

UNIT 6

Records and Procedures

(5 Hours) — Database Programming with PL/SQL

PL/SQL extends standard SQL with procedural constructs such as variables, control structures, and reusable program units. This unit covers two of its most important building blocks: records, which let related data items be grouped and handled as a single logical unit, and procedures, which let a sequence of PL/SQL statements be packaged, named, and reused with parameters.

6.1 Records

A record is a composite data structure in PL/SQL that groups together related data items (fields), possibly of different data types, under a single name. A record behaves like a single variable but internally holds several values that can each be accessed using dot notation (`record_name.field_name`). Records are similar to a "row" of a table, or to a structure ("struct") in languages like C.

- A record must be declared before it can be used, either in the DECLARE section of a block or as a stand-alone TYPE.
- Each field of a record has its own data type, which may itself be scalar (NUMBER, VARCHAR2, DATE, etc.) or another record.
- An entire record can be assigned to another record of the same type in a single statement.
- Individual fields are accessed and modified using the dot (.) operator.

6.1.1 Record Types

PL/SQL supports three ways of declaring a record type:

Record Type	Description	Example
%ROWTYPE record	Declared against a table or cursor; automatically inherits the same number, names, and data types of columns as the table/cursor.	<code>emp_rec employees%ROWTYPE;</code>
User-defined record	Declared with a programmer-defined TYPE ... IS RECORD, listing custom fields and data types.	<code>TYPE emp_rec_type IS RECORD (...);</code>
Cursor-based record	Declared using %ROWTYPE against an explicit cursor; matches the columns selected by that cursor.	<code>rec emp_cursor%ROWTYPE;</code>

Example 6.1 — %ROWTYPE Record

Declare a record that automatically matches the structure of the EMPLOYEES table, then fetch a row into it and print two of its fields.

Syntax / Code

```
DECLARE
    emp_rec    employees%ROWTYPE;
BEGIN
    SELECT * INTO emp_rec
    FROM employees
    WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE('Name : ' || emp_rec.first_name);
    DBMS_OUTPUT.PUT_LINE('Salary : ' || emp_rec.salary);
END;
/
```

Explanation

Because emp_rec is declared as employees%ROWTYPE, it automatically has one field for every column of the EMPLOYEES table — there is no need to list each column individually. The SELECT * INTO statement fills every field in a single step, and fields are then read using emp_rec.first_name and emp_rec.salary.

Example 6.2 — User-Defined Record Type

Declare a custom record type with hand-picked fields, rather than mirroring an entire table.

Syntax

```
DECLARE
    TYPE emp_rec_type IS RECORD (
        emp_id      employees.employee_id%TYPE,
        emp_name     employees.first_name%TYPE,
        emp_salary   employees.salary%TYPE DEFAULT 0
    );
    emp_record    emp_rec_type;
BEGIN
    emp_record.emp_id      := 101;
    emp_record.emp_name    := 'Rita Sharma';
    emp_record.emp_salary := 45000;
    DBMS_OUTPUT.PUT_LINE(emp_record.emp_name || ' earns ' ||
                          emp_record.emp_salary);
END;
/
```

Note: %TYPE anchors a field's data type to that of an existing column, so the record automatically stays in sync if the column's definition changes later.

6.1.2 Nested Records

A field of a record can itself be another record — this is called a nested record. Nested records are useful for modelling hierarchical, real-world data such as an employee record that contains an embedded address record, since the inner structure can be reused and manipulated independently.

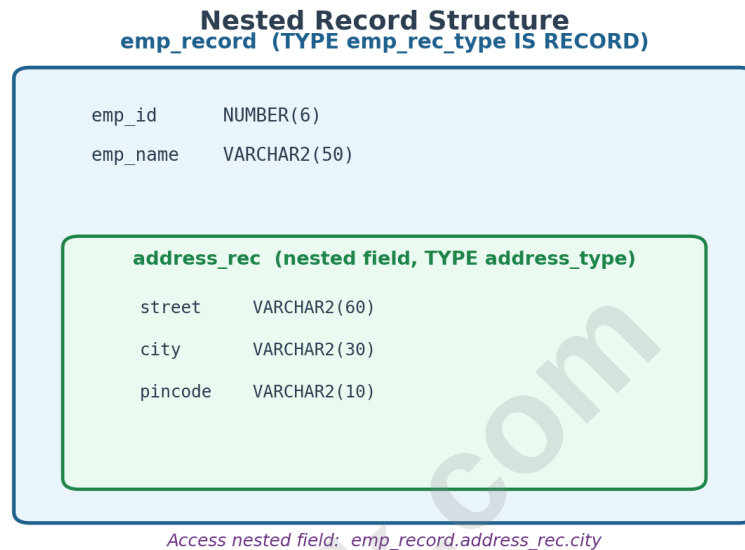


Fig 6.1 — A record with a nested record field (address_rec)

Example 6.3 — Declaring and Using a Nested Record

Create an address_type record, embed it as a field inside emp_rec_type, and access the nested field with two dots.

