Packages

Oracle Academy Study Materials

What Are PL/SQL Packages?

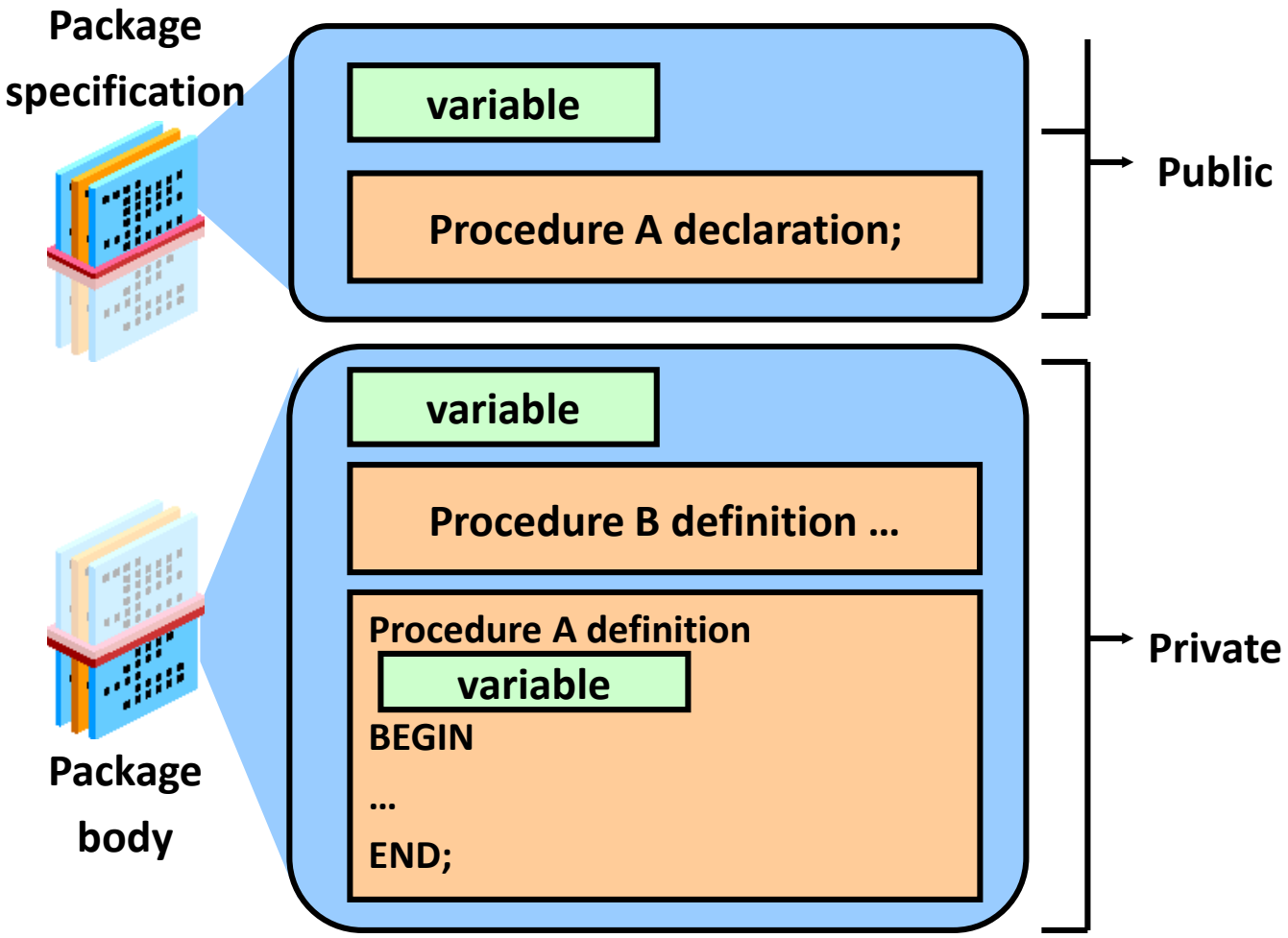
- A package is a schema object that groups logically related PL/SQL types, variables, and subprograms.
- Packages usually have two parts:
  - A specification (spec)
  - A body
- The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- The body defines the queries for the cursors and the code for the subprograms.
- Enable the Oracle server to read multiple objects into memory at once.

