into three types:

1. System-Defined Exceptions

These are predefined exceptions provided by PL/SQL. They cover common errors such as division by zero, invalid cursor operation, and others. Some common system-defined exceptions include:

`NO_DATA_FOUND`	Raised when a SELECT INTO statement returns no rows.
`TOO_MANY_ROWS`	Raised when a SELECT INTO statement returns more than one row.
`ZERO_DIVIDE`	Raised when attempting to divide by zero.
`INVALID_CURSOR`	Raised when attempting operations on an invalid cursor.
`CURSOR_ALREADY_OPEN`	Raised when attempting to open a cursor that is already open.
`LOGIN_DENIED`	Raised when login to Oracle database fails due to invalid username/password.

Example:

```
DECLARE
    -- Declare variables
BEGIN
    -- Main block of PL/SQL code
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Value Error occurred');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred');
END;
```

2. User-Defined Exceptions

These are exceptions defined by the user. Users can define their own exceptions to handle specific conditions in their application logic. User-defined exceptions are declared in the declaration section of PL/SQL blocks.

Example:

```
DECLARE
    my_exception EXCEPTION;
BEGIN
    IF condition THEN
        RAISE my_exception;
    END IF;
EXCEPTION
    WHEN my_exception THEN
        -- Handle the user-defined exception
END;
```

Creating Procedures

A PL/SQL procedure (stored procedure) is a named block that performs a specific task. Procedures are standalone blocks of a program that can be stored. Each procedure in PL/SQL has its own unique name by which it can be referred to and called. A procedure may or may not return a value.

Creating procedures involves defining a set of SQL and procedural statements to perform a specific task or set of tasks. The basic syntax to create procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter1 [IN | OUT | IN OUT] datatype [, ...])]
IS
    -- Declaration section (optional)
    variable_declarations;
BEGIN
    -- Executable section
    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, it contains one or more PL/SQL statements that define the logic of the procedure.
- `END` statement: Every PL/SQL block (including procedures) ends with the `END` keyword followed by the name of the procedure.

Example:- Following procedure is created to insert values in STUDENT table:

```
create or replace procedure INSERTSTUDENT
   (id IN NUMBER, name IN VARCHAR2)
is
begin
     insert into student values(id, name);
end;
/
```

Creating Function

The PL/SQL function is very similar to PL/SQL procedure. The main difference between procedure and a function is that, a function must always return a value and a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too. Following is syntax to create function:

```
CREATE [OR REPLACE] FUNCTION function_name
    [(parameter [, parameter])]
RETURN return_datatype
IS|AS
    [declaration_section]
BEGIN
    executable_section
[EXCEPTION
    exception_section]
END [function_name];
```

Where, the `function_name` specifies the name of the function. The `[OR REPLACE]` option allows modifying an existing function. The optional `parameter` list contains name, mode and types of the parameters. The function must contain a return statement. The `RETURN` clause specifies that data type you are going to return from the function. The `AS` keyword is used instead of the `IS` keyword for creating a standalone function.

Example :

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

Database Trigger

In PL/SQL, triggers are named blocks of code that are automatically executed (or "triggered") in response to specific events occurring in a database. These events can include DML statements such as INSERT, UPDATE, or DELETE, as well as DDL statements like CREATE, ALTER, or DROP.

Triggers are useful for enforcing data integrity constraints, auditing changes to data, implementing complex business rules, and automating repetitive tasks within the database. Triggers could be defined on the table, view, schema, or database.

Types of Triggers

Triggers can be classified based on the following parameters.

- Classification based on the **timing**
  - **BEFORE Trigger:** It fires before the specified event has occurred. Before triggers are commonly used to check the validity of the data before the action is performed.
  - **AFTER Trigger:** It fires after the specified event has occurred. For example, If after trigger is associated with INSERT command then it is fired after the row is inserted into the table.
- Classification based on the **level**
  - **STATEMENT level Trigger:** A statement trigger is fired only once for a DML statement irrespective of the number of rows affected by the statement. Statement-level trigger is the default type of trigger.
  - **ROW level Trigger:** A row trigger is fired once for each row that is affected by DML command. For example, if an UPDATE command updates 100 rows then row-level trigger is fired 100 times whereas a statement-level trigger is fired only for once. Row-level trigger are used to check for the validity of the data.

Creating a Trigger

CREATE TRIGGER command is used to create a trigger. The syntax is as follows:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER}
INSERT OR UPDATE [OF COLUMNS] OR DELETE
ON tablename
[FOR EACH ROW [WHEN (condition)]]
DECLARE
      Declaration statements
BEGIN
      Executable statements
EXCEPTION
      Exception handling statements
END;
```

Where, OR REPLACE is used to create a trigger even a procedure with the same name is already exists.

The BEFORE (or AFTER) in the trigger definition refers to when trigger wants to run, either before the actual database modification (update, delete, insert) or after.

The list of various statements, INSERT OR UPDATE [OF COLUMNS] OR DELETE refers to statements that fire this trigger. All three can be specified, or just one.

ON tablename specifies that the trigger is associated with the table.

If FOR EACH ROW option is used then it becomes a row-level trigger otherwise it is a statement-level trigger.

WHEN is used to fire the trigger only when the given condition is satisfied. This clause can be used only with row triggers.

The BEGIN and END is a usual code block where PL/SQL commands can be placed.

Examples-

1) Creating Before Trigger

