Release Date: August, 2016

Updates:

Database Programming with PL/SQL

10-3
Advanced Package Concepts

ORACLE ACADEMY

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

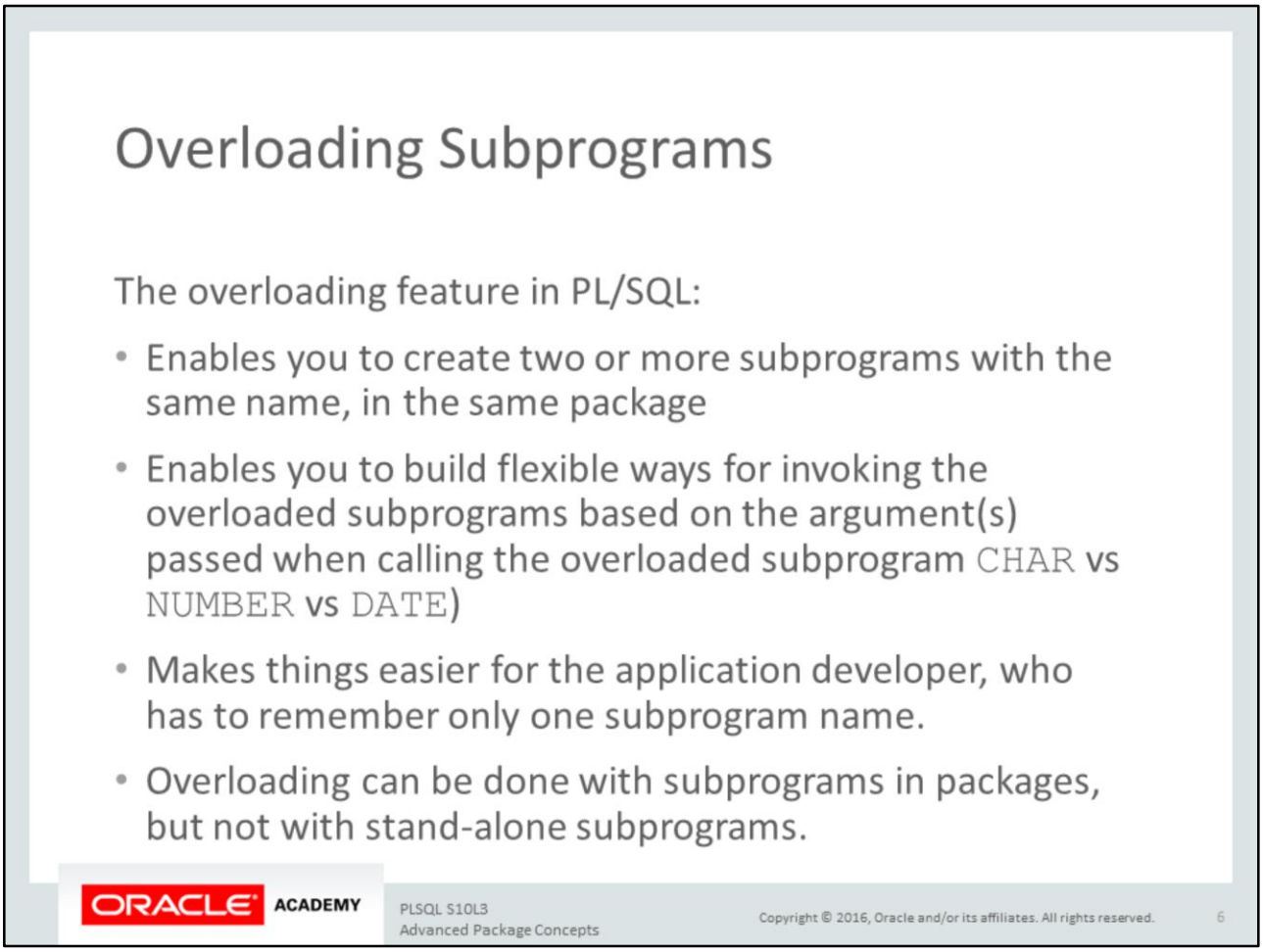
Purpose

- 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;
...
```



Overloaded subprograms have the same name, but are differentiated based on their parameters which must differ in quantity, order, or category of data type. When calling an overloaded subprogram, Oracle will determine which of the overloaded subprograms to use based on the argument(s) passed when calling the overloaded subprogram.

Overloading Subprograms

- Consider using overloading when the purposes of two or more subprograms are similar, but the type or number of parameters required 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 data type.
- 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;
```


ANSWER: The second one. The call's first actual parameter is a character string, not a number or a date, and the second overloaded program is the only one with a character data type category (in this case, VARCHAR2).

