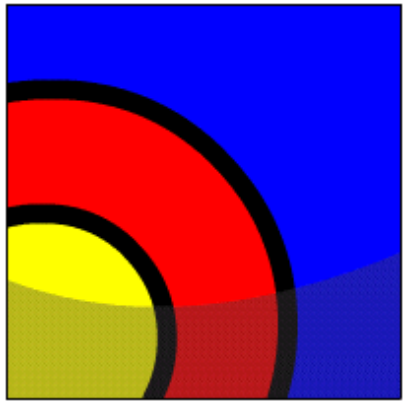
Creating Packages

What Will I Learn?

In this lesson, you will learn to:

- Describe the reasons for using a package
- Describe the two components of a package: specification and body
- Create packages containing related variables, cursors, constants, exceptions, procedures, and functions
- Create a PL/SQL block that invokes a package construct



Why Learn It?

You have already learned how to create and use stored procedures and functions.

Suppose you want to create several procedures and/or functions that are related to each other. An application can use either all of them or none of them.

Wouldn't it be easier to create and manage all the subprograms as a single database object: a package?

In this lesson, you learn what a package is and what its components are. You also begin to learn how to create and use packages.

Tell Me / Show Me

What Are PL/SQL Packages?

PL/SQL packages are containers that enable you to group together related PL/SQL subprograms, variables, cursors, and exceptions.

For example, a Human Resources package can contain hiring and firing procedures, commission and bonus functions, and tax-exemption variables.

Tell Me / Show Me

Components of a PL/SQL Package

A package consists of two parts stored separately in the database:

- Package specification: The interface to your applications. It must be created first. It declares the constructs (procedures, functions, variables, and so on) that are visible to the calling environment.
- Package body: This contains the executable code of the subprograms that were declared in the package specification. It can also contain its own variable declarations.

Package specification

Package body

Tell Me / Show Me

Components of a PL/SQL Package (continued)

The detailed package body code is invisible to the calling environment, which can see only the specification.

If changes to the code are needed, the body can be edited and recompiled without having to edit or recompile the specification.

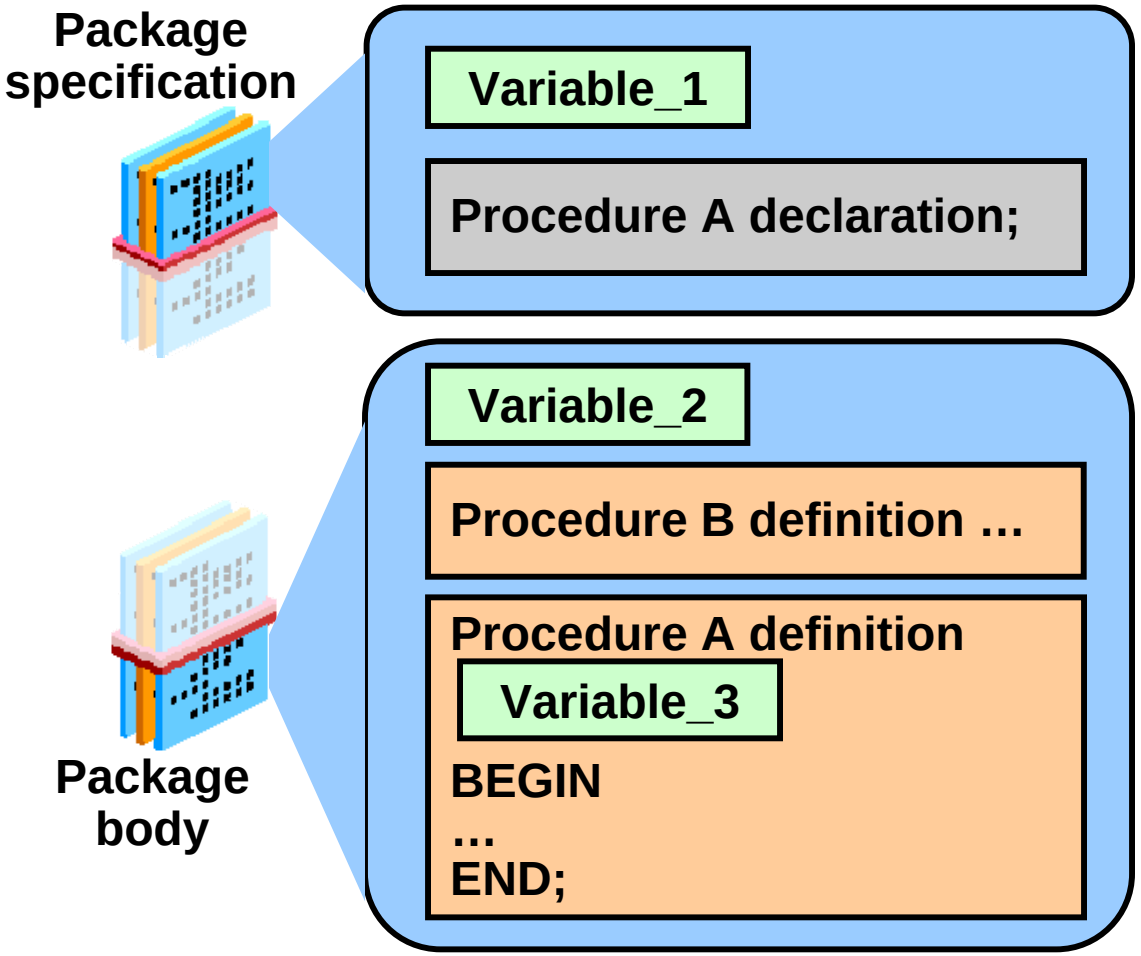
This two-part structure is an example of a modular programming principle called encapsulation.

Package specification

Package body

Tell Me / Show Me

Components of a PL/SQL Package (continued)



Tell Me / Show Me

Syntax for Creating the Package Specification

To create packages, you declare all public constructs within the package specification.

```
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
    public type and variable declarations
    public 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 EXECUTE privilege on the package.

Tell Me / Show Me

Syntax for Creating the Package Specification (continued)

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

- *package_name*: Specifies a name for the package that must be unique among objects within the owning schema. Including the package name after the END keyword is optional.
- *public type and variable declarations*: Declares public variables, constants, cursors, exceptions, user-defined types, and subtypes.
- *public subprogram specifications*: Declares the public procedures and/or functions in the package.

Tell Me / Show Me

Creating the Package Specification

"Public" means that the package construct (variable, procedure, function, and so on) can be seen and executed from outside the package. All constructs declared in the package specification are automatically public constructs.

The package specification should contain procedure and function headings terminated by a semicolon, without the `IS` (or AS) keyword and its PL/SQL block.

The implementation (writing the detailed code) of a procedure or function that is declared in a package specification is done in the package body.

The next two slides show code examples.

Tell Me / Show Me

Example of Package Specification: check_emp_pkg

```
CREATE OR REPLACE PACKAGE check_emp_pkg
IS
  g_max_length_of_service CONSTANT NUMBER := 100;
  PROCEDURE chk_hiredate
    (p_date    IN   employees.hire_date%TYPE);
  PROCEDURE chk_dept_mgr
    (p_empid   IN   employees.employee_id%TYPE,
     p_mgr     IN   employees.manager_id%TYPE);
END check_emp_pkg;
```

- G_MAX_LENGTH_OF_SERVICE is a constant declared and initialized in the specification.
- CHK_HIREDATE and CHK_DEPT_MGR are two public procedures declared in the specification. Their detailed code is written in the package body.

Tell Me / Show Me

Package Specification: A Second Example

```
CREATE OR REPLACE PACKAGE manage_jobs_pkg
IS
  g_todays_date        DATE := SYSDATE;
  CURSOR jobs_curs  IS
    SELECT employee_id, job_id FROM employees
      ORDER BY employee_id;
  PROCEDURE update_job
    (p_emp_id  IN   employees.employee_id%TYPE);
  PROCEDURE fetch_emps
    (p_job_id  IN   employees.job_id%TYPE,
     p_emp_id  OUT  employees.employee_id%TYPE);
END manage_jobs_pkg;
```

Remember that a cursor is a type of variable.

Tell Me / Show Me

Syntax for Creating the Package Body

Create a package body to contain the detailed code for all the subprograms declared in the specification.

