# Oracle Database 10*g*: Develop PL/SQL Program Units

**Volume 1 • Student Guide**

D17169GC21

Edition 2.1

December 2006

D48230

**ORACLE®**

## Authors

Tulika Srivastava
Glenn Stokol

## Technical Contributors and Reviewers

Chaitanya Koratamaddi
Dr. Christoph Burandt
Zarko Cesljas
Yanti Chang
Kathryn Cunningham
Burt Demchick
Laurent Dereac
Peter Driver
Bryan Roberts
Bryn Llewellyn
Nancy Greenberg
Craig Hollister
Thomas Hoogerwerf
Taj-Ul Islam
Inger Joergensen Eric Lee
Malika Marghadi
Hildegard Mayr
Nagavalli Pataballa
Sunitha Patel
Srinivas Putrevu
Denis Raphaely
Helen Robertson
Grant Spencer
Glenn Stokol
Tone Thomas
Priya Vennapusa
Lex Van Der Werff

## Graphic Designer

Satish Bettegowda

## Editors

Nita Pavitran
Richard Wallis

## Publisher

Sheryl Domingue

# Contents

Oracle University and En-Sof Informatica E Treinamento Ltda  use only

**2    Creating Stored Functions**

**7    Design Considerations for PL/SQL Code**

# Preface

**Profile**

### Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL and *i*SQL*Plus, as well as working experience in developing applications. Prerequisites are any of the following Oracle University courses or combinations of courses:

- *Oracle Database 10g: Introduction to SQL*
- *Oracle Database 10g: SQL Fundamentals I* and *Oracle Database 10g: SQL Fundamentals II*
- *Oracle Database 10g: SQL and PL/SQL Fundamentals*
- *Oracle Database 10g: PL/SQL Fundamentals*

### How This Course Is Organized

*Oracle Database 10g: Develop PL/SQL Program Units* is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and practice sessions reinforce the concepts and skills that are introduced.

**Related Publications**

**Oracle Publications**

| Title | Part Number |
| --- | --- |
| *Oracle Database Application Developer's Guide – Fundamentals (10g Release 1)* | *B10795-01* |
| *Oracle Database Application Developer's Guide – Large Objects (10g Release 1)* | *B10796-01* |
| *PL/SQL Packages and Types Reference (10g Release 1)* | *B10802-01* |
| *PL/SQL User's Guide and Reference (10g Release 1)* | *B10807-01* |

**Additional Publications**
- System release bulletins
- Installation and user's guides
- *Read-me* files
- International Oracle Users Group (IOUG) articles
- *Oracle Magazine*

## Typographic Conventions

### Typographic Conventions in Text

| Convention | Element | Example |
|---|---|---|
| Bold | Emphasized words and phrases in Web content only | To navigate within this application, do **not** click the Back and Forward buttons. |
| Bold italic | Glossary terms (if there is a glossary) | The ***algorithm*** inserts the new key. |
| Brackets | Key names | Press [Enter]. |
| Caps and lowercase | Buttons, check boxes, triggers, windows | Click the Executable button. Select the Registration Required check box. Assign a When-Validate-Item trigger. Open the Master Schedule window. |
| Carets | Menu paths | Select File > Save. |
| Commas | Key sequences | Press and release these keys one at a time: [Alt], [F], [D] |

**Typographic Conventions in Text (continued)**

| Convention | Object or Term | Example |
|---|---|---|
| Courier New, case sensitive | Code output, SQL and PL/SQL code elements, Java code elements, directory names, filenames, passwords, pathnames, URLs, user input, usernames | Code output: `debug.seti('I',300);` <br><br> SQL code elements: Use the `SELECT` command to view information stored in the `last_name` column of the `emp` table. <br><br> Java code elements: Java programming involves the `String` and `StringBuffer` classes. <br><br> Directory names: `bin` (DOS), `$FMHOME` (UNIX) <br><br> File names: Locate the `init.ora` file. <br><br> Passwords: Use `tiger` as your password. <br><br> Path names: Open `c:\my_docs\projects`. <br><br> URLs: Go to `http://www.oracle.com`. <br><br> User input: Enter `300`. <br><br> Usernames: Log on as `scott`. |
| Initial cap | Graphics labels (unless the term is a proper noun) | Customer address (*but* Oracle Payables) |
| Italic | Emphasized words and phrases in print publications, titles of books and courses, variables | Do *not* save changes to the database. <br><br> For further information, see *Oracle7 Server SQL Language Reference Manual*. <br><br> Enter *user_id@us.oracle.com*, where *user_id* is the name of the user. |
| Plus signs | Key combinations | Press and hold these keys simultaneously: [Control] + [Alt] + [Delete] |
| Quotation marks | Lesson and chapter titles in cross references, interface elements with long names that have only initial caps | This subject is covered in Unit II, Lesson 3, "Working with Objects." <br><br> Select the "Include a reusable module component" and click Finish. <br><br> Use the "`WHERE` clause of query" property. |

**Typographic Conventions in Navigation Paths**

This course uses simplified navigation paths to direct you through Oracle applications, as in the following example.

**Invoice Batch Summary**

(N) Invoice > Entry > Invoice Batches Summary (M) Query > Find (B) Approve

This simplified path translates to the following sequence of steps:
1. (N) From the Navigator window, select Invoice > Entry > Invoice Batches Summary.
2. (M) From the menu, select Query > Find.
3. (B) Click the Approve button.

**Notation:**

| | |
|---|---|
| (N) = Navigator | (I) = icon |
| (M) = menu | (H) = hyperlink |
| (T) = tab | (B) = button |

# I

# **Introduction**

# Lesson Objectives

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

- **Discuss the goals of the course**
- **Identify the modular components of PL/SQL:**
    - **Anonymous blocks**
    - **Procedures and functions**
    - **Packages**
- **Discuss the PL/SQL execution environment**
- **Describe the database schema and tables that are used in the course**
- **List the PL/SQL development environments that are available in the course**

ORACLE

**Lesson Aim**

PL/SQL supports many program constructs. In this lesson, you review program units in the form of anonymous blocks, and you are introduced to named PL/SQL blocks. The named PL/SQL blocks are also referred to as subprograms. The named PL/SQL blocks include procedures and functions.

The tables from the Human Resources (HR) schema (which is used for the practices in this course) are briefly discussed. The development tools for writing, testing, and debugging PL/SQL are listed.

# Course Objectives

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

- **Create, execute, and maintain:**
  - **Procedures and functions with** OUT **parameters**
  - **Package constructs**
  - **Database triggers**
- **Manage PL/SQL subprograms and triggers**
- **Use a subset of Oracle-supplied packages to:**
  - **Generate screen, file, and Web output**
  - **Schedule PL/SQL jobs to run independently**
- **Build and execute dynamic SQL statements**
- **Manipulate large objects (LOBs)**

ORACLE

**Course Objectives**

You can develop modularized applications with database procedures by using database objects such as the following:
- Procedures and functions
- Packages
- Database triggers

Modular applications improve:
- Functionality
- Security
- Overall performance

# Course Agenda

**Lessons for day 1:**

   I.   **Introduction**

   1.   **Creating Stored Procedures**

   2.   **Creating Stored Functions**

   3.   **Creating Packages**

   4.   **Using More Package Concepts**

ORACLE

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-4**

# Course Agenda

**Lessons for day 2:**

5.  **Using Oracle-Supplied Packages in Application Development**
6.  **Dynamic SQL and Metadata**
7.  **Design Considerations for PL/SQL Code**
8.  **Managing Dependencies**

ORACLE

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-5**

# Course Agenda

**Lessons for day 3:**

9. **Manipulating Large Objects**
10. **Creating Triggers**
11. **Applications for Triggers**
12. **Understanding and Influencing the PL/SQL Compiler**

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-6**
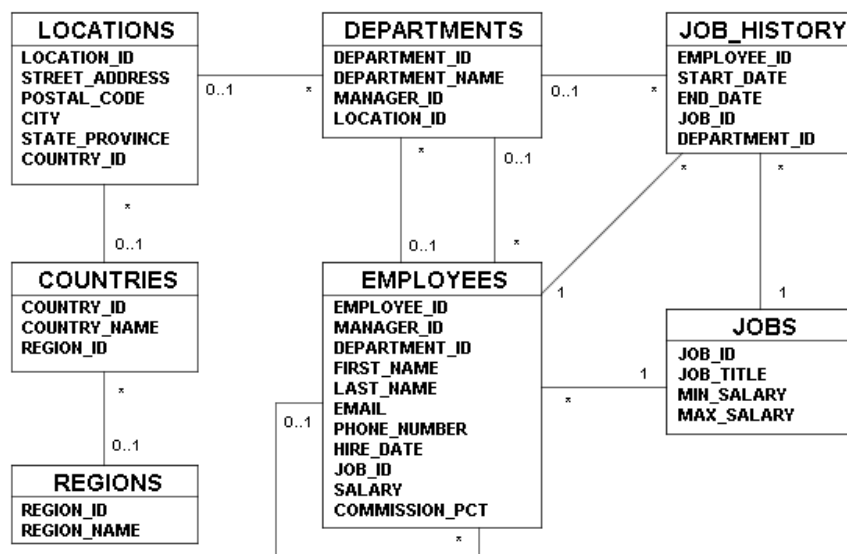
# Human Resources (HR) Schema

## Human Resources (HR) Schema

The Human Resources (HR) schema is part of the Oracle Sample Schemas that can be installed in an Oracle database. The practice sessions in this course use data from the HR schema.

**Table Descriptions**
- REGIONS contains rows that represent a region such as Americas, Asia, and so on.
- COUNTRIES contains rows for countries, each of which is associated with a region.
- LOCATIONS contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
- DEPARTMENTS shows details about the departments in which employees work. Each department may have a relationship representing the department manager in the EMPLOYEES table.
- EMPLOYEES contains details about each employee working for a department. Some employees may not be assigned to any department.
- JOBS contains the job types that can be held by each employee.
- JOB_HISTORY contains the job history of the employees. If an employee changes departments within a job or changes jobs within a department, then a new row is inserted into this table with the old job information of the employee.

**Oracle Database 10g: Develop PL/SQL Program Units   I-7**

# Creating a Modularized and Layered Subprogram Design
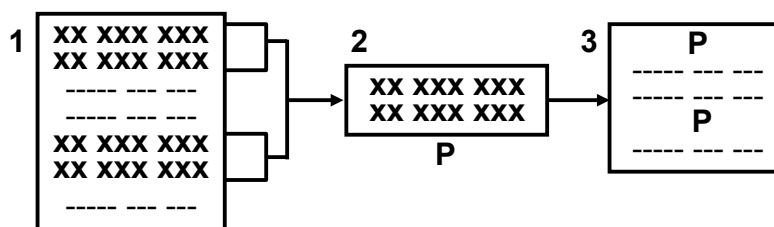


- **Modularize code into subprograms.**
  1. **Locate code sequences repeated more than once.**
  2. **Create subprogram P containing the repeated code.**
  3. **Modify original code to invoke the new subprogram.**
- **Create subprogram layers for your application.**
  - **Data access subprogram layer with SQL logic**
  - **Business logic subprogram layer, which may or may not use data access layer**

ORACLE

**Creating a Modularized and Layered Subprogram Design**

The diagram illustrates the principle of modularization with subprograms: the creation of smaller manageable pieces of flexible and reusable code. Flexibility is achieved by using subprograms with parameters, which in turn makes the same code reusable for different input values. To modularize existing code, perform the following steps:
1. Locate and identify repetitive sequences of code.
2. Move the repetitive code into a PL/SQL subprogram.
3. Replace the original repetitive code with calls to the new PL/SQL subprogram.

**Subprogram Layers**

Because PL/SQL allows SQL statements to be seamlessly embedded into the logic, it is too easy to have SQL statement spread all over the code. However, it is recommended that you keep the SQL logic separate from the business logic—that is, create a layered application design with a minimum of two layers:
- **Data access layer:** For subroutines to access the data by using SQL statements
- **Business logic layer:** For subprograms to implement the business processing rules, which may or may not call on the data access layer routines

Following this modular and layered approach can help you create code that is easier to maintain, particularly when the business rules change. In addition, keeping the SQL logic simple and free of complex business logic can benefit from the work of Oracle Database Optimizer, which can reuse parsed SQL statements for better use of server-side resources.

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-8**

# Modularizing Development
# with PL/SQL Blocks

- **PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:**
  - **Anonymous blocks**
  - **Procedures and functions**
  - **Packages**
  - **Database triggers**
- **The benefits of using modular program constructs are:**
  - **Easy maintenance**
  - **Improved data security and integrity**
  - **Improved performance**
  - **Improved code clarity**

ORACLE

**Modularizing Development with PL/SQL Blocks**

A subprogram is based on standard PL/SQL structures. It contains a declarative section, an executable section, and an optional exception-handling section (for example, anonymous blocks, procedures, functions, packages, and triggers). Subprograms can be compiled and stored in the database, providing modularity, extensibility, reusability, and maintainability.

Modularization converts large blocks of code into smaller groups of code called modules. After modularization, the modules can be reused by the same program or shared with other programs. It is easier to maintain and debug code that comprises smaller modules than it is to maintain code in a single large program. Modules can be easily extended for customization by incorporating more functionality, if required, without affecting the remaining modules of the program.

Subprograms provide easy maintenance because the code is located in one place and any modifications required to the subprogram can therefore be performed in this single location. Subprograms provide improved data integrity and security. The data objects are accessed through the subprogram, and a user can invoke the subprogram only if the appropriate access privilege is granted to the user.

**Note:** Knowing how to develop anonymous blocks is a prerequisite for this course. For detailed information about anonymous blocks, see the course titled *Oracle 10g: PL/SQL Fundamentals*.

# Review of Anonymous Blocks

**Anonymous blocks:**

- **Form the basic PL/SQL block structure**
- **Initiate PL/SQL processing tasks from applications**
- **Can be nested within the executable section of any PL/SQL block**

```
[DECLARE       -- Declaration Section (Optional)
  variable declarations; ... ]
BEGIN          -- Executable Section (Mandatory)
  SQL or PL/SQL statements;
[EXCEPTION     -- Exception Section (Optional)
  WHEN exception THEN statements; ]
END;           -- End of Block (Mandatory)
```

ORACLE

## Review of Anonymous Blocks

Anonymous blocks are typically used for:
- Writing trigger code for Oracle Forms components
- Initiating calls to procedures, functions, and package constructs
- Isolating exception handling within a block of code
- Nesting inside other PL/SQL blocks for managing code flow control

The DECLARE keyword is optional, but it is required if you declare variables, constants, and exceptions to be used within the PL/SQL block.

BEGIN and END are mandatory and require at least one statement between them, either SQL, PL/SQL, or both.

The exception section is optional and is used to handle errors that occur within the scope of the PL/SQL block. Exceptions can be propagated to the caller of the anonymous block by excluding an exception handler for the specific exception, thus creating what is known as an *unhandled* exception.

# Introduction to PL/SQL Procedures

**Procedures are named PL/SQL blocks that perform a sequence of actions.**

```
CREATE PROCEDURE getemp IS  -- header
  emp_id  employees.employee_id%type;
  lname   employees.last_name%type;
BEGIN
  emp_id := 100;
  SELECT last_name INTO lname
  FROM EMPLOYEES
  WHERE employee_id = emp_id;
  DBMS_OUTPUT.PUT_LINE('Last name: '||lname);
END;
/
```

**Introduction to PL/SQL Procedures**

A procedure is a named PL/SQL block that is created by using a CREATE PROCEDURE statement. When called or invoked, a procedure performs a sequence of actions. The anatomy of the procedure includes:

- **The header:** "PROCEDURE getemp IS"
- **The declaration section:** With variable declarations emp_id and lname
- **The executable section:** Contains the PL/SQL and SQL statements used to obtain the last name of employee 100. The executable section makes use of the DBMS_OUTPUT package to print the last name of the employee.
- **The exception section:** Optional and not used in the example

**Note:** Hard-coding the value of 100 for emp_id is inflexible. The procedure would be more reusable if used as a parameter to obtain the employee ID value. Using parameters is covered in the lesson titled "Creating Stored Procedures."

To call a procedure by using an anonymous block, use the following:

```
BEGIN
  getemp;
END;
```

# Introduction to PL/SQL Functions

**Functions are named PL/SQL blocks that perform a sequence of actions and return a value. A function can be invoked from:**

- **Any PL/SQL block**
- **A SQL statement (subject to some restrictions)**

```
CREATE FUNCTION avg_salary RETURN NUMBER IS
  avg_sal employees.salary%type;
BEGIN
  SELECT AVG(salary) INTO avg_sal
  FROM EMPLOYEES;
  RETURN avg_sal;
END;
/
```

## Introduction to PL/SQL Functions

A function is a named PL/SQL block that is created with a CREATE FUNCTION statement. Functions are used to compute a value that must be returned to the caller.

PL/SQL functions follow the same block structure as procedures. However, the header starts with the keyword FUNCTION followed by the function name. The header includes the return data type after the function name (using the keyword RETURN followed by the data type). The example declares a variable called avg_sal in the declaration section, and uses the RETURN statement to return the value retrieved from the SELECT statement. The value returned represents the average salary for all employees.

A function can be called from:
- Another PL/SQL block where its return value can be stored in a variable or supplied as a parameter to a procedure
- A SQL statement, subject to restrictions (This topic is covered in the lesson titled "Creating Stored Functions.")

To call a function from an anonymous block, use the following:
```
BEGIN
  dbms_output.put_line('Average Salary: ' ||
    avg_salary);
END;
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-12**

# Introduction to PL/SQL Packages

**PL/SQL packages have a specification and an optional body. Packages group related subprograms together.**

```
CREATE PACKAGE emp_pkg IS
  PROCEDURE getemp;
  FUNCTION avg_salary RETURN NUMBER;
END emp_pkg;
/
CREATE PACKAGE BODY emp_pkg IS
  PROCEDURE getemp IS ...
  BEGIN ... END;

  FUNCTION avg_salary RETURN NUMBER IS ...
  BEGIN ... RETURN avg_sal; END;
END emp_pkg;
/
```

ORACLE

### Introduction to PL/SQL Packages

A PL/SQL package is typically made up of two parts:
- A package specification declaring the public (accessible) components of the package
- A package body that implements public and private components of the package

A package is used to group related PL/SQL code and constructs together. This helps developers manage and maintain related code in one place. Packages are powerful constructs and provide a way to logically modularize code into application-specific or functional groups.

The example declares a package called emp_package that comprises a procedure getemp and a function avg_salary. The specification defines the procedure and function heading. The package body provides the full implementation of the procedure and function declared in the package specification. The example is incomplete but provides an introduction to the concept of a PL/SQL package. The details about PL/SQL packages are covered in the lesson titled "Creating Packages."

To call the package procedure from an anonymous block, use the following:

```
BEGIN
 emp_pkg.getemp;
END;
```

**Oracle Database 10*g*: Develop PL/SQL Program Units  I-13**

## Introduction to PL/SQL Triggers

**PL/SQL triggers are code blocks that execute when a specified application, database, or table event occurs.**

- **Oracle Forms application triggers are standard anonymous blocks.**
- **Oracle database triggers have a specific structure.**

```
CREATE TRIGGER check_salary
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
DECLARE
  c_min constant number(8,2) := 1000.0;
  c_max constant number(8,2) := 500000.0;
BEGIN
  IF :new.salary > c_max OR
     :new.salary < c_min THEN
    RAISE_APPLICATION_ERROR(-20000,
      'New salary is too small or large');
  END IF;
END;
/
```

### Introduction to PL/SQL Triggers

A trigger is a PL/SQL block that executes when a particular event occurs.

A PL/SQL trigger has two forms:

- **Application trigger:** Block of code that executes when a nominated application event occurs in an Oracle Forms execution environment
- **Database trigger:** Named block of code that is associated and executes when a nominated database or table event occurs

The check_salary trigger example shows a database trigger that executes before either an INSERT or an UPDATE operation occurs on the EMPLOYEES table. The trigger code checks whether the salary column value is within an acceptable range. If the value is outside the specified range, the code uses the RAISE_APPLICATION_ERROR built-in procedure to fail the operation. The :new syntax, which is used in the example, is a special bind/host variable that can be used in row-level triggers.

Triggers are covered in detail in the lesson titled "Creating Triggers."

# PL/SQL Execution Environment

## The PL/SQL run-time architecture:

**PL/SQL engine**

PL/SQL block → PL/SQL block — **procedural** → **Procedural statement executor**

**sql**

**Oracle server**

**SQL statement executor**
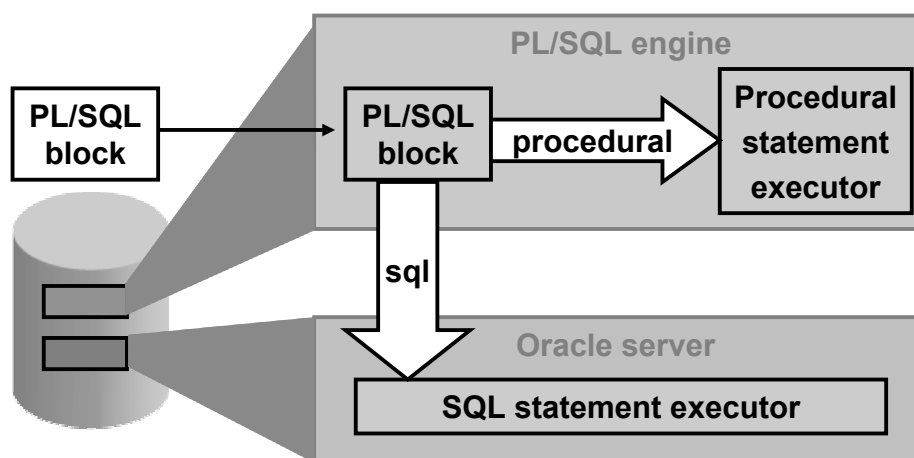
ORACLE

### PL/SQL Execution Environment

The diagram shows a PL/SQL block being executed by the PL/SQL engine. The PL/SQL engine resides in:
- The Oracle database for executing stored subprograms
- The Oracle Forms client when running client/server applications, or in the Oracle Application Server when using Oracle Forms Services to run Forms on the Web

Irrespective of the PL/SQL run-time environment, the basic architecture remains the same. Therefore, all PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle server processes.

The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL m-code instructions. When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the Oracle server processes. The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent statements in the PL/SQL block.

The Oracle Forms PL/SQL engine runs in the client for the client/server implementation, and in the application server for the Forms Services implementation. In either case, SQL statements are typically sent over a network to an Oracle server for processing.

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-15**

**PL/SQL Development Environments**

This course provides the following tools for developing PL/SQL code:

- **Oracle SQL\*Plus (GUI or command-line versions)**
- **Oracle *i*SQL\*Plus (used from a browser)**
- **Oracle JDeveloper IDE**

**PL/SQL Development Environments**

There are many tools that provide an environment for developing PL/SQL code. Oracle provides several tools that can be used to write PL/SQL code. Some of the development tools that are available for use in this course are:

- Oracle *i*SQL\*Plus: A browser-based application
- Oracle SQL\*Plus: A window or command-line application
- Oracle JDeveloper: A window-based integrated development environment (IDE)

**Note:** The code and screen examples presented in the course notes were generated from output in the *i*SQL\*Plus environment.

# Coding PL/SQL in *i*SQL*Plus

## Coding PL/SQL in *i*SQL*Plus

Oracle *i*SQL*Plus is a Web application that allows you to submit SQL statements and PL/SQL blocks for execution and receive the results in a standard Web browser.

*i*SQL*Plus is:
- Shipped with the database
- Installed in the middle tier
- Accessed from a Web browser by using a URL format that is similar to the following example:

      http://host:port/isqlplus

   The host and port are for the Web server name and HTTP listener port.

When coding PL/SQL subprograms in the *i*SQL*Plus tool, consider the following:
- You create subprograms by using the CREATE SQL statement.
- You execute subprograms by using either an anonymous PL/SQL block or the EXECUTE command.
- If you use the DBMS_OUTPUT package procedures to print text to the screen, you must first execute the SET SERVEROUTPUT ON command in your session.

# Coding PL/SQL in SQL*Plus

```
Oracle SQL*Plus                                    _ □ ×
File  Edit  Search  Options  Help

SQL*Plus: Release 10.2.0.1.0 - Production on Tue Apr 4 12:10:08 2006

Copyright (c) 1982, 2005, Oracle.  All rights reserved.


