4

Using More Package Concepts

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- **Overload package procedures and functions**
- **Use forward declarations**
- **Create an initialization block in a package body**
- **Manage persistent package data states for the life of a session**
- **Use PL/SQL tables and records in packages**
- **Wrap source code stored in the data dictionary so that it is not readable**

ORACLE

Lesson Aim

This lesson introduces the more advanced features of PL/SQL, including overloading, forward referencing, one-time-only procedures, and the persistency of variables, constants, exceptions, and cursors. It also explains the effect of packaging functions that are used in SQL statements.

Overloading Subprograms

The overloading feature 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**

Note: Overloading can be done with local subprograms, package subprograms, and type methods, but not with stand-alone subprograms.

ORACLE

Overloading Subprograms

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the `TO_CHAR` function has more than one way to be called, enabling you to convert a number or a date to a character string.

PL/SQL allows overloading of package subprogram names and object type methods.

The key rule is that you can use the same name for different subprograms as long as their formal parameters differ in number, order, or data type family.

Consider using overloading when:

- Processing rules for two or more subprograms are similar, but the type or number of parameters used varies
- Providing alternative ways for finding different data with varying search criteria. For example, you may want to find employees by their employee ID and also provide a way to find employees by their last name. The logic is intrinsically the same, but the parameters or search criteria differ.
- Extending functionality when you do not want to replace existing code

Note: Stand-alone subprograms cannot be overloaded. Writing local subprograms in object type methods is not discussed in this course.

Overloading Subprograms (continued)

Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same family (`NUMBER` and `DECIMAL` belong to the same family.)
- Two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family (`VARCHAR` and `STRING` are PL/SQL subtypes of `VARCHAR2`.)
- Two functions that differ only in return type, even if the types are in different families

You get a run-time error when you overload subprograms with the preceding features.

Note: The preceding restrictions apply if the names of the parameters are also the same. If you use different names for the parameters, you can invoke the subprograms by using named notation for the parameters.

Resolving Calls

The compiler tries to find a declaration that matches the call. It searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the name matches the name of the called subprogram. For similarly named subprograms at the same level of scope, the compiler needs an exact match in number, order, and data type between the actual and formal parameters.

Overloading: Example

```
CREATE OR REPLACE PACKAGE dept_pkg IS
  PROCEDURE add_department(deptno NUMBER,
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
  PROCEDURE add_department(
   name VARCHAR2 := 'unknown', loc NUMBER := 1700);
END dept_pkg;
/
```

ORACLE

Overloading: Example

The slide shows the `dept_pkg` package specification with an overloaded procedure called `add_department`. The first declaration takes three parameters that are used to provide data for a new department record inserted into the department table. The second declaration takes only two parameters because this version internally generates the department ID through an Oracle sequence.

Note: The example uses basic data types for its arguments to ensure that the example fits in the space provided. It is better to specify data types using the `%TYPE` attribute for variables that are used to populate columns in database tables, as in the following example:

```
PROCEDURE add_department
 (deptno departments.department_id%TYPE,
  name departments.department_name%TYPE := 'unknown',
  loc departments.location_id%TYPE := 1700);
```

Overloading: Example

```
CREATE OR REPLACE PACKAGE BODY dept_pkg  IS
  PROCEDURE add_department (deptno NUMBER,
   name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments(department_id,
      department_name, location_id)
    VALUES  (deptno, name, loc);
  END add_department;

  PROCEDURE add_department (
    name VARCHAR2:='unknown', loc NUMBER:=1700) IS
  BEGIN
    INSERT INTO departments (department_id,
      department_name, location_id)
    VALUES (departments_seq.NEXTVAL, name, loc);
  END add_department;
 END dept_pkg;
/
```

ORACLE

Overloading: Example (continued)

If you call `add_department` with an explicitly provided department ID, then PL/SQL uses the first version of the procedure. Consider the following example:

```
EXECUTE dept_pkg.add_department(980,'Education',2500)
SELECT * FROM departments
WHERE department_id = 980;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
980	Education		2500

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

```
EXECUTE dept_pkg.add_department ('Training', 2400)
SELECT * FROM departments
WHERE department_name = 'Training';
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
320	Training		2400

Overloading and the STANDARD Package

- A package named STANDARD defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. An example is the TO_CHAR function:

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN
VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN
VARCHAR2;
```

- A PL/SQL subprogram with the same name as a built-in subprogram overrides the standard declaration in the local context, unless you qualify the built-in subprogram with its package name.

ORACLE

Overloading and the STANDARD Package

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

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

If you redeclare a built-in subprogram as a stand-alone subprogram, then to access your subprogram you must qualify it with your schema name: for example, SCOTT.TO_CHAR.

Using Forward Declarations

- **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;
/
```

ORACLE

Using Forward Declarations

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

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

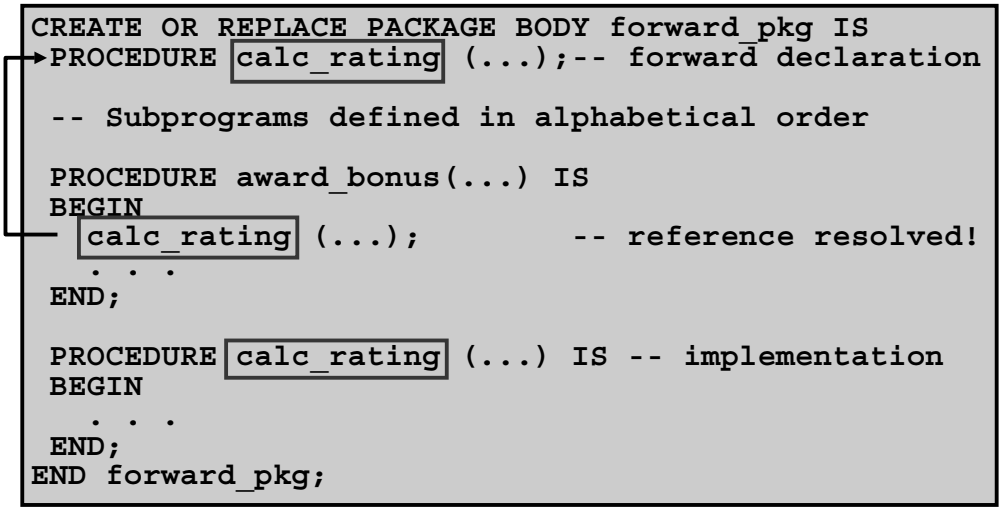
You can solve the illegal reference problem by reversing the order of the two procedures. However, this easy solution does not work if the coding rules require subprograms to be declared in alphabetical order.

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

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

Using Forward Declarations

In the package body, a forward declaration is a private subprogram specification terminated by a semicolon.



ORACLE

Using Forward Declarations (continued)

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

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

When creating a forward declaration:

- The formal parameters must appear in both the forward declaration and the subprogram body
- The subprogram body can appear anywhere after the forward declaration, but both must appear in the same program unit

Forward Declarations and Packages

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

Package Initialization Block

The 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
  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 tax
  FROM     tax_rates
  WHERE    rate_name = 'TAX';
END taxes;
/
```

ORACLE

Package Initialization Block

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

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

Consider the block of code at the end of a package body as a package initialization block that executes once, when the package is first invoked within the user session.

The example in the slide shows the `tax` public variable being initialized to the value in the `tax_rates` table the first time the `taxes` package is referenced.

Note: If you initialize the variable in the declaration by using an assignment operation, it is overwritten by the code in the initialization block at the end of the package body. The initialization block is terminated by the `END` keyword for the package body.

Using Package Functions in SQL and Restrictions

- **Package functions can be used in SQL statements.**
- **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.

ORACLE

Using Package Functions in SQL and Restrictions

When executing a SQL statement that calls a stored function, the Oracle server must know the purity level of stored functions, that is, whether the functions are free of the restrictions listed in the slide. In general, restrictions are changes to database tables or public package variables (those declared in a package specification). Restrictions can delay the execution of a query, yield order-dependent (therefore indeterminate) results, or require that the package state variables be maintained across user sessions. Various restrictions are not allowed when a function is called from a SQL query or a DML statement. Therefore, the following restrictions apply to stored functions called from SQL expressions:

- A function called from a query or DML statement cannot end the current transaction, create or roll back to a savepoint, or alter the system or session.
- A function called from a query statement or from a parallelized DML statement cannot execute a DML statement or otherwise modify the database.
- A function called from a DML statement cannot read or modify the particular table being modified by that DML statement.

Note: Prior to Oracle8*i*, the purity level was checked at compilation time by including `PRAGMA RESTRICT_REFERENCES` in the package specification. Since Oracle8*i*, the purity level of functions is checked at run time.

Package Function in SQL: Example

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
  FUNCTION tax (value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
  FUNCTION tax (value IN NUMBER) RETURN NUMBER IS
    rate NUMBER := 0.08;
  BEGIN
    RETURN (value * rate);
  END tax;
END taxes_pkg;
/
```

