**National University of Computer & Emerging Sciences, Karachi**

**Computer Science-School of Computing**

**Fall 2024, Lab Manual – 08**

|  |  |
| --- | --- |
| **Course Code: CL-2005** | **Course : Database Systems Lab** |
| **Instructor(s) :** | **Mubashir** |

# Contents:

1. PL/SQL
2. Block Structure
3. Variable & types
4. Conditional Logic
5. Loops
6. Views
7. Procedures
8. Functions
9. Cursors

# PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. PL/SQL is a completely portable, high-performance transaction-processing language.it provides a built-in, interpreted and OS independent programming environment.

It is tightly integrated with SQL and offers extensive error checking, numerous data types, and variety of programming structures. It also supports structured programming through functions and procedures, object-oriented programming, supports the development of web applications and server pages.

# Block Structure

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

**Note**: add the following command on the top of the script **“set serveroutput on”**

set serveroutput on

DECLARE

Sec\_Name varchar2(20) := 'Sec-A';

Course\_Name varchar2(20) := 'Database Systems Lab';

BEGIN

dbms\_output.put\_line('This is : '|| Sec\_Name || ' and the course is ' || Course\_Name);

END;

|  |  |  |  |
| --- | --- | --- | --- |
| **Delimiter** | **Description** | **Delimiter** | **Description** |
| **+, -, \*, /** | Addition, subtraction/ negation, multiplication, division | **:** | Host variable indicator |
| **%** | Attribute indicator | **,** | Item separator |
| **'** | Character string delimiter | **"** | Quoted identifier delimiter |
| **.** | Component selector | **=** | Relational operator |
| **(,)** | Expression or list delimiter | **@** | Remote access indicator |
| **||** | Concatenation operator | **;** | Statement terminator |
| **\*\*** | Exponentiation operator | **:=** | Assignment operator |
| **<<, >>** | Label delimiter (begin and end) | **=>** | Association operator |
| **/\*, \*/** | Multi-line comment delimiter (begin and end) | **<, >, <=, >=** | Relational operators |
| **--** | Single-line comment indicator | **<>, '=, ~=, ^=** | Different versions of NOT EQUAL |
| **..** | Range operator |  |  |

# Variable & types

set serveroutput on

DECLARE

a integer := 10;

b integer := 20;

c integer;

f real;

BEGIN

c := a + b;

dbms\_output.put\_line('Value of c: ' || c);

f := 70.0/3.0;

dbms\_output.put\_line('Value of f: ' || f);

END;

set serveroutput on

DECLARE

a integer := 10;

b integer := 20;

c integer;

f real;

BEGIN

c := a + b;

dbms\_output.put\_line('Value of c: ' || c);

f := 70.0/3.0;

dbms\_output.put\_line('Value of f: ' || f);

END;

DECLARE

-- Global variables

num1 number := 95;

num2 number := 85;

BEGIN

dbms\_output.put\_line('Outer Variable num1: ' || num1);

dbms\_output.put\_line('Outer Variable num2: ' || num2);

DECLARE

-- Local variables

num1 number := 195;

num2 number := 185;

BEGIN

dbms\_output.put\_line('Inner Variable num1: ' || num1);

dbms\_output.put\_line('Inner Variable num2: ' || num2);

END;

END;

DECLARE

e\_id employees.EMPLOYEE\_ID%type;

e\_name employees.FIRST\_NAME%type;

e\_lname employees.LAST\_NAME%type;

d\_name DEPARTMENTS.DEPARTMENT\_NAME%type;

BEGIN

SELECT EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,DEPARTMENT\_NAME

INTO e\_id, e\_name, e\_lname, d\_name

FROM employees inner join DEPARTMENTS

on employees.DEPARTMENT\_ID = DEPARTMENTS.DEPARTMENT\_ID and EMPLOYEE\_ID =100 ;

dbms\_output.put\_line('EMPLOYEE ID: ' ||e\_id);

dbms\_output.put\_line('EMPLOYEE First Name: ' ||e\_name);

dbms\_output.put\_line('EMPLOYEE Last Name: ' ||e\_lname);

dbms\_output.put\_line('DEPARTMENT Name: ' ||d\_name);

END;

# Conditional Logic

1. [IF - THEN statement](https://www.tutorialspoint.com/plsql/plsql_if_then.htm)
2. [IF-THEN-ELSE statement](https://www.tutorialspoint.com/plsql/plsql_if_then_else.htm)
3. [IF-THEN-ELSIF statement](https://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm)
4. [Case statement](https://www.tutorialspoint.com/plsql/plsql_case_statement.htm)
5. [Searched CASE statement](https://www.tutorialspoint.com/plsql/plsql_searched_case.htm)
6. Nested IF-THEN-ELSE