```
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.
- "Subprogram bodies" must contain the code of all the subprograms declared in the package specification.
- Private types and variables, and `BEGIN` initialization statements, are discussed in later lessons.

Tell Me / Show Me

Syntax for Creating the Package Body (continued)

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

- *package_name* specifies a name for the package that must be the same as its package specification. Using the package name after the END keyword is optional.
- *subprogram bodies* specifies the full implementation (the detailed PL/SQL code) of all private and/or public procedures or functions.

Tell Me / Show Me

Creating the Package Body

When creating a package body, do the following:

- Specify the `OR REPLACE` option to overwrite an existing package body.
- Define the subprograms in an appropriate order. The basic principle is that you must declare a variable or subprogram before it can be referenced by other components in the same package body.
- Every subprogram declared in the package specification must also be included in the package body.

Tell Me / Show Me

Example of Package Body: check_emp_pkg

```
CREATE OR REPLACE PACKAGE BODY check_emp_pkg IS
 PROCEDURE chk_hiredate
   (p_date    IN   employees.hire_date%TYPE)
   IS BEGIN
     IF MONTHS_BETWEEN(SYSDATE, p_date) >
       g_max_length_of_service * 12 THEN
       RAISE_APPLICATION_ERROR(-20200, 'Invalid Hiredate');
     END IF;
 END chk_hiredate;
 PROCEDURE chk_dept_mgr
   (p_empid   IN   employees.employee_id%TYPE,
    p_mgr     IN   employees.manager_id%TYPE)
   IS BEGIN ...
 END chk_dept_mgr;
END check_emp_pkg;
```

Tell Me / Show Me

Changing the Package Body Code

Suppose now you want to make a change to the `CHK_HIREDATE` procedure, for example, to raise a different error message.

You must edit and recompile the package body, but you do not need to recompile the specification. 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.

Tell Me / Show Me

Recompiling the Package Body: check_emp_pkg

```
CREATE OR REPLACE PACKAGE BODY check_emp_pkg IS
 PROCEDURE chk_hiredate
   (p_date    IN   employees.hire_date%TYPE)
   IS BEGIN
     IF MONTHS_BETWEEN(SYSDATE, p_date) >
       g_max_length_of_service * 12 THEN
       RAISE_APPLICATION_ERROR(-20201, 'Hiredate Too Old');
     END IF;
 END chk_hiredate;
 PROCEDURE chk_dept_mgr
   (p_empid   IN   employees.employee_id%TYPE,
    p_mgr     IN   employees.manager_id%TYPE)
   IS BEGIN ...
 END chk_dept_mgr;
END check_emp_pkg;
```

Tell Me / Show Me

Invoking Subprograms in Packages

You invoke packaged procedures and functions in the same way as non-packaged subprograms, except that you must dot-prefix the subprogram name with the package name. For example:

```
BEGIN
  check_emp_pkg.chk_hiredate('17-Jul-95');
END;
```

What if you forget the names of the procedures or which parameters you have to pass to them?

Tell Me / Show Me

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

1 - 3

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

Tell Me / Show Me

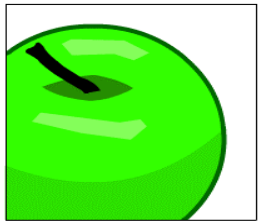
Reasons for Using Packages

- Modularity: Related programs and variables can be grouped together.

- Hiding information: Only the declarations in the package specification are visible to invokers. Application developers do not need to know the details of the package body code.

- Easier maintenance: You can change and recompile the package body code without having to recompile the specification. Therefore, applications that already use the package do not need to be recompiled.

Tell Me / Show Me



Terminology

Key terms used in this lesson include:

PL/SQL packages

Package specification

Package body

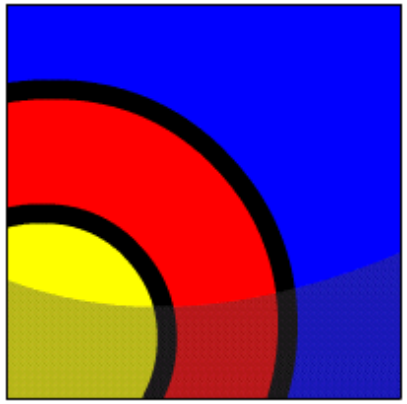
Encapsulation

OR REPLACE

Summary

In this lesson, you learned to:

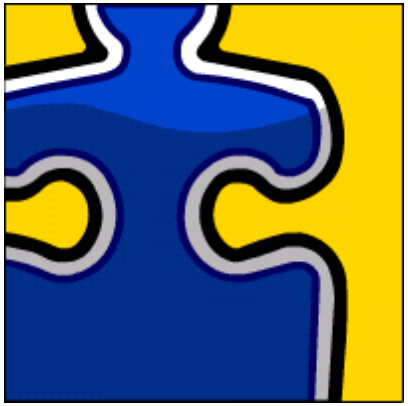
- Describe the reasons for using a package
- Describe the two components of a package: specification and body
- Create packages containing related variables, cursors, constants, exceptions, procedures, and functions
- Create a PL/SQL block that invokes a package construct



Try It / Solve It

The exercises in this lesson cover the following topics:

- Describing packages and listing their components
- Identifying a package specification and body
- Creating packages containing related variables, cursors, constants, exceptions, procedures, and functions
- Invoking a package construct

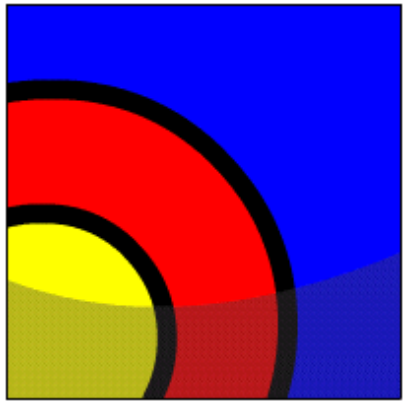


Managing Package Concepts

What Will I Learn?

In this lesson, you will learn to:

- Explain the difference between public and private package constructs
- Designate a package construct as either public or private
- Specify the appropriate syntax to drop packages
- Identify views in the Data Dictionary that manage packages
- Identify guidelines for using packages



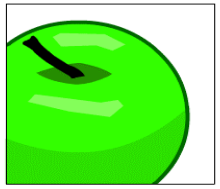
Why Learn It?

How would you create a procedure or function that cannot be invoked directly from an application (maybe for security reasons), but can be invoked only from other PL/SQL subprograms?

You would create a private subprogram within a package.

In this lesson, you learn how to create private subprograms. You also learn how to drop packages, and how to view them in the Data Dictionary. You also learn about the additional benefits of packages.

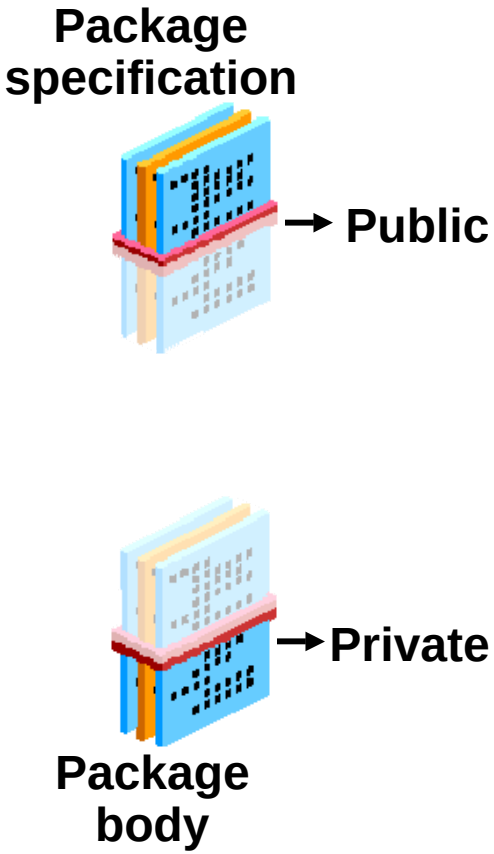
Tell Me / Show Me



Components of a PL/SQL Package

- Public components are declared in the package specification. You can invoke public components from any calling environment, provided the user has been granted EXECUTE privilege on the package.

- Private components are declared only in the package body and can be referenced only by other (public or private) constructs within the same package body. Private components can reference the package's public components.



