

4

Using More Package Concepts

Objectives

After completing this lesson, you should be able to do the following:

- **Overload package procedures and functions**
- **Use forward declarations**
- **Create an initialization block in a package body**
- **Manage persistent package data states for the life of a session**
- **Use PL/SQL tables and records in packages**
- **Wrap source code stored in the data dictionary so that it is not readable**

Overloading Subprograms

The overloading feature in PL/SQL:

- **Enables you to create two or more subprograms with the same name**
- **Requires that the subprogram's formal parameters differ in number, order, or data type family**
- **Enables you to build flexible ways for invoking subprograms with different data**
- **Provides a way to extend functionality without loss of existing code**

Note: Overloading can be done with local subprograms, package subprograms, and type methods, but not with stand-alone subprograms.

Overloading: Example

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department(deptno NUMBER,
    name VARCHAR2 := 'unknown', loc NUMBER := 1700);
  PROCEDURE add_department(
    name VARCHAR2 := 'unknown', loc NUMBER := 1700);
END dept_pkg;
/
```

Overloading: Example

```
CREATE OR REPLACE PACKAGE BODY dept_pkg IS
  PROCEDURE add_department (deptno NUMBER,
    name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES (deptno, name, loc);
  END add_department;

  PROCEDURE add_department (
    name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, name, loc);
  END add_department;
END dept_pkg;
```

/

Overloading and the STANDARD Package

- A package named STANDARD defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. An example is the TO_CHAR function:

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;  
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;  
FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN  
VARCHAR2;  
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN  
VARCHAR2;
```

- A PL/SQL subprogram with the same name as a built-in subprogram overrides the standard declaration in the local context, unless you qualify the built-in subprogram with its package name.

Using Forward Declarations

- **Block-structured languages (such as PL/SQL) must declare identifiers before referencing them.**
- **Example of a referencing problem:**

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE award_bonus(. . .) IS
  BEGIN

    calc_rating (. . .);      --illegal reference

  END;

  PROCEDURE calc_rating (. . .) IS
  BEGIN
    ...
  END;
END forward_pkg;
/
```

Using Forward Declarations

In the package body, a forward declaration is a private subprogram specification terminated by a semicolon.

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
→PROCEDURE calc_rating (...);-- forward declaration
    -- Subprograms defined in alphabetical order

    PROCEDURE award_bonus(...) IS
    BEGIN
        calc_rating (...);          -- reference resolved!
        . . .
    END;

    PROCEDURE calc_rating (...) IS -- implementation
    BEGIN
        . . .
    END;
END forward_pkg;
```


Package Initialization Block

The block at the end of the package body executes once and is used to initialize public and private package variables.

```
CREATE OR REPLACE PACKAGE taxes IS
    tax    NUMBER;
    ...    -- declare all public procedures/functions
END taxes;
/

CREATE OR REPLACE PACKAGE BODY taxes IS
    ...    -- declare all private variables
    ...    -- define public/private procedures/functions
BEGIN
    SELECT    rate_value INTO tax
    FROM      tax_rates
    WHERE     rate_name = 'TAX';
END taxes;
/
```

Using Package Functions in SQL and Restrictions

- **Package functions can be used in SQL statements.**
- **Functions called from:**
 - **A query or DML statement must not end the current transaction, create or roll back to a savepoint, or alter the system or session**
 - **A query or a parallelized DML statement cannot execute a DML statement or modify the database**
 - **A DML statement cannot read or modify the table being changed by that DML statement**

Note: A function calling subprograms that break the preceding restrictions is not allowed.