DECLARE

e\_id employees.EMPLOYEE\_ID%type := 100;

e\_sal employees.SALARY%type;

BEGIN

SELECT salary INTO e\_sal FROM employees WHERE EMPLOYEE\_ID = e\_id;

IF (e\_sal >=5000)

THEN

UPDATE employees SET salary = e\_sal+1000 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated');

END IF;

END;

Declare

n\_count number;

e\_id employees.EMPLOYEE\_ID%type := 1100;

Begin

Select count(1) into n\_count from employees Where EMPLOYEE\_ID = e\_id;

if n\_count > 0 then

dbms\_output.put\_line('record already exists.');

else

INSERT INTO employees

(employee\_id,first\_name,last\_name,email,phone\_number,hire\_date,job\_id,salary,commission\_pct,manager\_id,department\_id)

VALUES (e\_id,'Bruce','Austin','DAUSTIN7','590.423.4569','25-JUN-05','IT\_PROG',6000,0.2,100,60);

dbms\_output.put\_line('record inserted with Employee ID: ' ||e\_id);

end if;

End;

DECLARE

e\_id employees.EMPLOYEE\_ID%type := 100;

e\_sal employees.SALARY%type;

BEGIN

SELECT salary INTO e\_sal FROM employees WHERE EMPLOYEE\_ID = e\_id;

IF (e\_sal <=25000) THEN

UPDATE employees SET salary = e\_sal+100 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||e\_sal);

ELSIF (e\_sal >=20000) THEN

UPDATE employees SET salary = e\_sal+200 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

ELSIF (e\_sal <=15000) THEN

UPDATE employees SET salary = e\_sal+300 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

ELSE

UPDATE employees SET salary = e\_sal+400 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

END IF;

END;

DECLARE

e\_id employees.EMPLOYEE\_ID%type := 100;

e\_sal employees.SALARY%type;

e\_did employees.DEPARTMENT\_ID%type;

BEGIN

SELECT salary,DEPARTMENT\_ID INTO e\_sal,e\_did FROM employees WHERE EMPLOYEE\_ID = e\_id;

CASE e\_did

when 80 then

UPDATE employees SET salary = e\_sal+100 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||e\_sal);

when 50 then

UPDATE employees SET salary = e\_sal+200 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

when 40 then

UPDATE employees SET salary = e\_sal+300 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

ELSE

dbms\_output.put\_line('No such Record');

END CASE;

END;

DECLARE

e\_id employees.EMPLOYEE\_ID%type := 100;

e\_sal employees.SALARY%type;

e\_did employees.DEPARTMENT\_ID%type;

BEGIN

SELECT salary,DEPARTMENT\_ID INTO e\_sal,e\_did FROM employees WHERE EMPLOYEE\_ID = e\_id;

CASE

when e\_did = 80 then

UPDATE employees SET salary = e\_sal+100 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||e\_sal);

when e\_did = 50 then

UPDATE employees SET salary = e\_sal+200 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

when e\_did = 40 then

UPDATE employees SET salary = e\_sal+300 WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:'||e\_sal);

ELSE

dbms\_output.put\_line('No such Record');

END CASE;

END;

DECLARE

e\_id employees.EMPLOYEE\_ID%type := 100;

e\_sal employees.SALARY%type;

e\_did employees.DEPARTMENT\_ID%type;

e\_com employees.COMMISSION\_PCT%type;

BEGIN

SELECT salary,DEPARTMENT\_ID,COMMISSION\_PCT INTO e\_sal,e\_did,e\_com FROM employees WHERE EMPLOYEE\_ID = e\_id;

IF (e\_did=90) THEN

IF (e\_sal >=20000 AND e\_sal <=250000 ) THEN

UPDATE employees SET salary = (e\_sal+00)\*(1+e\_com) WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||e\_sal);

ELSIF (e\_sal >=15000 AND e\_sal <=20000 ) THEN

UPDATE employees SET salary = (e\_sal+20)\*(1+e\_com) WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||(e\_sal+100)\*(1+e\_com));

END IF;

END IF;

IF (e\_did=40) THEN

IF (e\_sal >=10000 AND e\_sal <=15000 ) THEN

UPDATE employees SET salary = (e\_sal+00)\*(1+e\_com) WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||e\_sal);