Connected to:
Oracle Database 10g Enterprise Edition Release 10.2.0.1.0 - Production
With the Partitioning, Oracle Label Security, OLAP and Data Mining Scoring Engine options

SQL> set serveroutput on
SQL> create procedure hello is
  2  begin
  3   dbms_output.put_line('Hello World');
  4  end;
  5  /

Procedure created.

SQL> execute hello
Hello World

PL/SQL procedure successfully completed.

SQL>
```

### Coding PL/SQL in SQL*Plus

Oracle SQL*Plus is a graphical user interface (GUI) or command-line application that enables you to submit SQL statements and PL/SQL blocks for execution and receive the results in an application or command window.

SQL*Plus is:
- Shipped with the database
- Installed on a client and on the database server system
- Accessed from an icon or the command line

When coding PL/SQL subprograms using SQL*Plus, consider the following:
- You create subprograms by using the CREATE SQL statement.
- You execute subprograms by using either an anonymous PL/SQL block or the EXECUTE command.
- If you use the DBMS_OUTPUT package procedures to print text to the screen, you must first execute the SET SERVEROUTPUT ON command in your session.

# Coding PL/SQL in Oracle JDeveloper
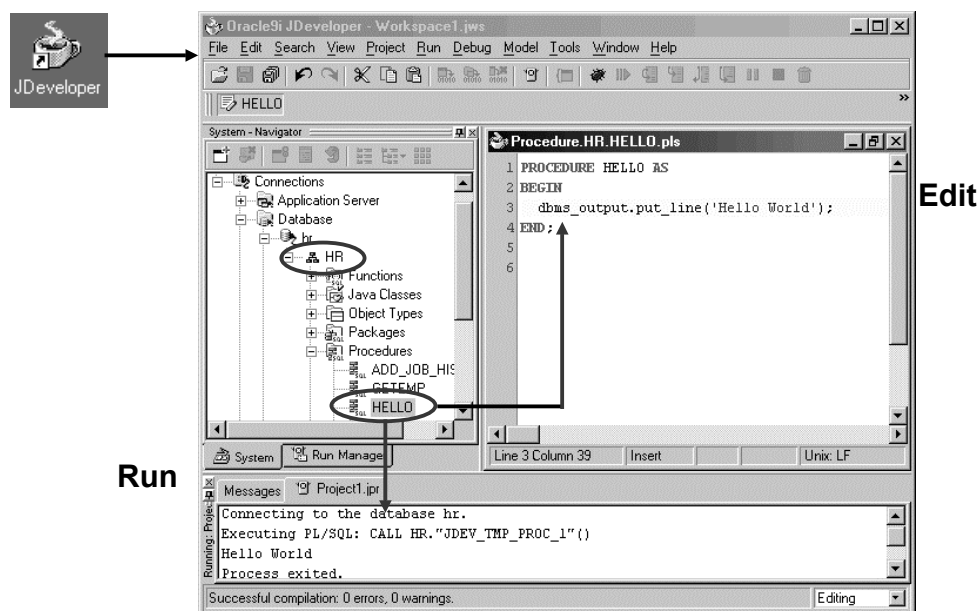
## Coding PL/SQL in Oracle JDeveloper

Oracle JDeveloper allows developers to create, edit, test, and debug PL/SQL code by using a sophisticated GUI. Oracle JDeveloper is a part of Oracle Developer Suite and is also available as a separate product.

When coding PL/SQL in JDeveloper, consider the following:
- You first create a database connection to enable JDeveloper to access a database schema owner for the subprograms.
- You can then use the JDeveloper context menus on the Database connection to create a new subprogram construct using the built-in JDeveloper Code Editor. The JDeveloper Code Editor provides an excellent environment for PL/SQL development, with features such as the following:
  - Different colors for syntactical components of the PL/SQL language
  - Code insight to rapidly locate procedures and functions in supplied packages
- You invoke a subprogram by using a Run command on the context menu for the named subprogram. The output appears in the JDeveloper Log Message window, as shown in the lower portion of the screenshot.

**Note:** JDeveloper provides color-coding syntax in the JDeveloper Code Editor and is sensitive to PL/SQL language constructs and statements.

**Oracle Database 10*g*: Develop PL/SQL Program Units   I-19**

# Summary

**In this lesson, you should have learned how to:**

- **Declare named PL/SQL blocks, including procedures, functions, packages, and triggers**
- **Use anonymous (unnamed) PL/SQL blocks to invoke stored procedures and functions**
- **Use *i*SQL\*Plus or SQL\*Plus to develop PL/SQL code**
- **Explain the PL/SQL execution environment:**
  - **The client-side PL/SQL engine for executing PL/SQL code in Oracle Forms and Oracle Reports**
  - **The server-side PL/SQL engine for executing PL/SQL code stored in an Oracle database**

ORACLE

**Summary**

The PL/SQL language provides different program constructs for blocks of reusable code. Unnamed or anonymous PL/SQL blocks can be used to invoke SQL and PL/SQL actions, procedures, functions, and package components. Named PL/SQL blocks, otherwise known as subprograms, include:

- Procedures
- Functions
- Package procedures and functions
- Triggers

Oracle supplies several tools to develop your PL/SQL functionality. Oracle provides a client-side or middle-tier PL/SQL run-time environment for Oracle Forms and Oracle Reports, and provides a PL/SQL run-time engine inside the Oracle database. Procedures and functions inside the database can be invoked from any application code that can connect to an Oracle database and execute PL/SQL code.

# Practice I: Overview

This practice covers the following topics:
- **Browsing the HR tables**
- **Creating a simple PL/SQL procedure**
- **Creating a simple PL/SQL function**
- **Using an anonymous block to execute the PL/SQL procedure and function**

**Practice I: Overview**

In this practice, you use *i*SQL*Plus to execute SQL statements to examine data in the HR schema. You also create a simple procedure and function that you invoke by using an anonymous block or the EXECUTE command in *i*SQL*Plus. Optionally, you can experiment by creating and executing the PL/SQL code in SQL*Plus.

**Note:** All written practices use *i*SQL*Plus as the development environment. However, you can use any of the tools that are provided in your course environment.

**Practice I**

1.  Launch *i*SQL*Plus using the icon that is provided on your desktop.
    a.  Log in to the database by using the username and database connect string details provided by your instructor (you may optionally write the information here for your records):
        Username: **ora___**
        Password: **oracle**
        Database Connect String/Tnsname: **T___**
    b.  Execute basic SELECT statements to query the data in the DEPARTMENTS, EMPLOYEES, and JOBS tables. Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides a description and some data from each table in the Human Resources schema.
2.  Create a procedure called HELLO to display the text Hello World.
    a.  Create a procedure called HELLO.
    b.  In the executable section, use the DBMS_OUTPUT.PUT_LINE procedure to print the text Hello World, and save the code in the database.
        **Note:** If you get compile-time errors, correct the PL/SQL code and replace the CREATE keyword with the text CREATE OR REPLACE.
    c.  Execute the SET SERVEROUTPUT ON command to ensure that the output from the DBMS_OUTPUT.PUT_LINE procedure is displayed in *i*SQL*Plus.
    d.  Create an anonymous block to invoke the stored procedure.
3.  Create a function called TOTAL_SALARY to compute the sum of all employee salaries.
    a.  Create a function called TOTAL_SALARY that returns a NUMBER.
    b.  In the executable section, execute a query to store the total salary of all employees in a local variable that you declare in the declaration section. Return the value stored in the local variable. Save and compile the code.
    c.  Use an anonymous block to invoke the function. To display the result computed by the function, use the DBMS_OUTPUT.PUT_LINE procedure.
        **Hint:** Either nest the function call inside the DBMS_OUTPUT.PUT_LINE parameter, or store the function result in a local variable of the anonymous block and use the local variable in the DBMS_OUTPUT.PUT_LINE procedure.

**If you have time, complete the following exercise:**

4.  Launch SQL*Plus using the icon that is provided on your desktop.
    a.  Invoke the procedure and function that you created in exercises 2 and 3.
    b.  Create a new procedure called HELLO_AGAIN to print Hello World again.
    c.  Invoke the HELLO_AGAIN procedure with an anonymous block.

# Creating Stored Procedures

1

ORACLE

# Objectives

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

- **Describe and create a procedure**
- **Create procedures with parameters**
- **Differentiate between formal and actual parameters**
- **Use different parameter-passing modes**
- **Invoke a procedure**
- **Handle exceptions in procedures**
- **Remove a procedure**

ORACLE

**Lesson Aim**

In this lesson, you learn to create, execute, and remove procedures with or without parameters. Procedures are the foundation of modular programming in PL/SQL. To make procedures more flexible, it is important that varying data is either calculated or passed into a procedure by using input parameters. Calculated results can be returned to the caller of a procedure by using OUT parameters.

To make your programs robust, you should always manage exception conditions by using the exception-handling features of PL/SQL.

# What Is a Procedure?

**A procedure:**

- **Is a type of subprogram that performs an action**
- **Can be stored in the database as a schema object**
- **Promotes reusability and maintainability**

ORACLE

## Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments). Generally, you use a procedure to perform an action. It has a header, a declaration section, an executable section, and an optional exception-handling section. A procedure is invoked by using the procedure name in the execution section of another PL/SQL block.

A procedure is compiled and stored in the database as a schema object. If you are using the procedures with Oracle Forms and Reports, then they can be compiled within the Oracle Forms or Oracle Reports executables.

Procedures promote reusability and maintainability. When validated, they can be used in any number of applications. If the requirements change, only the procedure needs to be updated.

# Syntax for Creating Procedures

- **Use `CREATE PROCEDURE` followed by the name, optional parameters, and keyword `IS` or `AS`.**
- **Add the `OR REPLACE` option to overwrite an existing procedure.**
- **Write a PL/SQL block containing local variables, a `BEGIN` statement, and an `END` statement (or `END procedure_name`).**

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode] datatype1,
   parameter2 [mode] datatype2, ...)]
IS|AS
   [local_variable_declarations; …]
BEGIN
   -- actions;
END [procedure_name];
```

→ **PL/SQL Block**

ORACLE

**Syntax for Creating Procedures**

You create new procedures with the `CREATE PROCEDURE` statement, which may declare a list of parameters and must define the actions to be performed by the standard PL/SQL block. The `CREATE` clause enables you to create stand-alone procedures that are stored in an Oracle database.

- PL/SQL blocks start with `BEGIN`, optionally preceded by the declaration of local variables. PL/SQL blocks end with either `END` or `END procedure_name`.
- The `REPLACE` option indicates that if the procedure exists, it is dropped and replaced with the new version created by the statement.

**Other Syntactic Elements**

- *parameter1* represents the name of a parameter.
- The *mode* option defines how a parameter is used: `IN` (default), `OUT`, or `IN OUT`.
- *datatype1* specifies the parameter data type, without any precision.

**Note:** Parameters can be considered as local variables. Substitution and host (bind) variables cannot be referenced anywhere in the definition of a PL/SQL stored procedure.

The `OR REPLACE` option does not require any change in object security, as long as you own the object and have the `CREATE [ANY] PROCEDURE` privilege.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-4**

# Developing Procedures



**file.sql**

1. Edit
2. Load
3. Create (compile and store)
4. Execute

ORACLE *i*SQL*Plus

**Use SHOW ERRORS for compilation errors**
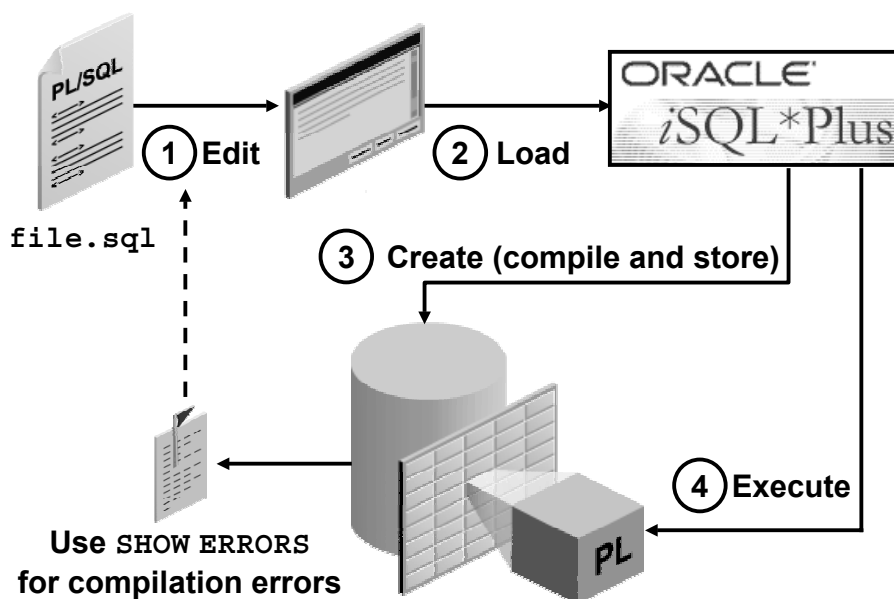
ORACLE

## Developing Procedures

To develop a stored procedure, perform the following steps:

1. Write the code to create a procedure in an editor or a word processor, and then save it as a SQL script file (typically with an `.sql` extension).
2. Load the code into one of the development tools such as SQL*Plus or *i*SQL*Plus.
3. Create the procedure in the database. The CREATE PROCEDURE statement compiles and stores source code and the compiled *m-code* in the database. If compilation errors exist, then the *m-code* is not stored and you must edit the source code to make corrections. You cannot invoke a procedure that contains compilation errors. To view the compilation errors in SQL*Plus or *i*SQL*Plus, use:
   - SHOW ERRORS for the most recently (last) compiled procedure
   - SHOW ERRORS PROCEDURE procedure_name for any procedure compiled previously
4. After successful compilation, execute the procedure to perform the desired action. Use the EXECUTE command from *i*SQL*Plus or an anonymous PL/SQL block from environments that support PL/SQL.

**Note:** If compilation errors occur, use a CREATE OR REPLACE PROCEDURE statement to overwrite the existing code if you previously used a CREATE PROCEDURE statement. Otherwise, DROP the procedure first and then execute the CREATE PROCEDURE statement.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-5**

# What Are Parameters?

**Parameters:**

- **Are declared after the subprogram name in the PL/SQL header**
- **Pass or communicate data between the caller and the subprogram**
- **Are used like local variables but are dependent on their parameter-passing mode:**
  - **An `IN` parameter (the default) provides values for a subprogram to process.**
  - **An `OUT` parameter returns a value to the caller.**
  - **An `IN OUT` parameter supplies an input value, which may be returned (output) as a modified value.**

ORACLE

## What Are Parameters?

Parameters are used to transfer data values to and from the calling environment and the procedure (or subprogram). Parameters are declared in the subprogram header, after the name and before the declaration section for local variables.

Parameters are subject to one of the three parameter-passing modes: IN, OUT, or IN OUT.
- An IN parameter passes a constant value from the calling environment into the procedure.
- An OUT parameter passes a value from the procedure to the calling environment.
- An IN OUT parameter passes a value from the calling environment to the procedure and a possibly different value from the procedure back to the calling environment using the same parameter.

Parameters can be thought of as a special form of local variable, whose input values are initialized by the calling environment when the subprogram is called, and whose output values are returned to the calling environment when the subprogram returns control to the caller.

# Formal and Actual Parameters

- **Formal parameters: Local variables declared in the parameter list of a subprogram specification**

  **Example:**

```
CREATE PROCEDURE raise_sal(id NUMBER,sal NUMBER) IS
BEGIN ...
END raise_sal;
```

- **Actual parameters: Literal values, variables, and expressions used in the parameter list of the called subprogram**

  **Example:**

```
emp_id := 100;
raise_sal(emp_id, 2000)
```

## Formal and Actual Parameters

Formal parameters are local variables that are declared in the parameter list of a subprogram specification. In the first example, in the raise_sal procedure, the variable id and sal identifiers represent the formal parameters.

The actual parameters can be literal values, variables, and expressions that are provided in the parameter list of a called subprogram. In the second example, a call is made to raise_sal, where the emp_id variable provides the actual parameter value for the id formal parameter and 2000 is supplied as the actual parameter value for sal. Actual parameters:
- Are associated with formal parameters during the subprogram call
- Can also be expressions, as in the following example:
  raise_sal(emp_id,raise+100);

The formal and actual parameters should be of compatible data types. If necessary, before assigning the value, PL/SQL converts the data type of the actual parameter value to that of the formal parameter.

# Procedural Parameter Modes

- **Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.**
- **The `IN` mode is the default if no mode is specified.**

```
CREATE PROCEDURE procedure(param [mode] datatype)
...
```

**Modes**

Calling environment → IN (default) → PL/SQL

OUT

IN OUT
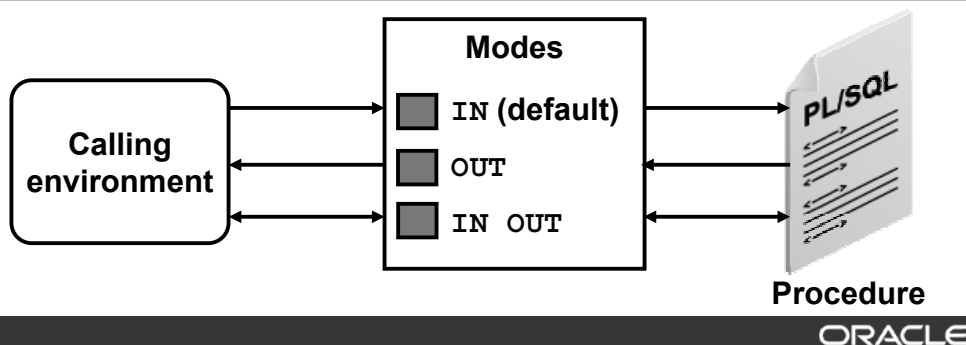
**Procedure**

## Procedural Parameter Modes

When you create the procedure, the formal parameter defines a variable name whose value is used in the executable section of the PL/SQL block. The actual parameter is used when invoking the procedure to provide input values or receive output results.

The parameter mode `IN` is the default passing mode—that is, if no mode is specified with a parameter declaration, the parameter is considered to be an `IN` parameter. The parameter modes `OUT` and `IN OUT` must be explicitly specified in their parameter declarations.

The `datatype` parameter is specified without a size specification. It can be specified:

- As an explicit data type
- Using the `%TYPE` definition
- Using the `%ROWTYPE` definition

**Note:** One or more formal parameters can be declared, with each separated by a comma.

# Using **IN** Parameters: Example

```
CREATE OR REPLACE PROCEDURE raise_salary
  (id      IN employees.employee_id%TYPE,
   percent IN NUMBER)
IS
BEGIN
  UPDATE employees
  SET     salary = salary * (1 + percent/100)
  WHERE   employee_id = id;
END raise_salary;
/
```

```
EXECUTE raise_salary(176,10)
```

## Using **IN** Parameters: Example

The example shows a procedure with two **IN** parameters. Running this first statement in *i*SQL*Plus creates the raise_salary procedure in the database. The second example invokes raise_salary and provides the first parameter value of 176 for the employee ID, and a percentage salary increase of 10 percent for the second parameter value.

To invoke a procedure in *i*SQL*Plus, use the following EXECUTE command:

```
EXECUTE raise_salary (176, 10)
```

To invoke a procedure from another procedure, use a direct call inside an executable section of the calling block. At the location of calling the new procedure, enter the procedure name and actual parameters. For example:

```
...
BEGIN
  raise_salary (176, 10);
END;
```

**Note:** IN parameters are passed as read-only values from the calling environment into the procedure. Attempts to change the value of an IN parameter result in a compile-time error.

# Using OUT Parameters: Example

```
CREATE OR REPLACE PROCEDURE query_emp
 (id     IN  employees.employee_id%TYPE,
  name   OUT employees.last_name%TYPE,
  salary OUT employees.salary%TYPE) IS
BEGIN
  SELECT   last_name, salary INTO name, salary
   FROM    employees
   WHERE   employee_id = id;
END query_emp;
```

```
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal); ...
END;
```

## Using OUT Parameters: Example

In this example, you create a procedure with OUT parameters to retrieve information about an employee. The procedure accepts the value 171 for employee ID and retrieves the name and salary of the employee with ID 171 into the two OUT parameters. The query_emp procedure has three formal parameters. Two of them are OUT parameters that return values to the calling environment, shown in the code box in the lower portion of the slide. The procedure accepts an employee ID value through the id parameter. The emp_name and emp_salary variables are populated with the information retrieved from the query into their two corresponding OUT parameters.

If you print the values returned into PL/SQL variables of the calling block shown in the second block of code, then the variables contain the following:
- emp_name holds the value Smith.
- emp_salary holds the value 7600.

**Note:** Make sure that the data type for the actual parameter variables used to retrieve values from OUT parameters has a size sufficient to hold the data values being returned.

Attempting to use or read OUT parameters inside the procedure that declares them results in a compilation error. The OUT parameters can be assigned values only in the body of the procedure in which they are declared.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-10**

# Viewing OUT Parameters with *i*SQL*Plus

- **Use PL/SQL variables that are printed with calls to the DBMS_OUTPUT.PUT_LINE procedure.**

```
SET SERVEROUTPUT ON
DECLARE
  emp_name employees.last_name%TYPE;
  emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, emp_name, emp_sal);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_name);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || emp_sal);
END;
```

- **Use *i*SQL*Plus host variables, execute QUERY_EMP using host variables, and print the host variables.**

```
VARIABLE name VARCHAR2(25)
VARIABLE sal  NUMBER
EXECUTE query_emp(171, :name, :sal)
PRINT name sal
```

ORACLE

### Viewing OUT Parameters with *i*SQL*Plus

The examples show two ways to view the values returned from OUT parameters.
- The first technique uses PL/SQL variables in an anonymous block to retrieve the OUT parameter values. The DBMS_OUPUT.PUT_LINE procedure is called to print the values held in the PL/SQL variables. The SET SERVEROUPUT must be ON.
- The second technique shows how to use *i*SQL*Plus variables that are created using the VARIABLE command. The *i*SQL*Plus variables are external to the PL/SQL block and are known as host or bind variables. To reference host variables from a PL/SQL block, you must prefix their names with a colon (:). To display the values stored in the host variables, you must use the *i*SQL*Plus PRINT command followed by the name of the *i*SQL*Plus variable (without the colon because this is not a PL/SQL command or context).

To use *i*SQL*Plus and host variables when calling a procedure with OUT parameters, perform the following steps:
1. Create an *i*SQL*Plus script file by using an editor.
2. Add commands to create the variables, execute the procedure, and print the variables.
3. Load and execute the *i*SQL*Plus script file.

**Note:** For details about the VARIABLE command, see the *i*SQL*Plus Command Reference.

# Calling PL/SQL Using Host Variables

**A host variable (also known as a *bind* or a *global* variable):**

- **Is declared and exists externally to the PL/SQL subprogram. A host variable can be created in:**
  - *i*SQL*Plus by using the `VARIABLE` command
  - Oracle Forms internal and UI variables
  - Java variables
- **Is preceded by a colon (`:`) when referenced in PL/SQL code**
- **Can be referenced in an anonymous block but not in a stored subprogram**
- **Provides a value to a PL/SQL block and receives a value from a PL/SQL block**

ORACLE

**Calling PL/SQL Using Host Variables**

The PL/SQL code that is stored in the database can be called from a variety of environments, such as:

- SQL*Plus or *i*SQL*Plus
- Oracle Forms and Oracle Reports
- Java and C applications

Each of the preceding environments provides ways to declare variables to store data in memory. The variable values in these applications are defined and held external to stored PL/SQL code. Each environment provides a way to pass the variable data to PL/SQL and receive updated values from the PL/SQL code. In general, most languages host calls to PL/SQL blocks or subprograms. The PL/SQL engine uses a technique called binding to associate values supplied from external locations to PL/SQL variables or parameters declared in the PL/SQL subprograms.

Unlike in Java, PL/SQL recognizes host variables by the presence of a colon prefixed to the external variable name when it is used in a PL/SQL block.

