Working with Packages

Objectives

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

In this lesson, you should have learned how to:

- Overload package procedures and functions
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Using PL/SQL Tables of Records in Packages

This lesson introduces the more advanced features of PL/SQL, including overloading, forward referencing, one-time-only procedures, and the persistency of variables, constants, exceptions, and cursors. It also explains the effect of packaging functions that are used in SQL statements.

Overloading Subprograms 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; that is, adding new parameters to existing subprograms
- Provides a way to overload local subprograms, package subprograms, and type methods, but not stand-alone subprograms

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the `TO_CHAR` function has more than one way to be called, enabling you to convert a number or a date to a character string.

PL/SQL allows overloading of package subprogram names and object type methods.

The key rule is that you can use the same name for different subprograms as long as their formal parameters differ in *number*, *order*, or *data type* family.

Consider using overloading when:

- Processing rules for two or more subprograms are similar, but the type or number of parameters used varies
- Providing alternative ways for finding different data with varying search criteria. For example, you may want to find employees by their employee ID and also provide a way to find employees by their last name. The logic is intrinsically the same, but the parameters or search criteria differ.
- Extending functionality when you do not want to replace existing code

Note: Stand-alone subprograms cannot be overloaded. Writing local subprograms in object

type methods is not discussed in this course.

Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same family (`NUMBER` and `DECIMAL` belong to the same family.)
- Two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family (`VARCHAR` and `STRING` are PL/SQL subtypes of `VARCHAR2`.)
- Two functions that differ only in return type, even if the types are in different families

You get a run-time error when you overload subprograms with the preceding features.

Note: The preceding restrictions apply if the names of the parameters are also the same. If you use different names for the parameters, you can invoke the subprograms by using named notation for the parameters.

Resolving Calls

The compiler tries to find a declaration that matches the call. It searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the name matches the name of the called subprogram. For similarly named subprograms at the same level of scope, the compiler needs an exact match in number, order, and data type between the actual and formal parameters.

Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department
    (p_deptno departments.department_id%TYPE,
     p_name departments.department_name%TYPE :='unknown',
     p_loc  departments.location_id%TYPE := 1700);

  PROCEDURE add_department
    (p_name departments.department_name%TYPE :='unknown',
     p_loc  departments.location_id%TYPE := 1700);
END dept_pkg;
/
```


Overloading: Example

The slide shows the `dept_pkg` package specification with an overloaded procedure called `add_department`. The first declaration takes three parameters that are used to provide data for a new department record inserted into the department table. The second declaration takes only two parameters because this version internally generates the department ID through an Oracle sequence.

It is better to specify data types using the `%TYPE` attribute for variables that are used to populate columns in database tables, as shown in the example in the slide; however, you can also specify the data types as follows:

```
CREATE OR REPLACE PACKAGE dept_pkg_method2 IS
  PROCEDURE add_department(p_deptno NUMBER,
   p_name VARCHAR2 := 'unknown', p_loc NUMBER := 1700);
...
```

Overloading Procedures Example: Creating the Package Body

```
-- Package body of package defined on previous slide.
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
PROCEDURE add_department -- First procedure's declaration
  (p_deptno departments.department_id%TYPE,
   p_name   departments.department_name%TYPE := 'unknown',
   p_loc    departments.location_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES  (p_deptno, p_name, p_loc);
  END add_department;
PROCEDURE add_department -- Second procedure's declaration
  (p_name   departments.department_name%TYPE := 'unknown',
   p_loc    departments.location_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
  END add_department;
 END dept_pkg; /
```


Overloading: Example

If you call `add_department` with an explicitly provided department ID, then PL/SQL uses the first version of the procedure. Consider the following example:

```
EXECUTE dept_pkg.add_department(980,'Education',2500)
SELECT * FROM departments
WHERE department_id = 980;
```

```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          980 Education                                        2500
```

If you call `add_department` with no department ID, PL/SQL uses the second version:

```
EXECUTE dept_pkg.add_department ('Training', 2400)
SELECT * FROM departments
WHERE department_name = 'Training';
```

```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          350 Training                                         2400
```

Package Function in SQL: Example

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

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


The first code example in the slide shows how to create the package specification and the body encapsulating the tax function in the `taxes_pkg` package. The second code example shows how to call the packaged tax function in the `SELECT` statement. The results are as follows:

```
TAXES_PKG.TAX(SALARY)   SALARY LAST_NAME
--------------------- -------- -------------------------
                 1920    24000 King
                 1360    17000 Kochhar
                 1360    17000 De Haan
                  720     9000 Hunold
                  480     6000 Ernst
                  384     4800 Austin
                  384     4800 Pataballa
                  336     4200 Lorentz
               960.64    12008 Greenberg
...

