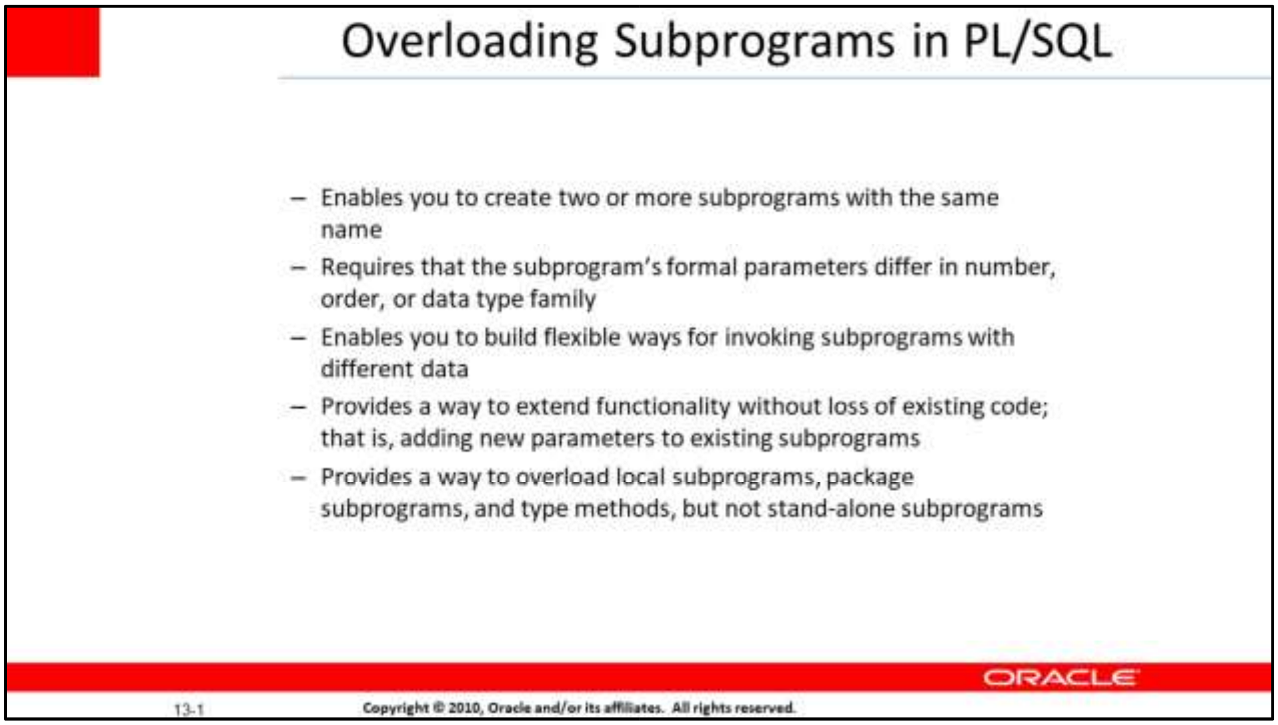


Overloading Subprograms in PL/SQL

- 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

13-1

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 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

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)

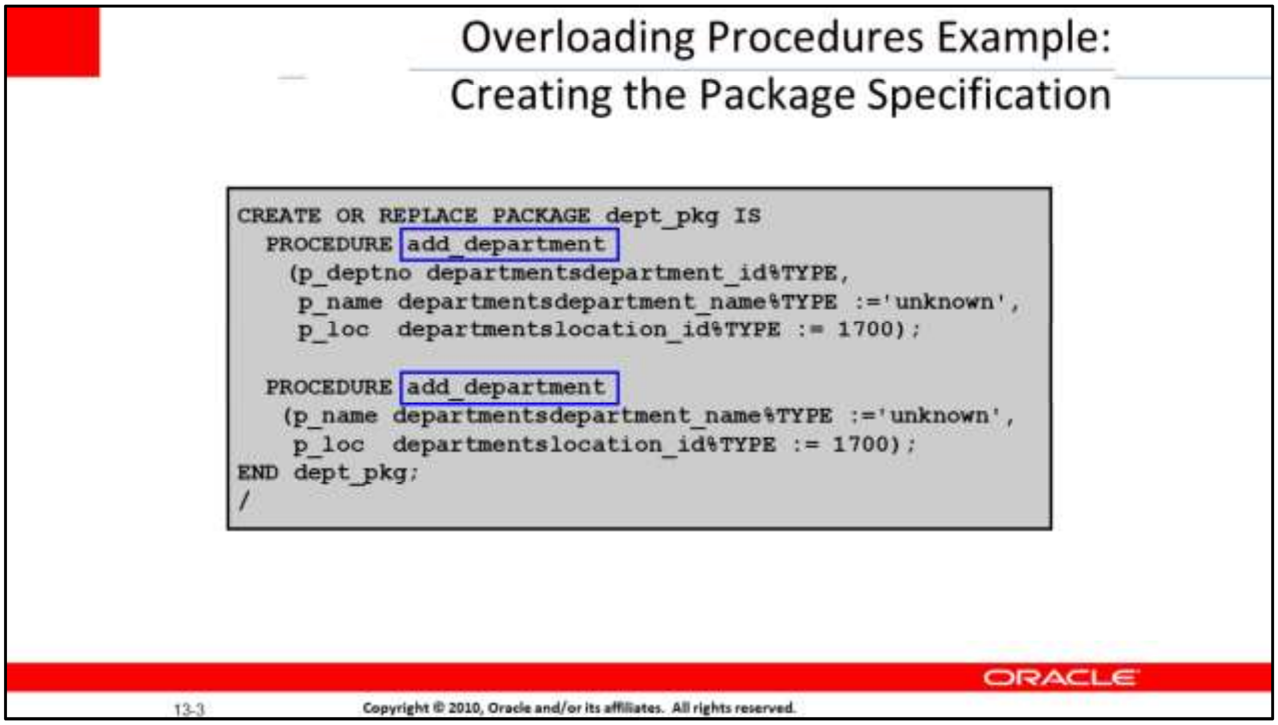
Two functions that differ only in return type, even if the types are in different families