You cannot store PL/SQL code with host variables because the compiler cannot resolve references to host variables. The binding process is done at run time.

# Using IN OUT Parameters: Example

**Calling environment**

| phone_no (before the call) | phone_no (after the call) |
|---|---|
| '8006330575' | '(800)633-0575' |

```
CREATE OR REPLACE PROCEDURE format_phone
  (phone_no IN OUT VARCHAR2) IS
BEGIN
  phone_no := '(' || SUBSTR(phone_no,1,3) ||
              ')' || SUBSTR(phone_no,4,3) ||
              '-' || SUBSTR(phone_no,7);
END format_phone;
/
```
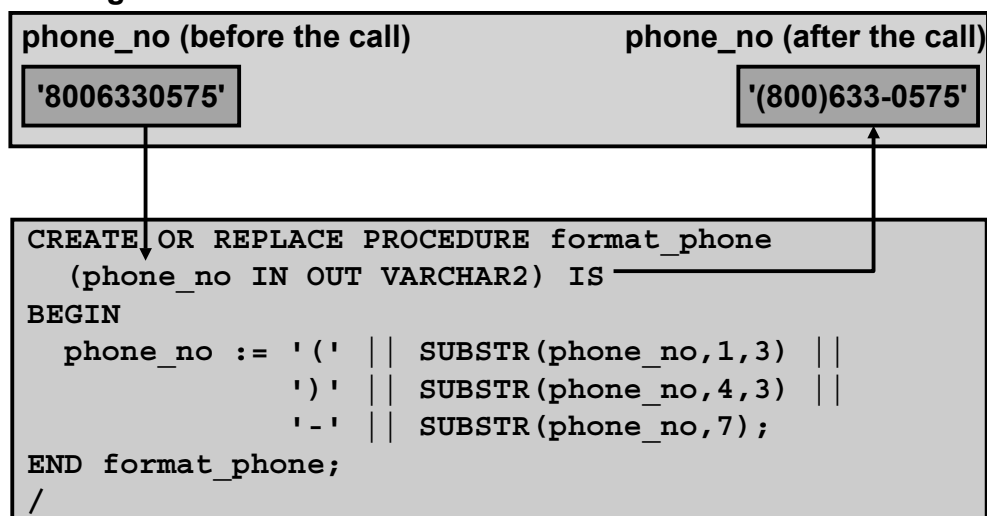
## Using IN OUT Parameters: Example

Using an IN OUT parameter, you can pass a value into a procedure that can be updated. The actual parameter value supplied from the calling environment can return as either of the following:

- The original unchanged value
- A new value that is set within the procedure

**Note:** An IN OUT parameter acts as an initialized variable.

The example in the slide creates a procedure with an IN OUT parameter to accept a 10-character string containing digits for a phone number. The procedure returns the phone number formatted with parentheses around the first three characters and a hyphen after the sixth digit—for example, the phone string 8006330575 is returned as (800)633-0575.

The following code uses the phone_no host variable of *i*SQL*Plus to provide the input value passed to the FORMAT_PHONE procedure. The procedure is executed and returns an updated string in the phone_no host variable.

```
VARIABLE phone_no VARCHAR2(15)
EXECUTE :phone_no := '8006330575'
PRINT phone_no
EXECUTE format_phone (:phone_no)
PRINT phone_no
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-13**

# Syntax for Passing Parameters

- **Positional:**
  - **Lists the actual parameters in the same order as the formal parameters**
- **Named:**
  - **Lists the actual parameters in arbitrary order and uses the association operator (=>) to associate a named formal parameter with its actual parameter**
- **Combination:**
  - **Lists some of the actual parameters as positional and some as named**

**Syntax for Passing Parameters**

For a procedure that contains multiple parameters, you can use a number of methods to specify the values of the parameters. The methods are:

- **Positional:** Lists the actual parameter values in the order in which the formal parameters are declared
- **Named:** Lists the actual values in arbitrary order and uses the association operator to associate each actual parameter with its formal parameter by name. The PL/SQL association operator is an "equal" sign followed by an "is greater than" sign, without spaces: =>.
- **Combination:** Lists the first parameter values by their position and the remainder by using the special syntax of the named method

The next page shows some examples of the first two methods.

# Parameter Passing: Examples

```
CREATE OR REPLACE PROCEDURE add_dept(
  name IN departments.department_name%TYPE,
  loc  IN departments.location_id%TYPE) IS
BEGIN
  INSERT INTO departments(department_id,
           department_name, location_id)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
/
```

- **Passing by positional notation:**

```
EXECUTE add_dept ('TRAINING', 2500)
```

- **Passing by named notation:**

```
EXECUTE add_dept (loc=>2400, name=>'EDUCATION')
```

ORACLE

**Parameter Passing: Examples**

In the example, the add_dept procedure declares two IN parameters: name and loc. The values of these parameters are used in the INSERT statement to set the department_name and location_id columns, respectively.

Passing parameters by position is shown in the first call to execute add_dept below the procedure definition. The first actual parameter supplies the value TRAINING for the name parameter. The second actual parameter value of 2500 is assigned by position to the loc parameter.

Passing parameters using the named notation is shown in the last example. Here, the loc parameter, which is declared as the second formal parameter, is referenced by name in the call, where it is associated to the actual value of 2400. The name parameter is associated to the value EDUCATION. The order of the actual parameters is irrelevant if all parameter values are specified.

**Note:** You must provide a value for each parameter unless the formal parameter is assigned a default value. Specifying default values for formal parameters is discussed next.

# Using the DEFAULT Option for Parameters

- **Defines default values for parameters:**

```
CREATE OR REPLACE PROCEDURE add_dept(
 name departments.department_name%TYPE:='Unknown',
 loc  departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments (...)
  VALUES (departments_seq.NEXTVAL, name, loc);
END add_dept;
```

- **Provides flexibility by combining the positional and named parameter-passing syntax:**

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', loc => 1200)
EXECUTE add_dept (loc => 1200)
```

ORACLE

**Using the DEFAULT Option for Parameters**

The code examples in the slide show two ways of assigning a default value to an IN parameter. The two ways shown use:
- The assignment operator (:=), as shown for the name parameter
- The DEFAULT option, as shown for the loc parameter

When default values are assigned to formal parameters, you can call the procedure without supplying an actual parameter value for the parameter. Thus, you can pass different numbers of actual parameters to a subprogram, either by accepting or by overriding the default values as required. It is recommended that you declare parameters without default values first. Then, you can add formal parameters with default values without having to change every call to the procedure.

**Note:** You cannot assign default values to OUT and IN OUT parameters.

The slide shows three ways of invoking the add_dept procedure:
- The first example assigns the default values for each parameter.
- The second example illustrates a combination of position and named notation to assign values. In this case, using named notation is presented as an example.
- The last example uses the default value for the name parameter and the supplied value for the loc parameter.

### Using the `DEFAULT` Option for Parameters (continued)

Usually, you can use named notation to override the default values of formal parameters. However, you cannot skip providing an actual parameter if there is no default value provided for a formal parameter.

**Note:** All the positional parameters should precede the named parameters in a subprogram call. Otherwise, you receive an error message, as shown in the following example:

```
EXECUTE add_dept(name=>'new dept', 'new location')
```

The following error message is generated:

```
ERROR at line 1:
ORA-06550: line 1, column 34:
PLS-00312: a positional parameter association may not follow a named association
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

# Summary of Parameter Modes

| IN | OUT | IN OUT |
|---|---|---|
| Default mode | Must be specified | Must be specified |
| Value is passed into subprogram | Returned to calling environment | Passed into subprogram; returned to calling environment |
| Formal parameter acts as a constant | Uninitialized variable | Initialized variable |
| Actual parameter can be a literal, expression, constant, or initialized variable | Must be a variable | Must be a variable |
| Can be assigned a default value | Cannot be assigned a default value | Cannot be assigned a default value |

## Summary of Parameter Modes

The IN parameter mode is the default mode if no mode is specified in the declaration. The OUT and IN OUT parameter modes must be explicitly specified with the parameter declaration.

A formal parameter of IN mode cannot be assigned a value and cannot be modified in the body of the procedure. By default, the IN parameter is passed by reference. An IN parameter can be assigned a default value in the formal parameter declaration, in which case the caller need not provide a value for the parameter if the default applies.

An OUT or IN OUT parameter must be assigned a value before returning to the calling environment. The OUT and IN OUT parameters cannot be assigned default values. To improve performance with OUT and IN OUT parameters, the NOCOPY compiler hint can be used to request to pass by reference.

**Note:** Using NOCOPY is discussed later in this course.

# Invoking Procedures

**You can invoke procedures by:**

- **Using anonymous blocks**
- **Using another procedure, as in the following example:**

```
CREATE OR REPLACE PROCEDURE process_employees
IS
   CURSOR emp_cursor IS
    SELECT employee_id
    FROM   employees;
BEGIN
   FOR emp_rec IN emp_cursor
   LOOP
     raise_salary(emp_rec.employee_id, 10);
   END LOOP;
   COMMIT;
END process_employees;
/
```

## Invoking Procedures

You can invoke procedures by using:
- Anonymous blocks
- Another procedure or PL/SQL subprogram

Examples on the preceding pages have illustrated how to use anonymous blocks (or the EXECUTE command in *i*SQL*Plus).

This example shows you how to invoke a procedure from another stored procedure. The PROCESS_EMPLOYEES stored procedure uses a cursor to process all the records in the EMPLOYEES table and passes each employee's ID to the RAISE_SALARY procedure, which results in a 10% salary increase across the company.

# Handled Exceptions

**Calling procedure**                    **Called procedure**

```
PROCEDURE                    PROCEDURE
 PROC1 ...                    PROC2 ...
IS                           IS
 ...                          ...
BEGIN                        BEGIN
 ...                          ...              ← Exception raised
  PROC2(arg1);              EXCEPTION
 ...                          ...              ← Exception handled
EXCEPTION                   END PROC2;
 ...
END PROC1;
```

**Control returns
to calling procedure**
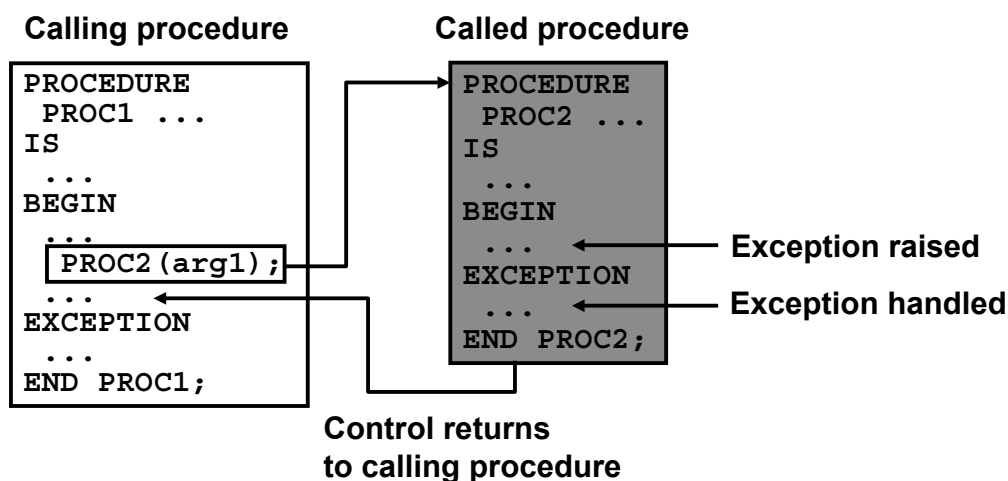
## Handled Exceptions

When you develop procedures that are called from other procedures, you should be aware of the effects that handled and unhandled exceptions have on the transaction and the calling procedure.

When an exception is raised in a called procedure, the control immediately goes to the exception section of that block. An exception is considered handled if the exception section provides a handler for the exception raised.

When an exception occurs and is handled, the following code flow takes place:
1. The exception is raised.
2. Control is transferred to the exception handler.
3. The block is terminated.
4. The calling program/block continues to execute as if nothing has happened.

If a transaction was started (that is, if any data manipulation language [DML] statements executed before executing the procedure in which the exception was raised), then the transaction is unaffected. A DML operation is rolled back if it was performed within the procedure before the exception.

**Note:** You can explicitly end a transaction by executing a COMMIT or ROLLBACK operation in the exception section.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-20**

# Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
    name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
EXCEPTION
 WHEN OTHERS THEN
  DBMS_OUTPUT.PUT_LINE('Err: adding dept: '||name);
END;
```

```
CREATE PROCEDURE create_departments IS
BEGIN
  add_department('Media', 100, 1800);
  add_department('Editing', 99, 1800);
  add_department('Advertising', 101, 1800);
END;
```

ORACLE

### Handled Exceptions: Example

The two procedures in the example are the following:
- The add_department procedure creates a new department record by allocating a new department number from an Oracle sequence, and sets the department_name, manager_id, and location_id column values using the name, mgr, and loc parameters, respectively.
- The create_departments procedure creates more than one department by using calls to the add_department procedure.

The add_department procedure catches all raised exceptions in its own handler. When create_departments is executed, the following output is generated:

```
Added Dept: Media
Err: Adding Dept: Editing
Added Dept: Advertising
```

The Editing department with manager_id of 99 is not inserted because of a foreign key integrity constraint violation on the manager_id. Because the exception was handled in the add_department procedure, the create_department procedure continues to execute. A query on the DEPARTMENTS table where the location_id is 1800 shows that Media and Advertising are added but the Editing record is not.

# Exceptions Not Handled

**Calling procedure**                    **Called procedure**

```
PROCEDURE                      PROCEDURE
 PROC1 ...                      PROC2 ...
IS                             IS
 ...                            ...
BEGIN                          BEGIN
 ...                            ...          ← Exception raised
  PROC2(arg1);                 EXCEPTION
 ...                            ...          ✖ Exception not
EXCEPTION                      END PROC2;       handled
 ...
END PROC1;
```

**Control returned
to exception section
of calling procedure**
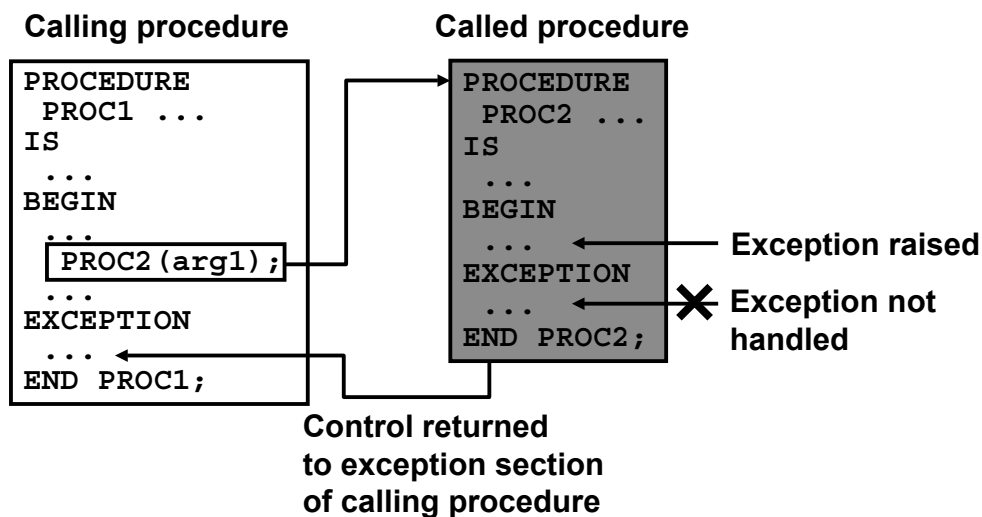
## Exceptions Not Handled

As discussed, when an exception is raised in a called procedure, control immediately goes to the exception section of that block. If the exception section does not provide a handler for the raised exception, then it is not handled. The following code flow occurs:

1. The exception is raised.
2. The block terminates because no exception handler exists; any DML operations performed within the procedure are rolled back.
3. The exception propagates to the exception section of the calling procedure—that is, control is returned to the exception section of the calling block, if one exists.

If an exception is not handled, then all the DML statements in the calling procedure and the called procedure are rolled back along with any changes to any host variables. The DML statements that are not affected are statements that were executed before calling the PL/SQL code whose exceptions are not handled.

# Exceptions Not Handled: Example

```
SET SERVEROUTPUT ON
CREATE PROCEDURE add_department_noex(
     name VARCHAR2, mgr NUMBER, loc NUMBER) IS
BEGIN
  INSERT INTO DEPARTMENTS (department_id,
    department_name, manager_id, location_id)
  VALUES (DEPARTMENTS_SEQ.NEXTVAL, name, mgr, loc);
  DBMS_OUTPUT.PUT_LINE('Added Dept: '||name);
END;
```

```
CREATE PROCEDURE create_departments_noex IS
BEGIN
  add_department_noex('Media', 100, 1800);
  add_department_noex('Editing', 99, 1800);
  add_department_noex('Advertising', 101, 1800);
END;
```

### Exceptions Not Handled: Example

The code example in the slide shows add_department_noex, which does not have an exception section. In this case, the exception occurs when the Editing department is added. Because of the lack of exception handling in either of the subprograms, no new department records are added into the DEPARTMENTS table. Executing the create_departments_noex procedure produces a result that is similar to the following:

```
Added Dept: Media
BEGIN create_departments_noex; END;

*
ERROR at line 1:
ORA-02291: integrity constraint (ORA1.DEPT_MGR_FK)
violated - parent key not
found
ORA-06512: at "ORA1.ADD_DEPARTMENT_NOEX", line 4
ORA-06512: at "ORA1.CREATE_DEPARTMENTS_NOEX", line 4
ORA-06512: at line 1
```

Although the results show that the Media department was added, its operation is rolled back because the exception was not handled in either of the subprograms invoked.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-23**

# Removing Procedures

**You can remove a procedure that is stored in the database.**

- **Syntax:**

```
DROP PROCEDURE procedure_name
```

- **Example:**

```
DROP PROCEDURE raise_salary;
```

**Removing Procedures**

When a stored procedure is no longer required, you can use the DROP PROCEDURE SQL statement to remove it.

**Note:** Whether successful or not, executing a data definition language (DDL) command such as DROP PROCEDURE commits any pending transactions that cannot be rolled back.

# Viewing Procedures in the Data Dictionary

**Information for PL/SQL procedures is saved in the following data dictionary views:**

- **View source code in the `USER_SOURCE` table to view the subprograms that you own, or the `ALL_SOURCE` table for procedures that are owned by others who have granted you the `EXECUTE` privilege.**

```
SELECT text
FROM   user_source
WHERE  name='ADD_DEPARTMENT' and type='PROCEDURE'
ORDER BY line;
```