Package Function in SQL: Example

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
    FUNCTION tax (value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
    FUNCTION tax (value IN NUMBER) RETURN NUMBER IS
        rate NUMBER := 0.08;
    BEGIN
        RETURN (value * rate);
    END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM   employees;
```

Persistent State of Packages

The collection of package variables and the values define the package state. The package state is:

- **Initialized when the package is first loaded**
- **Persistent (by default) for the life of the session**
 - **Stored in the User Global Area (UGA)**
 - **Unique to each session**
 - **Subject to change when package subprograms are called or public variables are modified**
- **Not persistent for the session but persistent for the life of a subprogram call when using `PRAGMA SERIALLY_REUSABLE` in the package specification**

Persistent State of Package Variables: Example

		State for: -Scott-		-Jones-	
Time	Events	STD	MAX	STD	MAX
9:00	Scott> EXECUTE comm_pkg.reset_comm(0.25)	0.10 0.25	0.4	-	0.4
9:30	Jones> INSERT INTO employees(last_name,commission_pct) VALUES('Madonna', 0.8);	0.25	0.4		0.8
9:35	Jones> EXECUTE comm_pkg.reset_comm (0.5)	0.25	0.4	0.1 0.5	0.8
10:00	Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'	0.25	0.4	0.5	0.8
11:00	Jones> ROLLBACK;	0.25	0.4	0.5	0.4
11:01	EXIT ...	0.25	0.4	-	0.4
12:00	EXEC comm_pkg.reset_comm(0.2)	0.25	0.4	0.2	0.4

Persistent State of a Package Cursor

```
CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  CURSOR c IS SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT c%ISOPEN THEN    OPEN c;  END IF;
  END open;
  FUNCTION next(n NUMBER := 1) RETURN BOOLEAN IS
    emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. n LOOP
      FETCH c INTO emp_id;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' || (emp_id));
    END LOOP;
    RETURN c%FOUND;
  END next;
  PROCEDURE close IS BEGIN
    IF c%ISOPEN THEN  CLOSE c;  END IF;
  END close;
END curs_pkg;
```

Executing CURS_PKG

```
SET SERVEROUTPUT ON
EXECUTE curs_pkg.open
DECLARE
    more BOOLEAN := curs_pkg.next(3);
BEGIN
    IF NOT more THEN
        curs_pkg.close;
    END IF;
END;
/
RUN  -- repeats execution on the anonymous block
EXECUTE curs_pkg.close
```

Using PL/SQL Tables of Records in Packages

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_table_type IS TABLE OF employees%ROWTYPE
    INDEX BY BINARY_INTEGER;
  PROCEDURE get_employees(emps OUT emp_table_type);
END emp_pkg;
/
```

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE get_employees(emps OUT emp_table_type) IS
    i BINARY_INTEGER := 0;
  BEGIN
    FOR emp_record IN (SELECT * FROM employees)
    LOOP
      emps(i) := emp_record;
      i := i+1;
    END LOOP;
  END get_employees;
END emp_pkg;
/
```


PL/SQL Wrapper

- **The PL/SQL wrapper is a stand-alone utility that hides application internals by converting PL/SQL source code into portable object code.**
- **Wrapping has the following features:**
 - Platform independence
 - Dynamic loading
 - Dynamic binding
 - Dependency checking
 - Normal importing and exporting when invoked

Running the Wrapper

The command-line syntax is:

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

- The **INAME** argument is required.
- The default extension for the input file is **.sql**, unless it is specified with the name.
- The **ONAME** argument is optional.
- The default extension for output file is **.plb**, unless specified with the **ONAME** argument.

Examples:

```
WRAP INAME=demo_04_hello.sql  
WRAP INAME=demo_04_hello  
WRAP INAME=demo_04_hello.sql ONAME=demo_04_hello.plb
```

Results of Wrapping

- **Original PL/SQL source code in input file:**

```
CREATE PACKAGE banking IS
  min_bal := 100;
  no_funds EXCEPTION;
  ...
END banking;
/
```

- **Wrapped code in output file:**

```
CREATE PACKAGE banking
wrapped
012abc463e ...
/
```

Guidelines for Wrapping

- **You must wrap only the package body, not the package specification.**
- **The wrapper can detect syntactic errors but cannot detect semantic errors.**
- **The output file should not be edited. You maintain the original source code and wrap again as required.**

Summary

In this lesson, you should have learned how to:

- **Create and call overloaded subprograms**
- **Use forward declarations for subprograms**
- **Write package initialization blocks**
- **Maintain persistent package state**
- **Use the PL/SQL wrapper to wrap code**

Practice 4: Overview

This practice covers the following topics:

- **Using overloaded subprograms**
- **Creating a package initialization block**
- **Using a forward declaration**
- **Using the `WRAP` utility to prevent the source code from being deciphered by humans**