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 departmentsdepartment_id%TYPE,
    p_name departmentsdepartment_name%TYPE :='unknown',
    p_loc  departmentslocation_id%TYPE := 1700);

  PROCEDURE add_department
   (p_name departmentsdepartment_name%TYPE :='unknown',
    p_loc  departmentslocation_id%TYPE := 1700);
END dept_pkg;
/
```

13-3 Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

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

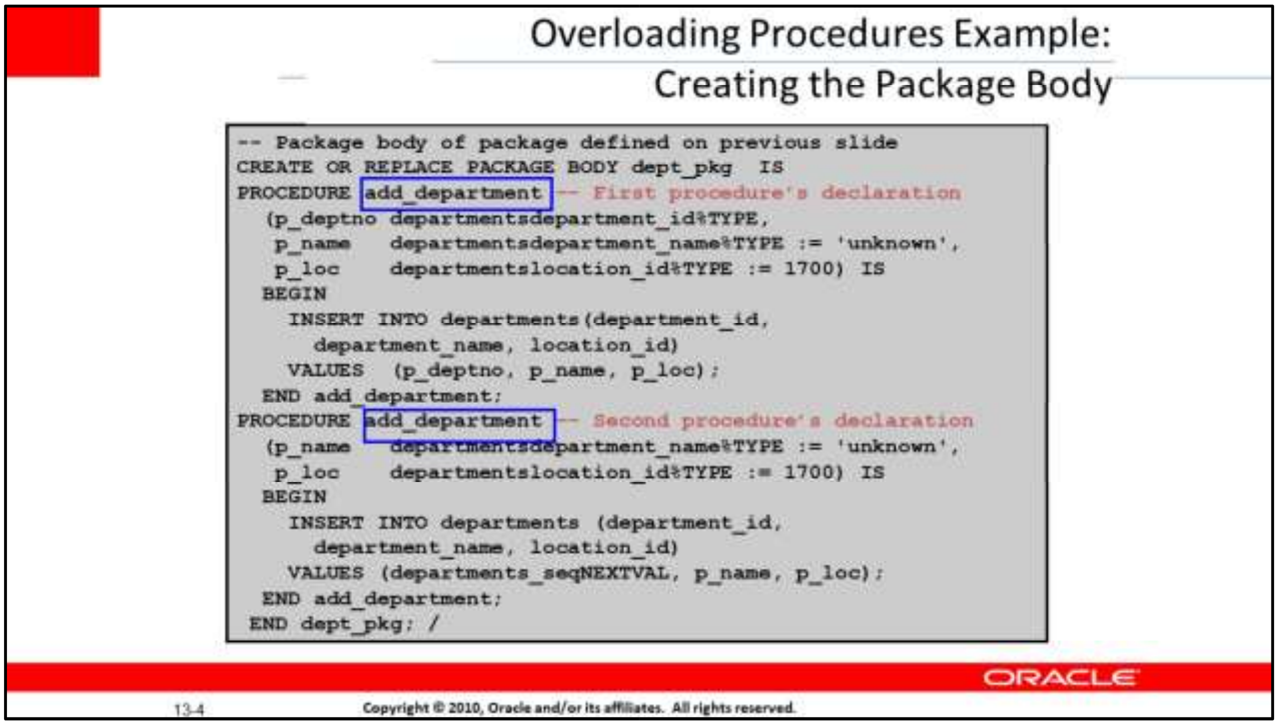
It is better to specify data types using the %TYPE attribute for variables that are used to populate columns in database tables, as shown in the example in the slide; however, you can also specify the data types as follows:

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

Overloading: Example (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:

```
EXECUTE dept_pkgadd_department(980,'Education',2500)
SELECT * FROM departments
```

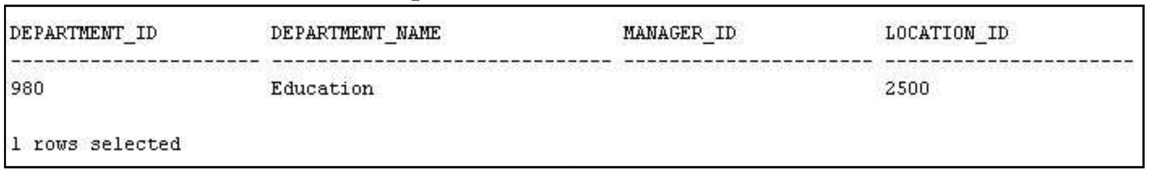


Overloading Procedures Example: Creating the Package Body

```
-- 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 departmentsdepartment_id%TYPE,
   p_name   departmentsdepartment_name%TYPE := 'unknown',
   p_loc    departmentslocation_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   departmentsdepartment_name%TYPE := 'unknown',
   p_loc    departmentslocation_id%TYPE := 1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seqNEXTVAL, p_name, p_loc);
  END add_department;
 END dept_pkg; /
```

13-4 Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

WHERE department_id = 980;



```
DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- ----------------------
980                    Education                                             2500

1 rows selected
```

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

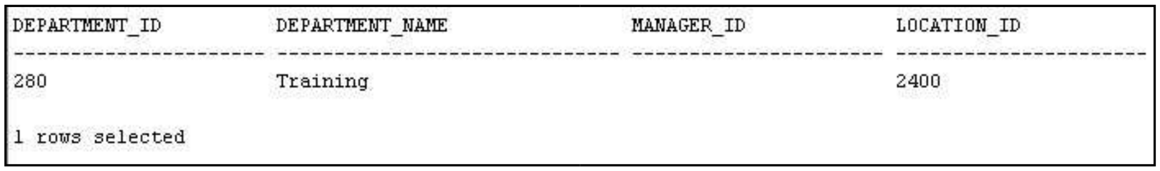
EXECUTE dept_pkgadd_department ('Training', 2400)

SELECT * FROM departments

WHERE department_name = 'Training';

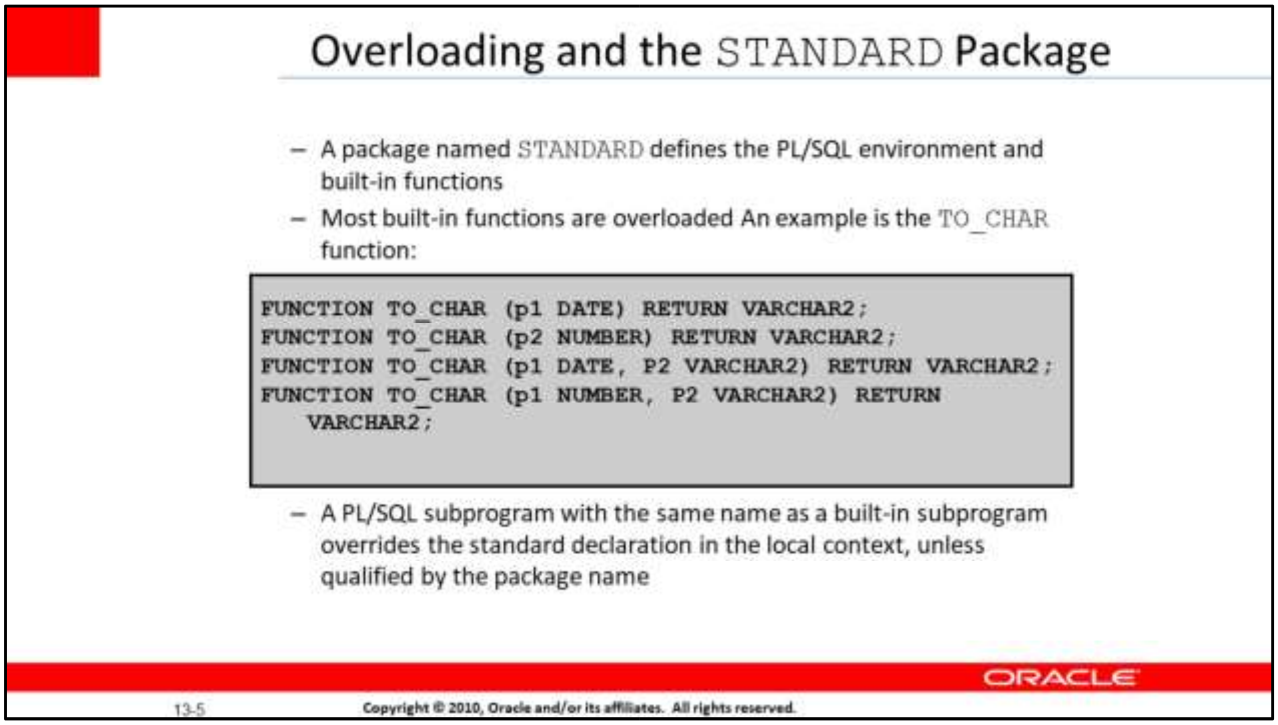
Overloading and the STANDARD Package

A package named STANDARD defines the PL/SQL environment and globally declares types, exceptions, and subprograms that are available automatically to PL/SQL programs Most of the built-in functions that are found in the STANDARD package are overloaded For example, the TO_CHAR function has four different declarations, as shown in the slide The TO_CHAR function can take either the DATE or the NUMBER data type and convert it to the character data type The format to which the date or number has to be converted can also be specified in the function call