- **View the names of procedures in `USER_OBJECTS`.**

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'PROCEDURE';
```

ORACLE

## Viewing Procedures in the Data Dictionary

The source code for PL/SQL subprograms is stored in the data dictionary tables. The source code is accessible to PL/SQL procedures that are successfully or unsuccessfully compiled. To view the PL/SQL source code stored in the data dictionary, execute a SELECT statement on the following tables:

- The USER_SOURCE table to display PL/SQL code that you own
- The ALL_SOURCE table to display PL/SQL code to which you have been granted the EXECUTE right by the owner of that subprogram code

The query example shows all the columns provided by the USER_SOURCE table:

- The TEXT column holds a line of PL/SQL source code.
- The NAME column holds the name of the subprogram in uppercase text.
- The TYPE column holds the subprogram type, such as PROCEDURE or FUNCTION.
- The LINE column stores the line number for each source code line.

The ALL_SOURCE table provides an OWNER column in addition to the preceding columns.

**Note:** You cannot display the source code for Oracle PL/SQL built-in packages, or PL/SQL whose source code has been wrapped by using a WRAP utility. The WRAP utility converts the PL/SQL source code into a form that cannot be deciphered by humans.

# Benefits of Subprograms

- **Easy maintenance**
- **Improved data security and integrity**
- **Improved performance**
- **Improved code clarity**

## Benefits of Subprograms

Procedures and functions have many benefits due to the modularizing of the code:
- **Easy maintenance** is realized because subprograms are located in one place. Modifications need to be done in only one place to affect multiple applications and minimize excessive testing.
- **Improved data security** can be achieved by controlling indirect access to database objects from nonprivileged users with security privileges. The subprograms are by default executed with definer's right. The execute privilege does not allow a calling user direct access to objects that are accessible to the subprogram.
- **Data integrity** is managed by having related actions performed together or not at all.
- **Improved performance** can be realized from reuse of parsed PL/SQL code that becomes available in the shared SQL area of the server. Subsequent calls to the subprogram avoid parsing the code again. Because PL/SQL code is parsed at compile time, the parsing overhead of SQL statements is avoided at run time. Code can be written to reduce the number of network calls to the database, and therefore, decrease network traffic.
- **Improved code clarity** can be attained by using appropriate names and conventions to describe the action of the routines, thereby reducing the need for comments and enhancing the clarity of the code.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-26**

# Summary

**In this lesson, you should have learned how to:**

- **Write a procedure to perform a task or an action**
- **Create, compile, and save procedures in the database by using the `CREATE PROCEDURE` SQL command**
- **Use parameters to pass data from the calling environment to the procedure by using three different parameter modes: `IN` (the default), `OUT`, and `IN OUT`**
- **Recognize the effect of handling and not handling exceptions on transactions and calling procedures**

ORACLE

## Summary

A procedure is a subprogram that performs a specified action. You can compile and save a procedure as a stored procedure in the database. A procedure can return zero or more values through its parameters to its calling environment. There are three parameter modes: `IN`, `OUT`, and `IN OUT`.

You should be able to handle and not handle exceptions, and you should understand how managing exceptions affects transactions and calling procedures. The exceptions are handled in the exception section of a subprogram.

Aut the right side reads:

# Summary

**In this lesson, you should have learned how to:**

- **Remove procedures from the database by using the `DROP PROCEDURE` SQL command**
- **Modularize your application code by using procedures as building blocks**

ORACLE

**Summary (continued)**

You can modify and remove procedures. Procedures are modular components that form the building blocks of an application. You can also create client-side procedures that can be used by client-side applications.

# Practice 1: Overview

**This practice covers the following topics:**

- **Creating stored procedures to:**
  - **Insert new rows into a table using the supplied parameter values**
  - **Update data in a table for rows that match the supplied parameter values**
  - **Delete rows from a table that match the supplied parameter values**
  - **Query a table and retrieve data based on supplied parameter values**
- **Handling exceptions in procedures**
- **Compiling and invoking procedures**

## Practice 1: Overview

In this practice, you create procedures that issue DML and query commands.

If you encounter compilation errors when you are using *i*SQL*Plus, use the SHOW ERRORS command.

If you correct any compilation errors in *i*SQL*Plus, do so in the original script file (rather than in the buffer) and then rerun the new version of the file. This saves a new version of the procedure to the data dictionary.

**Note:** It is recommended to use *i*SQL*Plus for this practice.

**Practice 1**

**Note:** You can find table descriptions and sample data in Appendix B, "Table Descriptions and Data." Click the Save Script button to save your subprograms as `.sql` files in your local file system.

Remember to enable `SERVEROUTPUT` if you have previously disabled it.

1. Create and invoke the `ADD_JOB` procedure and consider the results.
   a. Create a procedure called `ADD_JOB` to insert a new job into the `JOBS` table. Provide the ID and title of the job using two parameters.
   b. Compile the code; invoke the procedure with `IT_DBA` as job ID and `Database Administrator` as job title. Query the `JOBS` table to view the results.

| JOB_ID | JOB_TITLE | MIN_SALARY | MAX_SALARY |
|--------|-----------|------------|------------|
| IT_DBA | Database Administrator | | |

   c. Invoke your procedure again, passing a job ID of `ST_MAN` and a job title of `Stock Manager`. What happens and why?

   _____

   _____

2. Create a procedure called `UPD_JOB` to modify a job in the `JOBS` table.
   a. Create a procedure called `UPD_JOB` to update the job title. Provide the job ID and a new title using two parameters. Include the necessary exception handling if no update occurs.
   b. Compile the code; invoke the procedure to change the job title of the job ID `IT_DBA` to `Data Administrator`. Query the `JOBS` table to view the results.

| JOB_ID | JOB_TITLE | MIN_SALARY | MAX_SALARY |
|--------|-----------|------------|------------|
| IT_DBA | Data Administrator | | |

   Also check the exception handling by trying to update a job that does not exist. (You can use the job ID `IT_WEB` and the job title `Web Master`.)

3. Create a procedure called `DEL_JOB` to delete a job from the `JOBS` table.
   a. Create a procedure called `DEL_JOB` to delete a job. Include the necessary exception handling if no job is deleted.
   b. Compile the code; invoke the procedure using the job ID `IT_DBA`. Query the `JOBS` table to view the results.

   no rows selected

   Also, check the exception handling by trying to delete a job that does not exist. (Use the `IT_WEB` job ID.) You should get the message that you used in the exception-handling section of the procedure as output.

**Oracle Database 10*g*: Develop PL/SQL Program Units   1-30**

**Practice 1 (continued)**

4.  Create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.

    a.  Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Compile the code and remove the syntax errors.

    b.  Execute the procedure using host variables for the two OUT parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

| SALARY |
|---:|
| 8000 |

| JOB |
|---|
| ST_MAN |

    c.  Invoke the procedure again, passing an EMPLOYEE_ID of 300. What happens and why?

    _____

    _____

# 2

# **Creating Stored Functions**

# Objectives

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

- **Describe the uses of functions**
- **Create stored functions**
- **Invoke a function**
- **Remove a function**
- **Differentiate between a procedure and a function**

ORACLE

**Lesson Aim**

In this lesson, you learn how to create and invoke functions.

# Overview of Stored Functions

**A function:**

- **Is a named PL/SQL block that returns a value**
- **Can be stored in the database as a schema object for repeated execution**
- **Is called as part of an expression or is used to provide a parameter value**

## Overview of Stored Functions

A function is a named PL/SQL block that can accept parameters, be invoked, and return a value. In general, you use a function to compute a value. Functions and procedures are structured alike. A function must return a value to the calling environment, whereas a procedure returns zero or more values to its calling environment. Like a procedure, a function has a header, a declarative section, an executable section, and an optional exception-handling section. A function must have a RETURN clause in the header and at least one RETURN statement in the executable section.

Functions can be stored in the database as schema objects for repeated execution. A function that is stored in the database is referred to as a stored function. Functions can also be created on client-side applications.

Functions promote reusability and maintainability. When validated, they can be used in any number of applications. If the processing requirements change, only the function needs to be updated.
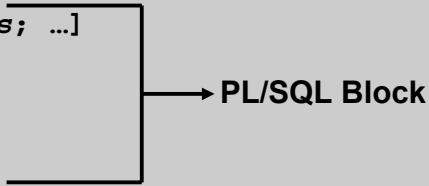
A function may also be called as part of a SQL expression or as part of a PL/SQL expression. In the context of a SQL expression, a function must obey specific rules to control side effects. In a PL/SQL expression, the function identifier acts like a variable whose value depends on the parameters passed to it.

**Oracle Database 10*g*: Develop PL/SQL Program Units   2-3**

# Syntax for Creating Functions

**The PL/SQL block must have at least one `RETURN` statement.**

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1, ...)]
RETURN datatype IS|AS
 [local_variable_declarations; …]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

→ **PL/SQL Block**

### Syntax for Creating Functions

A function is a PL/SQL block that returns a value. A `RETURN` statement must be provided to return a value with a data type that is consistent with the function declaration.

You create new functions with the `CREATE FUNCTION` statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

You should consider the following points about the `CREATE FUNCTION` statement:
- The `REPLACE` option indicates that if the function exists, it is dropped and replaced with the new version that is created by the statement.
- The `RETURN` data type must not include a size specification.
- The PL/SQL block starts with a `BEGIN` after the declaration of any local variables and ends with an `END`, optionally followed by the `function_name`.
- There must be at least one `RETURN expression` statement.
- You cannot reference host or bind variables in the PL/SQL block of a stored function.

**Note:** Although the `OUT` and `IN OUT` parameter modes can be used with functions, it is not good programming practice to use them with functions. However, if you need to return more than one value from a function, consider returning the values in a composite data structure such as a PL/SQL record or a PL/SQL table.

**Oracle Database 10g: Develop PL/SQL Program Units   2-4**

# Developing Functions



**1** Edit    **2** Load

**func.sql**

**3** Create (compile and store)

**4** Execute

Use `SHOW ERRORS` for compilation errors.
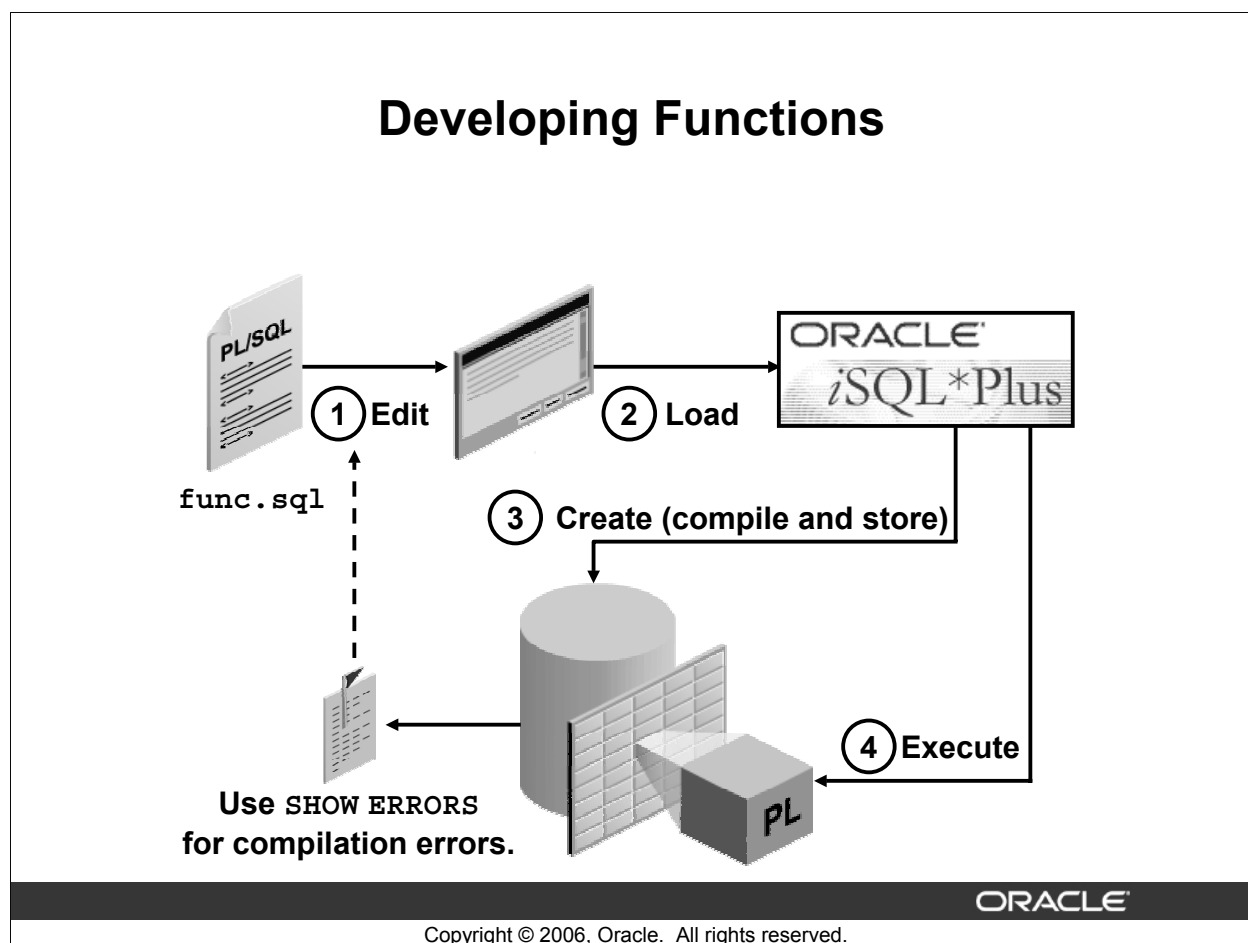
## How to Develop Stored Functions

The diagram illustrates the basic steps involved in developing a stored function. To develop a stored function, perform the following steps:

1. Create a file by using your favorite text or code editor to edit the function syntax, and saving the code in a file typically with a `.sql` extension.
2. Load the function code from the file into the buffer by using *i*SQL*Plus as the PL/SQL development environment.
3. Execute the `CREATE FUNCTION` statement to compile and store the function in the database.
4. After successful compilation, invoke the function from a PL/SQL environment or application.

### Returning a Value

- Add a `RETURN` clause with the data type in the header of the function.
- Include one `RETURN` statement in the executable section.

Multiple `RETURN` statements are allowed in a function (usually within an `IF` statement). Only one `RETURN` statement is executed because after the value is returned, processing of the block ceases.

Use the `SHOW ERRORS` or `SHOW ERRORS FUNCTION` function_name *i*SQL*Plus commands to view compilation errors.

**Oracle Database 10*g*: Develop PL/SQL Program Units 2-5**

# Stored Function: Example

- **Create the function:**

```
CREATE OR REPLACE FUNCTION get_sal
 (id employees.employee_id%TYPE) RETURN NUMBER IS
  sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO   sal
  FROM   employees
  WHERE  employee_id = id;
  RETURN sal;
END get_sal;
/
```

- **Invoke the function as an expression or as a parameter value:**

```
EXECUTE dbms_output.put_line(get_sal(100))
```

Oracle University and En-Sof Informatica E Treinamento Ltda  use only

## Stored Function: Example

The get_sal function is created with a single input parameter and returns the salary as a number. Execute the command as shown, or save it in a script file and run the script to create the get_sal function.

The get_sal function follows a common programming practice of using a single RETURN statement that returns a value assigned to a local variable. If your function has an exception section, then it may also contain a RETURN statement.

Invoke a function as part of a PL/SQL expression because the function will return a value to the calling environment. The second code box uses the *i*SQL*Plus EXECUTE command to call the DBMS_OUTPUT.PUT_LINE procedure whose argument is the return value from the function get_sal. In this case, get_sal is invoked first to calculate the salary of the employee with ID 100. The salary value returned is supplied as the value of the DBMS_OUTPUT.PUT_LINE parameter, which displays the result (if you have executed a SET SERVEROUTPUT ON).

**Note:** A function must always return a value. The example does not return a value if a row is not found for a given id.  Ideally, create an exception handler to return a value as well.

# Ways to Execute Functions

- **Invoke as part of a PL/SQL expression**
    - **Using a host variable to obtain the result:**

```
VARIABLE salary NUMBER
EXECUTE :salary := get_sal(100)
```

    - **Using a local variable to obtain the result:**

```
DECLARE sal employees.salary%type;
BEGIN
  sal := get_sal(100); ...
END;
```

- **Use as a parameter to another subprogram**

```
EXECUTE dbms_output.put_line(get_sal(100))
```

- **Use in a SQL statement (subject to restrictions)**

```
SELECT job_id, get_sal(employee_id) FROM employees;
```

### Ways to Execute Functions

If functions are designed thoughtfully, they can be powerful constructs. Functions can be invoked in the following ways:

- **As part of PL/SQL expressions:** You can use host or local variables to hold the returned value from a function. The first example in the slide uses a host variable and the second example uses a local variable in an anonymous block.
- **As a parameter to another subprogram:** The third example in the slide demonstrates this usage. The get_sal function with all its arguments is nested in the parameter required by the DBMS_OUTPUT.PUT_LINE procedure. This comes from the concept of nesting functions as discussed in the course titled *Oracle Database 10g: SQL Fundamentals I*.
- **As an expression in a SQL statement:** The last example shows how a function can be used as a single-row function in a SQL statement.

**Note:** The benefits and restrictions that apply to functions when used in a SQL statement are discussed in the next few pages.

**Advantages of User-Defined Functions in SQL Statements**

SQL statements can reference PL/SQL user-defined functions anywhere a SQL expression is allowed. For example, a user-defined function can be used anywhere that a built-in SQL function, such as UPPER(), can be placed.

**Advantages**
- Permits calculations that are too complex, awkward, or unavailable with SQL
- Increases data independence by processing complex data analysis within the Oracle server, rather than by retrieving the data into an application
- Increases efficiency of queries by performing functions in the query rather than in the application
- Manipulates new types of data (for example, latitude and longitude) by encoding character strings and using functions to operate on the strings

# Function in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(value IN NUMBER)
 RETURN NUMBER IS
BEGIN
    RETURN (value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM    employees
WHERE   department_id = 100;
```

Function created.

| EMPLOYEE_ID | LAST_NAME | SALARY | TAX(SALARY) |
|---|---|---|---|
| 108 | Greenberg | 12000 | 960 |
| 109 | Faviet | 9000 | 720 |
| 110 | Chen | 8200 | 656 |
| 111 | Sciarra | 7700 | 616 |
| 112 | Urman | 7800 | 624 |
| 113 | Popp | 6900 | 552 |

6 rows selected.

ORACLE

**Function in SQL Expressions: Example**

The example in the slide shows how to create a tax function to calculate income tax. The function accepts a NUMBER parameter and returns the calculated income tax based on a simple flat tax rate of 8%.

In *i*SQL*Plus, the tax function is invoked as an expression in the SELECT clause along with the employee ID, last name, and salary for employees in a department with ID 100. The return result from the tax function is displayed with the regular output from the query.

**Locations to Call User-Defined Functions**

**User-defined functions act like built-in single-row functions and can be used in:**

- **The `SELECT` list or clause of a query**
- **Conditional expressions of the `WHERE` and `HAVING` clauses**
- **The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query**
- **The `VALUES` clause of the `INSERT` statement**
- **The `SET` clause of the `UPDATE` statement**

ORACLE

**Locations to Call User-Defined Functions**

A PL/SQL user-defined function can be called from any SQL expression where a built-in single-row function can be called.

**Example:**
```
SELECT employee_id, tax(salary)
FROM   employees
WHERE  tax(salary) > (SELECT MAX(tax(salary))
                        FROM employees
                        WHERE department_id = 30)
ORDER BY tax(salary) DESC;
```

| EMPLOYEE_ID | TAX(SALARY) |
|---|---|
| 100 | 1920 |
| 101 | 1360 |
| 102 | 1360 |
| 145 | 1120 |
| 146 | 1080 |
| 201 | 1040 |

...

10 rows selected.

**Oracle Database 10*g*: Develop PL/SQL Program Units   2-10**

# Restrictions on Calling Functions from SQL Expressions

- **User-defined functions that are callable from SQL expressions must:**
  - **Be stored in the database**
  - **Accept only IN parameters with valid SQL data types, not PL/SQL-specific types**
  - **Return valid SQL data types, not PL/SQL-specific types**
- **When calling functions in SQL statements:**
  - **Parameters must be specified with positional notation**
  - **You must own the function or have the EXECUTE privilege**

**Restrictions on Calling Functions from SQL Expressions**

The user-defined PL/SQL functions that are callable from SQL expressions must meet the following requirements:
- The function must be stored in the database.
- The function parameters must be input only and valid SQL data types.
- The functions must return data types that are valid SQL data types. They cannot be PL/SQL-specific data types such as BOOLEAN, RECORD, or TABLE. The same restriction applies to the parameters of the function.

The following restrictions apply when calling a function in a SQL statement:
- Parameters must use positional notation. Named notation is not supported.
- You must own or have the EXECUTE privilege on the function.

Other restrictions on a user-defined function include the following:
- It cannot be called from the CHECK constraint clause of a CREATE TABLE or ALTER TABLE statement.
- It cannot be used to specify a default value for a column.

**Note:** Only stored functions are callable from SQL statements. Stored procedures cannot be called unless invoked from a function that meets the preceding requirements.

# Controlling Side Effects When Calling Functions from SQL Expressions

**Functions called from:**

- **A `SELECT` statement cannot contain DML statements**
- **An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`**
- **SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)**

**Note: Calls to subprograms that break these restrictions are also not allowed in the function.**

ORACLE

**Controlling Side Effects When Calling Functions from SQL Expressions**

To execute a SQL statement that calls a stored function, the Oracle server must know whether the function is free of specific side effects. The side effects are unacceptable changes to database tables.

Additional restrictions apply when a function is called in expressions of SQL statements:
- When a function is called from a `SELECT` statement or a parallel `UPDATE` or `DELETE` statement, the function cannot modify database tables.
- When a function is called from an `UPDATE` or `DELETE` statement, the function cannot query or modify database tables modified by that statement.
- When a function is called from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute directly or indirectly through another subprogram or SQL transaction control statements such as:
  - A `COMMIT` or `ROLLBACK` statement
  - A session control statement (such as `SET ROLE`)
  - A system control statement (such as `ALTER SYSTEM`)
  - Any DDL statements (such as `CREATE`) because they are followed by an automatic commit

**Oracle Database 10*g*: Develop PL/SQL Program Units   2-12**

## Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(sal NUMBER)
   RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                 email, hire_date, job_id, salary)
  VALUES(1, 'Frost', 'jfrost@company.com',
         SYSDATE, 'SA_MAN', sal);
  RETURN (sal + 100);
END;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170;
```

```
UPDATE employees SET salary = dml_call_sql(2000)
                 *
ERROR at line 1:
ORA-04091: table PLSQL.EMPLOYEES is mutating,
trigger/function may not see it
ORA-06512: at "PLSQL.DML_CALL_SQL", line 4
```

**Restrictions on Calling Functions from SQL: Example**

The dml_call_sql function in the slide contains an INSERT statement that inserts a new record into the EMPLOYEES table and returns the input salary value incremented by 100. This function is invoked in the UPDATE statement that modifies the salary of employee 170 to the amount returned from the function. The UPDATE statement fails with an error indicating that the table is mutating (that is, changes are already in progress in the same table). In the following example, the query_call_sql function queries the SALARY column of the EMPLOYEES table:

```
CREATE OR REPLACE FUNCTION query_call_sql(a NUMBER)
    RETURN NUMBER IS
    s NUMBER;
  BEGIN
    SELECT salary INTO s FROM employees
    WHERE employee_id = 170;
    RETURN (s + a);
  END;
```

When invoked from the following UPDATE statement, it returns the error message similar to the error message shown in the slide:

```
UPDATE employees SET salary = query_call_sql(100)
WHERE employee_id = 170;
```

**Oracle Database 10*g*: Develop PL/SQL Program Units 2-13**

# Removing Functions

**Removing a stored function:**

- **You can drop a stored function by using the following syntax:**

```
DROP FUNCTION function_name
```

**Example:**

```
DROP FUNCTION get_sal;
```

- **All the privileges that are granted on a function are revoked when the function is dropped.**

- **The CREATE OR REPLACE syntax is equivalent to dropping a function and re-creating it. Privileges granted on the function remain the same when this syntax is used.**

ORACLE

**Removing Functions**

When a stored function is no longer required, you can use a SQL statement in *i*SQL*Plus to drop it. To remove a stored function by using *i*SQL*Plus, execute the DROP FUNCTION SQL command.

**CREATE OR REPLACE Versus DROP and CREATE**

The REPLACE clause in the CREATE OR REPLACE syntax is equivalent to dropping a function and re-creating it. When you use the CREATE OR REPLACE syntax, the privileges granted on this object to other users remain the same. When you DROP a function and then re-create it, all the privileges granted on this function are automatically revoked.

# Viewing Functions in the Data Dictionary

**Information for PL/SQL functions is stored in the following Oracle data dictionary views:**

- **You can view source code in the `USER_SOURCE` table for subprograms that you own, or the `ALL_SOURCE` table for functions owned by others who have granted you the `EXECUTE` privilege.**

```
SELECT text
FROM   user_source
WHERE  type = 'FUNCTION'
ORDER BY line;
```

- **You can view the names of functions by using `USER_OBJECTS`.**

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'FUNCTION';
```

ORACLE

## Viewing Functions in the Data Dictionary

The source code for PL/SQL functions is stored in the data dictionary tables. The source code is accessible for PL/SQL functions that are successfully or unsuccessfully compiled. To view the PL/SQL function code stored in the data dictionary, execute a SELECT statement on the following tables where the TYPE column value is FUNCTION:

- The USER_SOURCE table to display the PL/SQL code that you own
- The ALL_SOURCE table to display the PL/SQL code to which you have been granted the EXECUTE right by the owner of that subprogram code

The first query example shows how to display the source code for all the functions in your schema. The second query, which uses the USER_OBJECTS data dictionary view, lists the names of all functions that you own.

# Procedures Versus Functions

| Procedures | Functions |
|---|---|
| Execute as a PL/SQL statement | Invoke as part of an expression |
| Do not contain `RETURN` clause in the header | Must contain a `RETURN` clause in the header |
| Can return values (if any) in output parameters | Must return a single value |
| Can contain a `RETURN` statement without a value | Must contain at least one `RETURN` statement |

### How Procedures and Functions Differ

You create a procedure to store a series of actions for later execution. A procedure can contain zero or more parameters that can be transferred to and from the calling environment, but a procedure does not have to return a value. A procedure can call a function to assist with its actions.

**Note:** A procedure containing a single `OUT` parameter would be better rewritten as a function returning the value.

You create a function when you want to compute a value that must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions typically return only a single value, and the value is returned through a `RETURN` statement. The functions used in SQL statements should not use `OUT` or `IN OUT` mode parameters. Although a function using output parameters can be used in a PL/SQL procedure or block, it cannot be used in SQL statements.

# Summary

**In this lesson, you should have learned how to:**

- **Write a PL/SQL function to compute and return a value by using the `CREATE FUNCTION` SQL statement**
- **Invoke a function as part of a PL/SQL expression**
- **Use stored PL/SQL functions in SQL statements**
- **Remove a function from the database by using the `DROP FUNCTION` SQL statement**

## Summary

A function is a named PL/SQL block that must return a value. Generally, you create a function to compute and return a value, and you create a procedure to perform an action.

A function can be created or dropped.

A function is invoked as a part of an expression.

# Practice 2: Overview

**This practice covers the following topics:**

- **Creating stored functions:**
  - **To query a database table and return specific values**
  - **To be used in a SQL statement**
  - **To insert a new row, with specified parameter values, into a database table**
  - **Using default parameter values**
- **Invoking a stored function from a SQL statement**
- **Invoking a stored function from a stored procedure**

ORACLE

## Practice 2: Overview

If you encounter compilation errors when using *i*SQL*Plus, use the SHOW ERRORS command.

If you correct any compilation errors in *i*SQL*Plus, do so in the original script file, not in the buffer, and then rerun the new version of the file. This saves a new version of the program unit to the data dictionary.

**Note:** It is recommended to use *i*SQL*Plus for this practice.

**Practice 2**

1. Create and invoke the `GET_JOB` function to return a job title.
   a. Create and compile a function called `GET_JOB` to return a job title.
   b. Create a `VARCHAR2` host variable called `TITLE`, allowing a length of 35 characters. Invoke the function with `SA_REP` job ID to return the value in the host variable. Print the host variable to view the result.

| TITLE |
|---|
| Sales Representative |

2. Create a function called `GET_ANNUAL_COMP` to return the annual salary computed from an employee's monthly salary and commission passed as parameters.
   a. Develop and store the `GET_ANNUAL_COMP` function, accepting parameter values for monthly salary and commission. Either or both values passed can be `NULL`, but the function should still return a non-`NULL` annual salary. Use the following basic formula to calculate the annual salary:
      `(salary*12) + (commission_pct*salary*12)`
   b. Use the function in a `SELECT` statement against the `EMPLOYEES` table for employees in department 30.

| EMPLOYEE_ID | LAST_NAME | Annual Compensation |
|---|---|---|
| 114 | Raphaely | 132000 |
| 115 | Khoo | 37200 |
| 116 | Baida | 34800 |
| 117 | Tobias | 33600 |
| 118 | Himuro | 31200 |
| 119 | Colmenares | 30000 |

6 rows selected.

