Database Programming with PL/SQL

Advanced Package Concepts

Objectives

This lesson covers the following objectives:

- Write packages that use the overloading feature
- Write packages that use forward declarations
- Explain the purpose of a package initialization block
- Create and use a bodiless package
- Invoke packaged functions from SQL
- Identify restrictions on using packaged functions in SQL statements
- Create a package that uses PL/SQL tables and records

Purpose

This lesson introduces additional advanced features of PL/SQL packages, including overloading, forward referencing, and a package initialization block.

It also explains the restrictions on package functions that are used in SQL statements.

Overloading 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.

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

Overloading Subprograms (cont.)

The overloading feature in PL/SQL:

- Enables you to create two or more subprograms with the same name, in the same package
- Enables you to build flexible ways for invoking the same subprograms with different data
- Makes things easier for the application developer, who has to remember only one subprogram name.

Overloading Subprograms (cont.)

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 category of data type.

Note: Overloading can be done with subprograms in packages, but not with stand-alone subprograms.

Overloading Subprograms (cont.)

Consider using overloading when the purposes of two or more subprograms are similar, but the type or number of parameters used varies.

Overloading can provide alternative ways for finding different data with varying search criteria. For example, you might want to find employees by their employee id, and also provide a way to find employees by their job id, or by their hire date. The purpose is the same, but the parameters or search criteria differ.

The next slide shows an example of this.

Overloading: Example

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE find_emp                                    -- 1
    (p_employee_id IN NUMBER, p_last_name OUT VARCHAR2);
  PROCEDURE find_emp                                    -- 2
    (p_job_id IN VARCHAR2, p_last_name OUT VARCHAR2);
  PROCEDURE find_emp                                    -- 3
    (p_hiredate IN DATE, p_last_name OUT VARCHAR2);
END emp_pkg;
```

The `emp_pkg` package specification contains an overloaded procedure called `find_emp`. The input arguments of the three declarations have different categories of datatype. Which of the declarations is executed by the following call?

```
DECLARE  v_last_name   VARCHAR2(30);
BEGIN    emp_pkg.find_emp('IT_PROG', v_last_name);
END;
```

Overloading Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same category (`NUMBER` and `INTEGER` belong to the same category; `VARCHAR2` and `CHAR` belong to the same category).
- Two functions that differ only in return type, even if the types are in different categories.

Overloading Restrictions (cont.)

These restrictions apply if the names of the parameters are also the same.

If you use different names for the parameters, then you can invoke the subprograms by using named notation for the parameters. The next slide shows an example of this.

Overloading: Example 2

```
CREATE PACKAGE sample_pack IS
  PROCEDURE  sample_proc (p_char_param IN CHAR);
  PROCEDURE  sample_proc (p_varchar_param IN VARCHAR2);
END sample_pack;
```

Now you invoke a procedure using positional notation:

```
BEGIN   sample_pack.sample_proc('Smith');   END;
```

This fails because `'Smith'` can be either `CHAR` or `VARCHAR2`. But the following invocation succeeds:

```
BEGIN sample_pack.sample_proc(p_char_param =>'Smith');  END;
```

Overloading: Example 3

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

In this example, the `dept_pkg` package specification contains 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.

Overloading: Example 3 (cont.)

```
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
  PROCEDURE add_department (p_deptno NUMBER,
    p_name VARCHAR2:='unknown', p_loc NUMBER:=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 (
    p_name VARCHAR2:='unknown', p_loc NUMBER:=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 3 (cont.)

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:

```
BEGIN
  dept_pkg.add_department(980,'Education',2500);
END;

SELECT * FROM departments
WHERE department_id = 980;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
980	Education	-	2500

Overloading: Example 3 (cont.)

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

```
BEGIN
  dept_pkg.add_department ('Training', 2500);
END;

SELECT * FROM departments
WHERE department_name = 'Training';
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
290	Training	-	2500

Overloading and the STANDARD Package

- A package named STANDARD defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. You have already seen that TO_CHAR is overloaded. Another example is the UPPER function:

```
FUNCTION UPPER (ch VARCHAR2) RETURN VARCHAR2;
FUNCTION UPPER (ch CLOB) RETURN CLOB;
```

- You do not prefix STANDARD package subprograms with the package name.

Overloading and the STANDARD Package (cont.)

Question: What if you create your own function with the same name as a STANDARD package function? For example, you create your own UPPER function. Then you invoke UPPER(argument). Which one is executed?

Answer: even though your function is in your own schema, the built-in STANDARD function is executed. To call your own function, you need to prefix it with your schema-name:

```
...
BEGIN
  v_return_value := your-schema-name.UPPER(argument);
END;
```

Using Forward Declarations

- Block-structured languages (such as PL/SQL) must declare identifiers before referencing them.
- Example of a referencing problem if `award_bonus` is public and `calc_rating` is private:

```
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;
```

- `calc_rating` is referenced (in `award_bonus`) before it has been declared.