Syntax

```
DECLARE
    TYPE address_type IS RECORD (
        street    VARCHAR2(60),
        city      VARCHAR2(30),
        pincode   VARCHAR2(10)
    );
    TYPE emp_rec_type IS RECORD (
        emp_id     NUMBER(6),
        emp_name   VARCHAR2(50),
        address_rec address_type    -- nested record field
    );
    emp_record    emp_rec_type;
BEGIN
    emp_record.emp_id    := 205;
    emp_record.emp_name := 'Aman Thapa';
    emp_record.address_rec.street := 'Baneshwor Road';
    emp_record.address_rec.city   := 'Kathmandu';
    emp_record.address_rec.pincode := '44600';
    DBMS_OUTPUT.PUT_LINE(emp_record.emp_name || ' lives in ' ||
                          emp_record.address_rec.city);
END;
/
```

Explanation

address_rec is declared with the type address_type, so it carries its own three fields (street, city, pincode) inside emp_record. To reach a field two levels deep, the dot operator is simply chained: emp_record.address_rec.city.

6.2 Procedures

A procedure is a named PL/SQL block that performs a specific action and can be invoked (called) by name from other blocks, procedures, triggers, or applications. Procedures allow logic to be written once, stored in the database, and reused — improving modularity, reducing code duplication, and keeping business logic close to the data it operates on.

6.2.1 PL/SQL Block Structure

Every PL/SQL program — whether an anonymous block, a procedure, or a function — is built from the same underlying block structure, made up of up to three sections plus a mandatory END:

PL/SQL Block Structure

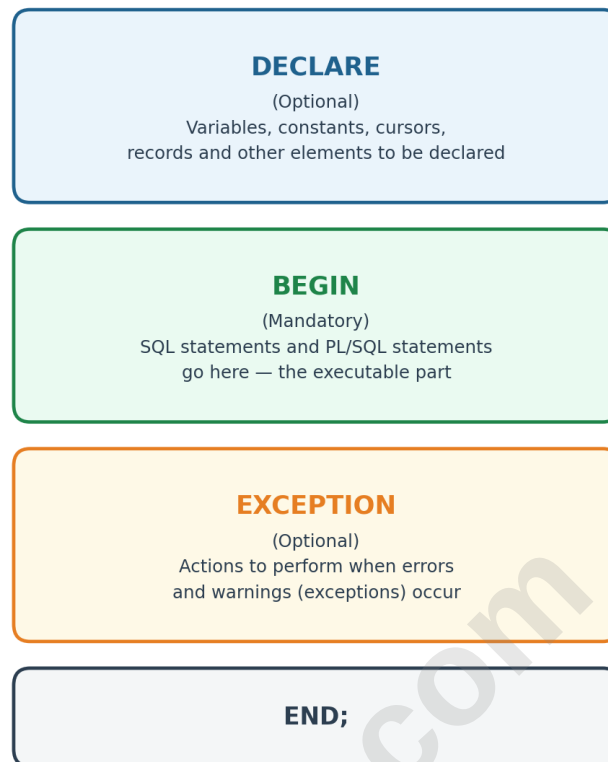


Fig 6.2 — The three sections of a PL/SQL block

Section	Keyword	Required?
Declarative	DECLARE	Optional — omitted entirely in an anonymous block that needs no declarations
Executable	BEGIN ... END	Mandatory — every block must have this section
Exception-handling	EXCEPTION	Optional — included only if errors need to be trapped

6.2.2 Anonymous Blocks

An anonymous block is a PL/SQL block that has no name and is not stored in the database — it is compiled and executed once, immediately, and then discarded. Anonymous blocks are commonly used for quick testing, one-off scripts, and ad-hoc data manipulation. All the DECLARE/BEGIN/END examples shown earlier in this unit are anonymous blocks.

Example 6.4 — A Simple Anonymous Block

An anonymous block has no CREATE PROCEDURE header — it begins directly with an optional DECLARE, or straight away with BEGIN.

Syntax

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('This is an anonymous block.');
```

```
END;
/
```

Note: Because an anonymous block is not stored, it must be re-submitted every time it needs to run. A procedure, by contrast, is compiled once and stored permanently in the database for repeated reuse.

6.2.3 Creating a Procedure

A stored procedure is created using the **CREATE [OR REPLACE] PROCEDURE** statement. Its structure closely follows the general block structure, but adds a header that names the procedure and lists its parameters.

Syntax — Creating a Procedure

```
CREATE OR REPLACE PROCEDURE procedure_name
    (parameter1 IN      datatype,
     parameter2      OUT datatype,
     parameter3 IN OUT datatype)
IS
    -- local variable declarations (optional)
BEGIN
    -- executable statements
EXCEPTION
    -- exception-handling statements (optional)
END procedure_name;
/
```

Example 6.5 — A Procedure with No Parameters

A basic stored procedure that prints a greeting message.

Syntax