Note: The overloaded programs use basic data types for their arguments for clarity and simplicity. It is better to specify data types using the %TYPE attribute for variables that correspond to database table columns.

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

- 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

```
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

- 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

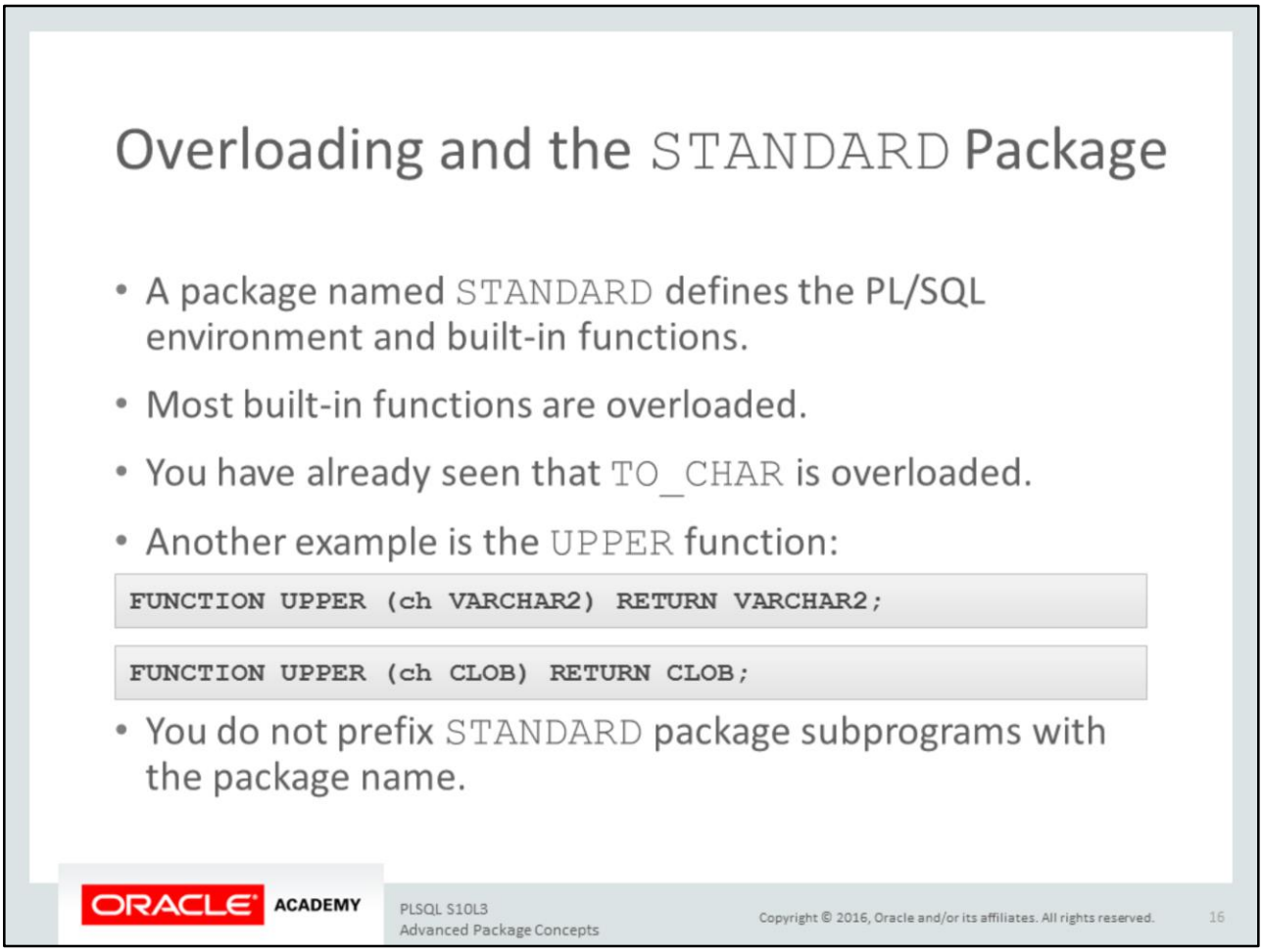
Overloading: Example 3

- 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



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 UPPER function has two different declarations, as shown in the slide.

The contents of package STANDARD are directly visible to applications. You do not need to qualify references to its contents by prefixing the package name.

Why not a third variant to accept and return a CHAR? Because CHAR and VARCHAR2 are in the same category. Although a CLOB stores character data, it is in a different category (see Section 11 for more information).

Overloading and the STANDARD Package

- 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;
```


Oracle strongly recommends that you do NOT create a function with the same name as one of the built-in functions.