3. Create a procedure, `ADD_EMPLOYEE`, to insert a new employee into the `EMPLOYEES` table. The procedure should call a `VALID_DEPTID` function to check whether the department ID specified for the new employee exists in the `DEPARTMENTS` table.
   a. Create a function `VALID_DEPTID` to validate a specified department ID and return a `BOOLEAN` value of `TRUE` if the department exists.
   b. Create the `ADD_EMPLOYEE` procedure to add an employee to the `EMPLOYEES` table. The row should be added to the `EMPLOYEES` table if the `VALID_DEPTID` function returns `TRUE`; otherwise, alert the user with an appropriate message. Provide the following parameters (with defaults specified in parentheses): `first_name`, `last_name`, `email`, `job` (`SA_REP`), `mgr` (145), `sal` (1000), `comm` (0), and `deptid` (30). Use the `EMPLOYEES_SEQ` sequence to set the `employee_id` column, and set `hire_date` to `TRUNC(SYSDATE)`.
   c. Call `ADD_EMPLOYEE` for the name `Jane Harris` in department 15, leaving other parameters with their default values. What is the result?
   d. Add another employee named `Joe Harris` in department 80, leaving remaining parameters with their default values. What is the result?

**Oracle Database 10*g*: Develop PL/SQL Program Units   2-19**

# Creating Packages

**3**

# Objectives

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

- **Describe packages and list their components**
- **Create a package to group together related variables, cursors, constants, exceptions, procedures, and functions**
- **Designate a package construct as either public or private**
- **Invoke a package construct**
- **Describe the use of a bodiless package**

ORACLE

**Lesson Aim**

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

# PL/SQL Packages: Overview

**PL/SQL packages:**

- **Group logically related components:**
  - **PL/SQL types**
  - **Variables, data structures, and exceptions**
  - **Subprograms: Procedures and functions**
- **Consist of two parts:**
  - **A specification**
  - **A body**
- **Enable the Oracle server to read multiple objects into memory at once**
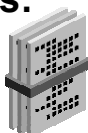
ORACLE

### PL/SQL Packages: Overview

PL/SQL packages enable you to bundle related PL/SQL types, variables, data structures, exceptions, and subprograms into one container. For example, a Human Resources package can contain hiring and firing procedures, commission and bonus functions, and tax exemption variables.

A package usually consists of two parts stored separately in the database:
- A specification
- A body (optional)

The package itself cannot be called, parameterized, or nested. After writing and compiling, the contents can be shared with many applications.

When a PL/SQL-packaged construct is referenced for the first time, the whole package is loaded into memory. Subsequent access to constructs in the same package do not require disk input/output (I/O).

# Components of a PL/SQL Package
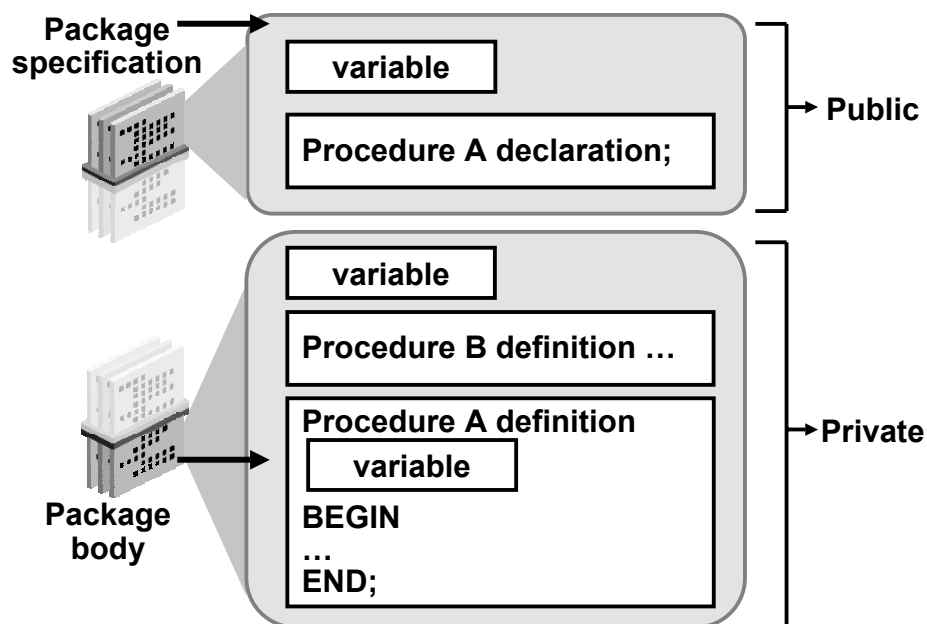
**Components of a PL/SQL Package**

You create a package in two parts:

- The **package specification** is the interface to your applications. It declares the public types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGMAs, which are directives to the compiler.
- The **package body** defines its own subprograms and must fully implement subprograms declared in the specification part. The package body may also define PL/SQL constructs, such as types, variables, constants, exceptions, and cursors.

**Public components** are declared in the package specification. The specification defines a public application programming interface (API) for users of package features and functionality—that is, public components can be referenced from any Oracle server environment that is external to the package.

**Private components** are placed in the package body and can be referenced only by other constructs within the same package body. Private components can reference the package public components.

**Note:** If a package specification does not contain subprogram declarations, then there is no requirement for a package body.

**Oracle Database 10g: Develop PL/SQL Program Units 3-4**
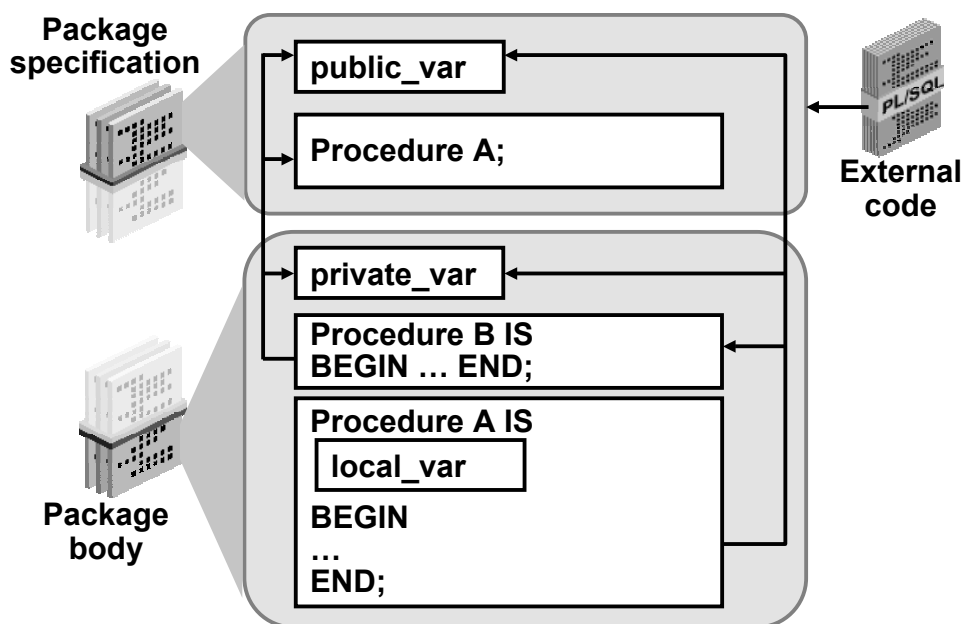
# Visibility of Package Components

## 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. The visibility of components depends on whether they are locally or globally declared.

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

- Variables defined in a subprogram can be referenced within that subprogram, and are not visible to external components—for example, `local_var` can be used in procedure `A`.
- Private package variables, which 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. For example, `private_var` can be used by procedures `A` and `B` within the package body, but not outside the package.

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

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

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

**Oracle Database 10*g*: Develop PL/SQL Program Units 3-5**
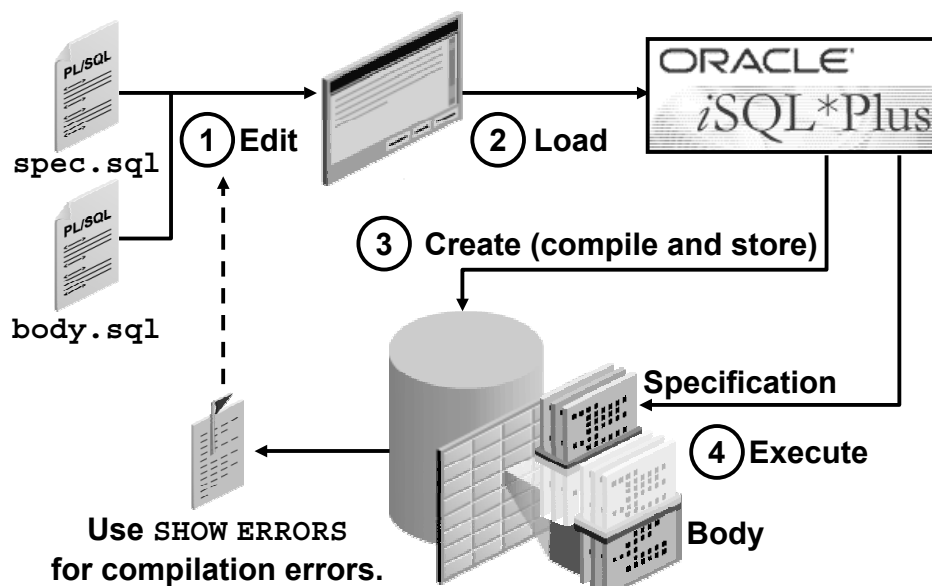
# Developing PL/SQL Packages

## Developing PL/SQL Packages

To develop a package, perform the following steps:
1. Edit the text for the specification by using the CREATE PACKAGE statement within a SQL script file. Edit the text for the body (only if required; see the guidelines below) by using the CREATE PACKAGE BODY statement within a SQL script file.
2. Load the script files into a tool such as *i*SQL*Plus.
3. Execute the script files to create (that is, to compile and store) the package and package body in the database.
4. Execute any public construct within the package specification from an Oracle server environment.

**Guidelines for Developing Packages**
- Consider saving the text for a package specification and a package body in two different script files to facilitate easier modifications to the package or its body.
- A package specification can exist without a package body—that is, when the package specification does not declare subprograms, a body is not required. However, a package body cannot exist without a package specification.

**Note:** The Oracle server stores the specification and body of a package separately. This enables you to change the implementation of a program construct in the package body without invalidating other schema objects that call or reference the program construct.

**Oracle Database 10*g*: Develop PL/SQL Program Units   3-6**

# Creating the Package Specification

**Syntax:**

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

- **The OR REPLACE option drops and re-creates the package specification.**
- **Variables declared in the package specification are initialized to NULL by default.**
- **All the constructs declared in a package specification are visible to users who are granted privileges on the package.**

ORACLE

### Creating the Package Specification

To create packages, you declare all public constructs within the package specification.
- Specify the OR REPLACE option, if overwriting an existing package specification.
- Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.

The following are definitions of items in the package syntax:
- **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.
- **subprogram specification** specifies the public procedure or function declarations.

**Note:** 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 of a procedure or function that is declared in a package specification is done in the package body.

# Example of Package Specification:
## `comm_pkg`

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

- `STD_COMM` **is a global variable initialized to** `0.10`.
- `RESET_COMM` **is a public procedure used to reset the standard commission based on some business rules. It is implemented in the package body.**

**Example of Package Specification: `comm_pkg`**

The example in the slide creates a package called `comm_pkg` used to manage business processing rules for commission calculations.

The `std_comm` public (global) variable is declared to hold a maximum allowable percentage commission for the user session, and it is initialized to `0.10` (that is, 10%).

The `reset_comm` public procedure is declared to accept a new commission percentage that updates the standard commission percentage if the commission validation rules are accepted. The validation rules for resetting the commission are not made public and do not appear in the package specification. The validation rules are managed by using a private function in the package body.

# Creating the Package Body

**Syntax:**

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

- **The `OR REPLACE` option drops and re-creates the package body.**
- **Identifiers defined in the package body are private and not visible outside the package body.**
- **All private constructs must be declared before they are referenced.**
- **Public constructs are visible to the package body.**

ORACLE

## Creating the Package Body

Create a package body to define and implement all public subprograms and supporting private constructs. When creating a package body, perform the following steps:
- 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. It is common to see all private variables and subprograms defined first and the public subprograms defined last in the package body.
- Complete the implementation for all procedures or functions declared in the package specification within the package body.

The following are definitions of items in the package body syntax:
- **`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.
- **`private type and variable declarations`** declares private variables, constants, cursors, exceptions, user-defined types, and subtypes.
- **`subprogram specification`** specifies the full implementation of any private and/or public procedures or functions.
- **`[BEGIN initialization statements]`** is an optional block of initialization code that executes when the package is first referenced.

**Oracle Database 10g: Develop PL/SQL Program Units 3-9**

# Example of Package Body: `comm_pkg`

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS
  FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS
    max_comm employees.commission_pct%type;
  BEGIN
    SELECT MAX(commission_pct) INTO max_comm
    FROM   employees;
    RETURN (comm BETWEEN 0.0 AND max_comm);
  END validate;
  PROCEDURE reset_comm (new_comm NUMBER) IS BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm; -- reset public var
    ELSE  RAISE_APPLICATION_ERROR(
            -20210, 'Bad Commission');
    END IF;
  END reset_comm;
END comm_pkg;
```

### Example of Package Body: `comm_pkg`

The slide shows the complete package body for `comm_pkg`, with a private function called `validate` to check for a valid commission. The validation requires that the commission be positive and lesser than the highest commission among existing employees. The `reset_comm` procedure invokes the private validation function before changing the standard commission in `std_comm`. In the example, note the following:

- The `std_comm` variable referenced in the `reset_comm` procedure is a public variable. Variables declared in the package specification, such as `std_comm`, can be directly referenced without qualification.
- The `reset_comm` procedure implements the public definition in the specification.
- In the `comm_pkg` body, the `validate` function is private and is directly referenced from the `reset_comm` procedure without qualification.

**Note:** The `validate` function appears before the `reset_comm` procedure because the `reset_comm` procedure references the `validate` function. It is possible to create forward declarations for subprograms in the package body if their order of appearance needs to be changed. If a package specification declares only types, constants, variables, and exceptions without any subprogram specifications, then the package body is unnecessary. However, the body can be used to initialize items declared in the package specification.

**Oracle Database 10*g*: Develop PL/SQL Program Units   3-10**

# Invoking Package Subprograms

- **Invoke a function within the same package:**

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...
  PROCEDURE reset_comm(new_comm NUMBER) IS
  BEGIN
    IF validate(new_comm) THEN
      std_comm := new_comm;
    ELSE ...
    END IF;
  END reset_comm;
END comm_pkg;
```

- **Invoke a package procedure from *i*SQL\*Plus:**

```
EXECUTE comm_pkg.reset_comm(0.15)
```

- **Invoke a package procedure in a different schema:**

```
EXECUTE scott.comm_pkg.reset_comm(0.15)
```

### Invoking Package Subprograms

After the package is stored in the database, you can invoke public or private subprograms within the same package, or public subprograms if external to the package. Fully qualify the subprogram with its package name when invoked externally from the package. Use the `package_name.subprogram` syntax.

Fully qualifying a subprogram when invoked within the same package is optional.

**Example 1:** Invokes the `validate` function from the `reset_comm` procedure within the same package. The package name prefix is not required; it is optional.

**Example 2:** Calls the `reset_comm` procedure from *i*SQL\*Plus (an environment external to the package) to reset the prevailing commission to 0.15 for the user session.

**Example 3:** Calls the `reset_comm` procedure that is owned in a schema user called `SCOTT`. Using *i*SQL\*Plus, the qualified package procedure is prefixed with the schema name. This can be simplified by using a synonym that references the `schema.package_name`.

Assume that a database link named `NY` has been created for a remote database in which the `reset_comm` package procedure is created. To invoke the remote procedure, use:

```
        EXECUTE comm_pkg.reset_comm@NY(0.15)
```

# Creating and Using Bodiless Packages

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_2_kilo    CONSTANT   NUMBER   :=   1.6093;
  kilo_2_mile    CONSTANT   NUMBER   :=   0.6214;
  yard_2_meter   CONSTANT   NUMBER   :=   0.9144;
  meter_2_yard   CONSTANT   NUMBER   :=   1.0936;
END global_consts;
```

```
BEGIN  DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
       20 * global_consts.mile_2_kilo || ' km');
END;
```

```
CREATE FUNCTION mtr2yrd(m NUMBER) RETURN NUMBER IS
BEGIN
  RETURN (m * global_consts.meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

## Creating and Using Bodiless Packages

The variables and constants declared within stand-alone subprograms exist only for the duration that the subprogram executes. To provide data that exists for the duration of the user session, create a package specification containing public (global) variables and constant declarations. In this case, create a package specification without a package body, known as a *bodiless package*. As discussed earlier in this lesson, if a specification declares only types, constants, variables, and exceptions, then the package body is unnecessary.

**Examples**

The first code box in the slide creates a bodiless package specification with several constants to be used for conversion rates. A package body is not required to support this package specification.

The second code box references the mile_2_kilo constant in the global_consts package by prefixing the package name to the identifier of the constant.

The third example creates a stand-alone function mtr2yrd to convert meters to yards, and uses the constant conversion rate meter_2_yard declared in the global_consts package. The function is invoked in a DBMS_OUTPUT.PUT_LINE parameter.

**Rule to be followed:** When referencing a variable, cursor, constant, or exception from outside the package, you must qualify it with the name of the package.

**Oracle Database 10*g*: Develop PL/SQL Program Units 3-12**

# Removing Packages

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

```
DROP PACKAGE package_name;
```

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

```
DROP PACKAGE BODY package_name;
```

**Removing Packages**

When a package is no longer required, you can use a SQL statement in *i*SQL*Plus to remove it. A package has two parts; therefore, you can remove the whole package, or you can remove only the package body and retain the package specification.

# 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 = 'COMM_PKG' AND type = 'PACKAGE';
```

- **To view the package body, use:**

```
SELECT text
FROM   user_source
WHERE  name = 'COMM_PKG' AND type = 'PACKAGE BODY';
```

ORACLE

### Viewing Packages in the Data Dictionary

The source code for PL/SQL packages is also stored in the data dictionary tables such as stand-alone procedures and functions. The source code is viewable in the data dictionary when you execute a SELECT statement on the USER_SOURCE and ALL_SOURCE tables.

When querying the package, use a condition in which the TYPE column is:
- Equal to 'PACKAGE' to display the source code for the package specification
- Equal to 'PACKAGE BODY' to display the source code for the package body

**Note:** The values of the NAME and TYPE columns must be uppercase.

## Guidelines for Writing Packages

- **Construct packages for general use.**
- **Define the package specification before the body.**
- **The package specification should contain only those constructs that you want to be public.**
- **Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.**
- **Changes to the package specification require recompilation of each referencing subprogram.**
- **The package specification should contain as few constructs as possible.**

ORACLE

### Guidelines for Writing Packages

Keep your packages as general as possible, so that they can be reused in future applications. Also, avoid writing packages that duplicate features provided by the Oracle server.

Package specifications reflect the design of your application, so define them before defining the package bodies. The package specification should contain only those constructs that must be visible to the users of the package. Thus, other developers cannot misuse the package by basing code on irrelevant details.

Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions. For example, declare a variable called NUMBER_EMPLOYED as a private variable if each call to a procedure that uses the variable needs to be maintained. When declared as a global variable in the package specification, the value of that global variable is initialized in a session the first time a construct from the package is invoked.

Changes to the package body do not require recompilation of dependent constructs, whereas changes to the package specification require recompilation of every stored subprogram that references the package. To reduce the need for recompiling when code is changed, place as few constructs as possible in a package specification.

# 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.**

ORACLE

## Advantages of Using Packages

Packages provide an alternative to creating procedures and functions as stand-alone schema objects, and they offer several benefits.

**Modularity and ease of maintenance:** You encapsulate logically related programming structures in a named module. Each package is easy to understand, and the interface between packages is simple, clear, and well defined.

**Easier application design:** All you need initially is the interface information in the package specification. You can code and compile a specification without its body. Then stored subprograms that reference the package can compile as well. You need not define the package body fully until you are ready to complete the application.

**Hiding information:** You decide which constructs are public (visible and accessible) and which are private (hidden and inaccessible). Declarations in the package specification are visible and accessible to applications. The package body hides the definition of the private constructs, so that only the package is affected (not your application or any calling programs) if the definition changes. This enables you to change the implementation without having to recompile the calling programs. Also, by hiding implementation details from users, you protect the integrity of the package.

# 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 of the same name**

**Advantages of Using Packages (continued)**

**Added functionality:** Packaged public variables and cursors persist for the duration of a session. Thus, they can be shared by all subprograms that execute in the environment. They also enable you to maintain data across transactions without having to store it in the database. Private constructs also persist for the duration of the session but can be accessed only within the package.

**Better performance:** When you call a packaged subprogram the first time, the entire package is loaded into memory. Later calls to related subprograms in the package therefore require no further disk I/O. Packaged subprograms also stop cascading dependencies and thus avoid unnecessary compilation.

**Overloading:** With packages, you can overload procedures and functions, which means you can create multiple subprograms with the same name in the same package, each taking parameters of different number or data type.

**Note:** Dependencies are covered in detail in the lesson titled "Managing Dependencies."

# Summary

**In this lesson, you should have learned how to:**

- **Improve code organization, management, security, and performance by using packages**
- **Create and remove package specifications and bodies**
- **Group related procedures and functions together in a package**
- **Encapsulate the code in a package body**
- **Define and use components in bodiless packages**
- **Change a package body without affecting a package specification**

ORACLE

## Summary

You group related procedures and function together in a package. Packages improve organization, management, security, and performance.

A package consists of a package specification and a package body. You can change a package body without affecting its package specification.

Packages enable you to hide source code from users. When you invoke a package for the first time, the entire package is loaded into memory. This reduces the disk access for subsequent calls.

# Summary

| Command | Task |
|---|---|
| CREATE [OR REPLACE] PACKAGE | **Create (or modify) an existing package specification.** |
| CREATE [OR REPLACE] PACKAGE BODY | **Create (or modify) an existing package body.** |
| DROP PACKAGE | **Remove both the package specification and package body.** |
| DROP PACKAGE BODY | **Remove only the package body.** |

**Summary (continued)**

You can create, delete, and modify packages. You can remove both package specification and body by using the DROP PACKAGE command. You can drop the package body without affecting its specification.

# Practice 3: Overview

**This practice covers the following topics:**
- **Creating packages**
- **Invoking package program units**

**Practice 3: Overview**

In this practice, you create package specifications and package bodies. You invoke the constructs in the packages by using sample data.

**Note:** If you are using SQL Developer, your compile time errors are displayed in the Message Log. If you are using SQL*Plus or *i*SQL*Plus to create your stored code, use the SHOW ERRORS to view compile errors.

**Practice 3**

1. Create a package specification and body called `JOB_PKG`, containing a copy of your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures as well as your `GET_JOB` function.
   **Tip:** Consider saving the package specification and body in two separate files (for example, `p3q1_s.sql` and `p3q1_b.sql` for the package specification and body, respectively). Include a `SHOW ERRORS` after the `CREATE PACKAGE` statement in each file. Alternatively, place all code in one file.
   **Note:** Use the code in your previously saved script files when creating the package.
   a. Create the package specification including the procedures and function headings as public constructs.
      **Note:** Consider whether you still need the stand-alone procedures and functions you just packaged.
   b. Create the package body with the implementations for each of the subprograms.
   c. Invoke your `ADD_JOB` package procedure by passing the values `IT_SYSAN` and `SYSTEMS ANALYST` as parameters.
   d. Query the `JOBS` table to see the result.

| JOB_ID | JOB_TITLE | MIN_SALARY | MAX_SALARY |
|--------|-----------|------------|------------|
| IT_SYSAN | Systems Analyst | | |

2. Create and invoke a package that contains private and public constructs.
   a. Create a package specification and package body called `EMP_PKG` that contains your `ADD_EMPLOYEE` and `GET_EMPLOYEE` procedures as public constructs, and include your `VALID_DEPTID` function as a private construct.
   b. Invoke the `EMP_PKG.ADD_EMPLOYEE` procedure, using department ID `15` for employee `Jane Harris` with the e-mail ID `JAHARRIS`. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.
   c. Invoke the `ADD_EMPLOYEE` package procedure by using department ID `80` for employee `David Smith` with the e-mail ID `DASMITH`.

# Using More Package Concepts

## 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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-3**

## 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;
/
```