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

ORACLE

Package Function in SQL: Example

The first code box shows how to create the package specification and the body encapsulating the `tax` function in the `taxes_pkg` package. The second code box shows how to call the packaged `tax` function in the `SELECT` statement. The output results are similar to:

TAXES_PACK.TAX(SALARY)	SALARY	LAST_NAME
1920	24000	King
1360	17000	Kochhar
1360	17000	De Haan
720	9000	Hunold
480	6000	Ernst
422.4	5280	Austin
422.4	5280	Pataballa
369.6	4620	Lorentz
960	12000	Greenberg

▪ ▪ ▪

109 rows selected.

Note: If you are using Oracle versions prior to 8*i*, then you must assert the purity level of the function in the package specification by using `PRAGMA RESTRICT_REFERENCES`. If this is not specified, you get an error message saying that the `TAX` function does not guarantee that it will not update the database while invoking the package function in a query.

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**
  - **Stored in the User Global Area (UGA)**
  - **Unique to each session**
  - **Subject to change when package subprograms are called or public variables are modified**
- **Not persistent for the session but persistent for the life of a subprogram call when using `PRAGMA SERIALLY_REUSABLE` in the package specification**

ORACLE

Persistent State of Packages

The collection of public and private package variables represents the package state for the user session. That is, the package state is the set of values stored in all the package variables at a given point in time. In general, the package state exists for the life of the user session.

Package variables are initialized the first time a package is loaded into memory for a user session. The package variables are, by default, unique to each session and hold their values until the user session is terminated. In other words, the variables are stored in the UGA memory allocated by the database for each user session.

The package state changes when a package subprogram is invoked and its logic modifies the variable state. Public package state can be directly modified by operations appropriate to its type.

Note: If you add `PRAGMA SERIALLY_RESUABLE` to the package specification, then the database stores package variables in the System Global Area (SGA) shared across user sessions. In this case, the package state is maintained for the life of a subprogram call or a single reference to a package construct. The `SERIALLY_REUSABLE` directive is useful if you want to conserve memory and if the package state does not need to persist for each user session.

Persistent State of Package Variables: Example

Time	Events	State for: -Scott- STD	-Scott- MAX	-Jones- STD	-Jones- MAX
9:00	Scott> EXECUTE comm_pkg.reset_comm(0.25)	0.10 0.25	0.4	-	0.4
9:30	Jones> INSERT INTO employees(last_name,commission_pct) VALUES('Madonna', 0.8);	0.25	0.4		0.8
9:35	Jones> EXECUTE comm_pkg.reset_comm (0.5)	0.25	0.4	0.1 0.5	0.8
10:00	Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'	0.25	0.4	0.5	0.8
11:00	Jones> ROLLBACK;	0.25	0.4	0.5	0.4
11:01	EXIT ...	0.25	0.4	-	0.4
12:00	EXEC comm_pkg.reset_comm(0.2)	0.25	0.4	0.2	0.4

ORACLE

Persistent State of Package Variables: Example

The slide sequence is based on two different users, Scott and Jones, executing `comm_pkg` (covered in the lesson titled "Creating Packages"), in which the `reset_comm` procedure invokes the `validate` function to check the new commission. The example shows how the persistent state of the `std_comm` package variable is maintained in each user session.

At 9:00: Scott calls `reset_comm` with a new commission value of 0.25, the package state for `std_comm` is initialized to 0.10 and then set to 0.25, which is validated because it is less than the database maximum value of 0.4.

At 9:30: Jones inserts a new row into the `EMPLOYEES` table with a new maximum `commission_pct` value of 0.8. This is not committed, so it is visible to Jones only. Scott's state is unaffected.

At 9:35: Jones calls `reset_comm` with a new commission value of 0.5. The state for Jones's `std_comm` is first initialized to 0.10 and then set to the new value 0.5 that is valid for his session with the database maximum value of 0.8.

At 10:00: Scott calls with `reset_comm` with a new commission value of 0.6, which is greater than the maximum database commission visible to his session, that is, 0.4 (Jones did not commit the 0.8 value.)

Between 11:00 and 12:00: Jones rolls back the transaction and exits the session. Jones logs in at 11:45 and successfully executes the procedure, setting his state to 0.2.

