Chatpter 19

Mastering Oracle PL/SQL Procedures

Mayko Freitas da Silva

Introduction

When writing large programs, it's often beneficial to break them down into smaller, manageable pieces. Think of it like building a house - instead of having one contractor do everything, you have specialists for plumbing, electrical work, and carpentry. In PL/SQL, procedures serve as these specialized units, each designed to perform a specific task.

In this chapter, you'll learn how to create and use PL/SQL procedures, which are named blocks of code that can be called repeatedly to perform specific tasks. We'll explore two main types of procedures: those nested within other PL/SQL blocks and those that stand alone as schema objects.

What You'll Learn

In this chapter, you will:

- Understand what procedures are and when to use them
- Learn to create nested procedures within PL/SQL blocks
- Master different parameter modes (IN, OUT, and IN OUT)
- Explore various ways to pass parameters using different notation types
- Discover how to create stand-alone procedures
- Practice error handling in procedures
- Work with real-world examples using the HR schema

Why Procedures Matter

Procedures are fundamental building blocks in PL/SQL that offer several benefits:

- Code reusability
- Easier maintenance
- Better organization
- Improved security through controlled access
- Reduced network traffic in client-server applications

Chapter Overview

We'll cover this material in two main labs:

Lab 19.1: Creating Nested Procedures

- Understanding nested procedures
- Working with parameter modes
- Using forward declarations

Lab 19.2: Creating Stand-Alone Procedures

- Creating schema-level procedures
- Managing procedure execution
- Implementing error handling

Lab 19.1: Creating Nested Procedures

Overview

A nested procedure is a procedure defined within another PL/SQL block, function, or procedure. Like a set of specialized tools kept in a specific toolbox, nested procedures are accessible only within their containing block, making them perfect for tasks that are unique to that block.

Objectives

After completing this lab, you will be able to:

- Create and implement nested procedures
- Use different parameter modes effectively
- Apply forward declaration when needed
- Understand procedure scope and accessibility

Nested Procedures

Let's start with a basic example that demonstrates the structure and scope of a nested procedure:

```
DECLARE
   -- Variables for the main block
   v_emp_count    NUMBER;
   v_dept_name    VARCHAR2(30);

   -- Nested procedure definition
   PROCEDURE count_dept_employees(
      p_dept_id    IN  NUMBER,
      p_dept_name  OUT VARCHAR2,
      p_emp_count  OUT NUMBER)
   IS
   BEGIN
      -- Get department name and employee count
      SELECT d.department_name, COUNT(e.employee_id)
        INTO p_dept_name, p_emp_count
        FROM departments d
        LEFT JOIN employees e ON (d.department_id = e.department_id)
       WHERE d.department_id = p_dept_id
       GROUP BY d.department_name;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         p_dept_name := 'Department not found';
         p_emp_count := 0;
   END count_dept_employees;

BEGIN
   -- Main block execution
   count_dept_employees(60, v_dept_name, v_emp_count);
   DBMS_OUTPUT.PUT_LINE('Department: ' || v_dept_name);
   DBMS_OUTPUT.PUT_LINE('Number of Employees: ' || v_emp_count);
END;
```

Key points about nested procedures:

- They are defined in the declaration section of a PL/SQL block
- Their scope is limited to the containing block
- They can access variables from the containing block
- They can have their own local variables

Parameter Modes

Parameter modes in PL/SQL determine how data is passed to and from a procedure. Think of these modes as different types of communication channels between your procedure and the calling program.

Understanding Parameter Modes

There are three parameter modes in PL/SQL:

- IN (default mode)
- OUT
- IN OUT

Let's explore each mode with practical examples using the HR schema:

```
DECLARE
   -- Variables for our examples
   v_salary       NUMBER;
   v_emp_name     VARCHAR2(50);
   v_commission   NUMBER := 0;

   -- Example 1: IN Parameter Mode
   PROCEDURE validate_employee_salary(
      p_emp_id    IN NUMBER,
      p_job_id    IN VARCHAR2)
   IS
      v_min_salary    jobs.min_salary%TYPE;
      v_max_salary    jobs.max_salary%TYPE;
      v_curr_salary   employees.salary%TYPE;
   BEGIN
      -- Get salary range for job
      SELECT min_salary, max_salary
        INTO v_min_salary, v_max_salary
        FROM jobs
       WHERE job_id = p_job_id;

      -- Get employee's current salary
      SELECT salary
        INTO v_curr_salary
        FROM employees
       WHERE employee_id = p_emp_id;

      IF v_curr_salary BETWEEN v_min_salary AND v_max_salary THEN
         DBMS_OUTPUT.PUT_LINE('Salary is within valid range');
      ELSE
         DBMS_OUTPUT.
```

```
PUT_LINE('Salary is out of range');
        END IF;
    END validate_employee_salary;

    -- Example 2: OUT Parameter Mode
    PROCEDURE get_employee_details(
        p_emp_id    IN  NUMBER,
        p_name      OUT VARCHAR2,
        p_salary    OUT NUMBER)
    IS
    BEGIN
        SELECT first_name || ' ' || last_name,
               salary
          INTO p_name, p_salary
          FROM employees
         WHERE employee_id = p_emp_id;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            p_name   := 'Employee not found';
            p_salary := 0;
    END get_employee_details;

    -- Example 3: IN OUT Parameter Mode
    PROCEDURE calculate_commission(
        p_salary       IN     NUMBER,
        p_commission    IN OUT NUMBER)
    IS
        v_commission_rate   NUMBER := 0.02;  -- 2% commission rate
    BEGIN
        -- Update commission based on salary
        p_commission := p_salary * v_commission_rate;

        -- Apply minimum commission rule
        IF p_commission < 100 THEN
            p_commission := 100;  -- Minimum commission
        END IF;
    END calculate_commission;

BEGIN
    -- Testing IN parameter mode
    validate_employee_salary(103, 'IT_PROG');

    -- Testing OUT parameter mode
    get_employee_details(103, v_emp_name, v_salary);
    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_emp_name);
    DBMS_OUTPUT.
```

```
PUT_LINE('Salary: $' || v_salary);

   -- Testing IN OUT parameter mode
   calculate_commission(v_salary, v_commission);
   DBMS_OUTPUT.PUT_LINE('Commission: $' || v_commission);
END;
```

Key Points About Parameter Modes:

1. IN Parameters:
    - Are read-only within the procedure
    - Cannot be modified by the procedure
    - Accept literals, expressions, or variables as arguments
    - Are passed by reference for large objects, by value for scalar datatypes
2. OUT Parameters:
    - Are undefined when the procedure starts
    - Must be assigned a value within the procedure
    - Cannot be used in expressions until assigned
    - Are always passed by reference
3. IN OUT Parameters:
    - Combine features of IN and OUT modes
    - Must be initialized before procedure call
    - Can be modified by the procedure
    - Are always passed by reference

Forward Declaration

Forward declaration allows you to define procedures that reference each other, solving the "chicken and egg" problem when procedures need to call each other. It's particularly useful in complex business logic scenarios.

Let's explore this concept with practical examples using the HR schema:

```
DECLARE
   -- Forward declarations
   PROCEDURE validate_department(p_dept_id IN NUMBER);
   PROCEDURE update_department_budget(p_dept_id IN NUMBER, p_budget IN NUMBER);
   PROCEDURE check_manager_salary(p_dept_id IN NUMBER);

   -- First procedure implementation
   PROCEDURE validate_department(p_dept_id IN NUMBER)
   IS
      v_dept_exists NUMBER;
      v_budget      NUMBER := 100000; -- Default budget
   BEGIN
      SELECT COUNT(*)
        INTO v_dept_exists
        FROM departments
       WHERE department_id = p_dept_id;

      IF v_dept_exists = 1 THEN
         check_manager_salary(p_dept_id);
         update_department_budget(p_dept_id, v_budget);
      ELSE
         RAISE_APPLICATION_ERROR(-20001, 'Department does not exist');
      END IF;
   END validate_department;

   -- Second procedure implementation
   PROCEDURE update_department_budget(
      p_dept_id IN NUMBER,
      p_budget  IN NUMBER)
   IS
      v_employee_count NUMBER;
      v_total_salary  NUMBER;
   BEGIN
      SELECT COUNT(*), NVL(SUM(salary), 0)
        INTO v_employee_count, v_total_salary
        FROM employees
       WHERE department_id = p_dept_id;

      IF v_total_salary > p_budget THEN
         DBMS_OUTPUT.
```

```
PUT_LINE('Warning: Department ' || p_dept_id ||
                        ' exceeds budget by ' ||
                        (v_total_salary - p_budget));
    END IF;
  END update_department_budget;

  -- Third procedure implementation
  PROCEDURE check_manager_salary(p_dept_id IN NUMBER)
  IS
    v_manager_id    NUMBER;
    v_manager_sal   NUMBER;
    v_avg_emp_sal   NUMBER;
  BEGIN
    -- Get department manager and their salary
    SELECT manager_id
      INTO v_manager_id
      FROM departments
     WHERE department_id = p_dept_id;

    -- Get manager's salary and average employee salary
    SELECT
       (SELECT salary
          FROM employees
         WHERE employee_id = v_manager_id) as mgr_salary,
       AVG(salary) as avg_salary
      INTO v_manager_sal, v_avg_emp_sal
      FROM employees
     WHERE department_id = p_dept_id
       AND employee_id != v_manager_id;

    -- Verify manager salary is appropriate
    IF v_manager_sal < (v_avg_emp_sal * 1.5) THEN
       DBMS_OUTPUT.PUT_LINE('Warning: Manager salary may need review');
    END IF;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
       DBMS_OUTPUT.PUT_LINE('No manager or employees found');
  END check_manager_salary;

BEGIN
  -- Main block execution
  validate_department(60);
END;
```