**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);
```

**Oracle Database 10*g*: Develop PL/SQL Program Units 4-5**

# 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;
/
```

**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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-8**

# Using Forward Declarations

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

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
PROCEDURE calc_rating (...);-- forward declaration

  -- Subprograms defined in alphabetical order

  PROCEDURE award_bonus(...) IS
  BEGIN
    calc_rating (...);          -- reference resolved!
    . . .
  END;

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

## 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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-11**

# 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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-12**

# 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

| | | State for: -Scott- | | -Jones- | |
|---|---|---|---|---|---|
| Time | Events | STD | MAX | STD | MAX |
| 9:00 | Scott> EXECUTE<br>   comm_pkg.reset_comm(0.25) | 0.10<br>0.25 | 0.4 | - | 0.4 |
| 9:30 | Jones> INSERT<br>   INTO employees(<br>   last_name,commission_pct)<br>   VALUES('Madonna', 0.8); | 0.25 | 0.4 | | 0.8 |
| 9:35 | Jones> EXECUTE<br>   comm_pkg.reset_comm (0.5) | 0.25 | 0.4 | 0.1<br>0.5 | 0.8 |
| 10:00 | Scott> EXECUTE<br>   comm_pkg.reset_comm(0.6)<br>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 |

**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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-14**

# 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;
```

## 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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-16**

# 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;
/
```

### 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
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-17**

# 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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-18**

# 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 ...

/
```

**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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-21**

# 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**

## 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.

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-24**

**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?

**Oracle Database 10*g*: Develop PL/SQL Program Units   4-25**

# Using Oracle-Supplied Packages
# in Application Development

Oracle University and En-Sof Informatica E Treinamento Ltda use only

# Objectives

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

- **Describe how the `DBMS_OUTPUT` package works**
- **Use `UTL_FILE` to direct output to operating system files**
- **Use the `HTP` package to generate a simple Web page**
- **Describe the main features of `UTL_MAIL`**
- **Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution**

ORACLE

**Lesson Aim**

In this lesson, you learn how to use some of the Oracle-supplied packages and their capabilities. This lesson focuses on the packages that generate text-based and Web-based output, e-mail processing, and the provided scheduling capabilities.

# Using Oracle-Supplied Packages

**The Oracle-supplied packages:**

- **Are provided with the Oracle server**
- **Extend the functionality of the database**
- **Enable access to certain SQL features that are normally restricted for PL/SQL**

**For example, the** `DBMS_OUTPUT` **package was originally designed to debug PL/SQL programs.**

ORACLE

## Using Oracle-Supplied Packages

Packages are provided with the Oracle server to allow either of the following:

- PL/SQL access to certain SQL features
- The extension of the functionality of the database

You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

Most of the standard packages are created by running `catproc.sql`. The `DBMS_OUTPUT` package is the one that you will be most familiar with during this course. You should know about this package if you attended the course *Oracle Database 10g: PL/SQL Fundamentals*.

# List of Some Oracle-Supplied Packages

**Here is an abbreviated list of some Oracle-supplied packages:**

- `DBMS_ALERT`
- `DBMS_LOCK`
- `DBMS_SESSION`
- `DBMS_OUTPUT`
- `HTP`
- `UTL_FILE`
- `UTL_MAIL`
- `DBMS_SCHEDULER`

ORACLE

### List of Some Oracle-Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. It would be impossible to cover the exhaustive set of packages and their functionality in this course. For more information, refer to the *PL/SQL Packages and Types Reference 10g* (previously known as the *PL/SQL Supplied Packages Reference)*. This lesson covers the last five packages in this list.

The following is a brief description about all the listed packages:
- `DBMS_ALERT` supports asynchronous notification of database events. Messages or alerts are sent on a `COMMIT` command.
- `DBMS_LOCK` is used to request, convert, and release locks through Oracle Lock Management services.
- `DBMS_SESSION` enables programmatic use of the `ALTER SESSION` SQL statement and other session-level commands.
- `DBMS_OUTPUT` provides debugging and buffering of text data.
- `HTP` package 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 and automated execution of PL/SQL blocks, stored procedures, and external procedures and executables.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-4**

# How the `DBMS_OUTPUT` Package Works

**The `DBMS_OUTPUT` package enables you to send messages from stored subprograms and triggers.**

- **`PUT` and `PUT_LINE` place text in the buffer.**
- **`GET_LINE` and `GET_LINES` read the buffer.**
- **Messages are not sent until the sender completes.**
- **Use `SET SERVEROUTPUT ON` to display messages in *i*SQL\*Plus.**
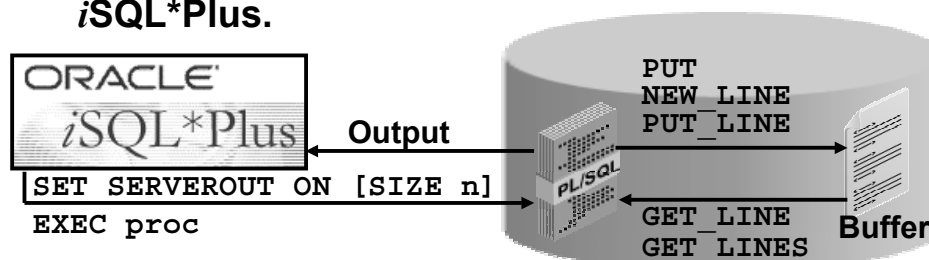
```
ORACLE
iSQL*Plus          Output

SET SERVEROUT ON [SIZE n]
EXEC proc
```

```
PUT
NEW_LINE
PUT_LINE

GET_LINE
GET_LINES     Buffer
```

PL/SQL

### How the `DBMS_OUTPUT` Package Works

The `DBMS_OUTPUT` package sends textual messages from any PL/SQL block into a buffer in the database. Procedures provided by the package include the following:

- `PUT` appends text from the procedure to the current line of the line output buffer.
- `NEW_LINE` places an end-of-line marker in the output buffer.
- `PUT_LINE` combines the action of `PUT` and `NEW_LINE` ( to trim leading spaces).
- `GET_LINE` retrieves the current line from the buffer into a procedure variable.
- `GET_LINES` retrieves an array of lines into a procedure-array variable.
- `ENABLE`/`DISABLE` enables and disables calls to `DBMS_OUTPUT` procedures.

The buffer size can be set by using:

- The `SIZE n` option appended to the `SET SERVEROUTPUT ON` command, where n is between 2,000 (the default) and 1,000,000 (1 million characters)
- An integer parameter between 2,000 and 1,000,000 in the `ENABLE` procedure

**Practical Uses**

- You can output results to the window for debugging purposes.
- You can trace code execution path for a function or procedure.
- You can send messages between subprograms and triggers.

**Note:** There is no mechanism to flush output during the execution of a procedure.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-5**

# Interacting with Operating System Files

**The `UTL_FILE` package extends PL/SQL programs to read and write operating system text files. `UTL_FILE`:**

- **Provides a restricted version of operating system stream file I/O for text files**
- **Can access files in operating system directories defined by a `CREATE DIRECTORY` statement. You can also use the `utl_file_dir` database parameter.**
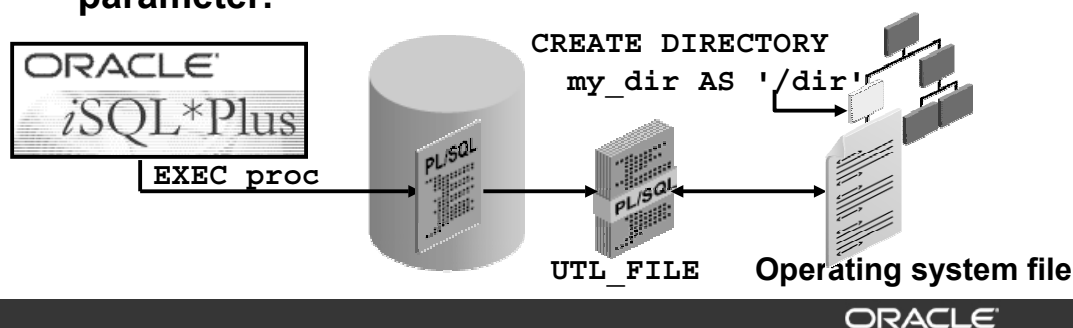
### Interacting with Operating System Files

The Oracle-supplied `UTL_FILE` package is used to access text files in the operating system of the database server. The database provides read and write access to specific operating system directories by using:

- A `CREATE DIRECTORY` statement that associates an alias with an operating system directory. The database directory alias can be granted the `READ` and `WRITE` privileges to control the type of access to files in the operating system. For example:
    ```
    CREATE DIRECTORY my_dir AS '/temp/my_files';
    GRANT READ, WRITE ON my_dir TO public.
    ```
- The paths specified in the `utl_file_dir` database initialization parameter

The preferred approach is to use the directory alias created by the `CREATE DIRECTORY` statement, which does not require the database to be restarted. The operating system directories specified by using either of these techniques should be accessible to and on the same machine as the database server processes. The path (directory) names may be case sensitive for some operating systems.

**Note:** The `DBMS_LOB` package can be used to read binary files on the operating system. `DBMS_LOB` is covered in the lesson titled "Manipulating Large Objects."

# File Processing Using the UTL_FILE Package

- **Reading a file:**

```
f:=FOPEN(dir,file,'r')
```

**Open for reading** → **Get lines from the text file** → **More to read?** — Yes (loop back) / No → **Close the text file**

```
GET_LINE(f,buf,len)
```
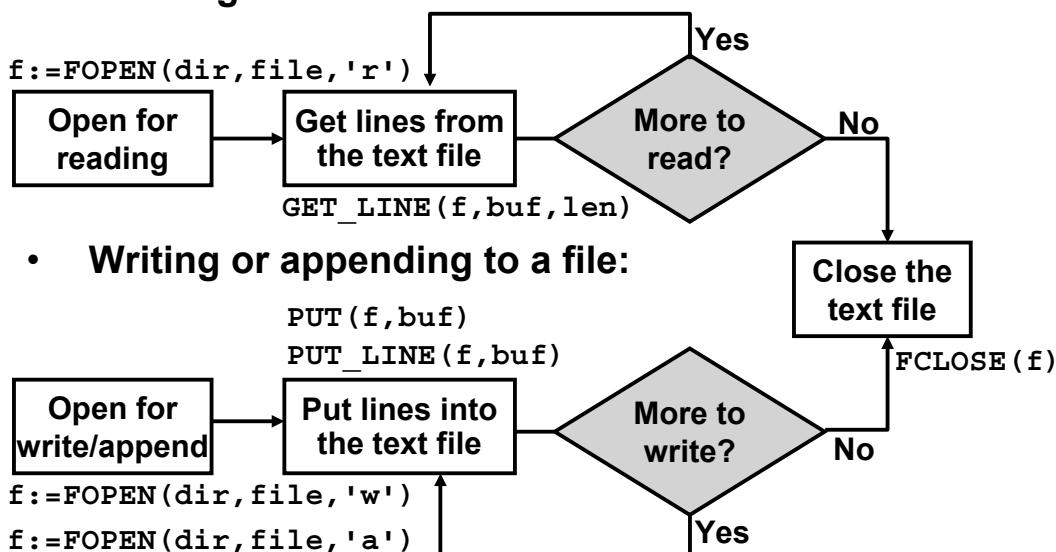
- **Writing or appending to a file:**
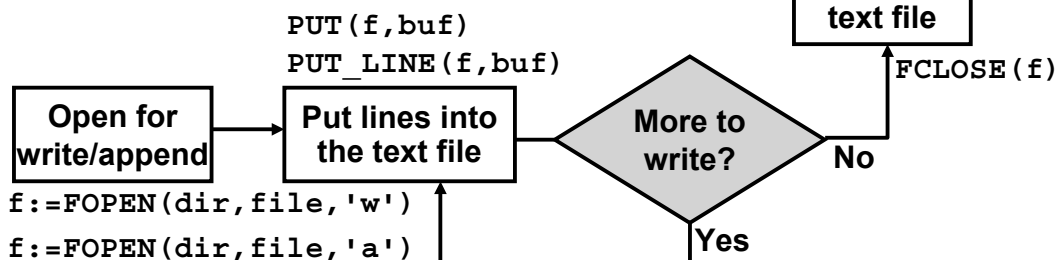
```
PUT(f,buf)
PUT_LINE(f,buf)
```

**Open for write/append** → **Put lines into the text file** → **More to write?** — Yes (loop back) / No → **Close the text file**

```
f:=FOPEN(dir,file,'w')
f:=FOPEN(dir,file,'a')
```

```
FCLOSE(f)
```

ORACLE

### File Processing Using the UTL_FILE Package

Using the procedures and functions in the UTL_FILE package, open files with the FOPEN function. You then either read from or write or append to the file until processing is done. After completing processing the file, close the file by using the FCLOSE procedure. The following are the subprograms:

- The FOPEN function opens a file in a specified directory for input/output (I/O) and returns a file handle used in subsequent I/O operations.
- The IS_OPEN function returns a Boolean value whenever a file handle refers to an open file. Use IS_OPEN to check whether the file is already open before opening the file.
- 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 unless you specify a larger size in the overloaded version of FOPEN.)
- The PUT and PUT_LINE procedures write text to the opened file.
- The PUTF procedure provides formatted output with two format specifiers: %s to substitute a value into the output string and \n for a new line character.
- The NEW_LINE procedure terminates a line in an output file.
- The FFLUSH procedure writes all data buffered in memory to a file.
- The FCLOSE procedure closes an opened file.
- The FCLOSE_ALL procedure closes all opened file handles for the session.

**Oracle Database 10g: Develop PL/SQL Program Units   5-7**

# Exceptions in the UTL_FILE Package

**You may have to handle one of these exceptions when using UTL_FILE subprograms:**

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

**Other exceptions not in the UTL_FILE package are:**

- NO_DATA_FOUND **and** VALUE_ERROR

ORACLE

### Exceptions in the UTL_FILE Package

The UTL_FILE package declares seven exceptions that indicate an error condition in the operating system file processing. The UTL_FILE exceptions are:

- INVALID_PATH if the file location or file name was invalid
- INVALID_MODE if the OPEN_MODE parameter in FOPEN was invalid
- INVALID_FILEHANDLE if the file handle was invalid
- INVALID_OPERATION if the file could not be opened or operated on as requested
- READ_ERROR if an operating system error occurred during the read operation
- WRITE_ERROR if an operating system error occurred during the write operation
- INTERNAL_ERROR if an unspecified error occurred in PL/SQL

**Note:** These exceptions must always be prefaced with the package name. UTL_FILE procedures can also raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR.

The NO_DATA_FOUND exception is raised when reading past the end of a file by using UTL_FILE.GET_LINE or UTL_FILE.GET_LINES.

# 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:**

```
CREATE PROCEDURE read_file(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_file;
```

## FOPEN and IS_OPEN Function Parameters

The parameters include the following:

- location parameter: Specifies the name of a directory alias defined by a CREATE DIRECTORY statement, or an operating system–specific path specified by using the utl_file_dir database parameter
- filename parameter: Specifies the name of the file, including the extension, without any path information
- open_mode string: Specifies how the file is to be opened. Values are:
    - 'r' for reading text (use GET_LINE)
    - 'w' for writing text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)
    - 'a' for appending text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

The return value from FOPEN is a file handle whose type is UTL_FILE.FILE_TYPE. The handle must be used on subsequent calls to routines that operate on the opened file.

The IS_OPEN function parameter is the file handle. The IS_OPEN function tests a file handle to see whether it identifies an opened file. It returns a Boolean value of TRUE if the file has been opened; otherwise it returns a value of FALSE indicating that the file has not been opened. The slide example shows how to combine the use of the two subprograms.

**Note:** For the full syntax, refer to the *PL/SQL Packages and Types Reference*.

**Oracle Database 10g: Develop PL/SQL Program Units 5-9**

# Using `UTL_FILE`: Example

```
CREATE OR REPLACE PROCEDURE sal_status(
  dir IN VARCHAR2, filename IN VARCHAR2) IS
  file UTL_FILE.FILE_TYPE;
CURSOR empc IS
  SELECT last_name, salary, department_id
  FROM employees ORDER BY department_id;
newdeptno employees.department_id%TYPE;
olddeptno employees.department_id%TYPE := 0;
BEGIN
  file:= UTL_FILE.FOPEN (dir, filename, 'w');
  UTL_FILE.PUT_LINE(file,
    'REPORT: GENERATED ON ' || SYSDATE);
  UTL_FILE.NEW_LINE (file); ...
```

## Using `UTL_FILE`: Example

In the example, the `sal_status` procedure creates a report of employees for each department, along with their salaries. The data is written to a text file by using the `UTL_FILE` package. In the code example, the variable `file` is declared as `UTL_FILE.FILE_TYPE`, a package type that is a record with a field called `ID` of the `BINARY_INTEGER` data type. For example:

```
TYPE file_type IS RECORD (id BINARY_INTEGER);
```

The field of `FILE_TYPE` record is private to the `UTL_FILE` package and should never be referenced or changed. The `sal_status` procedure accepts two parameters:

- The `dir` parameter for the name of the directory in which to write the text file
- The `filename` parameter to specify the name of the file

For example, to call the procedure, use the following:

```
EXECUTE sal_status('MY_DIR', 'salreport.txt')
```

**Note:** The directory location used (`MY_DIR`) must be in uppercase characters if it is a directory alias created by a `CREATE DIRECTORY` statement. When reading a file in a loop, the loop should exit when it detects the `NO_DATA_FOUND` exception. The `UTL_FILE` output is sent synchronously. `DBMS_OUTPUT` procedures do not produce output until the procedure is completed.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-10**

# Using `UTL_FILE`: Example

```
  FOR emp_rec IN empc LOOP
    IF emp_rec.department_id <> olddeptno THEN
      UTL_FILE.PUT_LINE (file,
        'DEPARTMENT: ' || emp_rec.department_id);
      UTL_FILE.NEW_LINE (file);
    END IF;
    UTL_FILE.PUT_LINE (file,
      '   EMPLOYEE: ' || emp_rec.last_name ||
      ' earns: ' || emp_rec.salary);
    olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (file);
  END LOOP;
  UTL_FILE.PUT_LINE(file,'*** END OF REPORT ***');
  UTL_FILE.FCLOSE (file);
EXCEPTION
 WHEN UTL_FILE.INVALID_FILEHANDLE THEN
  RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
 WHEN UTL_FILE.WRITE_ERROR THEN
  RAISE_APPLICATION_ERROR (-20002, 'Unable to
write to file');
END sal_status;
/
```

## Using `UTL_FILE`: Example (continued)

The output for this report in the `salreport.txt` file is as follows:

```
SALARY REPORT: GENERATED ON 08-MAR-01
     DEPARTMENT: 10
           EMPLOYEE: Whalen earns: 4400
     DEPARTMENT: 20
           EMPLOYEE: Hartstein earns: 13000
           EMPLOYEE: Fay earns: 6000
     DEPARTMENT: 30
           EMPLOYEE: Raphaely earns: 11000
           EMPLOYEE: Khoo earns: 3100
            ...
     DEPARTMENT: 100
           EMPLOYEE: Greenberg earns: 12000
          ...
     DEPARTMENT: 110
           EMPLOYEE: Higgins earns: 12000
           EMPLOYEE: Gietz earns: 8300
           EMPLOYEE: Grant earns: 7000
     *** END OF REPORT ***
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-11**

**Generating Web Pages with the HTP Package**

# Generating Web Pages with the HTP Package

The HTP package contains procedures that are used to generate HTML tags. The HTML tags that are generated typically enclose the data provided as parameters to the various procedures. The slide illustrates two ways in which the HTP package can be used:

- Most likely your procedures are invoked by the PL/SQL Gateway services, via the mod_plsql component supplied with Oracle HTTP Server, which is part of the Oracle Application Server product (represented by solid lines in the graphic).
- Alternatively (as represented by dotted lines in the graphic), your procedure can be called from *i*SQL*Plus that can display the generated HTML output, which can be copied and pasted to a file. This technique is used in this course because Oracle Application Server software is not installed as a part of the course environment.

**Note:** The HTP procedures output information to a session buffer held in the database server. In the Oracle HTTP Server context, when the procedure completes, the mod_plsql component automatically receives the buffer contents, which are then returned to the browser as the HTTP response. In SQL*Plus, you must manually execute:

- A SET SERVEROUTPUT ON command
- The procedure to generate the HTML into the buffer
- The OWA_UTIL.SHOWPAGE procedure to display the buffer contents

# Using the `HTP` Package Procedures

- **Generate one or more HTML tags. For example:**

```
htp.bold('Hello');                 -- <B>Hello</B>
htp.print('Hi <B>World</B>');  -- Hi <B>World</B>
```

- **Are used to create a well-formed HTML document:**

```
BEGIN                              -- Generates:
 htp.htmlOpen;      --------->  <HTML>
 htp.headOpen;      --------->  <HEAD>
 htp.title('Welcome');  -->     <TITLE>Welcome</TITLE>
 htp.headClose;     --------->  </HEAD>
 htp.bodyOpen;      --------->  <BODY>
 htp.print('My home page');     My home page
 htp.bodyClose;     --------->  </BODY>
 htp.htmlClose;     --------->  </HTML>
END;
```

ORACLE

### Using the `HTP` Package Procedures

The `HTP` package is structured to provide a one-to-one mapping of a procedure to standard HTML tags. For example, to display bold text on a Web page, the text must be enclosed in the HTML tag pair `<B>` and `</B>`. The first code box in the slide shows how to generate the word `Hello` in HTML bold text by using the equivalent `HTP` package procedure— that is, `HTP.BOLD`. The `HTP.BOLD` procedure accepts a text parameter and ensures that it is enclosed in the appropriate HTML tags in the HTML output that is generated.

The `HTP.PRINT` procedure copies its text parameter to the buffer. The example in the slide shows how the parameter supplied to the `HTP.PRINT` procedure can contain HTML tags. This technique is recommended only if you need to use HTML tags that cannot be generated by using the set of procedures provided in the `HTP` package.

The second example in the slide provides a PL/SQL block that generates the basic form of an HTML document. The example serves to illustrate how each of the procedures generates the corresponding HTML line in the enclosed text box on the right.

The benefit of using the `HTP` package is that you create well-formed HTML documents, eliminating the need to manually type the HTML tags around each piece of data.

**Note:** For information about all the `HTP` package procedures, refer to the *PL/SQL Packages and Types Reference*.

# Creating an HTML File with *i*SQL*Plus

**To create an HTML file with *i*SQL*Plus, perform the following steps:**

**1. Create a SQL script with the following commands:**

```
SET SERVEROUTPUT ON
ACCEPT procname PROMPT "Procedure: "
EXECUTE &procname
EXECUTE owa_util.showpage
UNDEFINE proc
```

**2. Load and execute the script in *i*SQL*Plus, supplying values for substitution variables.**

**3. Select, copy, and paste the HTML text that is generated in the browser to an HTML file.**

**4. Open the HTML file in a browser.**

ORACLE

### Creating an HTML File with *i*SQL*Plus

The slide example shows the steps for generating HTML by using any procedure and saving the output into an HTML file. You should perform the following steps:

1. Turn on server output with the `SET SERVEROUTPUT ON` command. Without this, you receive exception messages when running procedures that have calls to the `HTP` package.
2. Execute the procedure that contains calls to the `HTP` package.
   **Note:** This does *not* produce output, unless the procedure has calls to the `DBMS_OUTPUT` package.
3. Execute the `OWA_UTIL.SHOWPAGE` procedure to display the text. This call actually displays the HTML content that is generated from the buffer.

The `ACCEPT` command prompts for the name of the procedure to execute. The call to `OWA_UTIL.SHOWPAGE` displays the HTML tags in the browser window. You can then copy and paste the generated HTML tags from the browser window into an HTML file, typically with a `.htm` or `.html` extension.

**Note:** If you are using SQL*Plus, then you can use the `SPOOL` command to direct the HTML output directly to an HTML file. The `SPOOL` command is not supported in *i*SQL*Plus; therefore, the copy-and-paste technique is the only option.