ELSIF (e\_sal >=5000 AND e\_sal <=10000 ) THEN

UPDATE employees SET salary = (e\_sal+20)\*(1+e\_com) WHERE EMPLOYEE\_ID= e\_id;

dbms\_output.put\_line ('Salary updated:' ||(e\_sal+100)\*(1+e\_com));

END IF;

END IF;

END;

# Loops

SET SERVEROUTPUT ON;

DECLARE

BEGIN

FOR c IN (SELECT EMPLOYEE\_ID, FIRST\_NAME, SALARY FROM employees

WHERE DEPARTMENT\_ID = 90)

LOOP

DBMS\_OUTPUT.PUT\_LINE (

'Salary for the employee ' || c.FIRST\_NAME || ' is: ' || c.SALARY);

END LOOP;

END;

# Views

View is a virtual table that does not physically exist. Rather, it is created by a query joining one or more tables. A view contains no data itself. A view is simply any SELECT query that has been given a name and saved in the database. For this reason, a view is sometimes called a named query or a stored query.

**Benefits of using Views**

* Commonality of code being used. Since a view is based on one common set of SQL this means that when it is called it's less likely to require parsing.
* Views have long been used to hide the tables that actually contain the data you are querying. Also, views can be used to restrict the columns that a given user has access to.

**Types of views:**

1. **Updateable Views:**

The data dictionary views ALL\_UPDATABLE\_COLUMNS, DBA\_UPDATABLE\_COLUMNS,  
and USER\_UPDATABLE\_COLUMNS indicate which view columns are updatable. View does not hold any data so the impact of the DML operation will be direct on master/base table.

1. **Read-Only Views:**

A view is *read-only* if it is *not* delete-able, updatable, or insert-able. A view can be read-only if it is a view that does not comply with at least one of the rules for delete-able views.

1. **Materialized Views**

Materialized views are schema objects that can be used to summarize, pre compute, replicate, and distribute data. E.g. to construct a data warehouse. A materialized view provides indirect access to table data by storing the results of a query in a separate schema object. Unlike an ordinary view, which does not take up any storage space or contain any data. A materialized view can be stored in the same database as its base table(s) or in a different database. Materialized views stored in the same database as their base tables can improve query performance through query rewrites. Query rewrites are particularly useful in a data warehouse environment. A materialized view log is a schema object that records changes to a master table's data so that a materialized view defined on the master table can be refreshed incrementally.

CREATE or REPLACE VIEW EMP\_Det AS

SELECT DISTINCT EMPLOYEES.EMPLOYEE\_ID, EMPLOYEES.FIRST\_NAME, EMPLOYEES.EMAIL,DEPARTMENTS.DEPARTMENT\_NAME FROM EMPLOYEES INNER JOIN DEPARTMENTS

ON EMPLOYEES.EMPLOYEE\_ID = DEPARTMENTS.DEPARTMENT\_ID

WHERE EMPLOYEES.DEPARTMENT\_ID = 80;

select \* from emp\_det;

select \* from employees;

update emp\_det set FIRST\_NAME='Ali' where EMPLOYEE\_ID=170;

delete from emp\_det where EMPLOYEE\_ID=170;

create or replace view x as

select \* from employees /\* your query \*/

with read only;

select \* from x;

update x set salary = 100 where employee\_id =100;

CREATE MATERIALIZED VIEW MAT\_EMP\_Det

AS

SELECT DISTINCT EMPLOYEES.EMPLOYEE\_ID, EMPLOYEES.FIRST\_NAME, EMPLOYEES.EMAIL,DEPARTMENTS.DEPARTMENT\_NAME FROM EMPLOYEES INNER JOIN DEPARTMENTS

ON EMPLOYEES.EMPLOYEE\_ID = DEPARTMENTS.DEPARTMENT\_ID

WHERE EMPLOYEES.DEPARTMENT\_ID = 80;

update emp\_det set FIRST\_NAME='Fast' where EMPLOYEE\_ID=150;

select \* from employees where EMPLOYEE\_ID=150;

select \* from MAT\_EMP\_Det where EMPLOYEE\_ID=150;

# Functions

A stored function (also called a user function or user-defined function) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

1. Scalar Value functions
2. Inline table valued functions
3. Multi statement table valued functions

CREATE or replace FUNCTION CalculateSAL(DEPT\_ID in NUmber)

RETURN NUMBER

IS

Total\_Salary NUmber;

BEGIN

Select sum(Salary) into Total\_Salary from employees where DEPARTMENT\_ID= 80;

