PL/SQL

Why PL/SQL

PL/SQL sends entire block of statements to the Oracle engine at one time.

Provides facilities of conditional checking, branching and looping.

Handling of errors.

Declaration and use of variable in block code, to store intermediate results

Example

Write a PL/SQL procedure to calculate the area of a circle. The radius should be taken as input. Increase radius by a constant value to find area of circles with different radii.

```
declare
pi constant number(4,2):=3.14;
radius number(5);
area number(14,2);
begin
radius:=3;
loop
area:=pi * power(radius,2);
dbms_output.put_line(area);
radius := radius + 1;
exit when radius > 8;
end loop;
end;
```

Features

- Block Structure
- Constants and Variables
- Cursors
- Control Structure
- Modularity
- Data Abstraction
- Error Handling

Block Structure

PL/SQL is a block-structured language.

The basic units (procedures, functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks.

Each logical block corresponds to a problem or sub-problem to be solved.

Block Structure

DECLARE
---- Declarative section
BEGIN
-- Statement
-EXCEPTION handling
END;

Control Structure

Control structures are the most important PL/SQL extension to SQL.

It let you manipulate Oracle data and process the data using conditional, iterative, and sequential flow-of-control statements

Modularity

Modularity lets you break an application down into manageable, well-defined logic modules.

Besides blocks and subprograms, PL/SQL provides the package, which allows you to group related program items into larger units.

Data Abstraction

Data abstraction lets you extract the essential properties of data while ignoring unnecessary details.

Once you design a data structure, you can forget the details and focus on designing algorithms that manipulate the data structure.

Error Handling

PL/SQL makes it easy to detect and process predefined and user-defined error conditions called exceptions.

When an error occurs, an exception is raised.

Normal execution stops and control then transfers to the exception-handling part of your PL/SQL block or subprogram.

Error Handling

Predefined exceptions are raised implicitly by the runtime system.

For example, if you try to divide a number by zero, PL/SQL raises the predefined exception ZERO_DIVIDE automatically.

You must raise user-defined exceptions explicitly with the RAISE statement.

Condition Control

IF-THEN

IF-THEN-ELSE

IF-THEN-ELSIF

IF-THEN

Syntax:

```
IF condition THEN
sequence_of_statements;
END IF;
```

Example:

```
IF sales > quota THEN
        UPDATE payroll
        SET pay = pay + bonus
        WHERE empno = emp_id;
END IF;
```

IF-THEN-ELSE

Syntax:

```
IF condition THEN
  sequence_of_statements1;
ELSE
  sequence_of_statements2;
END IF;
```

Example:

```
IF trans_type = 'CR' THEN
        amt:=amt+deposit;
ELSE
        amt:=amt-deposit;
END IF;
```

IF-THEN-ELSIF

Syntax:

```
IF condition1 THEN   sequence_of_statements1;
ELSIF condition2 THEN
sequence_of_statements2;
ELSE   sequence_of_statements3;
END IF;
```

EXAMPLE:

```
BEGIN
  ...
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
END;
```

Example IF statement

```
DECALRE
acct_balance number(11,2);
acct_no varchar2(6);
debit_amt number(5):=2000;
min_bal constant number(5,2):=500.00;

BEGIN
 acct_no := &acct_no;
 Select bal INTO acct_balance FROM accounts where account_id=acct_no;
 acct_balance := acct_balance-debit_amt;
       if acct_balance>=min_bal Then
             Update accounts
             SET bal=bal-debit_amt where account_id=account_no;
       End If;
End;
```

Iterative Control

LOOP

EXIT

WHILE-LOOP

FOR-LOOP

LOOP

Syntax

```
LOOP
    sequence_of_statements;
END LOOP;
```

EXIT

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.

```
LOOP
  ...
  IF credit_rating < 3 THEN
    ...
    EXIT;  -- exit loop immediately
  END IF;
END LOOP;
-- control resumes here
```

WHILE-LOOP

- The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP

- Syntax

```
WHILE condition LOOP
  sequence_of_statements;
END LOOP;
```

WHILE-LOOP

```
WHILE total <= 25000 LOOP
   ...
   SELECT sal INTO salary FROM emp WHERE ...
   total := total + salary;
END LOOP;
```