# Using `UTL_MAIL`

**The `UTL_MAIL` package:**

• **Is a utility for managing e-mail that includes such commonly used e-mail features as attachments, CC, BCC, and return receipt**

• **Requires the `SMTP_OUT_SERVER` database initialization parameter to be set**

• **Provides the following procedures:**
  – **`SEND` for messages without attachments**
  – **`SEND_ATTACH_RAW` for messages with binary attachments**
  – **`SEND_ATTACH_VARCHAR2` for messages with text attachments**

ORACLE

## Using `UTL_MAIL`

The `UTL_MAIL` package is a utility for managing e-mail that includes commonly used e-mail features such as attachments, CC, BCC, and return receipt.

The `UTL_MAIL` package is not installed by default because of the `SMTP_OUT_SERVER` configuration requirement and the security exposure this involves. When installing `UTL_MAIL`, you should take steps to prevent the port defined by `SMTP_OUT_SERVER` being swamped by data transmissions. To install `UTL_MAIL`, log in as a DBA user in *i*SQL*Plus and execute the following scripts:

```
@$ORACLE_HOME/rdbms/admin/utlmail.sql
@$ORACLE_HOME/rdbms/admin/prvtmail.plb
```

You should define the `SMTP_OUT_SERVER` parameter in the `init.ora file` database initialization file:

```
SMTP_OUT_SERVER=mystmpserver.mydomain.com
```

The `SMTP_OUT_SERVER` parameter specifies the SMTP host and port to which `UTL_MAIL` delivers outbound e-mail. Multiple servers can be specified, separated by commas. If the first server in the list is unavailable, then `UTL_MAIL` tries the second server, and so on. If `SMTP_OUT_SERVER` is not defined, then this invokes a default setting derived from `DB_DOMAIN`, which is a database initialization parameter specifying the logical location of the database within the network structure. For example:

```
db_domain=mydomain.com
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-15**

## Installing and Using `UTL_MAIL`

- **As `SYSDBA`, using *i*SQL*Plus:**
  - **Set the `SMTP_OUT_SERVER` (requires DBMS restart).**

```
ALTER SYSTEM SET SMTP_OUT_SERVER='smtp.server.com'
SCOPE=SPFILE
```

  - **Install the `UTL_MAIL` package.**

```
@?/rdbms/admin/utlmail.sql
@?/rdbms/admin/prvtmail.plb
```

- **As a developer, invoke a `UTL_MAIL` procedure:**

```
BEGIN
 UTL_MAIL.SEND('otn@oracle.com','user@oracle.com',
   message => 'For latest downloads visit OTN',
   subject => 'OTN Newsletter');
END;
```

ORACLE

### Installing and Using `UTL_MAIL`

The slide shows how to configure the SMTP_OUT_SERVER parameter to the name of the SMTP host in your network, and how to install the UTL_MAIL package that is not installed by default. Changing the SMTP_OUT_SERVER parameter requires restarting the database instance. These tasks are performed by a user with SYSDBA capabilities.

The last example in the slide shows the simplest way to send a text message by using the UTL_MAIL.SEND procedure with at least a subject and a message. The first two required parameters are the following :
- The sender e-mail address (in this case, otn@oracle.com)
- The recipients e-mail address (for example, user@oracle.com). The value can be a comma-separated list of addresses.

The UTL_MAIL.SEND procedure provides several other parameters, such as cc, bcc, and priority with default values, if not specified. In the example, the message parameter specifies the text for the e-mail, and the subject parameter contains the text for the subject line. To send an HTML message with HTML tags, add the mime_type parameter (for example, mime_type=>'text/html').

**Note:** For details about all the UTL_MAIL procedure parameters, refer to the *PL/SQL Packages and Types Reference*.

# Sending E-Mail with a Binary Attachment

**Use the `UTL_MAIL.SEND_ATTACH_RAW` procedure:**

```
CREATE OR REPLACE PROCEDURE send_mail_logo IS
BEGIN
  UTL_MAIL.SEND_ATTACH_RAW(
    sender => 'me@oracle.com',
    recipients => 'you@somewhere.net',
    message =>
      '<HTML><BODY>See attachment</BODY></HTML>',
    subject => 'Oracle Logo',
    mime_type => 'text/html'
    attachment => get_image('oracle.gif'),
    att_inline => true,
    att_mime_type => 'image/gif',
    att_filename => 'oralogo.gif');
END;
/
```

### Sending E-Mail with a Binary Attachment

The slide shows a procedure calling the `UTL_MAIL.SEND_ATTACH_RAW` procedure to send a textual or an HTML message with a binary attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the e-mail message, the `SEND_ATTACH_RAW` procedure has the following highlighted parameters:

- The `attachment` parameter (required) accepts a `RAW` data type, with a maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is Boolean (default `TRUE`) to indicate that the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_image` function in the example uses a `BFILE` to read the image data. Using a `BFILE` requires creating a logical directory name in the database by using the `CREATE DIRECTORY` statement. The details about working with a `BFILE` are covered in the lesson titled "Manipulating Large Objects." The code for `get_image` is shown on the following page.

**Sending E-Mail with a Binary Attachment (continued)**

The `get_image` function uses the `DBMS_LOB` package to read a binary file from the operating system:

```
CREATE OR REPLACE FUNCTION get_image(
       filename VARCHAR2, dir VARCHAR2 := 'TEMP')
RETURN RAW IS
  image RAW(32767);
  file  BFILE := BFILENAME(dir, filename);
BEGIN
  DBMS_LOB.FILEOPEN(file, DBMS_LOB.FILE_READONLY);
  image := DBMS_LOB.SUBSTR(file);
  DBMS_LOB.CLOSE(file);
  RETURN image;
END;
/
```

To create the directory called `TEMP`, execute the following statement in *i*SQL*Plus:

```
CREATE DIRECTORY temp AS 'e:\temp';
```

**Note:** You need the `CREATE ANY DIRECTORY` system privilege to execute this statement.

# Sending E-Mail with a Text Attachment

**Use the `UTL_MAIL.SEND_ATTACH_VARCHAR2`
procedure:**

```
CREATE OR REPLACE PROCEDURE send_mail_file IS
BEGIN
  UTL_MAIL.SEND_ATTACH_VARCHAR2(
    sender => 'me@oracle.com',
    recipients => 'you@somewhere.net',
    message =>
      '<HTML><BODY>See attachment</BODY></HTML>',
    subject => 'Oracle Notes',
    mime_type => 'text/html'
    attachment => get_file('notes.txt'),
    att_inline => false,
    att_mime_type => 'text/plain',
    att_filename => 'notes.txt');
END;
/
```

### Sending E-Mail with a Text Attachment

The slide shows a procedure that calls the `UTL_MAIL.SEND_ATTACH_VARCHAR2`
procedure to send a textual or an HTML message with a text attachment. In addition to the
`sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that
provide values for the main part of the e-mail message, the `SEND_ATTACH_VARCHAR2`
procedure has the following parameters highlighted:

- The `attachment` parameter (required) accepts a `VARCHAR2` data type with a
  maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is a Boolean (default `TRUE`) to indicate that
  the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment.
  If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment.
  It is `NULL` by default, in which case, the name is assigned a default name.

The `get_file` function in the example uses a `BFILE` to read a text file from the
operating system directories for the value of the `attachment` parameter, which could
simply be populated from a `VARCHAR2` variable. The code for `get_file` is shown on
the following page.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-19**

### Sending E-Mail with a Text Attachment (continued)

The `get_file` function uses the `DBMS_LOB` package to read a binary file from the operating system, and uses the `UTL_RAW` package to convert the `RAW` binary data into readable text data in the form of a `VARCHAR2` data type:

```
CREATE OR REPLACE FUNCTION get_file(
  filename VARCHAR2, dir VARCHAR2 := 'TEMP')
RETURN VARCHAR2 IS
  contents VARCHAR2(32767);
  file  BFILE := BFILENAME(dir, filename);
BEGIN
  DBMS_LOB.FILEOPEN(file, DBMS_LOB.FILE_READONLY);
  contents := UTL_RAW.CAST_TO_VARCHAR2(
                  DBMS_LOB.SUBSTR(file));
  DBMS_LOB.CLOSE(file);
  RETURN contents;
END;
/
```

**Note:** Alternatively, you could read the contents of the text file into a `VARCHAR2` variable by using the `UTL_FILE` package functionality.

The preceding example requires the `TEMP` directory to be created similar to the following statement in *i*SQL*Plus:

```
CREATE DIRECTORY temp AS 'e:\temp';
```

**Note:** The `CREATE ANY DIRECTORY` system privilege is required to execute this statement.

# DBMS_SCHEDULER **Package**

**The database Scheduler comprises several components to enable jobs to be run. Use the DBMS_SCHEDULER package to create each job with:**

- **A unique job name**
- **A program ("what" should be executed)**
- **A schedule ("when" it should run)**

```
  Program ──── Job ──── Schedule
     │          │  ┆        ┆
  Arguments  Arguments ┄┄┄┄┄┘ ┆
              Job class     Window
```
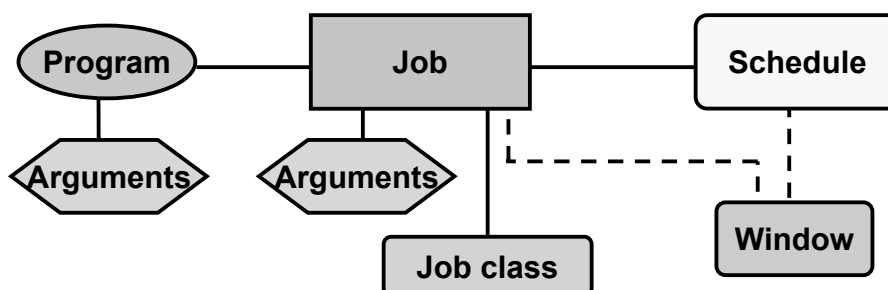
ORACLE

### DBMS_SCHEDULER **Package**

Oracle Database 10*g* provides a collection of subprograms in the DBMS_SCHEDULER package to simplify management and to provide a rich set of functionality for complex scheduling tasks. Collectively, these subprograms are called the Scheduler and can be called from any PL/SQL program. The Scheduler enables database administrators and application developers to control when and where various tasks take place. By ensuring that many routine database tasks occur without manual intervention, you can lower operating costs, implement more reliable routines, and minimize human error.

The diagram shows the following architectural components of the Scheduler:
- A **job** is the combination of a program and a schedule. Arguments required by the program can be provided with the program or the job. All job names have the format [schema.] name. When you create a job, you specify the job name, a program, a schedule, and (optionally) job characteristics that can be provided through a **job class**.
- A **program** determines what should be run. Every automated job involves a particular executable, whether it is a PL/SQL block, a stored procedure, a native binary executable, or a shell script. A program provides metadata about a particular executable and may require a list of arguments.
- A **schedule** specifies when and how many times a job is executed.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-21**

**`DBMS_SCHEDULER` Package (continued)**

- A **job class** defines a category of jobs that share common resource usage requirements and other characteristics. At any given time, each job can belong to only a single job class. A job class has the following attributes:
    - A database **service** name. The jobs in the job class will have an affinity to the particular service specified—that is, the jobs will run on the instances that cater to the specified service.
    - A **resource consumer group**, which classifies a set of user sessions that have common resource-processing requirements. At any given time, a user session or job class can belong to a single resource consumer group. The resource consumer group that the job class associates with determines the resources that are allocated to the job class.
- A **window** is represented by an interval of time with a well-defined beginning and end, and is used to activate different resource plans at different times.

The slide focuses on the job component as the primary entity. However, a program, a schedule, a window, and a job class are components that can be created as individual entities that can be associated with a job to be executed by the Scheduler. When a job is created, it may contain all the information needed in-line—that is, in the call that creates the job. Alternatively, creating a job may reference a program or schedule component that was previously defined. Examples of this are discussed in the next few pages.

For more information about the Scheduler, see the Online Course titled *Oracle Database 10g: Configure and Manage Jobs with the Scheduler*.

# Creating a Job

**A job can be created in several ways by using a combination of in-line parameters, named `Programs`, and named `Schedules`. You can create a job with the `CREATE_JOB` procedure by:**

- **Using in-line information with the "what" and the schedule specified as parameters**
- **Using a named (saved) program and specifying the schedule in-line**
- **Specifying what should be done in-line and using a named Schedule**
- **Using named Program and Schedule components**

**Note: Creating a job requires the `CREATE JOB` system privilege.**

ORACLE

## Creating a Job

The component that causes something to be executed at a specified time is called a **job**. Use the `DBMS_SCHEDULER.CREATE_JOB` procedure of the `DBMS_SCHEDULER` package to create a job, which is in a disabled state by default. A job becomes active and scheduled when it is explicitly enabled. To create a job, you:

- Provide a name in the format `[schema.]name`
- Need the `CREATE JOB` privilege

**Note:** A user with the `CREATE ANY JOB` privilege can create a job in any schema except the `SYS` schema. Associating a job with a particular class requires the `EXECUTE` privilege for that class.

In simple terms, a job can be created by specifying all the job details—the program to be executed (what) and its schedule (when)—in the arguments of the `CREATE_JOB` procedure. Alternatively, you can use predefined Program and Schedule components. If you have a named Program and Schedule, then these can be specified or combined with in-line arguments for maximum flexibility in the way a job is created.

A simple logical check is performed on the schedule information (that is, checking the date parameters when a job is created). The database checks whether the end date is after the start date. If the start date refers to a time in the past, then the start date is changed to the current date.

# Creating a Job with In-Line Parameters

**Specify the type of code, code, start time, and frequency of the job to be run in the arguments of the CREATE_JOB procedure.**

**Here is an example that schedules a PL/SQL block every hour:**

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB(
    job_name => 'JOB_NAME',
    job_type => 'PLSQL_BLOCK',
    job_action => 'BEGIN ...; END;',
    start_date => SYSTIMESTAMP,
    repeat_interval=>'FREQUENCY=HOURLY;INTERVAL=1',
    enabled => TRUE);
END;
/
```

### Creating a Job with In-Line Parameters

You can create a job to run a PL/SQL block, stored procedure, or external program by using the DBMS_SCHEDULER.CREATE_JOB procedure. The CREATE_JOB procedure can be used directly without requiring you to create Program or Schedule components.

The example in the slide shows how you can specify all the job details in-line. The parameters of the CREATE_JOB procedure define "what" is to be executed, the schedule, and other job attributes. The following parameters define what is to be executed:

- The job_type parameter can be one of three values:
  - PLSQL_BLOCK for any PL/SQL block or SQL statement. This type of job cannot accept arguments.
  - STORED_PROCEDURE for any stored stand-alone or packaged procedure. The procedures can accept arguments that are supplied with the job.
  - EXECUTABLE for an executable command-line operating system application
- The schedule is specified by using the following parameters:
  - The start_date accepts a time stamp, and the repeat_interval is string-specified as a calendar or PL/SQL expression. An end_date can be specified.

**Note:** String expressions that are specified for repeat_interval are discussed later. The example specifies that the job should run every hour.

**Oracle Database 10*g*: Develop PL/SQL Program Units 5-24**

# Creating a Job Using a Program

- **Use `CREATE_PROGRAM` to create a program:**

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM(
   program_name => 'PROG_NAME',
   program_type => 'PLSQL_BLOCK',
   program_action => 'BEGIN ...; END;');
END;
```

- **Use overloaded `CREATE_JOB` procedure with its `program_name` parameter:**

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   enabled => TRUE);
END;
```

ORACLE

## Creating a Job Using a Program

The DBMS_SCHEDULER.CREATE_PROGRAM procedure defines a program that must be assigned a unique name. Creating the program separately for a job enables you to:
- Define the action once and then reuse this action within multiple jobs
- Change the schedule for a job without having to re-create the PL/SQL block
- Change the program executed without changing all the jobs

The program action string specifies a procedure, executable name, or PL/SQL block depending on the value of the program_type parameter, which can be:
- PLSQL_BLOCK to execute an anonymous block or SQL statement
- STORED_PROCEDURE to execute a stored procedure, such as PL/SQL, Java, or C
- EXECUTABLE to execute operating system command-line programs

The example shown in the slide demonstrates calling an anonymous PL/SQL block. You can also call an external procedure within a program, as in the following example:

```
        DBMS_SCHEDULER.CREATE_PROGRAM(program_name =>
        'GET_DATE',
            program_action => '/usr/local/bin/date',
            program_type => 'EXECUTABLE');
```

To create a job with a program, specify the program name in the program_name argument in the call to the DBMS_SCHEDULER.CREATE_JOB procedure, as shown in the slide.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-25**

# Creating a Job for a Program with Arguments

- **Create a program:**

```
DBMS_SCHEDULER.CREATE_PROGRAM(
  program_name => 'PROG_NAME',
  program_type => 'STORED_PROCEDURE',
  program_action => 'EMP_REPORT');
```

- **Define an argument:**

```
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'PROG_NAME',
  argument_name => 'DEPT_ID',
  argument_position=> 1, argument_type=> 'NUMBER',
  default_value => '50');
```

- **Create a job specifying the number of arguments:**

```
DBMS_SCHEDULER.CREATE_JOB('JOB_NAME', program_name
  => 'PROG_NAME', start_date => SYSTIMESTAMP,
  repeat_interval => 'FREQ=DAILY',
  number_of_arguments => 1, enabled => TRUE);
```

ORACLE

## Creating a Job for a Program with Arguments

Programs, such as PL/SQL or external procedures, may require input arguments. Using the
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT procedure, you can define an
argument for an existing program. The DEFINE_PROGRAM_ARGUMENT procedure
parameters include the following:

- The program_name specifies an existing program that is to be altered.
- The argument_name specifies a unique argument name for the program.
- The argument_position specifies the position in which the argument is passed
  when the program is called.
- The argument_type specifies the data type of the argument value that is passed
  to the called program.
- The default_value specifies a default value that is supplied to the program if
  the job that schedules the program does not provide a value.

The slide shows how to create a job executing a program with one argument. The program
argument default value is 50. To change the program argument value for a job, use:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE(
    job_name => 'JOB_NAME',
    argument_name => 'DEPT_ID', argument_value => '80');
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-26**

# Creating a Job Using a Schedule

- **Use `CREATE_SCHEDULE` to create a schedule:**

```
BEGIN
  DBMS_SCHEDULER.CREATE_SCHEDULE('SCHED_NAME',
   start_date => SYSTIMESTAMP,
   repeat_interval => 'FREQ=DAILY',
   end_date => SYSTIMESTAMP +15);
END;
```

- **Use `CREATE_JOB` by referencing the schedule in the `schedule_name` parameter:**

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   schedule_name => 'SCHED_NAME',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN ...; END;',
   enabled => TRUE);
END;
```

## Creating a Job Using a Schedule

You can create a common schedule that can be applied to different jobs without having to specify the schedule details each time. The following are the benefits of creating a schedule:

- It is reusable and can be assigned to different jobs.
- Changing the schedule affects all jobs using the schedule. The job schedules are changed once, not multiple times.

A schedule is precise to only the nearest second. Although the TIMESTAMP data type is more accurate, the Scheduler rounds off anything with a higher precision to the nearest second.

The start and end times for a schedule are specified by using the TIMESTAMP data type. The end_date for a saved schedule is the date after which the schedule is no longer valid. The schedule in the example is valid for 15 days after using it with a specified job.

The repeat_interval for a saved schedule must be created by using a calendaring expression. A NULL value for repeat_interval specifies that the job runs only once.

**Note:** You cannot use PL/SQL expressions to express the repeat interval for a saved schedule.

# Setting the Repeat Interval for a Job

- **Using a calendaring expression:**

```
repeat_interval=> 'FREQ=HOURLY; INTERVAL=4'
repeat_interval=> 'FREQ=DAILY'
repeat_interval=> 'FREQ=MINUTELY;INTERVAL=15'
repeat_interval=> 'FREQ=YEARLY;
                   BYMONTH=MAR,JUN,SEP,DEC;
                   BYMONTHDAY=15'
```

- **Using a PL/SQL expression:**

```
repeat_interval=> 'SYSDATE + 36/24'
repeat_interval=> 'SYSDATE + 1'
repeat_interval=> 'SYSDATE + 15/(24*60)'
```

**Setting the Repeat Interval for a Job**

When scheduling repeat intervals for a job, you can specify either a PL/SQL expression (if it is within a job argument) or a calendaring expression.

The examples in the slide include the following:
- FREQ=HOURLY;INTERVAL=4 indicates a repeat interval of every four hours.
- FREQ=DAILY indicates a repeat interval of every day, at the same time as the start date of the schedule.
- FREQ=MINUTELY;INTERVAL=15 indicates a repeat interval of every 15 minutes.
- FREQ=YEARLY;BYMONTH=MAR,JUN,SEP,DEC;BYMONTHDAY=15 indicates a repeat interval of every year on March 15, June 15, September 15, and December 15.

With a calendaring expression, the next start time for a job is calculated using the repeat interval and the start date of the job.

**Note:** If no repeat interval is specified (that is, if a NULL value is provided in the argument), the job runs only once on the specified start date.

# Creating a Job Using a Named Program and Schedule

- **Create a named program called** PROG_NAME **by using the** CREATE_PROGRAM **procedure.**
- **Create a named schedule called** SCHED_NAME **by using the** CREATE_SCHEDULE **procedure.**
- **Create a job referencing the named program and schedule:**

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
   program_name => 'PROG_NAME',
   schedule_name => 'SCHED_NAME',
   enabled => TRUE);
END;
/
```

ORACLE

**Creating a Job Using a Named Program and Schedule**

The example in the slide shows the final form for using the
DBMS_SCHEDULER.CREATE_JOB procedure. In this example, the named program
(PROG_NAME) and schedule (SCHED_NAME) are specified in their respective parameters
in the call to the DBMS_SCHEDULER.CREATE_JOB procedure.

With this example, you can see how easy it is to create jobs by using a predefined program
and schedule.

Some jobs and schedules can be too complex to cover in this course. For example, you can
create windows for recurring time plans and associate a resource plan with a window. A
resource plan defines attributes about the resources required during the period defined by
execution window.

For more information, refer to the Online Course titled *Oracle Database 10g: Configure
and Manage Jobs with the Scheduler*.

# Managing Jobs

- **Run a job:**

```
DBMS_SCHEDULER.RUN_JOB('SCHEMA.JOB_NAME');
```

- **Stop a job:**

```
DBMS_SCHEDULER.STOP_JOB('SCHEMA.JOB_NAME');
```

- **Drop a job even if it is currently running:**

```
DBMS_SCHEDULER.DROP_JOB('JOB_NAME', TRUE);
```

## Managing Jobs

After a job has been created, you can:
- Run the job by calling the RUN_JOB procedure specifying the name of the job. The job is immediately executed in your current session.
- Stop the job by using the STOP_JOB procedure. If the job is running currently, it is stopped immediately. The STOP_JOB procedure has two arguments:
  - **job_name:** Is the name of the job to be stopped
  - **force:** Attempts to gracefully terminate a job. If this fails and force is set to TRUE, then the job slave is terminated. (Default value is FALSE.) To use force, you must have the MANAGE SCHEDULER system privilege.
- Drop the job with the DROP_JOB procedure. This procedure has two arguments:
  - **job_name:** Is the name of the job to be dropped
  - **force:** Indicates whether the job should be stopped and dropped if it is currently running (Default value is FALSE.)

If the DROP_JOB procedure is called and the job specified is currently running, then the command fails unless the force option is set to TRUE. If the force option is set to TRUE, then any instance of the job that is running is stopped and the job is dropped.

**Note:** To run, stop, or drop a job that belongs to another user, you need ALTER privileges on that job or the CREATE ANY JOB system privilege.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-30**

# Data Dictionary Views

- **[DBA | ALL | USER]_SCHEDULER_JOBS**
- **[DBA | ALL | USER]_SCHEDULER_RUNNING_JOBS**
- **[DBA | ALL]_SCHEDULER_JOB_CLASSES**
- **[DBA | ALL | USER]_SCHEDULER_JOB_LOG**
- **[DBA | ALL | USER]_SCHEDULER_JOB_RUN_DETAILS**
- **[DBA | ALL | USER]_SCHEDULER_PROGRAMS**

ORACLE

**Data Dictionary Views**

The DBA_SCHEDULER_JOB_LOG view shows all completed job instances, both successful and failed.