```
CREATE OR REPLACE PROCEDURE greet_user
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Welcome to the PL/SQL Procedures unit!');
END greet_user;
/
```

```
-- Calling the procedure:
EXECUTE greet_user;
```

6.2.4 IN, OUT and IN OUT Parameters in Procedures

Parameters let a procedure exchange data with the block that calls it. PL/SQL supports three parameter modes, specified when the procedure is created:

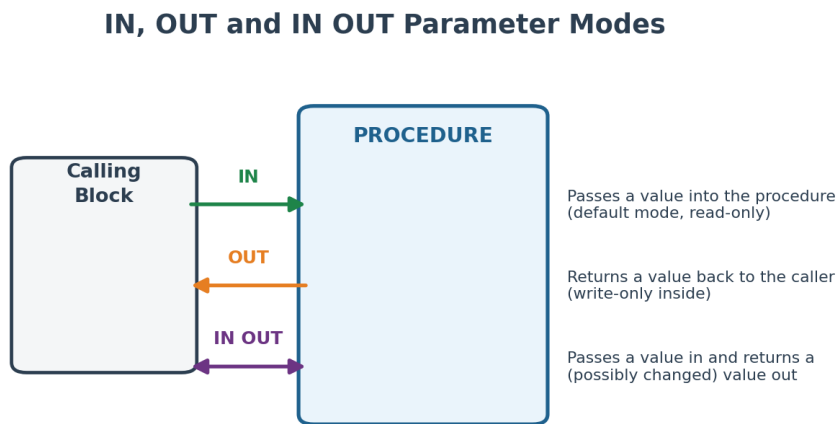


Fig 6.3 — Direction of data flow for IN, OUT and IN OUT parameters

Mode	Direction	Behaviour inside the procedure
IN	Caller → Procedure	Default mode. Passed by reference (read-only); the parameter acts like a constant and cannot be assigned a new value inside the procedure.
OUT	Procedure → Caller	Value is undefined on entry; the procedure must assign it a value, which is then returned to the caller when the procedure completes.
IN OUT	Caller ↔ Procedure	Combines both directions: the procedure receives an initial value and may also modify it; the final value is passed back to the caller.

Example 6.6 — Procedure Using IN and OUT Parameters

A procedure that accepts an employee ID (IN) and returns that employee's salary and a bonus amount (OUT).

Syntax

```
CREATE OR REPLACE PROCEDURE get_salary_details (  
    p_emp_id    IN    employees.employee_id%TYPE,  
    p_salary    OUT employees.salary%TYPE,  
    p_bonus     OUT NUMBER)  
IS  
BEGIN  
    SELECT salary INTO p_salary  
    FROM employees  
    WHERE employee_id = p_emp_id;  
    p_bonus := p_salary * 0.10;  
END get_salary_details;  
/  
-- Calling the procedure from an anonymous block:  
DECLARE  
    v_salary    employees.salary%TYPE;  
    v_bonus     NUMBER;  
BEGIN  
    get_salary_details(100, v_salary, v_bonus);  
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary || ' Bonus: ' || v_bonus)  
;  
END;  
/
```

Explanation

p_emp_id is IN, so the caller supplies 100 and the procedure only reads it. p_salary and p_bonus are OUT, so they start out empty inside the procedure, get assigned values during execution, and those values are copied back into the caller's variables v_salary and v_bonus once the procedure finishes.

Example 6.7 — Procedure Using an IN OUT Parameter

A procedure that receives a number and doubles it in place, using a single IN OUT parameter.

Syntax

```
CREATE OR REPLACE PROCEDURE double_value (p_num IN OUT NUMBER)
IS
BEGIN
    p_num := p_num * 2;
END double_value;
/
DECLARE
    v_num NUMBER := 15;
BEGIN
    double_value(v_num);
    DBMS_OUTPUT.PUT_LINE('Doubled value: ' || v_num);    -- prints 30
END;
/
```

Note: IN is the default mode and may be omitted, but it is good practice to state it explicitly for readability. OUT and IN OUT parameters cannot be given default values.

Key Takeaways

- ✓ A record groups related fields of possibly different data types under one name, accessed with dot notation.
- ✓ Records may be declared with %ROWTYPE (table or cursor-based) or as a programmer-defined TYPE ... IS RECORD.
- ✓ A nested record is a record whose field is itself another record, accessed by chaining the dot operator.
- ✓ Every PL/SQL block follows the same structure: optional DECLARE, mandatory BEGIN...END, optional EXCEPTION.
- ✓ An anonymous block is unnamed and not stored; a procedure is named, compiled once, and stored for reuse.
- ✓ A procedure is created with CREATE [OR REPLACE] PROCEDURE and can take IN, OUT, and IN OUT parameters.
- ✓ IN parameters are read-only inputs; OUT parameters return values to the caller; IN OUT parameters do both.

— End of Unit 6 —