Using Forward Declarations

- Block-structured languages (such as PL/SQL) must declare identifiers before referencing them.
- In the example below, if `award_bonus` and `calc_rating` are private, what will happen?

```
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;
```


ANSWER: The body will fail to compile because CALC_RATING is referenced (in AWARD_BONUS) before it has been declared. You have the same problem if AWARD_BONUS is public (meaning it is listed in the package specification) and CALC_RATING is private.

Using Forward Declarations

- All identifiers must be declared before being used, so you could 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 have the problem on the previous slide.
- Using forward declarations can solve this problem.

Note: If CALC_RATING is public (meaning it is listed in the package specification), the error does not occur because the compiler can resolve the reference.

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 (...);      -- resolved by forward declaration
   ...
 END;

 PROCEDURE calc_rating (...) IS -- implementation
 BEGIN
   ...
 END;
END forward_pkg;
```

Using Forward Declarations

Forward declarations help to:

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

Using Forward Declarations

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?

Your first thought might be something like this:

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
 g_tax  NUMBER := (SELECT rate_value INTO g_tax FROM tax_rates ...);
 …
END taxes_pkg;
```

Unfortunately, this doesn't work. You can't use SQL in a variable declaration.

Package Initialization Block

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

```
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 -- unnamed initialization block
  SELECT  rate_value INTO g_tax
   FROM   tax_rates
   WHERE  rate_name = 'TAX';
END taxes_pkg;
```


The unnamed initialization block is terminated by the END keyword for the package body.

Note: If you initialize the variable in the declaration by using an assignment operation, it is overwritten by the code in the unnamed initialization block at the end of the package body. This would not be a good programming practice. If a variable is to be initialized by the unnamed initialization block, do not set a default value when the variable is declared.

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.

Bodiless Packages

- Because it has no body, a bodiless package cannot contain any executable code: no procedures or functions.
- It can contain public (global) variables.
- Bodiless packages are often used to give names to unchanging constants, or to give names to non-predefined Oracle Server exceptions.

The main purpose of a bodiless package is to create standardized names which can be used everywhere in your organization.

Remember that all public (global) constructs in a package can be invoked from outside the package. Also remember that exceptions (like cursors) are a kind of variable.

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;
```


As with any other PL/SQL procedure, function, or package, a user must be granted EXECUTE privilege in order to use a bodiless package.

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;
```

- If we did not define these exceptions in a bodiless package, how else could we define them?

ANSWER: We could define the exceptions in every subprogram which invokes them. But do you really want the same exact code in multiple subprograms? What if the exception needs to be changed?

Invoking a Bodiless Package

- The block below converts 5,000 miles to kilometers using the constant defined in the GLOBAL_CONSTS package.

```
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;
```

- To test this code, create the GLOBAL_CONSTS package using the code on slide #27, then run the code above.

Invoking a Bodiless Package

- The block below uses the exception defined in the OUR_EXCEPTIONS package.

```
BEGIN
 INSERT INTO excep_test (number_col) VALUES (12345);
EXCEPTION
 WHEN our_exceptions.e_value_too_large THEN
  DBMS_OUTPUT.PUT_LINE('Value too big for column data
            type');
END;
```

- To test this code, create the OUR_EXCEPTIONS package using the code on slide #28, then create the EXCEP_TEST table using:

```
CREATE TABLE excep_test (number_col NUMBER(3));
```

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.

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;
```


This function works fine because it does not break any of the rules on the previous slide.

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(100), salary, last_name
 FROM employees;
```


This example breaks the rule that a function called from a query must not execute DML. Imagine that when the SELECT statement that calls the function SAL is executed, employee 100 has a salary of 10,000. What value would be returned in the second column (salary) of the query's output? Would it be 10,000, as it was when the SELECT statement started, or would it be 20,000, the result of running the function SAL? This ambiguity is the reason this code would result in an error.