EXAMPLE of WHILE Loop

```
DECLARE
pi constant number(4,2):=3.14;
radius number(5);
area number(14,2);

BEGIN
   radius :=3;
   while radius<=7
LOOP
 area:=pi * power(radius,2);
 Insert into areas values(radius,area);
 radius := radius +1;
END LOOP;
END;
```

FOR-LOOP

FOR loops iterate over a specified range of integers

Syntax

```
FOR counter IN [REVERSE] lower_bound..
higher_bound LOOP
  sequence_of_statements;
END LOOP;
```

Example

```
FOR i IN 1..3 LOOP
    sequence_of_stmt;
END LOOP;
```

Example of FOR Loop

```
DECALRE
 given_number varchar(5) :='5639';
 str_length number(2);
 inverted_number varchar(5);

BEGIN
 str_length := length(given_number);

FOR cntr IN REVERSE 1..str_length
LOOP
inverted_number := inverted_number || substr(given_number,cntr,1);
END LOOP;
 dbms_output.put_line('The given number is' || given_number);
 dbms_output.put_line('The inverted number is ' || inverted_number);
END;
```

Cursors

Oracle uses work areas called "private SQL areas" to execute SQL statements and store processing information

A PL/SQL construct called a "cursor" helps name a private SQL area and access its stored information

There are two kinds of cursors: **Implicit and Explicit**

PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row

For queries that return more than one row, a cursor can be explicitly declared to process the rows individually

Implicit Cursors

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicitly declared cursor

The values of cursor attributes always refer to the most recently executed SQL statement, wherever that statement appears

If an attribute attribute value has to be saved for later use, it should be assigned to a Boolean value

For example

```
UPDATE parts SET qty = qty - 1 WHERE partno = part_id;
sql_notfound := SQL%NOTFOUND;
check_parts;
IF sql_notfound THEN
  ...
END IF;
```

Explicit Cursors

The set of rows returned by a multirow query is called the "active set." Its size is the number of rows that meet your search criteria

An explicit cursor points to the current row in the active set. This allows the program to process the rows one at a time

Once a cursor is declared, three commands are used to control it: OPEN, FETCH, and CLOSE

Steps In Writing A Cursor

Declare the cursor mapped to a SQL statement that retrieves data for processing.

Initialize the cursor with the OPEN statement, which identifies the active set

Use the FETCH statement to retrieve the first row. FETCH can be used repeatedly until all rows have been retrieved ,into memory variable.

Process the data held in memory variable.

When the last row has been processed, you release the cursor with the CLOSE statement.

Declaring a Cursor

```
DECLARE
CURSOR c1 IS
SELECT * FROM emp;
      CURSOR c2 IS
SELECT a.emp_name, b.dept_name
FROM emp a, dept b
WHERE a.dept_no = b.dept_no;
```

OPEN Statement

The OPEN statement executes the query associated with an explicitly declared cursor

OPENing the cursor executes the query and identifies the active set, which consists of all rows that meet the query search criteria

Rows in the active set are not retrieved when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows

For example

```
OPEN c1;
```

FETCH Statement

The FETCH statement retrieves the rows in the active set one at a time into memory variable declared .

Each time FETCH is executed, the cursor advances to the next row in the active set

For Example :

```
FETCH c1 INTO my_empno, my_ename, my_deptno;
```

Typically, FETCH can be used as follows:

```
OPEN c1;
LOOP
  FETCH c1 INTO my_record;
  EXIT WHEN c1%NOTFOUND;
  -- process retrieved data
END LOOP;
```

CLOSE Statement

The CLOSE statement disables the cursor, and the active set becomes undefined

For example :

```
CLOSE c1;
```

Once a cursor is closed, you can reopen it. Any other operation on a closed cursor raises the predefined exception INVALID_CURSOR

EXAMPLE of CURSOR

Write a PL/SQL block which will create a cursor and utilize it to increase the salary of employees by a certain percentage according to their departments.

```
declare
cursor c1 is select * from employee;
begin
for c1rec in c1
loop
if c1rec.deptno = 10 then
update employee
set ebasic = ebasic + 500;
else
update employee
set ebasic = ebasic + 1500;
end if;
end loop;
end;
```

Reference Cursor

A REF CURSOR is a datatype that holds a cursor value in the same way that a VARCHAR2 variable will hold a string value.