109 rows selected
```

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

- persistent for the session but persistent for the life of a subprogram call when using `PRAGMA SERIALLY_REUSABLE` in the package specification

ORACLE

The collection of public and private package variables represents the package state for the user session. That is, the package state is the set of values stored in all the package variables at a given point in time. In general, the package state exists for the life of the user session.

Package variables are initialized the first time a package is loaded into memory for a user session. The package variables are, by default, unique to each session and hold their values until the user session is terminated. In other words, the variables are stored in the User Global Area (UGA) memory allocated by the database for each user session. The package state changes when a package subprogram is invoked and its logic modifies the variable state. Public package state can be directly modified by operations appropriate to its type.

`PRAGMA` signifies that the statement is a compiler directive. `PRAGMA`s are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler. If you add `PRAGMA SERIALLY_RESUABLE` to the package specification, then the database stores package variables in the System Global Area (SGA) shared across user sessions. In this case, the package state is maintained for the life of a subprogram call or a single reference to a package construct. The `SERIALLY_REUSABLE` directive is useful if you want to conserve memory and if the package state does not need to persist for each user session.

This `PRAGMA` is appropriate for packages that declare large temporary work areas that are used once and not needed during subsequent database calls in the same session.

You can mark a bodiless package as serially reusable. If a package has a spec and body, you must mark both. You cannot mark only the body.

The global memory for serially reusable packages is pooled in the System Global Area (SGA), not allocated to individual users in the User Global Area (UGA). That way, the package work area can be reused. When the call to the server ends, the memory is returned to the pool. Each time the package is reused, its public variables are initialized to their default values or to `NULL`.

Note: Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are called from SQL statements. If you try, the Oracle server generates an error.

Persistent State of a Package Cursor: Example

```
CREATE OR REPLACE PACKAGE curs_pkg IS -- Package spec
  PROCEDURE open;
  FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN;
  PROCEDURE close;
END curs_pkg;

CREATE OR REPLACE PACKAGE BODY curs_pkg IS
-- Package body
  CURSOR cur_c IS
    SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT cur_c%ISOPEN THEN
      OPEN cur_c;
    END IF;
  END open;
. . .  -- code continued on next slide
```


The example in the slide shows the `CURS_PKG` package specification and body. The body declaration is continued in the next slide.

To use this package, perform the following steps to process the rows:

1. Call the `open` procedure to open the cursor.

Persistent State of a Package Cursor: Example

```
. . .
  FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN IS
    v_emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. p_n LOOP
      FETCH cur_c INTO v_emp_id;
      EXIT WHEN cur_c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' ||(v_emp_id));
    END LOOP;
    RETURN cur_c%FOUND;
  END next;
  PROCEDURE close IS
    BEGIN
      IF cur_c%ISOPEN THEN
        CLOSE cur_c;
      END IF;
    END close;
END curs_pkg;
```


2. Call the `next` procedure to fetch one or a specified number of rows. If you request more rows than actually exist, the procedure successfully handles termination.
   It returns `TRUE` if more rows need to be processed; otherwise it returns `FALSE`.
3. Call the `close` procedure to close the cursor, before or at the end of processing the rows.

Note: The cursor declaration is private to the package. Therefore, the cursor state can be influenced by invoking the package procedure and functions listed in the slide.

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(p_emps OUT emp_table_type);
END emp_pkg;
```

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


Associative arrays used to be known as index by tables.

The `emp_pkg` package contains a `get_employees` procedure that reads rows from the `EMPLOYEES` table and returns the rows using the `OUT` parameter, which is an associative array (PL/SQL table of records). The key points include the following:

- `employee_table_type` is declared as a public type.
- `employee_table_type` is used for a formal output parameter in the procedure, and the `employees` variable in the calling block (shown below).

In SQL Developer, you can invoke the `get_employees` procedure in an anonymous PL/SQL block by using the `v_employees` variable, as shown in the following example and output:

```
SET SERVEROUTPUT ON
DECLARE
 v_employees  emp_pkg.emp_table_type;
BEGIN
  emp_pkg.get_employees(v_employees);
  DBMS_OUTPUT.PUT_LINE('Emp 5: '||v_employees(4).last_name);
END;
```

```
anonymous block completed
Emp 5: De Haan
```

Quiz

Overloading subprograms in PL/SQL:

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

Answer: a, b, c, d

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the `TO_CHAR` function has more than one way to be called, enabling you to convert a number or a date to a character string.

PL/SQL allows overloading of package subprogram names and object type methods.

The key rule is that you can use the same name for different subprograms as long as their formal parameters differ in *number*, *order*, or *data type* family.

Consider using overloading when:

- Processing rules for two or more subprograms are similar, but the type or number of parameters used varies
- Providing alternative ways for finding different data with varying search criteria. For example, you may want to find employees by their employee ID and also provide a way to find employees by their last name. The logic is intrinsically the same, but the parameters or search criteria differ.

- Extending functionality when you do not want to replace existing code

Summary

In this lesson, you should have learned how to:

- Overload package procedures and functions
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Using PL/SQL Tables of Records in Packages

Overloading is a feature that enables you to define different subprograms with the same name. It is logical to give two subprograms the same name when the processing in both the subprograms is the same but the parameters passed to them vary.

PL/SQL permits a special subprogram declaration called a forward declaration. A forward declaration enables you to define subprograms in logical or alphabetical order, define mutually recursive subprograms, and group subprograms in a package.

A package initialization block is executed only when the package is first invoked within the other user session. You can use this feature to initialize variables only once per session.

You can keep track of the state of a package variable or cursor, which persists throughout the user session, from the time the user first references the variable or cursor to the time the user disconnects.

Using the PL/SQL wrapper, you can obscure the source code stored in the database to protect your intellectual property.