Key Points About Forward Declaration:

1. Purpose:
    - Enables mutual recursion between procedures
    - Allows procedures to call each other regardless of their definition order
    - Helps organize code logically rather than sequentially
2. Rules:
    - Forward declaration must match the procedure specification exactly
    - Parameters in the declaration must match the implementation
    - Only the procedure specification is declared, not the implementation
    - All forward-declared procedures must be implemented
3. Best Practices:
    - Use forward declaration when procedures need to call each other
    - Group related forward declarations together
    - Maintain consistent parameter names between declaration and implementation
    - Document dependencies between procedures

Lab 19.2: Creating Stand-Alone Procedures

Overview

Stand-alone procedures are schema-level objects stored in the database. Unlike nested procedures, they can be accessed by multiple applications and users (with proper permissions), making them ideal for shared business logic.

Objectives

After completing this lab, you will be able to:

- Create and manage stand-alone procedures
- Implement error handling in procedures
- Use different parameter modes in stand-alone procedures
- Execute and maintain stand-alone procedures

Creating Stand-Alone Procedures

Here's our first example of a stand-alone procedure:

```
CREATE OR REPLACE PROCEDURE update_emp_salary(
   p_emp_id    IN  NUMBER,
   p_percent   IN  NUMBER,
   p_message   OUT VARCHAR2)
IS
   v_old_salary    NUMBER;
   v_new_salary    NUMBER;
   v_max_salary    NUMBER;
   v_min_salary    NUMBER;
BEGIN
   -- Get current and limit salaries
   SELECT e.salary, j.min_salary, j.max_salary
     INTO v_old_salary, v_min_salary, v_max_salary
     FROM employees e
     JOIN jobs j ON (e.job_id = j.job_id)
    WHERE e.employee_id = p_emp_id;

   -- Calculate new salary
   v_new_salary := v_old_salary * (1 + p_percent/100);

   -- Validate new salary against job limits
   IF v_new_salary BETWEEN v_min_salary AND v_max_salary THEN
      UPDATE employees
        SET salary = v_new_salary
       WHERE employee_id = p_emp_id;

      p_message := 'Salary updated successfully from ' ||
                   TO_CHAR(v_old_salary, 'FM999,999.00') || ' to ' ||
                   TO_CHAR(v_new_salary, 'FM999,999.00');
      COMMIT;
   ELSE
      p_message := 'New salary ' || TO_CHAR(v_new_salary, 'FM999,999.00') ||
                   ' is outside job grade limits (' ||
                   TO_CHAR(v_min_salary, 'FM999,999.00') || ' - ' ||
                   TO_CHAR(v_max_salary, 'FM999,999.00') || ')';
   END IF;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      p_message := 'Employee ' || p_emp_id || ' not found';
   WHEN OTHERS THEN
      p_message := 'Error: ' || SUBSTR(SQLERRM, 1, 200);
      ROLLBACK;
END update_emp_salary;
```

After create the procedure, you can call it:

```
DECLARE
   v_message VARCHAR2(200);
   v_avg_salary NUMBER;
   v_max_salary NUMBER;
BEGIN
   update_emp_salary(103, 10, v_message);
   DBMS_OUTPUT.PUT_LINE(v_message);

   -- Get average and maximum salary after update
   SELECT AVG(salary), MAX(salary)
   INTO v_avg_salary, v_max_salary
   FROM employees;

   DBMS_OUTPUT.PUT_LINE('Average of new/updated salaries: ' || TO_CHAR(v_avg_salary, 'FM999,999.00'));
   DBMS_OUTPUT.PUT_LINE('Maximum new/updated salary: ' || TO_CHAR(v_max_salary, 'FM999,999.00'));
END;
```