Tell Me / Show Me

Visibility of Package Components

The *visibility* of a component describes whether that component can be seen, that is, referenced and used by other components or objects. Visibility of components depends on where they are declared. You can declare components in three places within a package:

- Globally in the specification: These components are visible throughout the package body, and by the calling environment
- Locally in the package body, but outside any subprogram: These components are visible throughout the package body, but not by the calling environment
- Locally in the package body, within a specific subprogram: These components are visible only within that subprogram.

Tell Me / Show Me

Global/Local Compared to Public/Private:

Remember that public components declared in the specification are visible to the calling environment, while private components declared only within the body are not. Therefore all public components are global, while all private components are local.

So what's the difference between public and global, and between private and local?

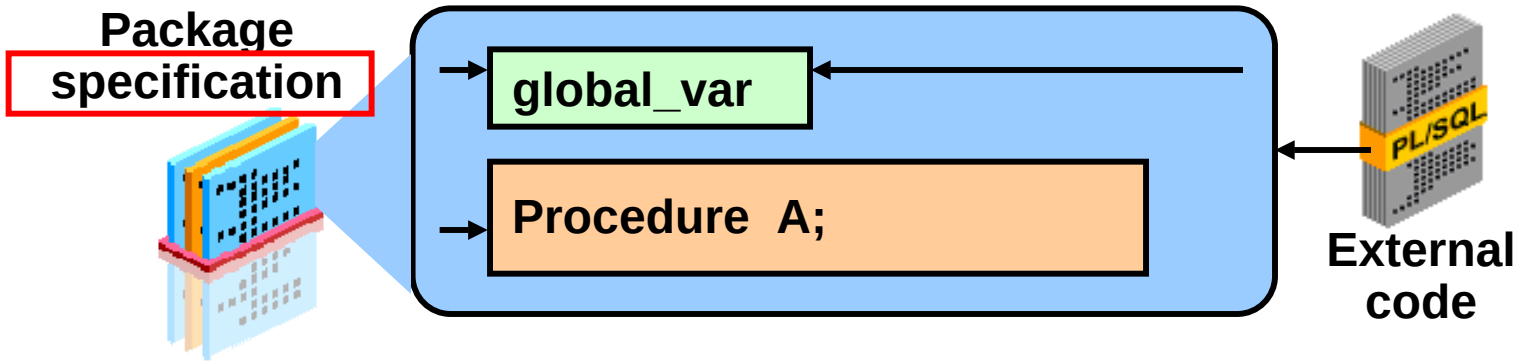
The answer is none, really. They are the same thing! But you use public/private when describing procedures and functions, and global/local when describing other components, such as variables, constants, and cursors.

Tell Me / Show Me

Visibility of Global (Public) Components

Globally declared components are visible internally and externally to the package, such as:

- A global variable declared in a package specification can be referenced and changed outside the package (for example, `global_var` can be referenced externally).
- A public subprogram declared in the specification can be called from external code sources (for example, `Procedure A` can be called from an environment external to the package).

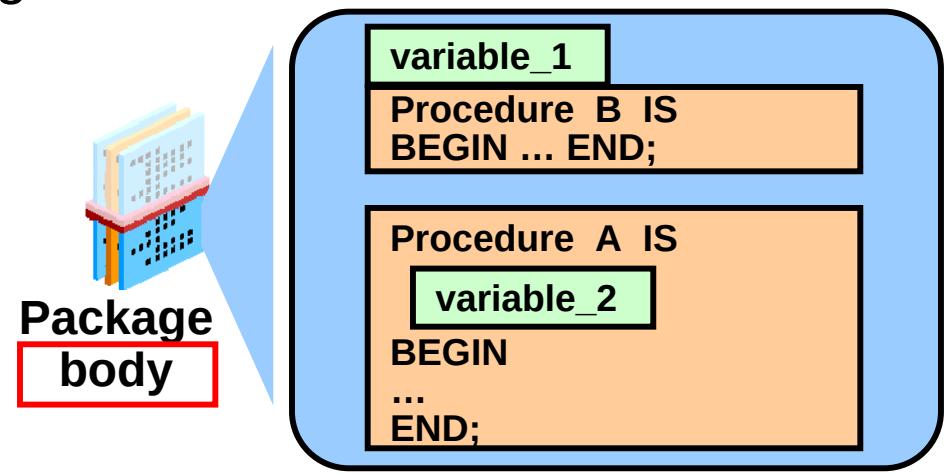


Tell Me / Show Me

Visibility of Local (Private) Components

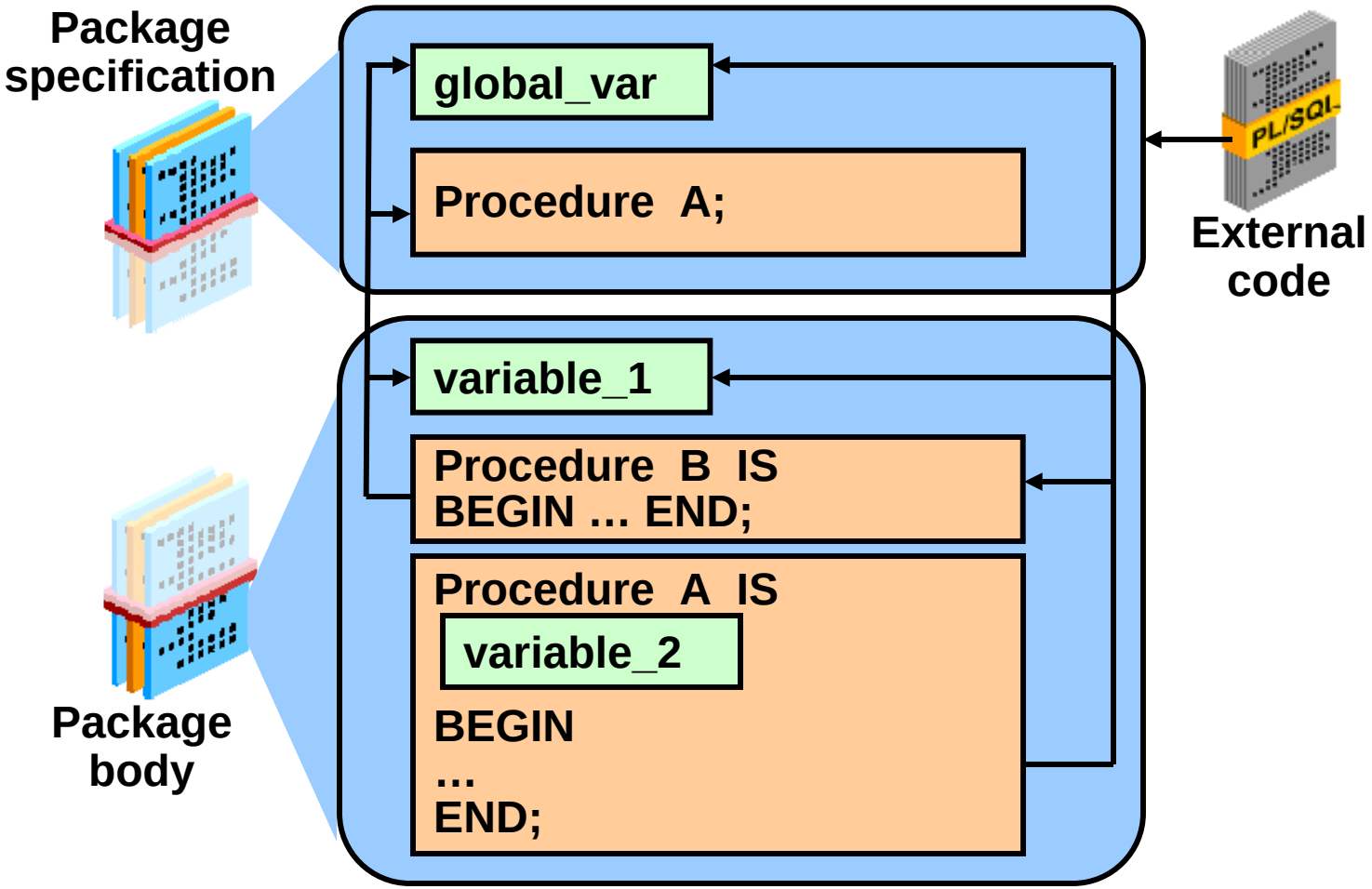
Local components are visible only within the structure in which they are declared, such as the following:

- Local variables defined within a specific subprogram can be referenced only within that subprogram, and are not visible to external components.
- Local variables that are declared in a package body can be referenced by other components in the same package body. They are not visible to any subprograms or objects that are outside the package.