```
create or replace trigger student_bi_row
before insert
on student
for each row
begin
  if :new.adm_dt >= sysdate then
     raise_application_error
    (-20002,'Admission date cannot be after system date.');
  end if;
end;
```

This is a simple **before trigger** and it is used to check whether date of admission of the student is less than or equal to system date. Otherwise it raises an error.

2) Creating After Trigger

```
create trigger item_tri
after insert
on item
for each row
begin
      insert into stock
      values(:new.ino, :new.iname, :new.qty, :new.rate);
end;
```

This is a simple **after trigger** and which insert a new row automatically into STOCK table whenever a new row is inserted into ITEM table.

INSTEAD-OF triggers

INSTEAD-OF triggers in PL/SQL are a special type of trigger that is primarily used with views. Unlike traditional BEFORE or AFTER triggers, which execute before or after an operation on a table, INSTEAD OF triggers are executed instead of the operation. They can be defined to execute INSTEAD OF INSERT, INSTEAD OF UPDATE, and INSTEAD OF DELETE operations on the view.

In Oracle, you can create an INSTEAD OF trigger for a view only. You cannot create an INSTEAD OF trigger for a table. The following is a syntax of creating an INSTEAD OF trigger:

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF {INSERT | UPDATE | DELETE}
ON view_name
FOR EACH ROW
BEGIN
    EXCEPTION
    ...
END;
```

In this syntax:

- First, specify the name of the trigger after the CREATE TRIGGER keywords. Use OR REPLACE if you want to modify an existing trigger.
- Second, use the INSTEAD OF keywords followed by an operation such as INSERT, UPDATE, and DELETE.
- Third, specify the name of the view with which the trigger is associated.
- Finally, specify the code that executes instead of the INSERT, UPDATE, and DELETE.

Example :

Suppose we have a view named employees_view that joins the employees table with the departments table, making it non-updatable directly due to the join operation. We can define an INSTEAD OF trigger to handle INSERT, UPDATE, and DELETE operations on the employees_view as below:

```
CREATE OR REPLACE TRIGGER instead_of_employees_view
INSTEAD OF INSERT OR UPDATE OR DELETE ON employees_view
FOR EACH ROW
BEGIN
    -- Custom logic to handle DML operations on the view
END;
```

Exercise

Fill in the blanks:

1. PL/SQL stands for …….
2. ……. section of PL/SQL block contains all the executable statements.
3. The …….. section of PL/SQL block is used to handle errors.
4. The ……. data type stores logical values TRUE or FALSE
5. The set of rows the cursor holds is referred to as ……..
6. …….. cursors are automatically created by Oracle whenever an SQL statement is executed.
7. The cursor attribute ……. Is used to check whether the cursor is open or not.
8. …….. cursors are programmer defined cursors.
9. A ……. is special stored procedure that runs when specific actions occur within a database.
10. ……. trigger is type of trigger fires before the specified event has occurred.

Choose the correct alternatives from the following:

1. ……. section of PL/SQL block contains all the executable statements.
   a) Declare b) Begin
   c) Exception d) End
2. This type of trigger fires one time for the specified event statement.
   a) Before b) After
   c) Statement Level d) Row Level
3. This statement uses to access one row at a time from explicit cursor.
   a) Open b) Access
   c) Read d) Fetch
4. User defined cursors are called as ……..
   a) Explicit Cursors b) Implicit Cursors
   c) Both (a) and (b) d) None of these
5. Which of the following executes the query and identifies the result set, consisting of all rows that meet the query search criteria.
   a) Fetching a cursor b) Opening a cursor
   c) Closing a cursor d) None of these

Answer in ONE sentence:

1. What is PL/SQL?
2. What is Cursor?
3. What is Implicit Cursor?
4. What is Explicit Cursor?
5. What is Trigger?
6. What is a purpose of Begin section is PL/SQL block?
7. What is a purpose of Exception section is PL/SQL block?
8. What is After triggers?
9. What is Before triggers?
10. How constant is declared in Pl/SQL?

Long answer questions:

1. What is PL/SQL? State the features of PL/SQL.
2. Explain the block structure of PL/SQL. Give example.
3. Explain the variables and constants in PL/SQL. Give examples.
4. Explain the following in PL/SQL.
   (i) Variable (ii) Constant (iii) Datatypes.
5. Explain datatypes supported by PL/SQL.
6. Explain the decision making statements supported by PL/SQL with example.
7. Explain the different control structure in PL/SQL with example.
8. Explain the loop control structures supported by PL/SQL with examples.
9. What is a cursor? Explain implicit and explicit cursors.
10. What is a cursor? Explain the steps to be carried out to use explicit cursor. Give example.
11. What is cursor? Explain the cursor attributes with example.
12. Explain cursor attributes.
13. Explain the following with respect to cursor.
    (i) Opening a cursor
    (ii) Declaration of cursor
    (iii) Fetching record from the cursor.

❐❐❐❐❐