```
DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- ----------------------
280                    Training                                              2400

1 rows selected
```



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 qualified by the package name

13-5 Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

If you re-declare a built-in subprogram in another PL/SQL program, then your local declaration overrides the standard or built-in subprogram To be able to access the built-in subprogram, you must qualify it with its package name For example, if you re-declare the TO_CHAR function to access the built-in function, you refer to it as STANDARDTO_CHAR

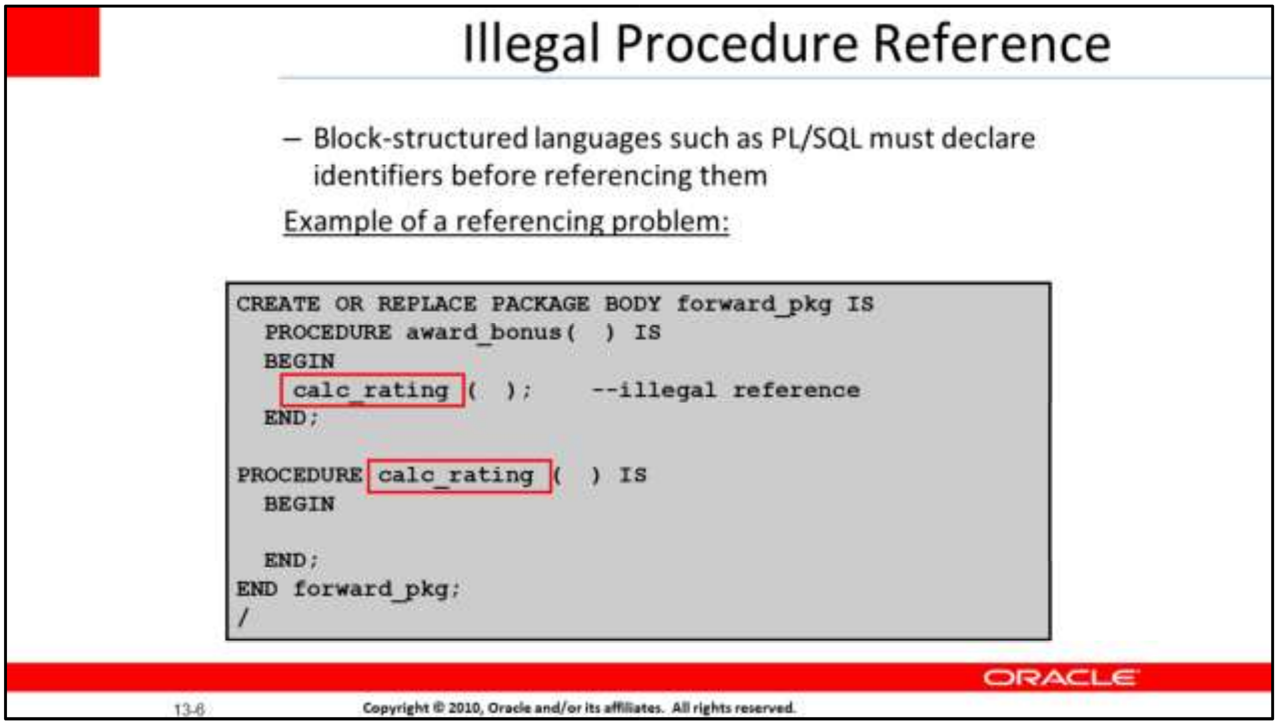
If you re-declare a built-in subprogram as a stand-alone subprogram, then, to access your subprogram, you must qualify it with your schema name: for example, SCOTTTO_CHAR

In the example in the slide, PL/SQL resolves a call to TO_CHAR by matching the number and data types of the formal and actual parameters

Using Forward Declarations

In general, PL/SQL is like other block-structured languages and does not allow forward references You must declare an identifier before using it For example, a subprogram must be declared before you can call it

Coding standards often require that subprograms be kept in alphabetical sequence to make them easy to find In this case, you may encounter problems, as shown in the slide, where the calc_rating procedure cannot be referenced because it has not yet been declared



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;
/
```