Tell Me / Show Me

Note: Private subprograms, such as `Procedure B`, can be invoked only with public subprograms, such as `Procedure A`, or other private package constructs.



Tell Me / Show Me

Example of Package Specification: `salary_pkg`:

You have a business rule that no employee's salary can be increased by more than 20 percent at one time.

```
CREATE OR REPLACE PACKAGE salary_pkg
IS
  g_max_sal_raise CONSTANT NUMBER := 0.20;
  PROCEDURE update_sal
     (p_employee_id   employees.employee_id%TYPE,
      p_new_salary    employees.salary%TYPE);
END salary_pkg;
```

- `g_max_sal_raise` is a global constant initialized to `0.20`.
- `update_sal` is a public procedure that updates an employee's salary.

Tell Me / Show Me

Example of Package Body: salary_pkg:

```
CREATE OR REPLACE PACKAGE BODY salary_pkg IS
  FUNCTION validate_raise  -- private function
    (p_old_salary employees.salary%TYPE,
     p_new_salary employees.salary%TYPE)
  RETURN BOOLEAN IS
  BEGIN
    IF  p_new_salary >
       (p_old_salary * (1 + g_max_sal_raise)) THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
  END validate_raise;

  -- next slide shows the public procedure
```

Tell Me / Show Me

```
...
  PROCEDURE update_sal   -- public procedure
    (p_employee_id   employees.employee_id%TYPE,
     p_new_salary    employees.salary%TYPE)
  IS  v_old_salary   employees.salary%TYPE; -- local variable
  BEGIN
    SELECT salary INTO v_old_salary FROM employees
       WHERE employee_id = p_employee_id;
    IF validate_raise(v_old_salary, p_new_salary) THEN
       UPDATE employees SET salary = p_new_salary
          WHERE employee_id = p_employee_id;
    ELSE
      RAISE_APPLICATION_ERROR(-20210, 'Raise too high');
    END IF;
  END update_sal;
END salary_pkg;
```

Tell Me / Show Me

Invoking Package Subprograms

After the package is stored in the database, you can invoke subprograms stored within the same package or stored in another package.

Within the same package	Specify the subprogram name
	`Subprogram;`
	You can fully qualify a subprogram within the same package, but this is optional
	`package_name.subprogram;`

External to the package	Fully qualify the (public) subprogram with its package name `package_name.subprogram;`

Tell Me / Show Me

Which of the following invocations from outside the `salary_pkg` are valid (assuming the caller either owns or has EXECUTE priviledge on the package)?

```
DECLARE
  v_bool    BOOLEAN;
  v_number  NUMBER;
BEGIN
  salary_pkg.update_sal(100,25000);                    -- 1
  update_sal(100,25000);                               -- 2
  v_bool := salary_pkg.validate_raise(24000,25000); -- 3
  v_number := salary_pkg.g_max_sal_raise;              -- 4
  v_number := salary_pkg.v_old_salary;                 -- 5
END;
```

Tell Me / Show Me

Removing Packages

- To remove the entire package, specification, and body, use the following syntax:

```
DROP PACKAGE package_name;
```

- To remove only the package body, use the following syntax:

```
DROP PACKAGE BODY package_name;
```

- You cannot remove the package specification on its own.

Tell Me / Show Me

Viewing Packages in the Data Dictionary

The source code for PL/SQL packages is maintained and is viewable through the USER_SOURCE and ALL_SOURCE tables in the Data Dictionary.

- To view the package specification, use:

```
SELECT text
  FROM   user_source
  WHERE  name = 'SALARY_PKG' AND type = 'PACKAGE'
  ORDER BY line;
```

- To view the package body, use:

```
SELECT text
  FROM   user_source
  WHERE  name = 'SALARY_PKG' AND type = 'PACKAGE BODY'
  ORDER BY line;
```

Tell Me / Show Me

Guidelines for Writing Packages

- Construct packages for general use.
- Create the package specification before the body.
- The package specification should contain only those constructs that you want to be public/global.
- Only recompile the package body, if possible, because changes to the package specification require recompilation of all programs that call the package.
- The package specification should contain as few constructs as possible.

Tell Me / Show Me

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.

Tell Me / Show Me

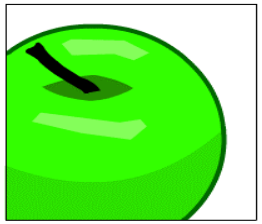
Advantages of Using Packages

- Added functionality: Persistency of variables and cursors
- Better performance:
  - The entire package is loaded into memory when the package is first referenced.
  - There is only one copy in memory for all users.
  - The dependency hierarchy is simplified.
- Overloading: Multiple subprograms having the same name.

Persistency and Overloading are covered in later lessons in this section.

Dependencies are covered in a later section of this course.

Tell Me / Show Me



Terminology

Key terms used in this lesson include:

Public components

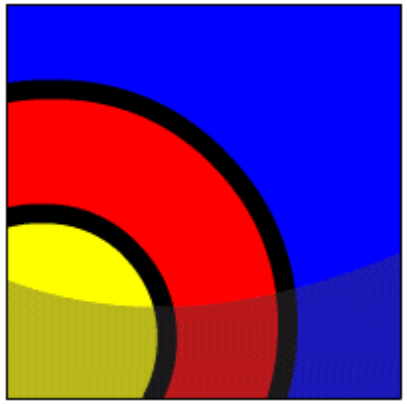
Private components

Visibility

Summary

In this lesson, you learned to:

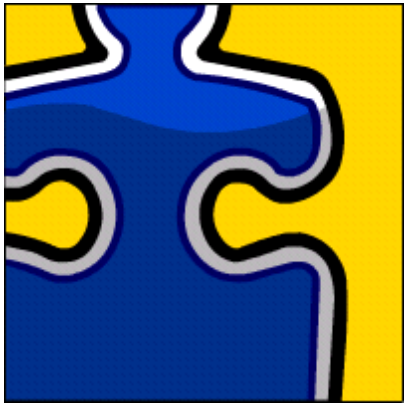
- Explain the difference between public and private package constructs
- Designate a package construct as either public or private
- Specify the appropriate syntax to drop packages
- Identify views in the data dictionary that manage packages
- Identify guidelines for using packages



Try It / Solve It

The exercises in this lesson cover the following topics:

- Designating a package construct as either public or private
- Invoking a package construct
- Identifying views in the data dictionary to manage packages
- Dropping packages
- Identifying guidelines and benefits of packages

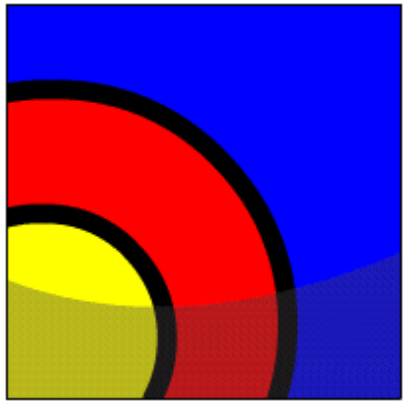


Advanced Package Concepts

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

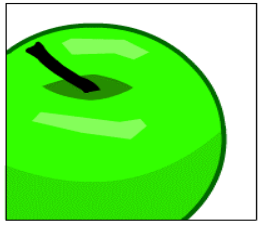


Why Learn It?

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.

