

UNIT 2

PL/SQL

(13 Hours) — Database Programming (CACS453)

1. PL/SQL Concepts

PL/SQL (Procedural Language extensions to SQL) is Oracle's procedural programming language, designed as an extension to standard SQL. While SQL is a declarative, set-based language used to manipulate data, it lacks procedural constructs such as variables, conditional branching, and loops. PL/SQL fills this gap by combining the data-manipulation power of SQL with the procedural capabilities of a conventional programming language.

1.1 Why PL/SQL?

- **Tight SQL Integration** — SQL statements can be used directly inside PL/SQL code without special APIs or drivers.
- **Improved Performance** — an entire block of statements is sent to the server in a single call, reducing network round-trips compared to statement-by-statement execution.
- **Procedural Capability** — supports variables, conditional logic (IF, CASE), loops (FOR, WHILE), and error handling (exceptions).
- **Modularity & Reusability** — code can be organized into reusable procedures, functions, and packages.
- **Portability** — PL/SQL code runs on any platform where Oracle Database is available, without modification.

1.2 SQL vs PL/SQL

Aspect	SQL	PL/SQL
Nature	Declarative — states what to do	Procedural — states how to do it
Execution	Statement executed individually	Block of statements executed as a unit
Control Structures	None	IF, CASE, LOOP, WHILE, FOR
Error Handling	Limited	Structured exception handling
Can embed the other?	Cannot embed PL/SQL	Can embed SQL statements

2. PL/SQL Architecture

PL/SQL code is not interpreted by the SQL engine directly. Instead, it passes through a distinct processing pipeline made up of two co-operating engines:

- **PL/SQL Engine** — compiles and executes the procedural parts of the block (variable assignments, loops, conditionals). It normally resides within the Oracle server, though a version also exists inside client tools such as SQL*Plus and Oracle Forms.
- **SQL Statement Executor** — handles every SQL statement encountered inside the block, sending it to the SQL engine for execution against the database.

When a PL/SQL block is submitted, the PL/SQL engine separates SQL statements from procedural statements. Procedural statements are executed directly by the PL/SQL engine, while SQL statements are handed off to the SQL statement executor, which processes them using the standard SQL engine and returns the results back to the PL/SQL engine.

Note: Because the whole block — procedural logic and embedded SQL — is sent to the server together, PL/SQL reduces the number of network round-trips compared to issuing each SQL statement separately from a client application.

3. PL/SQL Block Structure

Every unit of PL/SQL code is organized as a block, made up of up to four sections. Only the BEGIN...END executable section is mandatory; the others are optional.

Section	Keyword	Purpose	Required?
Header	(none, for anonymous blocks)	Names the block if it is a procedure/function	No
Declarative	DECLARE	Declares variables, constants, cursors, exceptions	No
Executable	BEGIN ... END	Contains the procedural and SQL statements	Yes
Exception-Handling	EXCEPTION	Handles runtime errors raised in the executable section	No

General Block Structure

```
DECLARE
    -- variable, constant, and cursor declarations
BEGIN
    -- executable statements (SQL and procedural)
EXCEPTION
    -- exception-handling statements
END;
/
```

Simple Anonymous Block — Example

```
DECLARE
    v_name VARCHAR2(30) := 'Anita';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, ' || v_name || '!');
END;
/
```

*Note: The forward slash (/) on its own line tells SQL*Plus to execute the block; it is a SQL*Plus command, not part of the PL/SQL syntax itself.*

Blocks may also be nested — an inner block can appear within the executable or exception section of an outer block, each with its own DECLARE and EXCEPTION sections.

4. Executing PL/SQL Script

PL/SQL blocks can be typed directly at the SQL*Plus prompt or saved to a script file and executed with START or @, exactly as with SQL scripts.

Running a Saved PL/SQL Script

```
SQL> @greet.sql

-- or

SQL> START greet.sql
```

Before output such as DBMS_OUTPUT.PUT_LINE can be seen in SQL*Plus, output must be enabled for the session:

```
SQL> SET SERVEROUTPUT ON
```

Note: If SET SERVEROUTPUT ON is omitted, the block still executes successfully, but any text sent via DBMS_OUTPUT.PUT_LINE will not be displayed on screen.