13-6 Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

You can solve the illegal reference problem by reversing the order of the two procedures However, this easy solution does not work if the coding rules require subprograms to be declared in alphabetical order

The solution in this case is to use forward declarations provided in PL/SQL A forward declaration enables you to declare the heading of a subprogram, that is, the subprogram specification terminated by a semicolon Note:

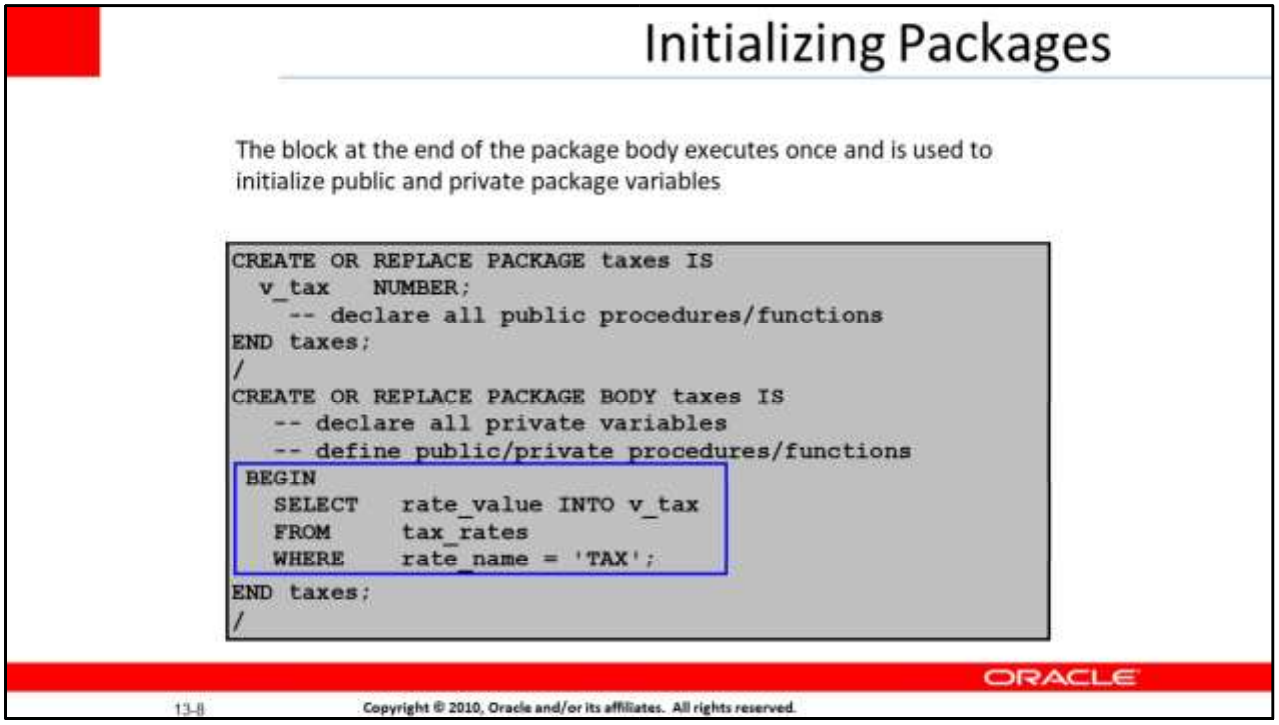
The compilation error for calc_rating occurs only if calc_rating is a private packaged procedure If calc_rating is declared in the package specification, it is already declared as if it is a forward declaration, and its reference can be resolved by the compiler

Using Forward Declarations (continued)

As previously mentioned, PL/SQL enables you to create a special subprogram declaration called a forward declaration A forward declaration may be required for private subprograms in the package body, and consists of the subprogram specification terminated by a semicolon 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