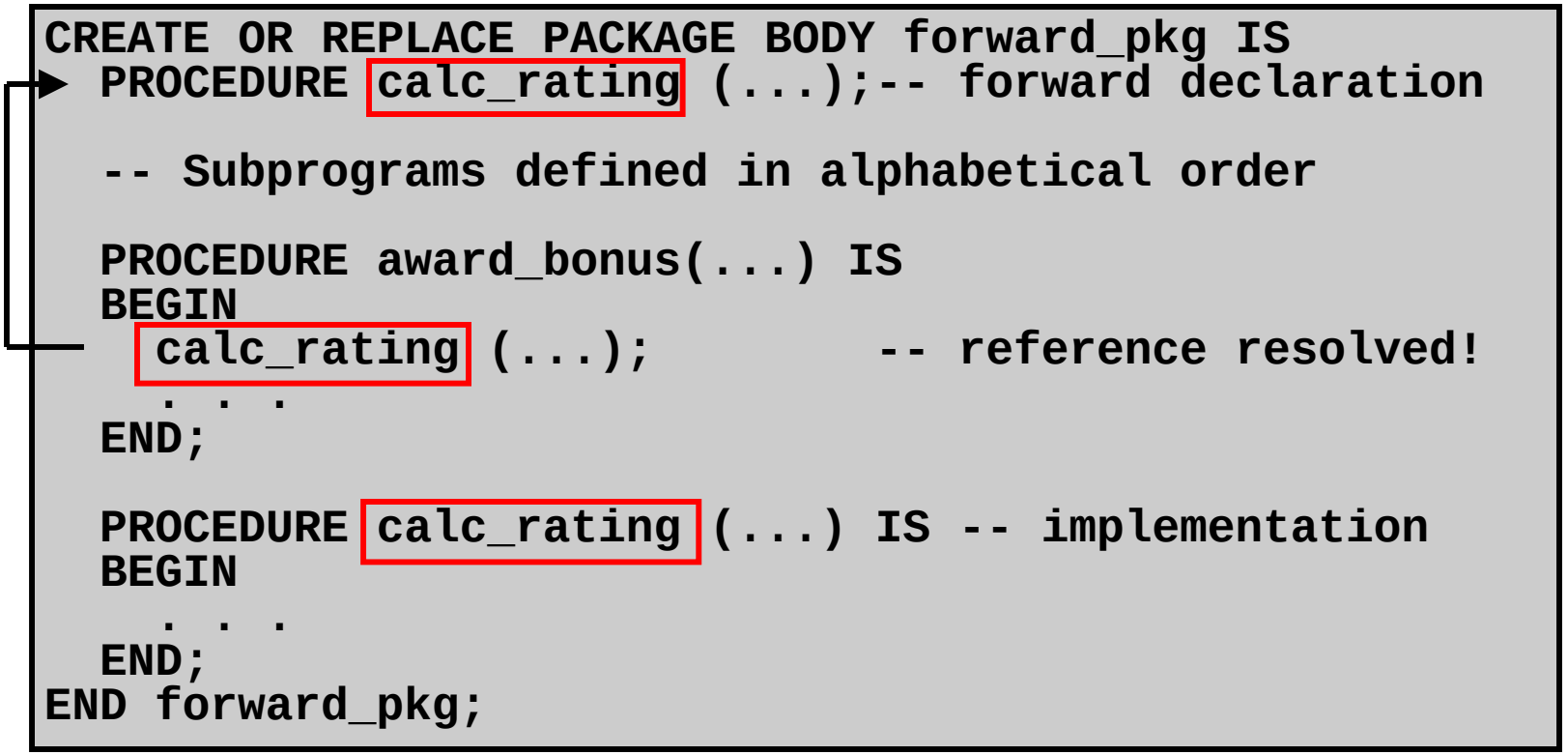
However, coding standards often require that subprograms be kept in alphabetical sequence to make them easy to find. In this case, you might encounter problems, as shown in the slide example.

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.



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

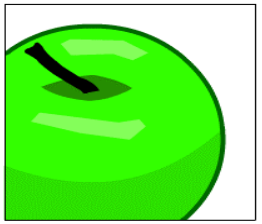
Package Function in SQL: Example 2

```
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

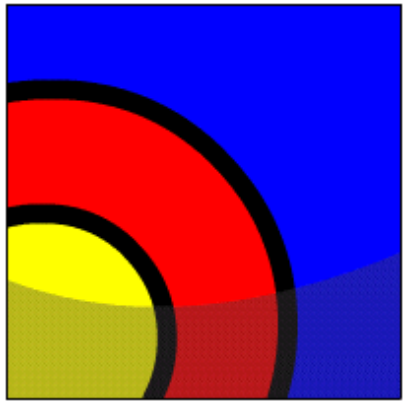
Forward declaration

Initialization block

Summary

In this lesson, you learned to:

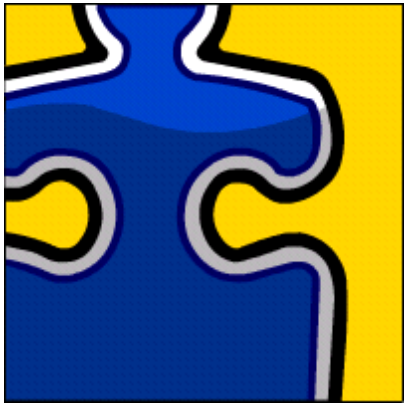
- Write packages that use the overloading feature
- Write packages that use forward declarations
- Explain the purpose of a package initialization block
- Identify restrictions on using packaged functions in SQL statements



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

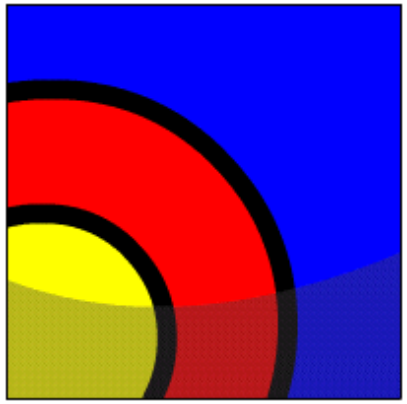


Persistent State of Package Variables

What Will I Learn?

In this lesson, you will learn to:

- Identify persistent states of package variables
- Control the persistent state of a package cursor



Why Learn It?

Suppose you connect to the database and modify the value in a package variable, for example from 10 to 20. Later, you (or someone else) invoke the package again to read the value of the variable. What will you/they see: 10 or 20? It depends!

Real applications often invoke the same package many times. It is important to understand when the values in package variables are kept (persist) and when they are lost.

Tell Me / Show Me

Package State

The collection of package variables and their current 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 session's private memory area
  - Unique to each session even if the second session is started by the same user
  - Subject to change when package subprograms are called or public variables are modified.

Other sessions each have their own package state, and do not see your changes.

Tell Me / Show Me

The following is a simple package that initializes a single global variable and contains a procedure to update it:

```
CREATE OR REPLACE PACKAGE pers_pkg IS
  g_var    NUMBER := 10;
  PROCEDURE upd_g_var (p_var IN NUMBER);
END pers_pkg;

CREATE OR REPLACE PACKAGE BODY pers_pkg IS
  PROCEDURE upd_g_var (p_var IN NUMBER) IS
    BEGIN
      g_var := p_var;
    END upd_g_var;
END pers_pkg;

GRANT EXECUTE ON pers_pkg TO SCOTT, JONES;
```

SCOTT and JONES call the procedure to update the variable.

Tell Me / Show Me

The following sequence of events occurs:

Time	Event	State for: Scott	Jones
9:00	Scott> .. svar := pers_pkg.g_var;	10	-
9:30	Jones> .. jvar := pers_pks.g_var; Jones> .. pers_pkg.upd_g_var(20); Scott> .. svar := pers_pkg.g_var;	10	10 20
9:35	Scott> .. pers_pkg.upd_g_var(50); Jones> .. jvar := pers_pks.g_var;	50	20
10:00	Scott disconnects and reconnects İn a new session		
10:05	Scott> .. svar := pers_pkg.g_var;	10	

Tell Me / Show Me

Explanation of the events on the previous slide:

- At 9:00: Scott connects and reads the variable, seeing the initialized value 10.
- At 9:30: Jones connects and also reads the variable, also seeing the initialized value 10. At this point there are two separate and independent copies of the value, one in each session's private memory area. Jones now updates his own session's value to 20 using the procedure. Scott then re-reads the variable but does not see Jones's change.
- At 9:35: Scott updates his own session's value to 50. Again, Jones cannot see the change.
- At 10:00: Scott disconnects and reconnects, creating a new session.
- At 10:05: Scott reads the variable and sees the initialized value 10.

- These changes would not be visible in other sessions even if both sessions are connected under the same user name.

Tell Me / Show Me

Persistent State of a Package Cursor

A cursor declared in the package specification is a type of global variable, and follows the same persistency rules as any other variable. A cursor's state is not defined by a single numeric or other value. A cursor's state consists of the following attributes:

- Whether the cursor is open or closed
- If open, how many rows have been fetched from the cursor (`%ROWCOUNT`) and whether the most recent fetch was successful (`%FOUND or %NOTFOUND`).

The next three slides show the definition of a cursor and its repeated use in a calling application.

Tell Me / Show Me

Persistent State of a Package Cursor: Package Specification