Packages

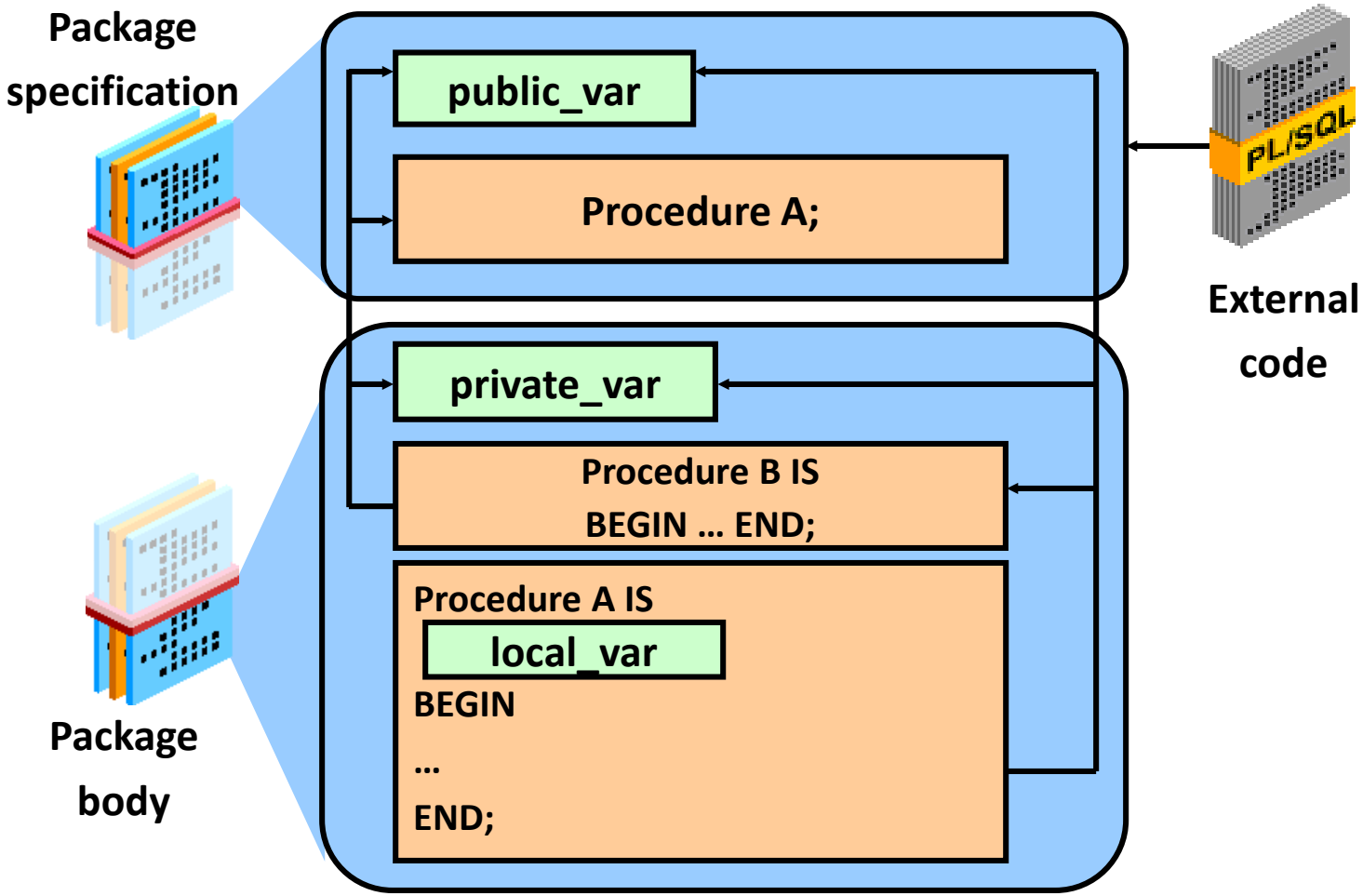
Advantages of Using Packages

- Modularity: Encapsulating related constructs
- Easier maintenance: Keeping logically related functionality together
- Easier application design: Coding and compiling the specification and body separately
- Hiding information:
  - Only the declarations in the package specification are visible and accessible to applications
  - Private constructs in the package body are hidden and inaccessible
  - All coding is hidden in the package body