5. DBMS_OUTPUT.PUT_LINE Statement

DBMS_OUTPUT is a built-in Oracle package used to display output from a PL/SQL block — primarily useful for debugging and demonstration, since PL/SQL itself has no console output of its own. The PUT_LINE procedure writes a line of text to a buffer, which SQL*Plus then displays once SERVEROUTPUT is enabled.

Syntax & Example

```
SET SERVEROUTPUT ON
```

```
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Total salary is: ' || 55000);
```

```
END;
```

```
/
```

Output

```
Total salary is: 55000
```

Related Procedure	Purpose
DBMS_OUTPUT.PUT(text)	Writes text to the buffer without a trailing newline
DBMS_OUTPUT.PUT_LINE(text)	Writes text to the buffer followed by a newline
DBMS_OUTPUT.NEW_LINE	Writes a newline to the buffer

6. Substitution Variable Feature

As with plain SQL scripts, PL/SQL blocks run in SQL*Plus can use substitution variables (&) so that a block prompts for a value each time it runs, making the same script reusable with different inputs.

Example — Prompting Inside a PL/SQL Block

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
    v_salary NUMBER := &salary_input;
```

```
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Annual salary: ' || (v_salary * 12));
```

```
END;
```

```
/
```

When executed, SQL*Plus prompts Enter value for salary_input: and substitutes the typed value into the DECLARE section before the block is compiled and run.

*Note: SET VERIFY OFF can be added at the top of a script to suppress the 'old value / new value' echo that SQL*Plus normally prints after a substitution.*

7. PL/SQL Language Fundamentals

7.1 Character Set, Identifiers & Comments

- Identifiers (variable, constant, and block names) may contain letters, digits, underscore (_), dollar sign (\$), and hash (#), and must begin with a letter.
- PL/SQL is not case-sensitive for identifiers and keywords.
- Single-line comments begin with --; multi-line comments are enclosed in /* ... */.

7.2 Variables and Constants

Variables are declared in the DECLARE section, specifying a name and a datatype, and optionally an initial value using the assignment operator :=. Constants are declared similarly but with the CONSTANT keyword, and must be initialized at declaration.

Declaring Variables and Constants

```
DECLARE
  v_emp_name  VARCHAR2(30);
  v_salary    NUMBER(8,2) := 50000;
  v_hire_date DATE := SYSDATE;
  c_tax_rate  CONSTANT NUMBER := 0.13;
BEGIN
  v_emp_name := 'Ramesh Shrestha';
  ...
END;
/
```

7.3 Common Datatypes

Datatype	Description
VARCHAR2(n)	Variable-length character string, up to n characters
NUMBER(p,s)	Numeric value with precision p and scale s
DATE	Date and time value
BOOLEAN	TRUE, FALSE, or NULL — usable only within PL/SQL, not in SQL statements
%TYPE	Anchors a variable's datatype to a column or another variable's type
%ROWTYPE	Declares a record whose structure matches an entire table row

%TYPE and %ROWTYPE — Example

```
DECLARE
  v_salary employees.salary%TYPE;
  v_emp_row employees%ROWTYPE;
BEGIN
  SELECT salary INTO v_salary FROM employees WHERE emp_id = 101;
  SELECT * INTO v_emp_row FROM employees WHERE emp_id = 101;
END;
/
```

Note: %TYPE and %ROWTYPE let a variable automatically track a column's datatype, so the PL/SQL code keeps working even if the underlying table's column definitions change.

7.4 Operators

Category	Operators
Arithmetic	+ - * / ** (exponentiation)
Relational	= <> or != < > <= >=
Logical	AND OR NOT
Concatenation	(joins two strings)
Range / Membership	BETWEEN ... AND ... IN (...)
Pattern Matching / Null Test	LIKE IS NULL IS NOT NULL

8. DML Statements in PL/SQL

INSERT, UPDATE, and DELETE statements can be embedded directly inside a PL/SQL block, just as in ordinary SQL, allowing data manipulation to be combined with procedural logic such as loops and conditionals. A SELECT statement inside PL/SQL must use the INTO clause to store its result into variables, since PL/SQL has no built-in way to display a result set directly.

SELECT ... INTO — Example

DECLARE

v_salary NUMBER;

BEGIN

SELECT salary INTO v_salary

FROM employees WHERE emp_id = 101;

DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);

END;

/

INSERT — Example

BEGIN

INSERT INTO employees (emp_id, emp_name, hire_date, salary)

VALUES (102, 'Sita Gurung', SYSDATE, 48000);

END;

/

UPDATE & DELETE — Example