A REF Cursor allows a cursor to be opened on the server and passed to the client as a unit rather than fetching one row at a time.

One can use a Ref Cursor as target of an assignment, and it can be passed as parameter to other program units.

Ref Cursors are opened with an OPEN FOR statement. In most other ways they behave similar to normal cursors.

Difference between cursor and ref cursor

A REF Cursor have a return type, but Cursor doesn't have return type

REF Cursor can be associated with many no. of SQL statements where Cursor can be associated only with one SQL statement.

REF Cursor is dynamic ,Cursor is static

SYNTAX of REF Cursor

```
TYPE ref_cursor_name IS REF CURSOR
[RETURN record type];
```

EXAMPLE

- CREATE OR REPLACE PROCEDURE pass_ref_cur(p_cursor SYS_REFCURSOR) IS
  TYPE array_t IS TABLE OF VARCHAR2(4000)
  INDEX BY BINARY_INTEGER;
  rec_array array_t;
  BEGIN
    FETCH p_cursor BULK COLLECT INTO rec_array;
    FOR i IN rec_array.FIRST .. rec_array.LAST
    LOOP
      dbms_output.put_line(rec_array(i));
    END LOOP;
  END pass_ref_cur;
  /

EXAMPLE (Contd.)

```
set serveroutput on

    DECLARE
     rec_array SYS_REFCURSOR;
    BEGIN
      OPEN rec_array FOR
      'SELECT empname FROM employees';
      pass_ref_cur(rec_array);
      CLOSE rec_array;
    END;
    /
```

Triggers

- Trigger defines an action the database should take when some event occurs. They can be used to:
  - supplement declarative integrity
  - enforce complex business rules
  - audit changes to data
- The code within a trigger is made up of PL/SQL blocks.
- A trigger is executed implicitly and it does not accept any arguments.

Triggers

- A SQL trigger is **a database object which fires when an event occurs in a database**. We can execute a SQL query that will "do something" in a database when a change occurs on a database table such as a record is inserted or updated or deleted. For example, a trigger can be set on a record insert in a database table.

Triggers

- A trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Types of Triggers

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE

Triggers

```
CREATE TRIGGER audit_sal
   AFTER UPDATE OF sal ON emp
   FOR EACH ROW
BEGIN
   INSERT INTO emp_audit VALUES ...
END;
```

```
CREATE OR REPLACE TRIGGER triggername
BEFORE | AFTER triggeringEvent ON tablename
[ FOR EACH ROW ] [ WHEN condition]
DECLARE
   declaration
BEGIN
   body
EXCEPTION
   exceptions
END;
```

Trigger Example

```
create trigger stud_marks
before INSERT on
Student
for each row
set Student.total = Student.subj1 + Student.
    subj2 + Student.subj3, Student.per = Student.
    total * 60 / 100;
```

DML trigger with a reminder message

```
CREATE TRIGGER reminder1
ON Sales.Customer
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Customer Relations', 16,
   10);
GO
```

Trigger Example

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab
  FOR EACH ROW
WHEN (new.Empno > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff  := :new.sal  - :old.sal;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put('  New salary: ' || :new.sal);
    dbms_output.put_line('  Difference ' || sal_diff);
END;
/
```

Trigger Example

```
CREATE OR REPLACE TRIGGER Log_salary_increase
AFTER UPDATE ON Emp_tab
FOR EACH ROW
WHEN (new.Sal > 1000)
BEGIN
    INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
      VALUES (:new.Empno, SYSDATE, :new.SAL, 'NEW SAL');
END;
```

```
UPDATE Emp_tab SET Sal = Sal + 1000.0
    WHERE Deptno = 20;
```

If there are 6 employees in DeptNo 20 with sal > 100 , then trigger is fired 6 times