Using Forward Declarations

You can solve the illegal reference problem by reversing the order of the two procedures.

However, coding standards often require that subprograms be kept in alphabetical sequence to make them easy to find. In this case, you might encounter problems, as shown in the slide example.

Using Forward Declarations (cont.)

Note: The compilation error for `calc_rating` occurs only if `calc_rating` is a private packaged subprogram. If `calc_rating` is declared in the package specification, then it is already declared as if it was a forward declaration, and its reference can be resolved by the PL/SQL compiler.

Using Forward Declarations (cont.)

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;
```

Using Forward Declarations (cont.)

Forward declarations help to:

- Define subprograms in logical or alphabetical order.
- Define mutually recursive subprograms. Mutually recursive programs are programs that call each other directly or indirectly.
- Group and logically organize subprograms in a package body.

Using Forward Declarations (cont.)

When creating a forward declaration:

- The formal parameters must appear in both the forward declaration and the subprogram body.
- The subprogram body can appear anywhere after the forward declaration, but both must appear in the same package body.

Package Initialization Block

Suppose you want to automatically execute some code every time you make the first call to a package in your session? For example, you want to automatically load a tax rate into a package variable.

If the tax rate is a constant, you can initialize the package variable as part of its declaration:

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  g_tax   NUMBER  := 0.20;
  ...
END taxes_pkg;
```

But what if the tax rate is stored in a database table?

Package Initialization Block (cont.)

Optionally, you can include an un-named block at the end of the package body. This block automatically executes once and is used to initialize public and private package variables.

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

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

Bodiless Packages

Every package must have two parts, a specification and a body. Right? Wrong.

You can create a useful package which has a specification but no body. This is called a bodiless package. Because it has no body, a bodiless package cannot contain any executable code: no procedures or functions. It can contain only public (global) variables.

Bodiless packages are often used to give names to unchanging constants, or to give names to non-predefined Oracle Server exceptions.

Bodiless Packages: Example 1

This package gives names to several constant ratios used in converting distances between two different systems of measurement.

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_to_kilo    CONSTANT  NUMBER  :=  1.6093;
  kilo_to_mile    CONSTANT  NUMBER  :=  0.6214;
  yard_to_meter   CONSTANT  NUMBER  :=  0.9144;
  meter_to_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;

GRANT EXECUTE ON global_consts TO PUBLIC;
```

Bodiless Packages: Example 2

This package declares two non-predefined Oracle Server exceptions.

```
CREATE OR REPLACE PACKAGE our_exceptions IS
  e_cons_violation     EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_cons_violation, -2292);
  e_value_too_large    EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_value_too_large, -1438);
END our_exceptions;

GRANT EXECUTE ON our_exceptions TO PUBLIC;
```

Invoking a Bodiless Package

The first invocation converts 500 miles to kilometers.
The second tests for constraint violations.

```
DECLARE
  distance_in_miles  NUMBER(5) := 5000;
  distance_in_kilo   NUMBER(6,2);
BEGIN
  distance_in_kilo :=
    distance_in_miles * global_consts.mile_to_kilo;
  DBMS_OUTPUT.PUT_LINE(distance_in_kilo);
END;
```

```
DECLARE
  ...
BEGIN
  ...
EXCEPTION
  WHEN our_exceptions.e_cons_violation THEN ...;
END;
```

Restrictions on Using Package Functions in SQL Statements

Package functions, like standalone functions, can be used in SQL statements and they must follow the same rules.

Restrictions on Using Package Functions in SQL Statements (cont.)

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 1

```
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;
```

Package Function in SQL: Example 2

```
CREATE OR REPLACE PACKAGE sal_pkg IS
  FUNCTION sal (p_emp_id IN NUMBER) RETURN NUMBER;
END sal_pkg;

CREATE OR REPLACE PACKAGE BODY sal_pkg IS
  FUNCTION sal (p_emp_id IN NUMBER) RETURN NUMBER IS
    v_sal  employees.salary%TYPE;
  BEGIN
    UPDATE employees SET salary = salary * 2
      WHERE employee_id = p_emp_id;
    SELECT salary INTO v_sal FROM employees
      WHERE employee_id = p_emp_id;
    RETURN (v_sal);
  END sal;
END sal_pkg;
```

