Chapter 21

Packages in Oracle PL/SQL

Mayko Freitas da Silva

Introduction

Imagine you're organizing your workspace. Instead of having all your tools scattered around, you decide to group them into specialized toolboxes. This is exactly what PL/SQL packages do for your code. In this chapter, you'll learn how to create and use these "code toolboxes" to make your database programming more efficient and organized.

In this chapter, you will learn about:

1. Creating Packages
2. Package Instantiation and Initialization
3. SERIALLY_REUSABLE Packages

A package in PL/SQL is like a toolbox that holds related PL/SQL objects such as procedures, functions, variables, and more. By grouping these objects together, you gain several benefits that we'll explore throughout this chapter.

Lab 21.1: Creating Packages

After this lab, you will be able to:

- Create a Package Specification
- Create a Package Body

The Power of Packages

Packages offer numerous advantages in PL/SQL programming:

1. **Modularity**: They help you organize related code, making it easier to manage and maintain.
2. **Encapsulation**: You can hide implementation details, allowing you to change the internals without affecting other parts of your application.
3. **Performance**: Once loaded into memory, subsequent calls to package elements are faster.

Think of a package as a well-organized toolbox. The package specification is like the label on the toolbox, telling you what tools are inside. The package body is the actual contents of the toolbox, where the real work happens.

Creating a Package Specification

The package specification is your "toolbox label". It declares the public elements of your package - the things other parts of your code can see and use.

Here's the basic syntax for creating a package specification:

```
CREATE [OR REPLACE] PACKAGE package_name AS
  -- Public declarations go here
END package_name;
```

Let's create a package to manage employee-related operations using the HR schema:

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
  -- Define a record type for employee information
  TYPE emp_info_type IS RECORD (
    emp_id     employees.employee_id%TYPE,
    full_name VARCHAR2(100),
    job_title jobs.job_title%TYPE,
    dept_name departments.department_name%TYPE
  );

  -- Define a table type to hold multiple employee records
  TYPE emp_table_type IS TABLE OF emp_info_type INDEX BY PLS_INTEGER;

  -- Declare public procedures and functions
  PROCEDURE get_employee(p_emp_id IN employees.employee_id%TYPE, p_emp_info OUT emp_info_type);
  FUNCTION is_manager(p_emp_id IN employees.employee_id%TYPE) RETURN BOOLEAN;
  PROCEDURE get_department_employees(p_dept_id IN departments.department_id%TYPE, p_emp_table OUT e
mp_table_type);
END emp_mgmt;
/
```

In this specification, we've declared a record type, a table type, and three subprograms. These are all public and can be used by any code that has access to this package.

Creating a Package Body

The package body is where you implement the subprograms declared in the specification. It's like the inside of your toolbox, where the actual tools reside.

Here's the basic syntax for creating a package body:

```
CREATE [OR REPLACE] PACKAGE BODY package_name AS
  -- Private declarations
  -- Subprogram implementations
END package_name;
```

Let's implement the body for our emp_mgmt package:

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
  -- Private cursor
  CURSOR emp_cur(p_dept_id departments.department_id%TYPE) IS
    SELECT e.employee_id, e.first_name || ' ' || e.last_name AS full_name,
           j.job_title, d.department_name
    FROM employees e
    JOIN jobs j ON e.job_id = j.job_id
    JOIN departments d ON e.department_id = d.department_id
    WHERE e.department_id = p_dept_id;

  -- Implement get_employee procedure
  PROCEDURE get_employee(p_emp_id IN employees.employee_id%TYPE, p_emp_info OUT emp_info_type) IS
  BEGIN
    SELECT e.employee_id, e.first_name || ' ' || e.last_name,
           j.job_title, d.department_name
    INTO p_emp_info
    FROM employees e
    JOIN jobs j ON e.job_id = j.job_id
    JOIN departments d ON e.department_id = d.department_id
    WHERE e.employee_id = p_emp_id;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      p_emp_info := NULL;
  END get_employee;

  -- Implement is_manager function
  FUNCTION is_manager(p_emp_id IN employees.employee_id%TYPE) RETURN BOOLEAN IS
    v_direct_reports NUMBER;
  BEGIN
    SELECT COUNT(*)
    INTO v_direct_reports
    FROM employees
    WHERE manager_id = p_emp_id;

    RETURN v_direct_reports > 0;
  END is_manager;

  -- Implement get_department_employees procedure
  PROCEDURE get_department_employees(p_dept_id IN departments.department_id%TYPE, p_emp_table OUT e
mp_table_type) IS
    v_index PLS_INTEGER := 1;
  BEGIN
    p_emp_table.
```

```
DELETE;  -- Clear the table
    FOR emp_rec IN emp_cur(p_dept_id) LOOP
      p_emp_table(v_index) := emp_rec;
      v_index := v_index + 1;
    END LOOP;
  END get_department_employees;

END emp_mgmt;
/
```

In this body, we've implemented all the subprograms declared in the specification. We've also added a private cursor that's only accessible within the package.

Using the Package

Now that we've created our package, let's see how to use it:

```
DECLARE
  v_emp_info emp_mgmt.emp_info_type;
  v_emp_table emp_mgmt.emp_table_type;
BEGIN
  -- Get information for employee 103
  emp_mgmt.get_employee(103, v_emp_info);
  DBMS_OUTPUT.PUT_LINE('Employee: ' || v_emp_info.full_name || ' (' || v_emp_info.job_title || ')');

  -- Check if employee 103 is a manager
  IF emp_mgmt.is_manager(103) THEN
    DBMS_OUTPUT.PUT_LINE('This employee is a manager');
  ELSE
    DBMS_OUTPUT.PUT_LINE('This employee is not a manager');
  END IF;

  -- Get all employees in department 60
  emp_mgmt.get_department_employees(60, v_emp_table);
  DBMS_OUTPUT.PUT_LINE('Employees in department 60:');
  FOR i IN 1..v_emp_table.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_table(i).full_name);
  END LOOP;
END;
/
```

This example demonstrates how to use the public elements of our package. Notice how we use the package name followed by a dot to access its members.