RETURN(Total\_Salary);

END;

select CalculateSAL(80) from dual;

CREATE or replace FUNCTION CalculateTOTALSAL

RETURN NUMBER

IS

Total\_Salary NUmber;

BEGIN

Select sum(Salary) into Total\_Salary from employees;

RETURN(Total\_Salary);

END;

select CalculateTOTALSAL from dual;

CREATE or replace TYPE EMP\_OBJ\_TYPE as OBJECT (

EMPLOYEE\_ID NUMBER(6,0),

FIRST\_NAME VARCHAR(30),

LAST\_NAME VARCHAR(30),

DEPARTMENT\_ID NUMBER(4,0)

);

CREATE TYPE EMP\_TBL\_TYPE as TABLE OF EMP\_OBJ\_TYPE;

CREATE OR REPLACE FUNCTION GETALL

RETURN EMP\_TBL\_TYPE

IS

EMPLOYEE\_ID NUMBER(6,0);

FIRST\_NAME VARCHAR(30);

LAST\_NAME VARCHAR(30);

DEPARTMENT\_ID NUMBER(4,0);

-- NESTED TABLE VARIALE DECLARATION AND INITIALIZATION

EMP\_DETAILS EMP\_TBL\_TYPE := EMP\_TBL\_TYPE();

BEGIN

-- EXTENDING THE NESTED TABLE

EMP\_DETAILS.EXTEND();

---- GET THE REQUIRED DATA INTO VARIABLES

SELECT EMPLOYEE\_ID,FIRST\_NAME, LAST\_NAME,DEPARTMENT\_ID INTO EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,DEPARTMENT\_ID FROM EMPLOYEES where EMPLOYEE\_ID=100;

-- USING A OBJECT CONSTRUCTOR, TO INSERT THE DATA INTO THE NESTED TABLE

EMP\_DETAILS(1) := EMP\_OBJ\_TYPE(EMPLOYEE\_ID,FIRST\_NAME,LAST\_NAME,DEPARTMENT\_ID);

RETURN EMP\_DETAILS;

END;

/

SELECT \* FROM TABLE(GETALL);

SELECT \* FROM TABLE(GETALL);

CREATE OR REPLACE FUNCTION GETALL1

RETURN EMP\_TBL\_TYPE

IS

EMPLOYEE\_ID NUMBER(6,0);

FIRST\_NAME VARCHAR(30);

LAST\_NAME VARCHAR(30);

DEPARTMENT\_ID NUMBER(4,0);

-- NESTED TABLE VARIALE DECLARATION AND INITIALIZATION

EMP\_DETAILS EMP\_TBL\_TYPE := EMP\_TBL\_TYPE();

BEGIN

-- EXTENDING THE NESTED TABLE

EMP\_DETAILS.EXTEND();

---- GET THE REQUIRED DATA INTO VARIABLES

SELECT EMP\_OBJ\_TYPE( EMPLOYEE\_ID,FIRST\_NAME, LAST\_NAME,DEPARTMENT\_ID) bulk collect INTO EMP\_DETAILS FROM EMPLOYEES;

-- USING A OBJECT CONSTRUCTOR, TO INSERT THE DATA INTO THE NESTED TABLE

RETURN EMP\_DETAILS;

END;

/

SELECT \* FROM TABLE(GETALL1);

# Stored Procedures

A stored procedure is a PL/SQL block which performs a specific task or a set of tasks. A procedure has a name, contains SQL queries and is able to receive parameters and return results. A procedure Is similar to functions (or methods) in programming languages.

**Benefits of stored procedure**

**Reusability**: Create a procedure once and use it any number of times at any number of places. You just need to call it and your task is done.

**Easy maintenance:** If instead of using a procedure, you repeat the SQL everywhere and if there is a change in logic, then you need to update it at all the places.  
 With stored procedure, the change needs to be done at only one place.