```
BEGIN
  UPDATE employees SET salary = salary * 1.05
  WHERE department_id = 20;
  DELETE FROM employees WHERE emp_id = 999;
END;
/
```

Note: A SELECT INTO that returns no rows raises the NO_DATA_FOUND exception, and one that returns more than one row raises TOO_MANY_ROWS — both should normally be handled in the EXCEPTION section.

9. Transaction Control in PL/SQL

Transaction control statements manage the permanence of DML changes made within a PL/SQL block. They behave the same way inside PL/SQL as they do in plain SQL.

Statement	Purpose
COMMIT	Makes all changes since the last COMMIT/ROLLBACK permanent
ROLLBACK	Undoes all changes since the last COMMIT
SAVEPOINT name	Marks a point within a transaction to which a partial rollback can occur
ROLLBACK TO SAVEPOINT name	Undoes changes made after the named savepoint only

SAVEPOINT — Example

```
BEGIN
  UPDATE employees SET salary = salary + 1000 WHERE emp_id = 101;
  SAVEPOINT after_raise;
  DELETE FROM employees WHERE emp_id = 102;
  ROLLBACK TO SAVEPOINT after_raise; -- undoes only the DELETE
  COMMIT;
END;
/
```

10. Conditional Control

PL/SQL provides IF and CASE constructs to alter the flow of execution based on conditions evaluated at runtime.

10.1 Simple IF

IF ... THEN ... END IF — Example

```
IF v_salary > 50000 THEN
    DBMS_OUTPUT.PUT_LINE('High earner');
END IF;
```

10.2 IF...THEN...ELSE

IF ... THEN ... ELSE ... END IF — Example

```
IF v_salary > 50000 THEN
    DBMS_OUTPUT.PUT_LINE('High earner');
ELSE
    DBMS_OUTPUT.PUT_LINE('Standard earner');
END IF;
```

10.3 IF...THEN...ELSIF (Nested Condition Chain)

IF ... ELSIF ... ELSE ... END IF — Example

```
IF v_salary >= 80000 THEN
    DBMS_OUTPUT.PUT_LINE('Senior band');
ELSIF v_salary >= 50000 THEN
    DBMS_OUTPUT.PUT_LINE('Mid band');
ELSE
    DBMS_OUTPUT.PUT_LINE('Junior band');
END IF;
```

10.4 Nested IF

An IF statement may itself contain another IF statement within its THEN or ELSE branch, allowing multiple conditions to be evaluated in sequence.

Nested IF — Example

```
IF v_department = 'IT' THEN
  IF v_salary > 60000 THEN
    DBMS_OUTPUT.PUT_LINE('Senior IT staff');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Junior IT staff');
  END IF;
ELSE
  DBMS_OUTPUT.PUT_LINE('Non-IT staff');
END IF;
```

10.5 CASE Statement

The CASE statement selects one of several alternatives based on the value of an expression. PL/SQL supports both the simple CASE (comparing an expression to fixed values) and the searched CASE (evaluating independent boolean conditions).

Simple CASE — Example

```
CASE v_grade
  WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
  WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Good');
  WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Average');
  ELSE DBMS_OUTPUT.PUT_LINE('Needs Improvement');
END CASE;
```

Searched CASE — Example