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.
- 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

- 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;
END sel_emp_dept;
```

Using a User-defined Type as a Parameter

- 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

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

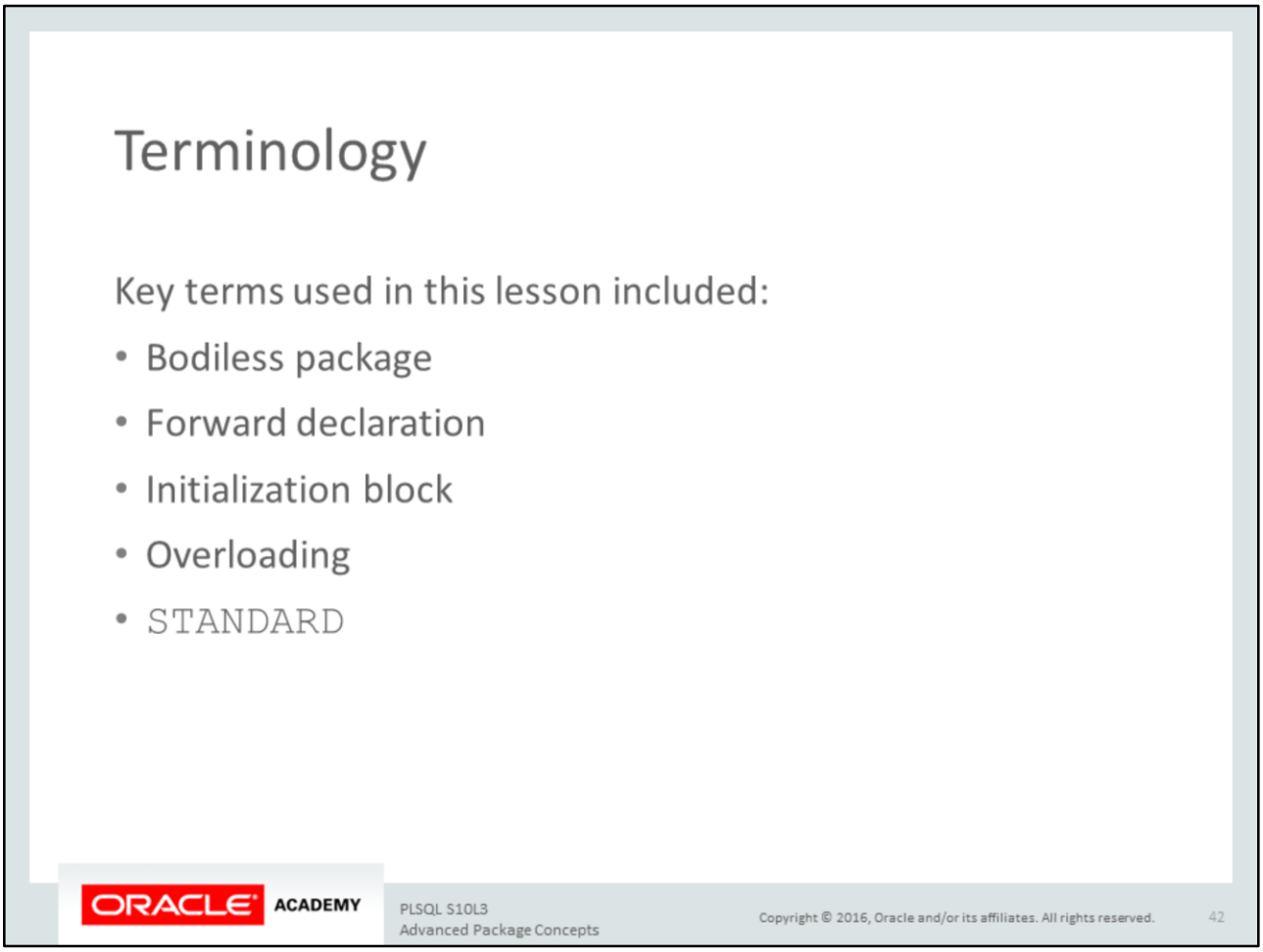
The procedure uses a cursor to populate the INDEX BY table with employee rows, and return this data in a single OUT parameter.

```
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

By creating the EMP_PKG package, the entire EMPLOYEES table can be fetched with a single procedure call where ever it is needed.

```
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;
```



Bodiless Package – A package that has a specification but no executable code containing only public variables.

Forward Declaration – Defines subprograms in logical or alphabetical order, defines mutually recursive subprograms, and groups and logically organizes subprograms in a package body.

Initialization Block – An un-named block at the end of the package body that automatically executes once and is used to initialize public and private package variables.

Overloading – Enables you to develop two or more packaged subprograms with the same name.

STANDARD – A package that defines the PL/SQL environment and globally declares types, exceptions, and subprograms that are available automatically to PL/SQL programs.

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


ORACLE
ACADEMY