Initializing Packages

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
  v_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 v_tax
   FROM     tax_rates
   WHERE    rate_name = 'TAX';
END taxes;
/
```

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 program unit

Forward Declarations and Packages

Typically, the subprogram specifications go in the package specification, and the subprogram bodies go in the package body The public subprogram declarations in the package specification do not require forward declarations

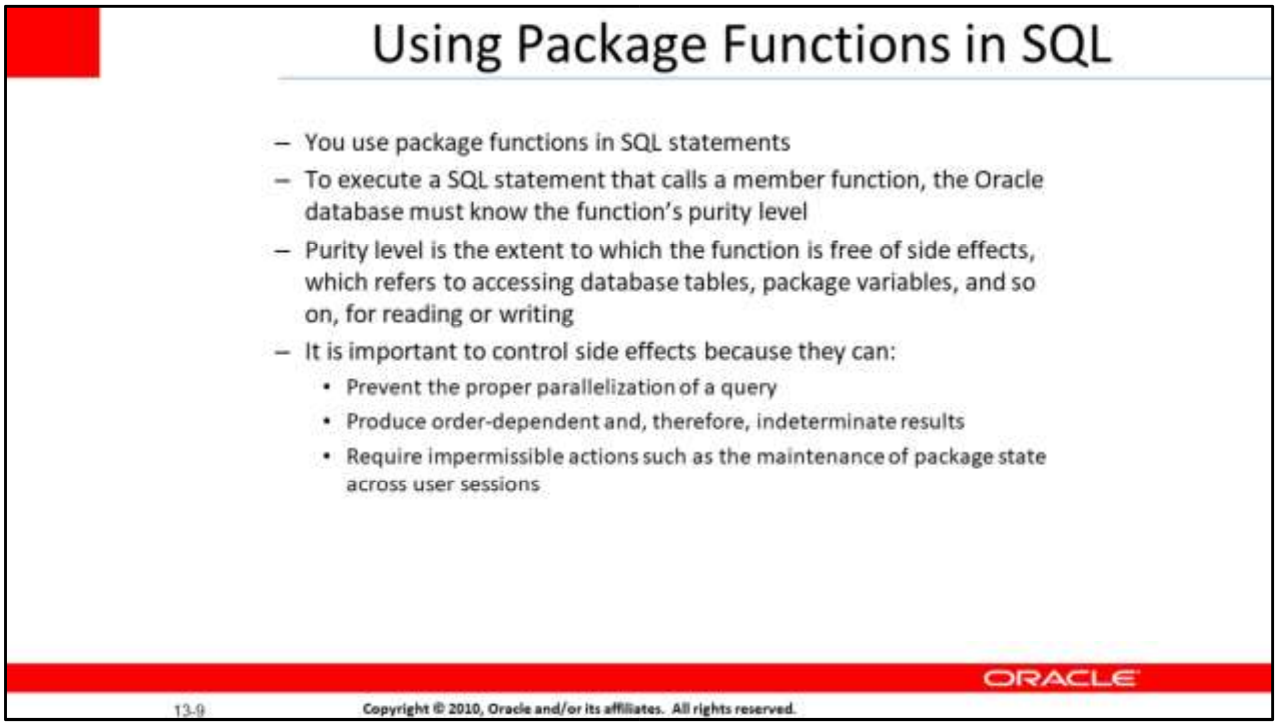
Package Initialization Block

The first time a component in a package is referenced, the entire package is loaded into memory for the user session By default, the initial value of variables is NULL (if not explicitly initialized) To initialize package variables, you can:

- Use assignment operations in their declarations for simple initialization tasks
- Add code block to the end of a package body for more complex initialization tasks

Consider the block of code at the end of a package body as a package initialization block that executes once, when the package is first invoked within the user session
The example in the slide shows the v_tax public variable being initialized to the value in the tax_rates table the first time the taxes package is referenced
Note: If you initialize the variable in the declaration by using an assignment operation, it is overwritten by the code in the initialization block at the end of the package body The initialization block is terminated by the END keyword

for the package body



Using Package Functions in SQL

- You use package functions in SQL statements
- To execute a SQL statement that calls a member function, the Oracle database must know the function's purity level