Components of a PL/SQL Package



The Visibility of a Package's Components



Packages

Creating the Package Specification Using the CREATE PACKAGE Statement

```
CREATE [OR REPLACE] PACKAGE package_name IS|AS
    public type and variable declarations
    subprogram specifications
END [package_name];
```

- The OR REPLACE option drops and re-creates the package specification.
- Variables declared in the package specification are initialized to NULL by default.
- All the constructs declared in a package specification are visible to users who are granted privileges on the package.
- Including the package name after the END keyword is optional.

Packages

– Example of a Package Specification: comm_pkg

```
-- The package spec with a public variable and a
-- public procedure that are accessible from
-- outside the package.

CREATE OR REPLACE PACKAGE comm_pkg IS
  v_std_comm NUMBER := 0.10;  --initialized to 0.10
  PROCEDURE reset_comm(p_new_comm NUMBER);
END comm_pkg;
/
```

– V_STD_COMM is a public global variable initialized to 0.10.

– RESET_COMM is a public procedure used to reset the standard commission based on some business rules. It is implemented in the package body.

Creating the Package Body

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS
    private type and variable declarations
    subprogram bodies
[BEGIN initialization statements]
END [package_name];
```

- The OR REPLACE option drops and re-creates the package body.
- Identifiers defined in the package body are private and not visible outside the package body.
- All private constructs must be declared before they are referenced.
- Public constructs are visible to the package body.
- If changes to the code are needed, the body can be edited and recompiled without having to edit or recompile the specification.
- Every subprogram declared in the package specification must also be included in the package body.

Example of a Package Body: comm_pkg

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS

FUNCTION validate(p_comm NUMBER) RETURN BOOLEAN IS
    v_max_comm       employees.commission_pct%type;
  BEGIN
    SELECT MAX(commission_pct) INTO v_max_comm
    FROM   employees;
    RETURN (p_comm BETWEEN 0.0 AND v_max_comm);
  END validate;

  PROCEDURE reset_comm (p_new_comm NUMBER) IS BEGIN
    IF validate(p_new_comm) THEN
      v_std_comm := p_new_comm; -- reset public var
    ELSE  RAISE_APPLICATION_ERROR(
            -20210, 'Bad Commission');
    END IF;
  END reset_comm;
END comm_pkg;
```

Invoking the Package Subprograms: Examples

```
-- Invoke a function within the same packages:
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...
  PROCEDURE reset_comm(p_new_comm NUMBER) IS
  BEGIN
    IF validate(p_new_comm) THEN
      v_std_comm := p_new_comm;
    ELSE ...
    END IF;
  END reset_comm;
END comm_pkg;
```

```
-- Invoke a package procedure from SQL*Plus:
EXECUTE comm_pkg.reset_comm(0.15)
```

```
-- Invoke a package procedure in a different schema:
EXECUTE scott.comm_pkg.reset_comm(0.15)
```

Changing the Package Body Code

- Changing the Package Body Code – Remarks:
- You must edit and recompile the package body, but you do not need to recompile the specification unless the name or parameters have changed.
- Remember, the specification can exist without the body (but the body cannot exist without the specification).
- Because the specification is not recompiled, you do not need to recompile any applications (or other PL/SQL subprograms) that are already invoking the package procedures.