Lab 21.2: Package Instantiation and Initialization

After this lab, you will be able to:

- Understand Package Instantiation and Initialization
- Describe the Package State

Package Instantiation and Initialization

When you first reference a package in a session, Oracle instantiates it. This is like unpacking your toolbox when you start a job. If multiple people are working (multiple sessions), each gets their own instance of the toolbox.

The instantiation process follows these steps:

1. Assign initial values to public constants

2. Assign initial values to public variables (if specified)
3. Execute the initialization section of the package body (if it exists)

Let's modify our emp_mgmt package to include some initialization:

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
  -- Public variable
  last_accessed_emp NUMBER;

  -- Rest of the package specification remains the same
  ...
END emp_mgmt;
/

CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
  -- Package initialization
  BEGIN
    last_accessed_emp := 0;
    DBMS_OUTPUT.PUT_LINE('emp_mgmt package initialized');
  END;

  -- Rest of the package body remains the same
  ...
END emp_mgmt;
/
```

Now, when the package is first referenced, it will initialize last_accessed_emp to 0 and print a message.

Package State

A package's state includes the values of its variables and the status of its cursors. Packages can be stateful or stateless:

- **Stateful**: Packages that declare variables, cursors, or constants.
- **Stateless**: Packages that don't declare any of these, or only declare compile-time constants.

Our emp_mgmt package is stateful because it declares the last_accessed_emp variable.

The package state persists for the duration of a session, with a few exceptions:

1. When the package is recompiled
2. When the package is marked as SERIALLY_REUSABLE (covered in the next lab)
3. When a session references multiple packages and one becomes invalid

Here's an example of a stateless package:

```
CREATE OR REPLACE PACKAGE constants_pkg AS
  c_max_salary CONSTANT NUMBER := 30000;
  c_min_salary CONSTANT NUMBER := 2000;
END constants_pkg;
/
```

This package only contains compile-time constants, so Oracle treats it as stateless.

Lab 21.3: SERIALLY_REUSABLE Packages

After this lab, you will be able to:

Use SERIALLY_REUSABLE Packages

Understanding SERIALLY_REUSABLE Packages

SERIALLY_REUSABLE packages are like disposable cups at a water cooler. Instead of each person (session) having their own cup (package state) for the entire day, they use a cup briefly and then discard it, allowing others to reuse it.

These packages help with memory management and scalability. Their state is stored in the System Global Area (SGA) and exists only for the duration of a call to the package.

To create a SERIALLY_REUSABLE package, add the pragma in both the specification and body:

```
CREATE OR REPLACE PACKAGE sr_emp_mgmt AS
  PRAGMA SERIALLY_REUSABLE;
  -- Package contents
END sr_emp_mgmt;
/

CREATE OR REPLACE PACKAGE BODY sr_emp_mgmt AS
  PRAGMA SERIALLY_REUSABLE;
  -- Package body contents
END sr_emp_mgmt;
/
```

Let's see the difference between a regular package and a SERIALLY_REUSABLE package:

```
CREATE OR REPLACE PACKAGE regular_pkg AS
  v_count NUMBER := 0;
  PROCEDURE increment;
END regular_pkg;
/

CREATE OR REPLACE PACKAGE BODY regular_pkg AS
  PROCEDURE increment IS
  BEGIN
    v_count := v_count + 1;
  END increment;
END regular_pkg;
/

CREATE OR REPLACE PACKAGE sr_pkg AS
  PRAGMA SERIALLY_REUSABLE;
  v_count NUMBER := 0;
  PROCEDURE increment;
END sr_pkg;
/

CREATE OR REPLACE PACKAGE BODY sr_pkg AS
  PRAGMA SERIALLY_REUSABLE;
  PROCEDURE increment IS
  BEGIN
    v_count := v_count + 1;
  END increment;
END sr_pkg;
/

-- Test the packages
BEGIN
  -- Regular package
  regular_pkg.increment;
  regular_pkg.increment;
  DBMS_OUTPUT.PUT_LINE('Regular package count: ' || regular_pkg.v_count);

  -- SERIALLY_REUSABLE package
  sr_pkg.increment;
  sr_pkg.increment;
  DBMS_OUTPUT.PUT_LINE('SERIALLY_REUSABLE package count: ' || sr_pkg.v_count);
END;
/
```

The regular package will output 2, while the SERIALLY_REUSABLE package will output 1. This is because the state of the SERIALLY_REUSABLE package is reset after each call.

Remember, SERIALLY_REUSABLE packages have some limitations:

- They can't be referenced from triggers
- They can't be referenced from SQL statements
- They can't be referenced from PL/SQL functions called from SQL statements

Summary

In this chapter, you've learned about PL/SQL packages, powerful tools for organizing and modularizing your code. You've seen how to create package specifications and bodies, how packages are instantiated and initialized, and how package state works. You've also explored SERIALLY_REUSABLE packages and their unique behavior.

Packages offer numerous benefits:

- Improved code organization and modularity

- Encapsulation of implementation details
- Better performance through memory management

By mastering packages, you'll be able to create more efficient, maintainable, and scalable PL/SQL applications.

Packages in Oracle PL/SQL