- Purity level is the extent to which the function is free of side effects, which refers to accessing database tables, package variables, and so on, for reading or writing
- It is important to control side effects because they can:
  - Prevent the proper parallelization of a query
  - Produce order-dependent and, therefore, indeterminate results
  - Require impermissible actions such as the maintenance of package state across user sessions

ORACLE

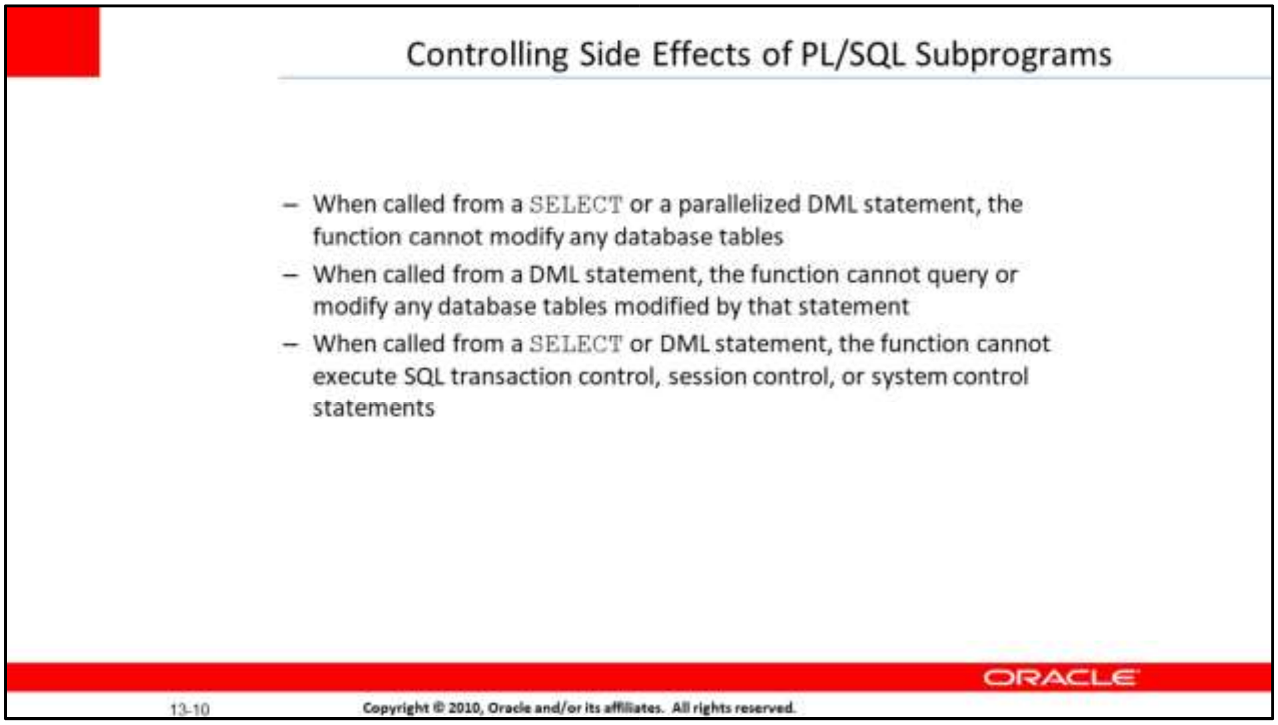
13-9

Using Package Functions in SQL and Restrictions

To execute a SQL statement that calls a stored function, the Oracle Server must know the purity level of the function, or the extent to which the function is free of side effects The term side effect refers to accessing database tables, package variables, and so forth for reading or writing It is important to control side effects because they can prevent the proper parallelization of a query, produce order-dependent and therefore indeterminate results, or require impermissible actions such as the maintenance of package state across user sessions

In general, restrictions are changes to database tables or public package variables (those declared in a package specification) Restrictions can delay the execution of a query, yield order-dependent (therefore indeterminate) results, or require that the package state variables be maintained across user sessions

Various restrictions are not allowed when a function is called from a SQL query or a DML statement



Controlling Side Effects of PL/SQL Subprograms

- When called from a SELECT or a parallelized DML statement, the function cannot modify any database tables
- When called from a DML statement, the function cannot query or modify any database tables modified by that statement