```
CREATE OR REPLACE PACKAGE curs_pkg IS
  CURSOR emp_curs IS SELECT employee_id FROM employees
              ORDER BY employee_id;
  PROCEDURE open_curs;
  FUNCTION fetch_n_rows(n NUMBER := 1) RETURN BOOLEAN;
  PROCEDURE close_curs;
END curs_pkg;
```

The cursor declaration is declared globally within the package specification. Therefore, any or all of the package procedures can reference it.

Tell Me / Show Me

Persistent State of a Package Cursor: Package Body

```
CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  PROCEDURE open_curs IS
  BEGIN
    IF NOT emp_curs%ISOPEN THEN   OPEN emp_curs;  END IF;
  END open_curs;
  FUNCTION fetch_n_rows(n NUMBER := 1) RETURN BOOLEAN IS
    emp_id employees.employee_id%TYPE;
  BEGIN
    FOR count IN 1 .. n LOOP
      FETCH emp_curs INTO emp_id;
      EXIT WHEN emp_curs%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Id: ' ||(emp_id));
    END LOOP;
    RETURN emp_curs%FOUND;
  END fetch_n_rows;
  PROCEDURE close_curs IS BEGIN
    IF emp_curs%ISOPEN THEN  CLOSE emp_curs;  END IF;
  END close_curs;
END curs_pkg;
```

Tell Me / Show Me

Invoking CURS_PKG

```
DECLARE
  v_more_rows_exist BOOLEAN := TRUE;
BEGIN
  curs_pkg.open_curs;                                    --1
  LOOP
    v_more_rows_exist := curs_pkg.fetch_n_rows(3);       --2
    DBMS_OUTPUT.PUT_LINE('-------');
    EXIT WHEN NOT v_more_rows_exist;
  END LOOP;
  curs_pkg.close_curs;                                   --3
END;
```

Step 1 opens the cursor. Step 2 (in a loop) fetches and displays the next three rows from the cursor until all rows have been fetched. Step 3 closes the cursor.

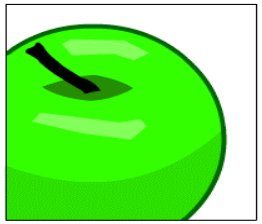
Tell Me / Show Me

Invoking CURS_PKG (continued)

```
DECLARE
  v_more_rows_exist BOOLEAN := TRUE;
BEGIN
  curs_pkg.open_curs;                                    --1
  LOOP
    v_more_rows_exist := curs_pkg.fetch_n_rows(3);       --2
    DBMS_OUTPUT.PUT_LINE('-------');
    EXIT WHEN NOT v_more_rows_exist;
  END LOOP;
  curs_pkg.close_curs;                                   --3
END;
```

- The first looped call to `fetch_n_rows` displays the first three rows. The second time round the loop, the next three rows are fetched and displayed. And so on.
- This technique is often used in applications that need to FETCH a large number of rows from a cursor, but can only display them to the user one screenful at a time.

Tell Me / Show Me



Terminology

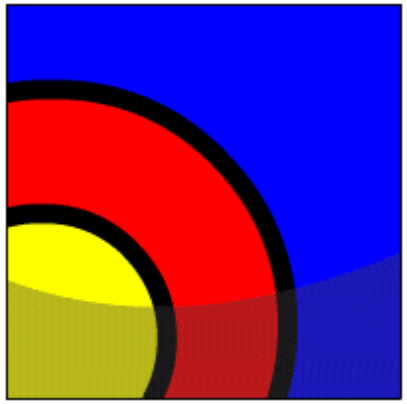
Key terms used in this lesson include:

Package state

Summary

In this lesson, you learned to:

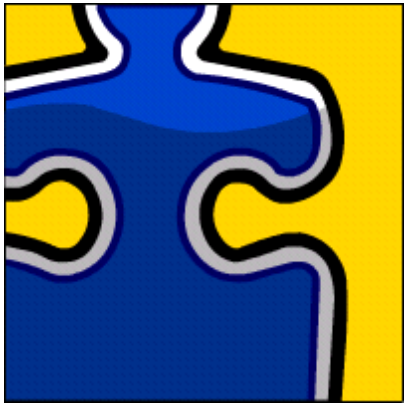
- Identify persistent states of package variables
- Control the persistent state of a package cursor



Try It / Solve It

The exercises in this lesson cover the following topics:

- Identifying persistent states of package variables
- Controlling the persistent state of a package cursor

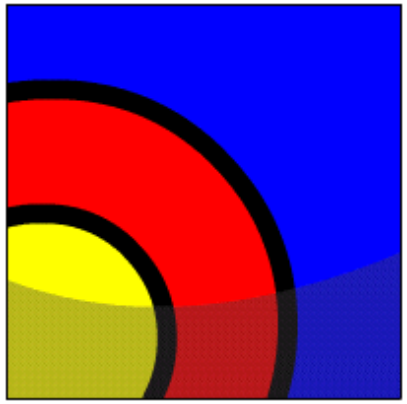


Using Oracle-Supplied Packages

What Will I Learn?

In this lesson, you will learn to:

- Describe two common uses for the DBMS_OUTPUT server-supplied package
- Recognize the correct syntax to specify messages for the DBMS_OUTPUT package
- Describe the purpose for the UTL_FILE server-supplied package.
- Recall the exceptions used in conjunction with the UTL_FILE server-supplied package.



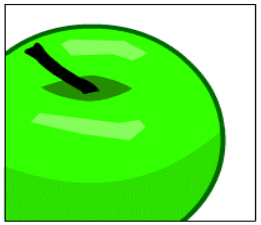
Why Learn It?

You already know that Oracle supplies a number of SQL functions (UPPER, TO_CHAR, and so on) that you can use in your SQL statements when required. It would be wasteful for everyone to have to "re-invent the wheel" by writing their own functions to do these things.

In the same way, Oracle supplies a number of ready-made PL/SQL packages to do things that most application developers and/or database administrators need to do from time to time.

In this lesson, you learn how to use two of the Oracle-supplied PL/SQL packages. These packages focus on generating text output and manipulating text files.

Tell Me / Show Me



Using Oracle-Supplied Packages

You can use these packages directly by invoking them from your own application, exactly as you would invoke packages that you had written yourself.

Or, you can use these packages as ideas when you create your own subprograms.

Think of these packages as ready-made “building blocks” that you can invoke from your own applications.

Tell Me / Show Me

List of Some Oracle-Supplied Packages

DBMS_LOB	Enables manipulation of Oracle Large Object column datatypes: CLOB, BLOB and BFILE
DBMS_LOCK	Used to request, convert, and release locks in the database through Oracle Lock Management services
DBMS_OUTPUT	Provides debugging and buffering of messages
HTP	Writes HTML-tagged data into database buffers
UTL_FILE	Enables reading and writing of operating system text files
UTL_MAIL	Enables composing and sending of e-mail messages
DBMS_SCHEDULER	Enables scheduling of PL/SQL blocks, stored procedures, and external procedures or executables

Tell Me / Show Me

The DBMS_OUTPUT Package

The DBMS_OUTPUT package sends text messages from any PL/SQL block into a private memory area, from which the message can be displayed on the screen.

Common uses of DBMS_OUTPUT include:

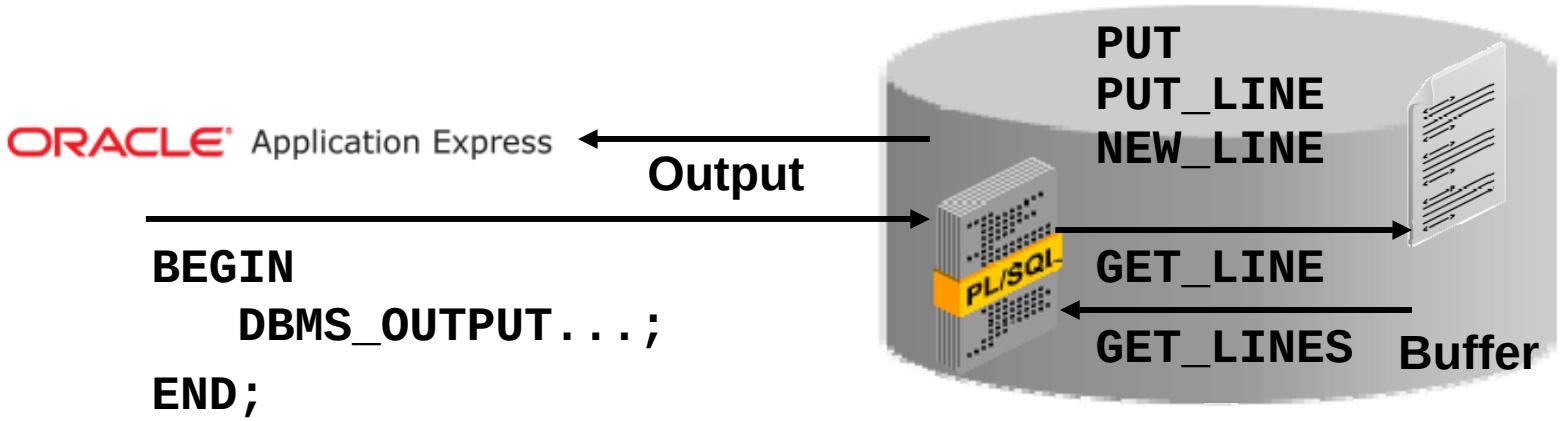
- You can output results back to the developer during testing for debugging purposes.
- You can trace the code execution path for a function or procedure.

Tell Me / Show Me

How the DBMS_OUTPUT Package Works

The DBMS_OUTPUT package enables you to send messages from stored subprograms and anonymous blocks.

- PUT places text in the buffer.
- NEW_LINE sends the buffer to the screen.
- PUT_LINE does a PUT followed by a NEW_LINE.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until after the calling block finishes.



Tell Me / Show Me

Using DBMS_OUTPUT: Example 1

You have already used DBMS_OUTPUT.PUT_LINE. This writes a text message into a buffer, then displays the buffer on the screen:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('The cat sat on the mat');
END;
```