```
SELECT sal_pkg.sal(employee_id), salary, last_name
FROM   employees;
```

Using a Record Structure as a Parameter

Earlier in the course, you learned how to declare and use composite data types such as records, either by using `%ROWTYPE` or by declaring your own `TYPE`.

What if you want to use a whole record as a procedure parameter? For example, you want your procedure to `SELECT` a whole row (many columns) from the `EMPLOYEES` table and pass it back to the calling environment.

Using a Record Structure as a Parameter (cont.)

The data type of a parameter can be any kind of PL/SQL variable, scalar or composite. The next slide shows how.

Using a Record Structure as a Parameter (cont.)

Create the procedure:

```
CREATE OR REPLACE PROCEDURE sel_one_emp
  (p_emp_id    IN  employees.employee_id%TYPE,
   p_emprec    OUT employees%ROWTYPE)
IS BEGIN
  SELECT * INTO p_emprec FROM employees
    WHERE employee_id = p_emp_id;
EXCEPTION
  WHEN NO_DATA_FOUND THEN ...
END sel_one_emp;
```

And invoke it from an anonymous block:

```
DECLARE
  v_emprec    employees%ROWTYPE;
BEGIN
  sel_one_emp(100, v_emprec);
  ...
  dbms_output.put_line(v_emprec.last_name);
  ...
END;
```

Using a User-defined Type as a Parameter

You can also use your own declared types as parameters, but you need to be careful. What is wrong with this code?

```
CREATE OR REPLACE PROCEDURE sel_emp_dept
  (p_emp_id       IN  employees.employee_id%TYPE,
   p_emp_dept_rec OUT ed_type)
IS
  TYPE ed_type IS RECORD (f_name employees.first_name%TYPE,
                          l_name employees.last_name%TYPE,
                          d_name departments.department_name%TYPE);
BEGIN
  SELECT e.first_name, e.last_name, d.department_name
    INTO ed_type.f_name, ed_type.l_name, ed_type.d_name
    FROM employees e JOIN departments d USING (employee_id)
    WHERE employee_id = p_emp_id;
EXCEPTION
  WHEN NO_DATA_FOUND THEN ...
END sel_emp_dept;
```

Using a User-defined Type as a Parameter (cont.)

Types must be declared before you can use them. And in a standalone procedure or function, the parameters (and their data types) are declared in the subprogram header, before we can declare our own types.

So how can we declare a type before declaring a parameter of that type? We must create a package.

We declare the type in the specification, before declaring any procedures or functions which have parameters of that type.

Using a User-defined Type as a Parameter (cont.)

ED_TYPE is declared globally in the specification and can be used outside the package.

```
CREATE OR REPLACE PACKAGE emp_dept_pkg
IS
  TYPE ed_type IS RECORD (f_name employees.first_name%TYPE,
                          l_name employees.last_name%TYPE,
                          d_name departments.department_name%TYPE);
  PROCEDURE sel_emp_dept
        (p_emp_id       IN  employees.employee_id%TYPE,
         p_emp_dept_rec OUT ed_type);
END emp_dept_pkg;
-- And create the package body as usual
```

```
DECLARE
  v_emp_dept_rec   emp_dept_pkg.ed_type;
BEGIN
  emp_dept_pkg.sel_emp_dept(100, v_emprec);
END;
```

Using an INDEX BY Table of Records in a Package

Because an INDEX BY table is also a kind of variable, it can be declared in a package specification. This allows it to be used by any subprogram within and outside the package:

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

Using an INDEX BY Table of Records in a Package (cont.)

The procedure uses a cursor to populate the INDEX BY table with employee rows, and return this data in a single OUT parameter.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE get_employees(p_emp_table OUT emprec_type) IS
  BEGIN
    FOR emp_record IN (SELECT * FROM employees)
    LOOP
      p_emp_table(emp_record.employee_id) := emp_record;
    END LOOP;
  END get_employees;
END emp_pkg;
```

Using an INDEX BY Table of Records in a Package (cont.)

The whole EMPLOYEES table has been fetched with a single procedure call. How can you do this in a standalone procedure or function (not in a package)?

```
DECLARE
  v_emp_table  emp_pkg.emprec_type;
BEGIN
  emp_pkg.read_emp_table(v_emp_table);
  FOR i IN v_emp_table.FIRST..v_emp_table.LAST
  LOOP
    IF v_emp_table.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE(v_emp_table(i).employee_id ...);
    END IF;
  END LOOP;
END;
```

Terminology

Key terms used in this lesson included:

- Bodiless package
- Forward declaration
- Initialization block
- Overloading
- `STANDARD`

Summary

In this lesson, you should have learned how to:

- Write packages that use the overloading feature
- Write packages that use forward declarations
- Explain the purpose of a package initialization block
- Create and use a bodiless package
- Invoke packaged functions from SQL
- Identify restrictions on using packaged functions in SQL statements
- Create a package that uses PL/SQL tables and records