Let's create another stand-alone procedure that handles employee transfers:

```
CREATE OR REPLACE PROCEDURE transfer_employee
 (p_emp_id IN NUMBER,
 p_new_dept_id IN NUMBER,
 p_new_job_id IN VARCHAR2,
 p_new_salary IN NUMBER,
 p_status OUT VARCHAR2)
IS
 v_old_dept_id NUMBER;
 v_old_job_id VARCHAR2(10);
 v_dept_name VARCHAR2(100); -- Increased size
 v_manager_id NUMBER;
 v_hire_date DATE;
BEGIN
 -- Get current employee details including hire_date
 SELECT department_id, job_id, hire_date
 INTO v_old_dept_id, v_old_job_id, v_hire_date
 FROM employees
 WHERE employee_id = p_emp_id;

 -- Get department details
 SELECT department_name, manager_id
 INTO v_dept_name, v_manager_id
 FROM departments
 WHERE department_id = p_new_dept_id;

 -- Validate manager
 IF v_manager_id = p_emp_id THEN
 RAISE_APPLICATION_ERROR(-20002, 'Employee cannot be transferred to a department they manage');
 END IF;

 -- Update employee record
 -- This will trigger UPDATE_JOB_HISTORY
 UPDATE employees
 SET department_id = p_new_dept_id,
 job_id = p_new_job_id,
 salary = p_new_salary
 WHERE employee_id = p_emp_id;

 p_status := 'Transfer successful to ' || v_dept_name;
 COMMIT;

EXCEPTION
 WHEN NO_DATA_FOUND THEN
 p_status := 'Error: Invalid department or employee ID';
 ROLLBACK;
 WHEN OTHERS THEN
 p_status := SQLERRM;
```

```
ROLLBACK;
END transfer_employee;
```

Now you can call the procedure:

```
DECLARE
   v_status VARCHAR2(200);
BEGIN
   transfer_employee(
      p_emp_id => 193,
      p_new_dept_id => 60,
      p_new_job_id => 'IT_PROG',
      p_new_salary => 6000,
      p_status => v_status
   );
   DBMS_OUTPUT.PUT_LINE(v_status);
END;
```

Summary

What We've Learned

In this chapter, we explored PL/SQL procedures, essential building blocks for creating modular and maintainable code. Let's review the key concepts and their practical applications:

1. Understanding Procedures

We learned that procedures are named PL/SQL blocks that:

- Perform specific tasks
- Can accept parameters
- Can be called multiple times
- Help organize code into manageable units
- Promote code reuse and maintainability

2. Types of Procedures

We covered two main types of procedures:

Nested Procedures

- Defined within another PL/SQL block
- Limited scope to the containing block
- Useful for organizing code within a specific context
- Can access variables from the containing block
- Perfect for tasks specific to one program unit

Stand-alone Procedures

- Stored as schema-level objects
- Available to multiple applications and users
- Can be granted permissions to other users
- Ideal for shared business logic
- Maintain consistency across applications

3. Parameter Modes

We mastered three parameter modes and their specific uses:

IN Parameters

- Default mode if not specified
- Read-only within the procedure
- Accept literals, expressions, or variables
- Best for passing values into procedures
- Cannot be modified inside the procedure

OUT Parameters

- Used to return values from procedures
- Must be assigned values within the procedure
- Undefined when procedure starts
- Perfect for returning multiple values
- Always passed by reference

IN OUT Parameters

- Combination of IN and OUT modes
- Must be initialized before procedure call
- Can be modified within the procedure
- Useful for values that need modification
- Always passed by reference

4. Parameter Notations

We explored different ways to pass parameters:

Positional Notation

```
update_emp_salary(103, 10, v_message);
```

Named Notation

```
update_emp_salary(
   p_emp_id  => 103,
   p_percent => 10,
   p_message => v_message
);
```

Mixed Notation

```
update_emp_salary(103, p_percent => 10, p_message => v_message);
```

5. Forward Declaration

We learned how to:

- Declare procedures before their implementation
- Handle mutual recursion between procedures
- Organize code logically rather than sequentially
- Manage procedure dependencies effectively

6. Best Practices

Throughout the chapter, we covered important best practices:

Error Handling

- Always include exception handling
- Use specific exception handlers when possible
- Provide meaningful error messages
- Implement proper transaction management

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    p_status := 'Error: Record not found';
  WHEN OTHERS THEN
    p_status := 'Error: ' || SQLERRM;
    ROLLBACK;
```

Parameter Naming

- Use consistent naming conventions
- Prefix parameters (p_) to distinguish from variables
- Use meaningful and descriptive names
- Document parameter purpose and usage

Code Organization

- Group related procedures together
- Use proper indentation and formatting
- Include appropriate comments
- Follow modular programming principles

7. Practical Applications

We implemented procedures for common business scenarios:

- Employee salary updates
- Department transfers
- Budget calculations
- Data validation
- Employee record management

8. Key Considerations

Important points to remember:

- Choose appropriate procedure type based on scope
- Consider security implications for stand-alone procedures
- Plan for maintainability and reusability
- Document dependencies and requirements
- Test thoroughly with various scenarios

Moving Forward

With the knowledge gained from this chapter, you can:

- Create efficient and maintainable code
- Implement complex business logic
- Manage database operations effectively
- Build scalable applications
- Ensure code consistency across projects

Remember that procedures are fundamental to PL/SQL development, and mastering them is essential for creating robust database applications. Practice creating both nested and stand-alone procedures, and experiment with different parameter modes to understand their practical applications fully.

Mastering Oracle PL/SQL Procedures