Working with Packages

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

Overloading Subprograms (continued)

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 Procedures Example: Creating the Package Body

```
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 and the STANDARD Package

```
--overloading the standard package
create or replace package override
is
/*we defined to_char function and this function already exist
as oracle bulit-in function
*/

function to_char( p1 number, p2 date )
return varchar2;

procedure print;

end;
```

```
create or replace package body override
is

      function to_char( p1 number, p2 date )
      return varchar2
      is
      begin
      return p1||p2;
      end;

      procedure print
      is
      begin
      dbms_output.put_line(to_char(1,'1-jan-81' ));
      dbms_output.put_line(standard.to_char(10));
      end;

end;
```

the oracle will understand that to_char is the function from the package, not the built-in

when you prefix standard, this to tell oracle to use the built-in

Note:
Standard only used in PL/SQL

You can not do this outside PL/SQL

Select standard.to_char….

Illegal Procedure Reference

- 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 to Solve Illegal Procedure Reference

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;
```

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

```
create or replace package Persistent_state
is
g_var number:=10;

procedure update_g_var ( p_no number);

end;
```

```
create or replace package body Persistent_state
is
      procedure update_g_var ( p_no number)
      is
      begin
      g_var:=p_no;
      dbms_output.put_line(g_var);
      end;
end;
```

```
execute Persistent_state.update_g_var(90);
```

the value 90 will be in g_var untill end of your session

Thank You