If you wanted to build a message a little at a time, you could code:

```
BEGIN
  DBMS_OUTPUT.PUT('The cat sat ');
  DBMS_OUTPUT.PUT('on the mat');
  DBMS_OUTPUT.NEW_LINE;
END;
```

Tell Me / Show Me

Using DBMS_OUTPUT: Example 2

You can trace the flow of execution of a block with complex IF … ELSE, CASE or looping structures:

```
DECLARE
  v_bool1    BOOLEAN := true;
  v_bool2    BOOLEAN := false;
  v_number   NUMBER;
BEGIN
  ...
  IF v_bool1 AND NOT v_bool2 AND v_number < 25 THEN
    DBMS_OUTPUT.PUT_LINE('IF branch was executed');
  ELSE
    DBMS_OUTPUT.PUT_LINE('ELSE branch was executed');
  END IF;
  ...
END;
```

Tell Me / Show Me

DBMS_OUTPUT Is Designed for Debugging Only

You would not use DBMS_OUTPUT in PL/SQL programs that are called from a "real" application, which can include its own application code to display results on the user's screen. Instead, you would return the text to be displayed as an OUT argument from the subprogram. For example:

```
PROCEDURE do_some_work (...) IS BEGIN
  ... DBMS_OUTPUT.PUT_LINE('string'); ...  END;
```

Would be converted to:

```
PROCEDURE do_some_work (... p_output OUT VARCHAR2) IS
BEGIN
  ... p_output := 'string'; ...
END;
```

Tell Me / Show Me

DBMS_OUTPUT Is Designed for Debugging Only (continued)

For this reason, you should not use DBMS_OUTPUT in subprograms, but only in anonymous PL/SQL blocks for testing purposes. Instead of:

```
CREATE OR REPLACE PROCEDURE do_some_work IS BEGIN
  ... DBMS_OUTPUT.PUT_LINE('string'); ...  END;

BEGIN    do_some_work;    END;     -- Test the procedure
```

You should use:

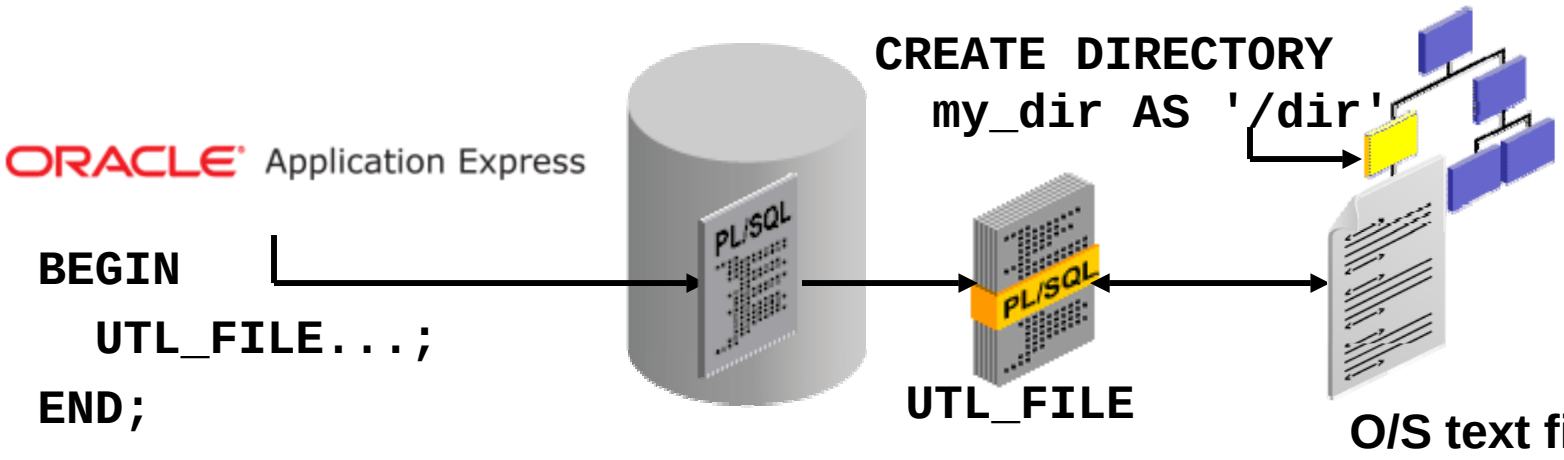
```
CREATE OR REPLACE PROCEDURE do_some_work
    (p_output OUT VARCHAR2) IS BEGIN
   ... p_output := 'string'; ...  END;

DECLARE v_output VARCHAR2(100);  BEGIN  --Test the procedure
  do_some_work(v_output);
  DBMS_OUTPUT.PUT_LINE(v_output);    END;
```

Tell Me / Show Me

The UTL_FILE Package:

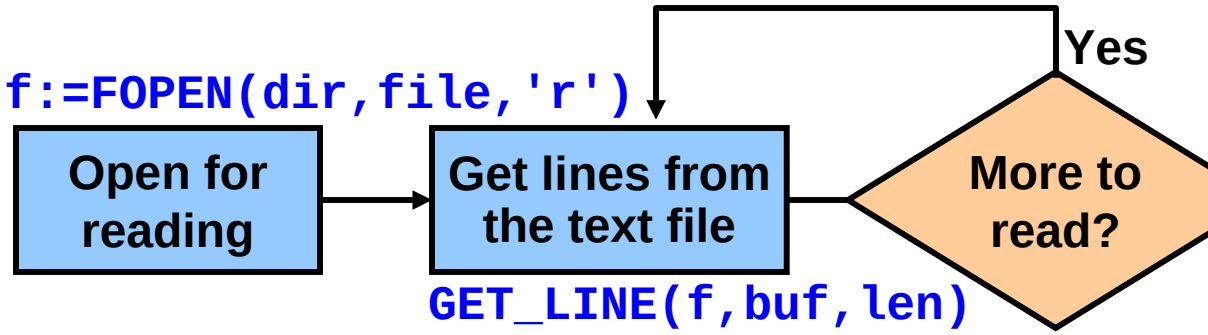
- Allows PL/SQL programs to read and write operating system text files.
- Can access text files in operating system directories defined by a `CREATE DIRECTORY` statement. You can also use the `utl_file_dir` database parameter.