```
CASE
  WHEN v_salary >= 80000 THEN DBMS_OUTPUT.PUT_LINE('Senior band');
  WHEN v_salary >= 50000 THEN DBMS_OUTPUT.PUT_LINE('Mid band');
  ELSE DBMS_OUTPUT.PUT_LINE('Junior band');
END CASE;
```

Note: A CASE expression (as opposed to a CASE statement) can also be used inline to return a value directly, e.g. within a SELECT list or a variable assignment.

11. Repetitive Control (Loops)

PL/SQL provides several looping constructs to repeat a sequence of statements — the basic LOOP, the WHILE loop, and the FOR loop — along with EXIT/CONTINUE statements and loop labels for controlling nested loops.

11.1 Simple LOOP

The basic LOOP repeats indefinitely until an explicit EXIT statement (usually conditional, via EXIT WHEN) is encountered.

Simple LOOP — Example

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 5;
  END LOOP;
END;
/
```

11.2 WHILE Loop

The WHILE loop evaluates a condition before each iteration and continues only while that condition remains TRUE.

WHILE Loop — Example

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter <= 5 LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

11.3 FOR Loop

The FOR loop iterates automatically over a fixed range, implicitly declaring and incrementing (or decrementing, with REVERSE) the loop counter — there is no need to declare the counter variable beforehand.

FOR Loop — Example

```
BEGIN
  FOR v_counter IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
  END LOOP;
END;
/
```

FOR Loop with REVERSE — Example

```
BEGIN
  FOR v_counter IN REVERSE 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
  END LOOP;
END;
/
```

Note: The FOR loop counter is implicitly declared, exists only within the loop, and cannot be assigned a value inside the loop body.

11.4 Nested Loops and Loop Labels

Loops can be nested inside one another. When loops are nested, a loop label (written as <<label_name>> before the LOOP keyword) allows an EXIT or CONTINUE statement in an inner loop to refer to — and control — an outer loop directly.

Nested Loop with Label — Example

```
BEGIN
  <<outer_loop>>
  FOR i IN 1..3 LOOP
    FOR j IN 1..3 LOOP
      IF j = 2 THEN
        CONTINUE outer_loop;
      END IF;
      DBMS_OUTPUT.PUT_LINE('i=' || i || ' j=' || j);
    END LOOP;
  END LOOP outer_loop;
END;
/
```

11.5 CONTINUE Statement

CONTINUE skips the remaining statements in the current iteration and moves directly to the next iteration of the loop. Like EXIT, it can be used unconditionally, with WHEN, or combined with a loop label to affect an outer loop from within a nested one.

CONTINUE WHEN — Example

```
BEGIN
  FOR v_counter IN 1..10 LOOP
    CONTINUE WHEN MOD(v_counter, 2) = 0; -- skip even numbers
    DBMS_OUTPUT.PUT_LINE('Odd number: ' || v_counter);
  END LOOP;
END;
/
```

Statement	Effect
EXIT	Terminates the loop immediately (unconditionally)
EXIT WHEN condition	Terminates the loop only when the condition is TRUE
CONTINUE	Skips to the next iteration immediately (unconditionally)
CONTINUE WHEN condition	Skips to the next iteration only when the condition is TRUE
EXIT / CONTINUE label	Refers to a specific (usually outer) labelled loop in a nested structure

Key Takeaways

- ✓ PL/SQL extends SQL with variables, conditional logic, loops, and exception handling, and sends whole blocks to the server to reduce round-trips.
- ✓ The PL/SQL engine executes procedural statements; embedded SQL is delegated to the SQL statement executor.
- ✓ Every block follows DECLARE / BEGIN / EXCEPTION / END, with only BEGIN...END mandatory.
- ✓ SET SERVEROUTPUT ON is required before DBMS_OUTPUT.PUT_LINE text becomes visible in SQL*Plus.
- ✓ SELECT statements inside PL/SQL require an INTO clause; DML and transaction control (COMMIT, ROLLBACK, SAVEPOINT) work as in plain SQL.
- ✓ Conditional control is provided by IF / ELSIF / nested IF and simple/searched CASE statements.
- ✓ Repetitive control includes the simple LOOP, WHILE loop, and FOR loop, with EXIT, CONTINUE, and loop labels for controlling nested loops precisely.

— *End of Unit 2* —