Creating and Using Bodiless Packages

```
CREATE OR REPLACE PACKAGE global_consts IS
  c_mile_2_kilo    CONSTANT  NUMBER  :=  1.6093;
  c_kilo_2_mile    CONSTANT  NUMBER  :=  0.6214;
  c_yard_2_meter   CONSTANT  NUMBER  :=  0.9144;
  c_meter_2_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;
```

```
BEGIN  DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
        20 * global_consts.c_mile_2_kilo || ' km');
END;
```

```
CREATE FUNCTION mtr2yrd(p_m NUMBER) RETURN NUMBER IS
BEGIN
  RETURN (p_m * global_consts.c_meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

Removing Packages Using SQL DROP Statement

```
-- Remove the package specification and body
DROP PACKAGE package_name;
```

```
-- Remove the package body only
DROP PACKAGE BODY package_name;
```

Packages

– You can DESCRIBE a package in the same way as you can DESCRIBE a table or view:

```
DESCRIBE check_emp_pkg
```

Object Type PACKAGE Object CHECK_EMP_PKG

Package Name	Procedure	Argument	In Out	Datatype
CHECK_EMP_PKG	CHK_DEPT_MGR	P_EMPID	IN	NUMBER
		P_MGR	IN	NUMBER
	CHK_HIREDATE	P_DATE	IN	DATE

– You cannot DESCRIBE individual packaged subprograms, only the whole package.

Viewing Packages Using the Data Dictionary

```
-- View the package specification.
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE';
```

	TEXT
1	PACKAGE comm_pkg IS
2	std_comm NUMBER := 0.10; --initialized to 0.10
3	PROCEDURE reset_comm(new_comm NUMBER);
4	END comm_pkg;

```
-- View the package body.
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE BODY';
```

	TEXT
1	PACKAGE BODY comm_pkg IS
2	FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS
3	max_comm employees.commission_pct%type;
4	BEGIN
5	SELECT MAX(commission_pct) INTO max_comm
6	FROM employees;
7	RETURN (comm BETWEEN 0.0 AND max_comm);

Packages

Guidelines for Writing Packages:

- Develop packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout
  a session or across transactions.
- The fine-grain dependency management reduces the need to recompile referencing subprograms when a package specification changes.
- The package specification should contain as few constructs as possible.

Overloading Subprograms in PL/SQL

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 is useful when you want a subprogram to accept similar sets of parameters that have different data types.

Overloading Subprograms in PL/SQL

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 Subprograms in PL/SQL

– 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 Subprograms in PL/SQL

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 do not prefix STANDARD package subprograms with the package name.
- 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;
...
```

- Another example is the UPPER function:

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

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

Overloading Subprograms in PL/SQL

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.

Overloading Subprograms in PL/SQL

Overloading Restrictions – Example:

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

– The following invocation fails:

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

– The following invocation succeeds:

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

Packages - 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;
/
```

Packages

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;
```

Packages

Using Forward Declarations

- Forward declarations help to:
  - Define subprograms in logical or alphabetical order.
  - 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.

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;
/
```

Packages

– 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 := 0.08;
  BEGIN
    RETURN (p_value * v_rate);
  END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM   employees;
```

Persistent State of Packages

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:
  - Unique to each session
- Subject to change when package subprograms are called or public variables are modified

Packages

Using PL/SQL Tables of Records in Packages

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

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE get_employees(p_emps OUT emp_table_type) IS
    v_i BINARY_INTEGER := 0;
  BEGIN
    FOR emp_record IN (SELECT * FROM employees)
    LOOP
      emps(v_i) := emp_record;
      v_i:= v_i + 1;
    END LOOP;
  END get_employees;
END emp_pkg;
```

Oracle-Supplied Packages

- The Oracle-supplied packages:
  - Are provided with the Oracle server
  - Extend the functionality of the database
  - Enable access to certain SQL features that are normally restricted for PL/SQL
- For example, the DBMS_OUTPUT package was originally designed to debug PL/SQL programs.

Oracle-Supplied Packages

How the DBMS_OUTPUT Package Works:

- The DBMS_OUTPUT package enables you to send messages from stored subprograms and triggers.
- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until the sending subprogram or trigger completes.
- Use SET SERVEROUTPUT ON to display messages in SQL Developer and SQL*Plus.