

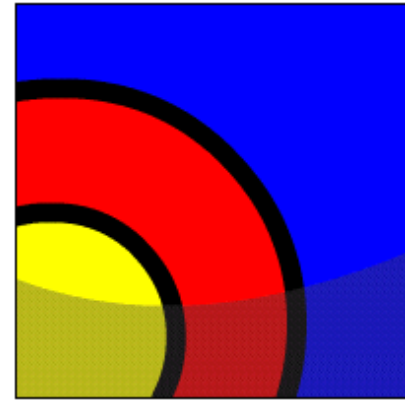
# Advanced Package Concepts



## What Will I Learn?

## In this lesson, you will learn to:

- Write packages that use the overloading feature
- Write packages that use forward declarations
- Explain the purpose of a package initialization block
- Identify restrictions on using packaged functions in SQL statements





## Why Learn It?

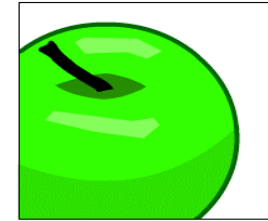
This lesson introduces additional advanced features of PL/SQL packages, including overloading, forward referencing, and a package initialization block.

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



# Tell Me / Show Me

## 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;  
...
```



## Tell Me / Show Me

### Overloading Subprograms (continued)

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.

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 standalone subprograms.



# Tell Me / Show Me

## Overloading Subprograms (continued)

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.



# Tell Me / Show Me

## 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;
```



# Tell Me / Show Me

## 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

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.





# Tell Me / Show Me

## 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;
```



# Tell Me / Show Me

## 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.



# Tell Me / Show Me

## Overloading: Example 3 (continued)

```
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;
```



# Tell Me / Show Me

## Overloading: Example 3 (continued)

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

1 rows returned in 0.00 seconds

[Download](#)



# Tell Me / Show Me

## Overloading: Example 3 (continued)

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

1 rows returned in 0.01 seconds

[Download](#)



# Tell Me / Show Me

## 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.



## Tell Me / Show Me

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;
```



# Tell Me / Show Me

## 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.





## Tell Me / Show Me

### 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.

**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.



# Tell Me / Show Me

## 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;
```

# Tell Me / Show Me

## Using Forward Declarations

- 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
- 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.

## Tell Me / Show Me

### 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 ?



# Tell Me / Show Me

## Package Initialization Block

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;
```



# Tell Me / Show Me

## 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.
- 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.



# Tell Me / Show Me

## 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;
```



# Tell Me / Show Me

## 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;
```



# Tell Me / Show Me

## Terminology

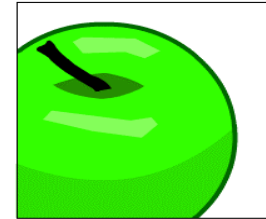
Key terms used in this lesson include:

Overloading

STANDARD

Forward declaration

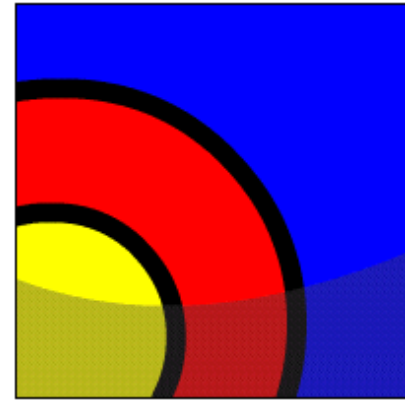
Initialization block



## Summary

In this lesson, you learned to:

- Write packages that use the overloading feature
- Write packages that use forward declarations
- Explain the purpose of a package initialization block
- Identify restrictions on using packaged functions in SQL statements





## Try It / Solve It

The exercises in this lesson cover the following topics:

- Writing packages that use the overloading feature
- Writing packages that use forward declarations
- Identifying restrictions on using packaged functions in SQL statements