- When called from a SELECT or DML statement, the function cannot execute SQL transaction control, session control, or system control statements

13-10 Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

Controlling Side Effects of PL/SQL Subprograms

The fewer side effects a function has, the better it can be optimized within a query, particularly when the PARALLEL_ENABLE or DETERMINISTIC hints are used

To be callable from SQL statements, a stored function (and any subprograms that it calls) must obey the purity rules listed in the slide The purpose of those rules is to control side effects

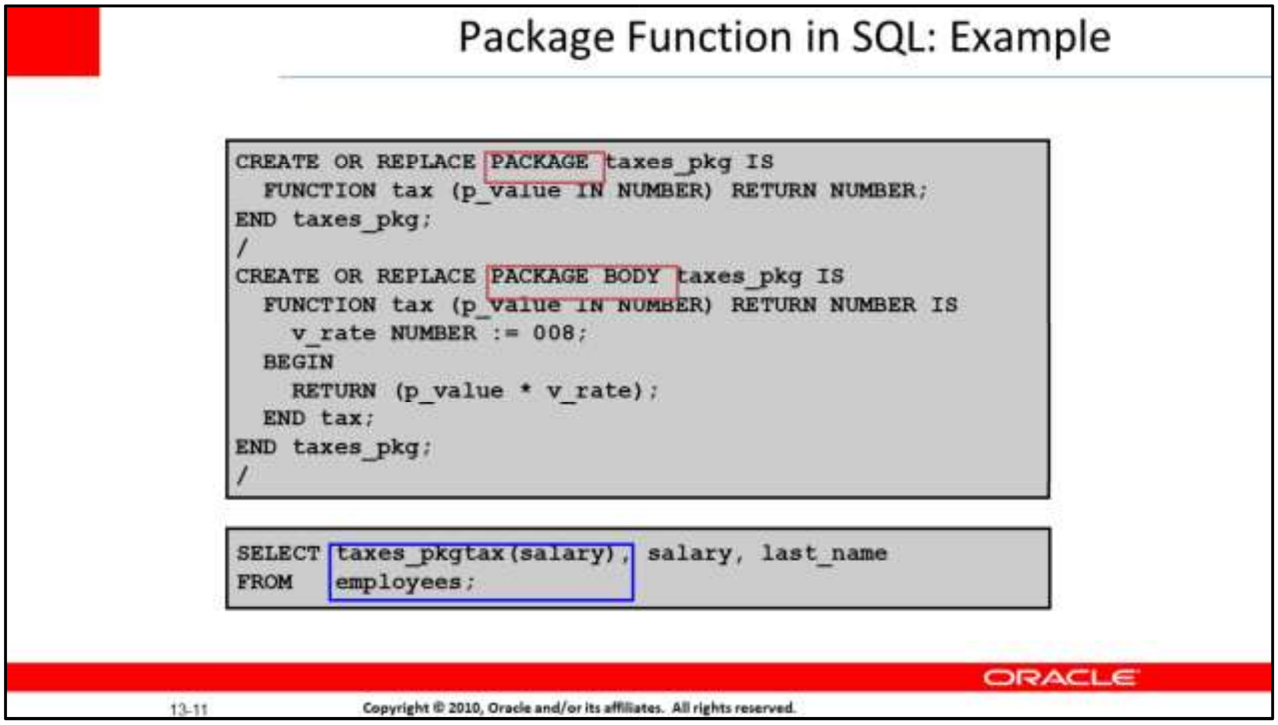
If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed)

To check for purity rule violations at compile time, use the RESTRICT_REFERENCES pragma to assert that a function does not read or write database tables or package variables

Note

In the slide, a DML statement refers to an INSERT, UPDATE, or DELETE statement

For information about using the RESTRICT_REFERENCES pragma, refer to the Oracle Database PL/SQL Language Reference



Package Function in SQL: Example

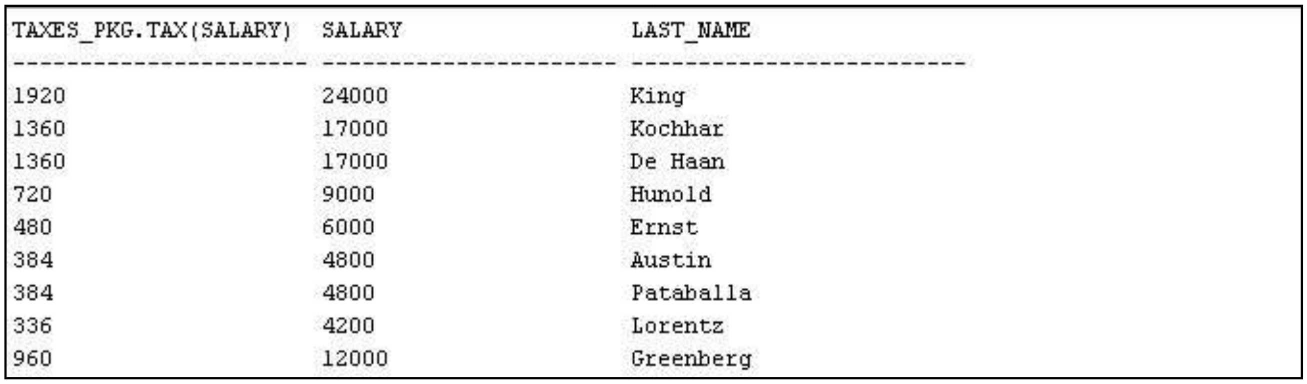
```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  FUNCTION tax (p_value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
  FUNCTION tax (p_value IN NUMBER) RETURN NUMBER IS
    v_rate NUMBER := 008;
  BEGIN
    RETURN (p_value * v_rate);
  END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkgtax(salary), salary, last_name
FROM   employees;
```

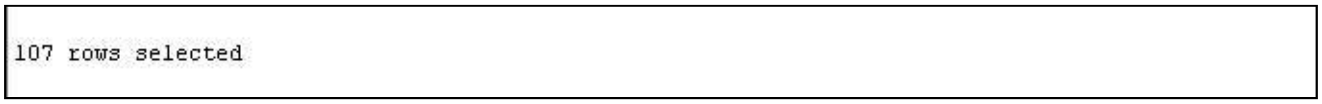
13-11

Package Function in SQL: Example

- 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                    12000                  Greenberg
```



```
107 rows selected
```