Persistent State of a Package Cursor

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

ORACLE

Persistent State of a Package Cursor

The code in the slide shows the package body for CURS_PKG to support the following package specification:

```
CREATE OR REPLACE PACKAGE curs_pkg IS
  PROCEDURE open;
  FUNCTION next(n NUMBER := 1) RETURN BOOLEAN;
  PROCEDURE close;
END curs_pkg;
```

To use this package, perform the following steps to process the rows:

- Call the `open` procedure to open the cursor.
- Call the `next` procedure to fetch one or a specified number of rows. If you request more rows than actually exist, the procedure successfully handles termination. It returns `TRUE` if more rows need to be processed; otherwise it returns `FALSE`.
- Call the `close` procedure to close the cursor, before or at the end of processing the rows.

Note: The cursor declaration is private to the package. Therefore the cursor state can be influenced by invoking the package procedure and functions listed in the slide.

Executing CURS_PKG

```
SET SERVEROUTPUT ON
EXECUTE curs_pkg.open
DECLARE
  more BOOLEAN := curs_pkg.next(3);
BEGIN
  IF NOT more THEN
    curs_pkg.close;
  END IF;
END;
/
RUN  -- repeats execution on the anonymous block
EXECUTE curs_pkg.close
```

ORACLE

Executing CURS_PKG

Recall that the state of a package variable or cursor persists across transactions within a session. However, the state does not persist across different sessions for the same user. The database tables hold data that persists across sessions and users. The `SET SERVEROUTPUT ON` command prepares *i*SQL*Plus to display output results. The call to `curs_pkg.open` opens the cursor, which remains open until the session is terminated, or the cursor is explicitly closed. The anonymous block executes the `next` function in the `Declaration` section, initializing the `BOOLEAN` variable `more` to `TRUE`, as there are more than three rows in the employees table. The block checks for the end of the result set and closes the cursor, if appropriate. When the block executes, it displays the first three rows:

```
Id :100
Id :101
Id :102
```

The `RUN` command executes the anonymous block again, and the next three rows are displayed:

```
Id :103
Id :104
Id :105
```

The `EXECUTE curs_pkg.close` command closes the cursor in the package.

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(emps OUT emp_table_type);
END emp_pkg;
/
```

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

ORACLE

Using Tables of Records of Procedures or Functions in Packages

The `emp_pkg` package contains a `get_employees` procedure that reads rows from the `EMPLOYEES` table and returns the rows using the `OUT` parameter, which is a PL/SQL table of records. The key points include the following:

- `employee_table_type` is declared as a public type.
- `employee_table_type` is used for a formal output parameter in the procedure, and the `employees` variable in the calling block (shown below).

In *i*SQL*Plus, you can invoke the `get_employees` procedure in an anonymous PL/SQL block by using the `employees` variable, as shown in the following example:

```
DECLARE
  employees  emp_pkg.emp_table_type;
BEGIN
  emp_pkg.get_employees(employees);
  DBMS_OUTPUT.PUT_LINE('Emp 4:
'||employees(4).last_name);
END;
/
```

This results in the following output:

```
Emp 4: Ernst
```

PL/SQL Wrapper

- **The PL/SQL wrapper is a stand-alone utility that hides application internals by converting PL/SQL source code into portable object code.**
- **Wrapping has the following features:**
  - **Platform independence**
  - **Dynamic loading**
  - **Dynamic binding**
  - **Dependency checking**
  - **Normal importing and exporting when invoked**

ORACLE

PL/SQL Wrapper

The PL/SQL wrapper is a stand-alone utility that converts PL/SQL source code into portable object code. Using it, you can deliver PL/SQL applications without exposing your source code, which may contain proprietary algorithms and data structures. The wrapper converts the readable source code into unreadable code. By hiding application internals, it prevents misuse of your application.

Wrapped code, such as PL/SQL stored programs, has several features:

- It is platform independent, so you do not need to deliver multiple versions of the same compilation unit.
- It permits dynamic loading, so users need not shut down and relink to add a new feature.
- It permits dynamic binding, so external references are resolved at load time.
- It offers strict dependency checking, so that invalidated program units are recompiled automatically when they are invoked.
- It supports normal importing and exporting, so the import/export utility can process wrapped files.

Running the Wrapper

The command-line syntax is:

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

- **The `INAME` argument is required.**
- **The default extension for the input file is `.sql`, unless it is specified with the name.**
- **The `ONAME` argument is optional.**
- **The default extension for output file is `.plb`, unless specified with the `ONAME` argument.**

Examples:

```
WRAP INAME=demo_04_hello.sql
WRAP INAME=demo_04_hello
WRAP INAME=demo_04_hello.sql ONAME=demo_04_hello.plb
```

ORACLE

Running the Wrapper

The wrapper is an operating system executable called `WRAP`. To run the wrapper, enter the following command at your operating system prompt:

```
WRAP INAME=input_file_name [ONAME=output_file_name]
```

Each of the examples shown in the slide takes a file called demo_04_hello`.sql` as input and creates an output file called `demo_04_hello.plb`.

After the wrapped file is created, execute the `.plb` file from *i*SQL*Plus to compile and store the wrapped version of the source code, as you would execute SQL script files.

Note

- Only the `INAME` argument is required. If the `ONAME` argument is not specified, then the output file acquires the same name as the input file with an extension of `.plb`.
- The input file can have any extension, but the default is `.sql`.
- Case sensitivity of the `INAME` and `ONAME` values depends on the operating system.
- Generally, the output file is much larger than the input file.
- Do not put spaces around the equal signs in the `INAME` and `ONAME` arguments and values.

Results of Wrapping

- **Original PL/SQL source code in input file:**

```
CREATE PACKAGE banking IS
  min_bal := 100;
  no_funds EXCEPTION;
...
END banking;
/
```

- **Wrapped code in output file:**

```
CREATE PACKAGE banking
wrapped
012abc463e ...
/
```

ORACLE

Results of Wrapping

When it is wrapped, an object type, package, or subprogram has the following form: header, followed by the word `wrapped`, followed by the encrypted body.

The input file can contain any combination of SQL statements. However, the PL/SQL wrapper wraps only the following `CREATE` statements:

- `CREATE [OR REPLACE] TYPE`
- `CREATE [OR REPLACE] TYPE BODY`
- `CREATE [OR REPLACE] PACKAGE`
- `CREATE [OR REPLACE] PACKAGE BODY`
- `CREATE [OR REPLACE] FUNCTION`
- `CREATE [OR REPLACE] PROCEDURE`

All other SQL `CREATE` statements are passed intact to the output file.

Guidelines for Wrapping

- **You must wrap only the package body, not the package specification.**
- **The wrapper can detect syntactic errors but cannot detect semantic errors.**
- **The output file should not be edited. You maintain the original source code and wrap again as required.**

Guidelines for Wrapping

Guidelines include the following:

- When wrapping a package or object type, wrap only the body, not the specification. Thus, you give other developers the information they need to use the package without exposing its implementation.
- If your input file contains syntactic errors, the PL/SQL wrapper detects and reports them. However, the wrapper cannot detect semantic errors because it does not resolve external references. For example, the wrapper does not report an error if the table or view `amp` does not exist:

```
CREATE PROCEDURE raise_salary (emp_id INTEGER, amount
NUMBER) AS
BEGIN
   UPDATE amp  -- should be emp
     SET sal = sal + amount WHERE empno = emp_id;
END;
```

  However, the PL/SQL compiler resolves external references. Therefore, semantic errors are reported when the wrapper output file (`.plb` file) is compiled.
- Because its contents are not readable, the output file should not be edited. To change a wrapped object, you need to modify the original source code and wrap the code again.

Summary

In this lesson, you should have learned how to:

- **Create and call overloaded subprograms**
- **Use forward declarations for subprograms**
- **Write package initialization blocks**
- **Maintain persistent package state**
- **Use the PL/SQL wrapper to wrap code**

ORACLE

Summary

Overloading is a feature that enables you to define different subprograms with the same name. It is logical to give two subprograms the same name when the processing in both the subprograms is the same but the parameters passed to them vary.

PL/SQL permits a special subprogram declaration called a forward declaration. A forward declaration enables you to define subprograms in logical or alphabetical order, define mutually recursive subprograms, and group subprograms in a package.

A package initialization block is executed only when the package is first invoked within the other user session. You can use this feature to initialize variables only once per session.

You can keep track of the state of a package variable or cursor, which persists throughout the user session, from the time the user first references the variable or cursor to the time the user disconnects.

Using the PL/SQL wrapper, you can obscure the source code stored in the database to protect your intellectual property.

Practice 4: Overview

This practice covers the following topics:

- **Using overloaded subprograms**
- **Creating a package initialization block**
- **Using a forward declaration**
- **Using the `WRAP` utility to prevent the source code from being deciphered by humans**

ORACLE

Practice 4: Overview

In this practice, you modify an existing package to contain overloaded subprograms and you use forward declarations. You also create a package initialization block within a package body to populate a PL/SQL table. You use the `WRAP` command-line utility to prevent the source code from being readable in the data dictionary tables.

Practice 4

1. Copy and modify the code for the EMP_PKG package that you created in Practice 3, Exercise 2, and overload the ADD_EMPLOYEE procedure.
   a. In the package specification, add a new procedure called ADD_EMPLOYEE that accepts three parameters: the first name, last name, and department ID. Save and compile the changes.
   b. Implement the new ADD_EMPLOYEE procedure in the package body so that it formats the e-mail address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name. The procedure should call the existing ADD_EMPLOYEE procedure to perform the actual INSERT operation using its parameters and formatted e-mail to supply the values. Save and compile the changes.
   c. Invoke the new ADD_EMPLOYEE procedure using the name Samuel Joplin to be added to department 30.
2. In the EMP_PKG package, create two overloaded functions called GET_EMPLOYEE.
   a. In the specification, add a GET_EMPLOYEE function that accepts the parameter called emp_id based on the employees.employee_id%TYPE type, and a second GET_EMPLOYEE function that accepts the parameter called family_name of type employees.last_name%TYPE. Both functions should return an EMPLOYEES%ROWTYPE. Save and compile the changes.
   b. In the package body, implement the first GET_EMPLOYEE function to query an employee by his or her ID, and the second to use the equality operator on the value supplied in the family_name parameter. Save and compile the changes.
   c. Add a utility procedure PRINT_EMPLOYEE to the package that accepts an EMPLOYEES%ROWTYPE as a parameter and displays the department_id, employee_id, first_name, last_name, job_id, and salary for an employee on one line, using DBMS_OUTPUT. Save and compile the changes.
   d. Use an anonymous block to invoke the EMP_PKG.GET_EMPLOYEE function with an employee ID of 100 and family name of 'Joplin'. Use the PRINT_EMPLOYEE procedure to display the results for each row returned.
3. Because the company does not frequently change its departmental data, you improve performance of your EMP_PKG by adding a public procedure INIT_DEPARTMENTS to populate a private PL/SQL table of valid department IDs. Modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values.
   a. In the package specification, create a procedure called INIT_DEPARTMENTS with no parameters.
   b. In the package body, implement the INIT_DEPARTMENTS procedure to store all department IDs in a private PL/SQL index-by table named valid_departments containing BOOLEAN values. Use the department_id column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of TRUE. Declare the valid_departments variable and its type definition boolean_tabtype before all procedures in the body.

Practice 4 (continued)

c. In the body, create an initialization block that calls the `INIT_DEPARTMENTS` procedure to initialize the table. Save and compile the changes.

4. Change `VALID_DEPTID` validation processing to use the private PL/SQL table of department IDs.
   a. Modify `VALID_DEPTID` to perform its validation by using the PL/SQL table of department ID values. Save and compile the changes.
   b. Test your code by calling `ADD_EMPLOYEE` using the name `James Bond` in department 15. What happens?
   c. Insert a new department with ID 15 and name Security, and commit the changes.
   d. Test your code again, by calling `ADD_EMPLOYEE` using the name `James Bond` in department 15. What happens?
   e. Execute the `EMP_PKG.INIT_DEPARTMENTS` procedure to update the internal PL/SQL table with the latest departmental data.
   f. Test your code by calling `ADD_EMPLOYEE` using the employee name `James Bond`, who works in department 15. What happens?
   g. Delete employee `James Bond` and department 15 from their respective tables, commit the changes, and refresh the department data by invoking the `EMP_PKG.INIT_DEPARTMENTS` procedure.
5. Reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.
   a. Edit the package specification and reorganize subprograms alphabetically. In *i*SQL*Plus, load and compile the package specification. What happens?
   b. Edit the package body and reorganize all subprograms alphabetically. In *i*SQL*Plus, load and compile the package specification. What happens?
   c. Fix the compilation error using a forward declaration in the body for the offending subprogram reference. Load and re-create the package body. What happens? Save the package code in a script file.

If you have time, complete the following exercise:

6. Wrap the `EMP_PKG` package body and re-create it.
   a. Query the data dictionary to view the source for the `EMP_PKG` body.
   b. Start a command window and execute the `WRAP` command-line utility to wrap the body of the `EMP_PKG` package. Give the output file name a `.plb` extension.
      Hint: Copy the file (which you saved in step 5c) containing the package body to a file called `emp_pkb_b.sql`.
   c. Using *i*SQL*Plus, load and execute the `.plb` file containing the wrapped source.
   d. Query the data dictionary to display the source for the `EMP_PKG` package body again. Are the original source code lines readable?