SET SERVEROUTPUT ON;

CREATE OR REPLACE PROCEDURE Insert\_Data(STREET\_ADDRESS IN VARCHAR,POSTAL\_CODE IN VARCHAR Default 'NULL', CITY VARCHAR, STATE\_PROVINCE VARCHAR,COUNTRY\_ID CHAR)

IS

Total\_record INT;

LOCATION\_ID Number;

BEGIN

SELECT count(LOCATION\_ID) into LOCATION\_ID from LOCATIONS;

LOCATION\_ID :=LOCATION\_ID+1;

Total\_record :=LOCATION\_ID;

INSERT INTO LOCATIONS(LOCATION\_ID,STREET\_ADDRESS,POSTAL\_CODE,CITY,STATE\_PROVINCE) VALUES(LOCATION\_ID,STREET\_ADDRESS,POSTAL\_CODE,CITY,STATE\_PROVINCE);

dbms\_output.put\_line('NEW RECORD INSERTED WITH ID : ' || LOCATION\_ID);

dbms\_output.put\_line('TOTAL NO OF RECORDS : ' || Total\_record);

END;

exec Insert\_Data('DHA','1234','KARACHI','SINDH','PK');

# Cursors

A cursor is a pointer that points to a result of a query. PL/SQL has two types of cursors: implicit cursors and explicit cursors.

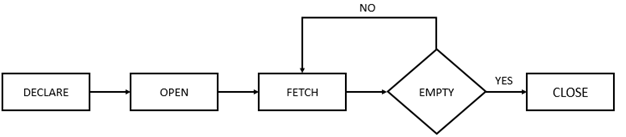
**Implicit cursors**

Whenever Oracle executes an SQL statement such as SELECT INTO, INSERT, UPDATE, and DELETE, it automatically creates an implicit cursor Oracle internally manages the whole execution cycle of implicit cursors and reveals only the cursor’s information and statuses such as SQL%ROWCOUNT, SQL%ISOPEN, SQL%FOUND, and SQL%NOTFOUND.

**Explicit cursors**

An explicit cursor is an SELECT statement declared explicitly in the declaration section of the current block or a package specification. For an explicit cursor, you have control over its execution cycle from OPEN, FETCH, and CLOSE.

The following illustration shows the execution cycle of an explicit cursor:



SET SERVEROUTPUT ON;

DECLARE

CURSOR Cursor\_EMP IS

SELECT \* FROM employees ORDER BY salary DESC;

-- record

row\_emp Cursor\_EMP%ROWTYPE;

BEGIN

OPEN Cursor\_EMP;

-- LOOP

FETCH Cursor\_EMP INTO row\_emp;

--EXIT WHEN Cursor\_EMP%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE( 'EMPLOYEE id: ' ||row\_emp.EMPLOYEE\_ID || ' EMPLOYEE NAME: ' || row\_emp.FIRST\_NAME || ' EMPLOYEE CONTACT: ' || row\_emp.PHONE\_NUMBER || '.');

-- END LOOP;

CLOSE Cursor\_EMP;

END;

SET SERVEROUTPUT ON;

DECLARE

CURSOR Cursor\_EMP IS

SELECT \* FROM employees ORDER BY salary DESC;

-- record

row\_emp Cursor\_EMP%ROWTYPE;

BEGIN

OPEN Cursor\_EMP;

LOOP

FETCH Cursor\_EMP INTO row\_emp;

EXIT WHEN Cursor\_EMP%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE( 'EMPLOYEE id: ' ||row\_emp.EMPLOYEE\_ID || ' EMPLOYEE NAME: ' || row\_emp.FIRST\_NAME || ' EMPLOYEE CONTACT: ' || row\_emp.PHONE\_NUMBER || '.');

END LOOP;

CLOSE Cursor\_EMP;

END;

**LAB Tasks**

1. Write a PL/SQL block to update the salary of the employee with a 10% increase whose empno is to be passed as an argument for the procedure
2. Write a procedure to add an amount of Rs.1000 for the employees whose salaries is greater than 5000 and who belongs to the deptno passed as an argument.
3. Create views for following purposes: -
   1. Display each designation and number of employees with that particular designation.
   2. The organization wants to display only the details like empno, empname , deptno , deptname of all the employee except king.
   3. The organization wants to display only the details empno, empname, deptno, deptname of the employees.
4. Write a PL/SQL code that takes two inputs from user, add them and store the sum in new variable and show the output.
5. Write a PL/SQL code that takes two inputs, lower boundary and upper boundary, then print the sum of all the numbers between the boundaries INCLUSIVE.
6. Write a PL/SQL code to retrieve the employee name, hiredate, and the department name in which he works, whose number is input by the user.
7. Write a PL/SQL code to check whether the given number is palindrome or not.
8. Write a PL/SQL code that takes all the required inputs from the user for the Employee table and then insert it into the Employee and Department table in the database.
9. Write a PL/SQL code to find the first employee who has a salary over $2500 and is higher in the chain of command than employee 90. Note: For chain, use of LOOP is necessary.
10. Write a PL/SQL code to print the sum of first 100 numbers.