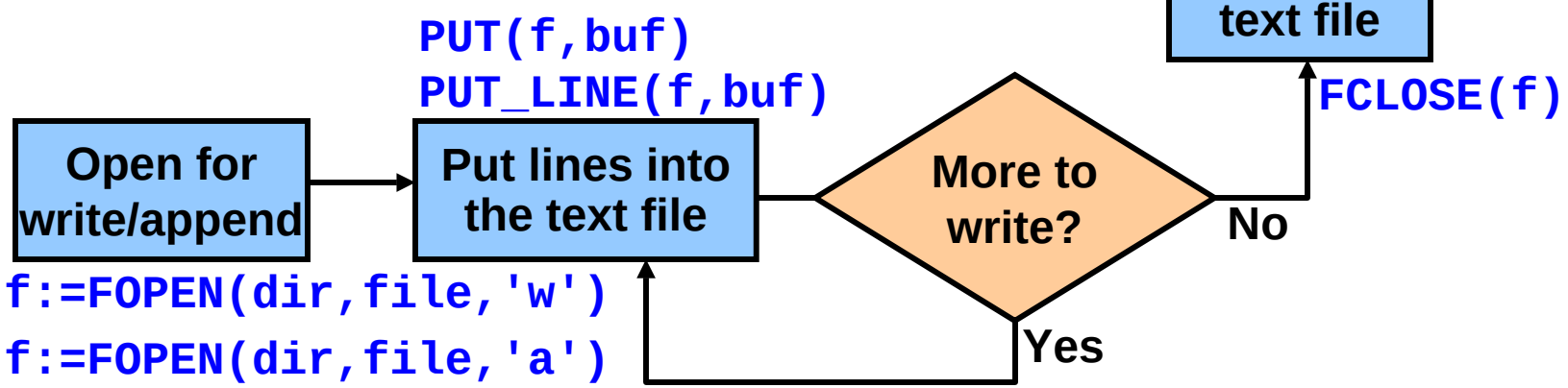
Tell Me / Show Me

File Processing Using the UTL_FILE Package

- Reading a file



- Writing or appending to a file



Tell Me / Show Me

File Processing Using the UTL_FILE Package

You open files for reading or writing with the FOPEN function. You then either read from or write or append to the file until processing is done. Then close the file by using the FCLOSE procedure. The following are the main subprograms:

- The FOPEN function opens a file in a specified directory for input/output (I/O) and returns a file handle used in later I/O operations.
- The IS_OPEN function checks whether the file is already open, returning a Boolean.
- The GET_LINE procedure reads a line of text from the file into an output buffer parameter.The maximum input record size is 1,023 bytes.
- The PUT and PUT_LINE procedures write text to the opened file.
- The NEW_LINE procedure terminates a line in an output file.
- The FCLOSE procedure closes an opened file.

Tell Me / Show Me

Exceptions in the UTL_FILE Package

You might have to handle one or more of these exceptions when using UTL_FILE subprograms:

- INVALID_PATH
- INVALID_MODE
- INVALID_FILEHANDLE
- INVALID_OPERATION
- READ_ERROR
- WRITE_ERROR
- INTERNAL_ERROR

The other exceptions not specific to the UTL_FILE package are:

- NO_DATA_FOUND and VALUE_ERROR

Tell Me / Show Me

FOPEN and IS_OPEN Function Parameters

```
FUNCTION FOPEN (location IN VARCHAR2,
                filename IN VARCHAR2,
                open_mode IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

```
FUNCTION IS_OPEN (file IN FILE_TYPE)
RETURN BOOLEAN;
```

Example:

```
PROCEDURE read(dir VARCHAR2, filename VARCHAR2) IS
  file UTL_FILE.FILE_TYPE;
BEGIN
  IF NOT UTL_FILE.IS_OPEN(file) THEN
    file := UTL_FILE.FOPEN (dir, filename, 'r');
  END IF; ...
END read;
```

Tell Me / Show Me

Using UTL_FILE: Example (continued on next two slides)

In this example, the sal_status procedure uses UTL_FILE to create a text report of employees for each department, along with their salaries. In the code, the variable v_file is declared as UTL_FILE.FILE_TYPE, a BINARY_INTEGER datatype that is declared globally by the UTL_FILE package.

The sal_status procedure accepts two IN parameters: p_dir for the name of the directory in which to write the text file, and p_filename to specify the name of the file.

To invoke the procedure, use (for example):

```
BEGIN sal_status('MY_DIR', 'salreport.txt'); END;
```

Tell Me / Show Me

Using UTL_FILE: Example (continued on next slide)

```
CREATE OR REPLACE PROCEDURE sal_status(
  p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
  v_file UTL_FILE.FILE_TYPE;
  CURSOR empc IS
    SELECT last_name, salary, department_id
      FROM employees ORDER BY department_id;
  v_newdeptno employees.department_id%TYPE;
  v_olddeptno employees.department_id%TYPE := 0;
BEGIN
  v_file:= UTL_FILE.FOPEN (p_dir, p_filename, 'w');    -- 1
  UTL_FILE.PUT_LINE(v_file,                            -- 2
    'REPORT: GENERATED ON ' || SYSDATE);
  UTL_FILE.NEW_LINE (v_file); ...                      -- 3
```

Tell Me / Show Me

Using UTL_FILE: Example (continued)

```
  FOR emp_rec IN empc LOOP
     UTL_FILE.PUT_LINE (v_file,                              --4
        '  EMPLOYEE: ' || emp_rec.last_name ||
        'earns: ' || emp_rec.salary);
  END LOOP;
  UTL_FILE.PUT_LINE(v_file,'*** END OF REPORT ***');  --5
  UTL_FILE.FCLOSE (v_file);                                --6
EXCEPTION
  WHEN UTL_FILE.INVALID_FILEHANDLE THEN                    --7
    RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
  WHEN UTL_FILE.WRITE_ERROR THEN                           --8
    RAISE_APPLICATION_ERROR (-20002, 'Unable to
                             write to file');
END sal_status;
```

Tell Me / Show Me

Using UTL_FILE: Example: Invocation and Output Report

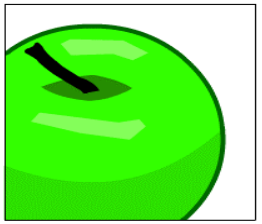
Suppose you invoke your procedure by:

```
BEGIN   sal_status('MYDIR', 'salreport.txt');   END;
```

The output contained in the file is:

```
SALARY REPORT: GENERATED ON 29-NOV-06
    EMPLOYEE: Whalen earns: 4400
    EMPLOYEE: Hartstein earns: 13000
    EMPLOYEE: Fay earns: 6000
        ...
    EMPLOYEE: Higgins earns: 12000
    EMPLOYEE: Gietz earns: 8300
    EMPLOYEE: Grant earns: 7000
*** END OF REPORT ***
```

Tell Me / Show Me



Terminology

Key terms used in this lesson include:

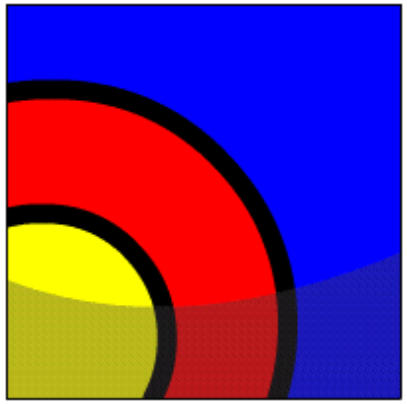
`DBMS_OUTPUT` package

`UTL_FILE` package

Summary

In this lesson, you learned to:

- Describe two common uses for the DBMS_OUTPUT server-supplied package
- Recognize the correct syntax to specify messages for the DBMS_OUTPUT package
- Describe the purpose for the UTL_FILE server-supplied package.
- Recall the exceptions used in conjunction with the UTL_FILE server-supplied package.



Try It / Solve It

The exercises in this lesson cover the following topics:

- Describing two common uses for the `DBMS_OUTPUT` server-supplied package
- Recognizing the correct syntax to specify messages for the `DBMS_OUTPUT` package
- Describing the purpose for the `UTL_FILE` server-supplied package.
- Recalling the exceptions used in conjunction with the `UTL_FILE` server-supplied package.