To view the state of your jobs, use the following query:

```
SELECT job_name, program_name, job_type, state
FROM USER_SCHEDULER_JOBS;
```

To determine which instance a job is running on, use the following query:

```
SELECT owner, job_name, running_instance,
resource_consumer_group
FROM DBA_SCHEDULER_RUNNING_JOBS;
```

To determine information about how a job ran, use the following query:

```
SELECT job_name, instance_id, req_start_date,
actual_start_date, status
FROM ALL_SCHEDULER_JOB_RUN_DETAILS;
```

To determine the status of your jobs, use the following query:

```
SELECT job_name, status, error#, run_duration, cpu_used
FROM USER_SCHEDULER_JOB_RUN_DETAILS;
```

# Summary

**In this lesson, you should have learned how to:**
- **Use various preinstalled packages that are provided by the Oracle server**
- **Use the following packages:**
  - `DBMS_OUTPUT` **to buffer and display text**
  - `UTL_FILE` **to write operating system text files**
  - `HTP` **to generate HTML documents**
  - `UTL_MAIL` **to send messages with attachments**
  - `DBMS_SCHEDULER` **to automate processing**
- **Create packages individually or by using the** `catproc.sql` **script**

ORACLE

## Summary

This lesson covers a small subset of packages provided with the Oracle database. You have extensively used `DBMS_OUTPUT` for debugging purposes and displaying procedurally generated information on the screen in *i*SQL*Plus.

In this lesson, you should have learned how to use the power features provided by the database to create text files in the operating system by using `UTL_FILE`, generate HTML reports with the `HTP` package, send e-mail with or without binary or text attachments by using the `UTL_MAIL` package, and schedule PL/SQL and external code for execution with the `DBMS_SCHEDULER` package.

**Note:** For more information about all PL/SQL packages and types, refer to the *PL/SQL Packages and Types Reference*.

# Practice 5: Overview

**This practice covers the following topics:**
- **Using `UTL_FILE` to generate a text report**
- **Using `HTP` to generate a Web page report**
- **Using `DBMS_SCHEDULER` to automate report processing**

ORACLE

**Practice 5: Overview**

In this practice, you use `UTL_FILE` to generate a text file report of employees in each department. You create a procedure to generate an HTML version of the employee report, and write a SQL script file to spool the results to a text file. You use the `DBMS_SCHEDULER` to run the first report that creates a text file every five minutes.

**Practice 5**

1. Create a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their departments.

   a. Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.
   **Note:** Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.

   b. Invoke the program, using the second parameter with a name such as sal_rpt*xx*.txt, where *xx* represents your user number (for example, 01, 15, and so on). The following is a sample output from the report file:

   ```
   Employees who earn more than average salary:
   REPORT GENERATED ON 26-FEB-04
   Hartstein                          20        $13,000.00
   Raphaely                           30        $11,000.00
   Marvis                             40         $6,500.00
       ...
       *** END OF REPORT ***
   ```

   **Note:** The data displays the employee's last name, department ID, and salary. Ask your instructor to provide instructions on how to obtain the report file from the server using the Putty PSFTP utility.

2. Create a new procedure called WEB_EMPLOYEE_REPORT that generates the same data as the EMPLOYEE_REPORT.

   a. First, execute SET SERVEROUTPUT ON, and then execute htp.print('hello') followed by executing OWA_UTIL.SHOWPAGE. The exception messages generated can be ignored.

   b. Write the WEB_EMPLOYEE_REPORT procedure by using the HTP package to generate an HTML report of employees with a salary greater than the average for their departments. If you know HTML, create an HTML table; otherwise, create simple lines of data.
   **Hint:** Copy the cursor definition and the FOR loop from the EMPLOYEE_REPORT procedure for the basic structure for your Web report.

   c. Execute the procedure using *i*SQL*Plus to generate the HTML data into a server buffer, and execute the OWA_UTIL.SHOWPAGE procedure to display contents of the buffer. Remember that SERVEROUTPUT should be ON before you execute the code.

   d. Create an HTML file called web_employee_report.htm containing the output result text that you select and copy from the opening <HTML> tag to the closing </HTML> tag. Paste the copied text into the file and save it to disk. Double-click the file to display the results in your default browser.

## Practice 5 (continued)

3. Your boss wants to run the employee report frequently. You create a procedure that uses the DBMS_SCHEDULER package to schedule the EMPLOYEE_REPORT procedure for execution. You should use parameters to specify a frequency, and an optional argument to specify the number of minutes after which the scheduled job should be terminated.

   a. Create a procedure called SCHEDULE_REPORT that provides the following two parameters:
      - interval: To specify a string indicating the frequency of the scheduled job
      - minutes: To specify the total life in minutes (default of 10) for the scheduled job, after which it is terminated. The code divides the duration by the quantity (24 × 60) when it is added to the current date and time to specify the termination time.

      When the procedure creates a job, with the name of EMPSAL_REPORT by calling DBMS_SCHEDULER.CREATE_JOB, the job should be enabled and scheduled for the PL/SQL block to start immediately. You must schedule an anonymous block to invoke the EMPLOYEE_REPORT procedure so that the file name can be updated with a new time, each time the report is executed. The EMPLOYEE_REPORT is given the directory name supplied by your instructor for task 1, and the file name parameter is specified in the following format: sal_rptxx_hh24-mi-ss.txt, where xx is your assigned user number and hh24-mi-ss represents the hours, minutes, and seconds.

      Use the following local PL/SQL variable to construct a PL/SQL block:

      ```
      plsql_block VARCHAR2(200) :=
      'BEGIN'||
      ' EMPLOYEE_REPORT(''UTL_FILE'','||
      '''sal_rptXX_''||to_char(sysdate,''HH24-MI-
      SS'')||''.txt'');'||
      'END;';
      ```

      This code is provided to help you because it is a nontrivial PL/SQL string to construct. In the PL/SQL block, **XX** is your student number.

   b. Test the SCHEDULE_REPORT procedure by executing it with a parameter specifying a frequency of every two minutes and a termination time 10 minutes after it starts.
      **Note:** You must connect to the database server by using PSFTP to check whether your files are created.

   c. During and after the process, you can query the job_name and enabled columns from the USER_SCHEDULER_JOBS table to check whether the job still exists.
      **Note:** This query should return no rows after 10 minutes have elapsed.

**Oracle Database 10*g*: Develop PL/SQL Program Units   5-35**

# Dynamic SQL and Metadata

**6**

## Objectives

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

- **Describe the execution flow of SQL statements**
- **Build and execute SQL statements dynamically using Native Dynamic SQL (that is, with `EXECUTE IMMEDIATE` statements)**
- **Compare Native Dynamic SQL with the `DBMS_SQL` package approach**
- **Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects**

ORACLE

**Lesson Aim**

In this lesson, you learn to construct and execute SQL statements dynamically—that is, at run time using the Native Dynamic SQL statements in PL/SQL. You compare Native Dynamic SQL to the `DBMS_SQL` package, which provides similar capabilities.

You learn how to use the `DBMS_METADATA` package to retrieve metadata from the database dictionary as Extensible Markup Language (XML) or creation DDL and to submit the XML to re-create the object.

# Execution Flow of SQL

- **All SQL statements go through various stages:**
  - **Parse**
  - **Bind**
  - **Execute**
  - **Fetch**
- **Some stages may not be relevant for all statements—for example, the fetch phase is applicable to queries.**

**Note: For embedded SQL statements (`SELECT`, DML, `COMMIT`, and `ROLLBACK`), the parse and bind phases are done at compile time. For dynamic SQL statements, all phases are performed at run time.**

## Steps to Process SQL Statements

All SQL statements have to go through various stages. However, some stages may not be relevant for all statements. The following are the key stages:

- **Parse:** Every SQL statement must be parsed. Parsing the statement includes checking the statement's syntax and validating the statement, ensuring that all references to objects are correct and that the relevant privileges to those objects exist.
- **Bind:** After parsing, the Oracle server may need values from or for any bind variable in the statement. The process of obtaining these values is called binding variables. This stage may be skipped if the statement does not contain bind variables.
- **Execute:** At this point, the Oracle server has all necessary information and resources, and the statement is executed. For nonquery statements, this is the last phase.
- **Fetch:** In the fetch stage, which is applicable to queries, the rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched.

# Dynamic SQL

**Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:**

- **Is constructed and stored as a character string within the application**
- **Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)**
- **Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL**
- **Is executed with Native Dynamic SQL statements or the `DBMS_SQL` package**

ORACLE

**Dynamic SQL**

The embedded SQL statements available in PL/SQL are limited to SELECT, INSERT, UPDATE, DELETE, COMMIT, and ROLLBACK, all of which are parsed at compile time—that is, they have a fixed structure. You need to use dynamic SQL functionality if you require:

- The structure of a SQL statement to be altered at run time
- Access to data definition language (DDL) statements and other SQL functionality in PL/SQL

To perform these kinds of tasks in PL/SQL, you must construct SQL statements dynamically in character strings and execute them using either of the following:

- Native Dynamic SQL statements with EXECUTE IMMEDIATE
- The DBMS_SQL package

The process of using SQL statements that are not embedded in your source program and are constructed in strings and executed at run time is known as "dynamic SQL." The SQL statements are created dynamically at run time and can access and use PL/SQL variables. For example, you create a procedure that uses dynamic SQL to operate on a table whose name is not known until run time, or execute a DDL statement (such as CREATE TABLE), a data control statement (such as GRANT), or a session control statement (such as ALTER SESSION).

# Native Dynamic SQL

- **Provides native support for dynamic SQL directly in the PL/SQL language**
- **Provides the ability to execute SQL statements whose structure is unknown until execution time**
- **Is supported by the following PL/SQL statements:**
  - `EXECUTE IMMEDIATE`
  - `OPEN-FOR`
  - `FETCH`
  - `CLOSE`

ORACLE

**Oracle University and En-Sof Informatica E Treinamento Ltda use only**

### Native Dynamic SQL

In Oracle 8 and earlier releases, the only way to implement dynamic SQL in a PL/SQL application was by using the `DBMS_SQL` package. With Oracle 8*i* and later releases, the PL/SQL environment provides Native Dynamic SQL as an alternative.

Native Dynamic SQL provides the ability to dynamically execute SQL statements whose structure is constructed at execution time. The following statements have been added or extended in PL/SQL to support Native Dynamic SQL:

- **`EXECUTE IMMEDIATE`:** Prepares a statement, executes it, returns variables, and then deallocates resources
- **`OPEN-FOR`:** Prepares and executes a statement using a cursor variable
- **`FETCH`:** Retrieves the results of an opened statement by using the cursor variable
- **`CLOSE`:** Closes the cursor used by the cursor variable and deallocates resources

You can use bind variables in the dynamic parameters in the `EXECUTE IMMEDIATE` and `OPEN` statements. Native Dynamic SQL includes the following capabilities:

- Define a dynamic SQL statement.
- Bind instances of any SQL data types supported in PL/SQL.
- Handle `IN`, `IN OUT`, and `OUT` bind variables that are bound by position, not by name.

# Using the EXECUTE IMMEDIATE Statement

**Use the EXECUTE IMMEDIATE statement for Native Dynamic SQL or PL/SQL anonymous blocks:**

```
EXECUTE IMMEDIATE dynamic_string
  [INTO {define_variable
      [, define_variable] ... | record}]
  [USING [IN|OUT|IN OUT] bind_argument
      [, [IN|OUT|IN OUT] bind_argument] ... ];
```

*   **INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.**
*   **USING is used to hold all bind arguments. The default parameter mode is IN, if not specified.**

ORACLE

### Using the EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement can be used to execute SQL statements or PL/SQL anonymous blocks. The syntactical elements include the following:

*   dynamic_string is a string expression that represents a dynamic SQL statement (without terminator) or a PL/SQL block (with terminator).
*   define_variable is a PL/SQL variable that stores the selected column value.
*   record is a user-defined or %ROWTYPE record that stores a selected row.
*   bind_argument is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.
*   The INTO clause specifies the variables or record into which column values are retrieved. It is used only for single-row queries. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.
*   The USING clause holds all bind arguments. The default parameter mode is IN.

You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL).

**Note:** Use OPEN-FOR, FETCH, and CLOSE for a multirow query. The syntax shown in the slide is not complete because support exists for bulk-processing operations (which is a topic that is not covered in this course).

**Oracle Database 10g: Develop PL/SQL Program Units   6-6**

# Dynamic SQL with a DDL Statement

- **Create a table:**

```
CREATE PROCEDURE create_table(
  table_name VARCHAR2, col_specs  VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE '||table_name||
                    ' (' || col_specs || ')';
END;
/
```

- **Call example:**

```
BEGIN
  create_table('EMPLOYEE_NAMES',
   'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```

ORACLE

**Dynamic SQL with a DDL Statement**

The code examples show the creation of a `create_table` procedure that accepts the table name and column definitions (specifications) as parameters.

The call shows the creation of a table called `EMPLOYEE_NAMES` with two columns:

- An ID column with a `NUMBER` data type used as a primary key
- A name column of up to 40 characters for the employee name

Any DDL statement can be executed by using the syntax shown in the slide, whether the statement is dynamically constructed or specified as a literal string. You can create and execute a statement that is stored in a PL/SQL string variable, as in the following example:

```
CREATE PROCEDURE add_col(table_name VARCHAR2,
                         col_spec VARCHAR2) IS
  stmt VARCHAR2(100) := 'ALTER TABLE ' || table_name ||
                        ' ADD '||col_spec;
BEGIN
  EXECUTE IMMEDIATE stmt;
END;
/
```

To add a new column to a table, enter the following:

```
EXECUTE add_col('employee_names', 'salary number(8,2)')
```

**Oracle Database 10g: Develop PL/SQL Program Units   6-7**

# Dynamic SQL with DML Statements

- **Delete rows from any table:**

```
CREATE FUNCTION del_rows(table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM '||table_name;
  RETURN SQL%ROWCOUNT;
END;
```

```
BEGIN DBMS_OUTPUT.PUT_LINE(
  del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');
END;
```

- **Insert a row into a table with two columns:**

```
CREATE PROCEDURE add_row(table_name VARCHAR2,
    id NUMBER, name VARCHAR2) IS
BEGIN
  EXECUTE IMMEDIATE 'INSERT INTO '||table_name||
        ' VALUES (:1, :2)' USING id, name;
END;
```

ORACLE

## Dynamic SQL with DML Statements

The examples in the slide demonstrate the following:
- The del_rows function deletes rows from a specified table and returns the number of rows deleted by using the implicit SQL cursor %ROWCOUNT attribute. Executing the function is shown below the example for creating a function.
- The add_row procedure shows how to provide input values to a dynamic SQL statement with the USING clause. The bind variable names :1 and :2 are not important, but the order of the variable names (id and name) in the USING clause is associated to the bind variables by position, in the order of their respective appearance. Therefore, the PL/SQL variable id is assigned to the :1 placeholder, and the name variable is assigned to the :2 placeholder. Placeholder/bind variable names can be alphanumeric but must be preceded with a colon.

**Note:** The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes the dynamic SQL statement. Dynamic SQL statements are always parsed.

Also, note that a COMMIT operation is not performed in either of the examples. Therefore, the operations can be undone with a ROLLBACK statement.

# Dynamic SQL with a Single-Row Query

**Example of a single-row query:**

```
CREATE FUNCTION get_emp(emp_id NUMBER)
RETURN employees%ROWTYPE IS
  stmt VARCHAR2(200);
  emprec employees%ROWTYPE;
BEGIN
  stmt := 'SELECT * FROM employees ' ||
          'WHERE employee_id = :id';
  EXECUTE IMMEDIATE stmt INTO emprec USING emp_id;
  RETURN emprec;
END;
/
```

```
DECLARE
  emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: '||emprec.last_name);
END;
/
```

ORACLE

### Dynamic SQL with a Single-Row Query

The single-row query example demonstrates the get_emp function that retrieves an
EMPLOYEES record into a variable specified in the INTO clause. It also shows how to
provide input values for the WHERE clause.

The anonymous block is used to execute the get_emp function and return the result into
a local EMPLOYEES record variable.

The example could be enhanced to provide alternative WHERE clauses depending on input
parameter values, making it more suitable for dynamic SQL processing.

# Dynamic SQL with a Multirow Query

**Use** `OPEN-FOR`, `FETCH`, **and** `CLOSE` **processing:**

```
CREATE PROCEDURE list_employees(deptid NUMBER) IS
  TYPE emp_refcsr IS REF CURSOR;
  emp_cv   emp_refcsr;
  emprec   employees%ROWTYPE;
  stmt varchar2(200) := 'SELECT * FROM employees';
BEGIN
  IF deptid IS NULL THEN OPEN emp_cv FOR stmt;
  ELSE
    stmt := stmt || ' WHERE department_id = :id';
    OPEN emp_cv FOR stmt USING deptid;
  END IF;
  LOOP
    FETCH emp_cv INTO emprec;
    EXIT WHEN emp_cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emprec.department_id||
                    ' ' ||emprec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
```

ORACLE

**Dynamic SQL with a Multirow Query**

The example in the slide shows how to execute a multirow query by performing the following programming steps:

- Declaring a `REF CURSOR` type
- Declaring a cursor variable based on the `REF CURSOR` type name that you declare
- Executing an `OPEN-FOR` statement that uses the cursor variable
- Using a `FETCH` statement referencing the cursor variable until all records are processed
- Executing the `CLOSE` statement by using the cursor variable

This process is the same as using static cursor definitions. However, the `OPEN-FOR` syntax accepts a string literal or variable specifying the `SELECT` statement, which can be dynamically constructed.

**Note:** The next page provides a brief introduction to the `REF CURSOR` type and cursor variables. An alternative to this is using the `BULK COLLECT` syntax supported by Native Dynamic SQL statements (a topic that is not covered in this course).

# Declaring Cursor Variables

- **Declare a cursor type as** `REF CURSOR`:

```
CREATE PROCEDURE process_data IS
  TYPE ref_ctype IS REF CURSOR; -- weak ref cursor
  TYPE emp_ref_ctype IS REF CURSOR -- strong
    RETURN employees%ROWTYPE;
  :
```

- **Declare a cursor variable using the cursor type:**

```
  :
  dept_csrvar ref_ctype;
  emp_csrvar  emp_ref_ctype;
BEGIN
  OPEN emp_csrvar FOR SELECT * FROM employees;
  OPEN dept_csrvar FOR SELECT * from departments;
  -- Then use as normal cursors
END;
```

ORACLE

**Declaring Cursor Variables**

A cursor variable is a PL/SQL identifier whose type name has been declared as a REF
CURSOR type. Creating a cursor variable involves two steps:

- Declaring a type name as a REF CURSOR type
- Declaring a PL/SQL variable by using the type name declared as a REF CURSOR
  type

The slide examples create two reference cursor types:

- The ref_ctype is a generic reference cursor, known as a weak reference cursor. A
  weak reference cursor can be associated with any query.
- The emp_ref_ctype is a strong reference cursor type that must be associated with
  a type-compatible query: the query must return data that is compatible with the type
  specified after the RETURN keyword (for example, an EMPLOYEES row type).

After a cursor variable is declared by using a reference cursor type name, the cursor
variable that is associated with a query is opened by using the OPEN-FOR syntax shown
in the slide. The standard FETCH, cursor attributes, and CLOSE operations used with
explicit cursors are also applicable with cursor variables. To compare cursor variables with
explicit cursors:

- A cursor variable can be associated with more than one query at run time
- An explicit cursor is associated with one query at compilation time

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-11**

# Dynamically Executing a PL/SQL Block

**Execute a PL/SQL anonymous block dynamically:**

```
CREATE FUNCTION annual_sal(emp_id NUMBER)
RETURN NUMBER IS
  plsql varchar2(200) :=
    'DECLARE '||
    ' emprec employees%ROWTYPE; '||
    'BEGIN '||
    ' emprec := get_emp(:empid); ' ||
    ' :res := emprec.salary * 12; ' ||
    'END;';
  result NUMBER;
BEGIN
 EXECUTE IMMEDIATE plsql
          USING IN emp_id, OUT result;
  RETURN result;
END;
/
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

## Dynamically Executing a PL/SQL Block

The `annual_sal` function dynamically constructs an anonymous PL/SQL block. The PL/SQL block contains bind variables for:

- The input of the employee ID using the `:empid` placeholder
- The output result computing the annual employees' salary using the placeholder called `:res`

**Note:** This example demonstrates how to use the `OUT` result syntax (in the `USING` clause of the `EXECUTE IMMEDIATE` statement) to obtain the result calculated by the PL/SQL block. The procedure output variables and function return values can be obtained in a similar way from a dynamically executed anonymous PL/SQL block.

# Using Native Dynamic SQL to Compile PL/SQL Code

**Compile PL/SQL code with the `ALTER` statement:**

- `ALTER PROCEDURE name COMPILE`
- `ALTER FUNCTION name COMPILE`
- `ALTER PACKAGE name COMPILE SPECIFICATION`
- `ALTER PACKAGE name COMPILE BODY`

```
CREATE PROCEDURE compile_plsql(name VARCHAR2,
 plsql_type VARCHAR2, options VARCHAR2 := NULL) IS
  stmt varchar2(200) := 'ALTER '|| plsql_type ||
                        ' '|| name || ' COMPILE';
BEGIN
 IF options IS NOT NULL THEN
   stmt := stmt || ' ' || options;
 END IF;
 EXECUTE IMMEDIATE stmt;
END;
/
```

ORACLE

**Using Native Dynamic SQL to Compile PL/SQL Code**

The `compile_plsql` procedure in the example can be used to compile different PL/SQL code using the `ALTER` DDL statement. Four basic forms of the `ALTER` statement are shown to compile:

- A procedure
- A function
- A package specification
- A package body

**Note:** If you leave out the keyword `SPECIFICATION` or `BODY` with the `ALTER PACKAGE` statement, then the specification and body are both compiled.

Here are examples of calling the procedure in the slide for each of the four cases, respectively:

```
EXEC compile_plsql ('list_employees', 'procedure')
EXEC compile_plsql ('get_emp', 'function')
EXEC compile_plsql ('mypack', 'package',
'specification')
EXEC compile_plsql ('mypack', 'package', 'body')
```

Here is an example of compiling with `debug` enabled for the `get_emp` function:

```
EXEC compile_plsql ('get_emp', 'function', 'debug')
```

**Oracle Database 10*g*: Develop PL/SQL Program Units  6-13**

# Using the `DBMS_SQL` Package

The `DBMS_SQL` package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:

- `OPEN_CURSOR`
- `PARSE`
- `BIND_VARIABLE`
- `EXECUTE`
- `FETCH_ROWS`
- `CLOSE_CURSOR`

ORACLE

## Using the `DBMS_SQL` Package

Using `DBMS_SQL`, you can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL, such as executing DDL statements in PL/SQL—for example, executing a `DROP TABLE` statement. The operations provided by this package are performed under the current user, not under the package owner `SYS`. The `DBMS_SQL` package provides the following subprograms to execute dynamic SQL:

- `OPEN_CURSOR` to open a new cursor and return a cursor ID number
- `PARSE` to parse the SQL statement—that is, it checks the statement syntax and associates it with the opened cursor. DDL statements are immediately executed when parsed.
- `BIND_VARIABLE` to bind a given value to a bind variable identified by its name in the statement being parsed. This is not needed if the statement does not have bind variables.
- `EXECUTE` to execute the SQL statement and return the number of rows processed
- `FETCH_ROWS` to retrieve the next row for a query (use in a loop for multiple rows)
- `CLOSE_CURSOR` to close the specified cursor

**Note:** Using the `DBMS_SQL` package to execute DDL statements can result in a deadlock. For example, the most likely reason is that the package is being used to drop a procedure that you are still using.

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-14**

# Using `DBMS_SQL` with a DML Statement

**Example of deleting rows:**

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (table_name VARCHAR2) RETURN NUMBER IS
  csr_id INTEGER;
  rows_del    NUMBER;
BEGIN
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id,
    'DELETE FROM '||table_name, DBMS_SQL.NATIVE);
  rows_del := DBMS_SQL.EXECUTE (csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  RETURN rows_del;
END;
/
```

```
CREATE table temp_emp as select * from employees;
BEGIN
 DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

ORACLE

## Using `DBMS_SQL` with a DML Statement

In the slide, the table name is passed into the `delete_all_rows` function. The function uses dynamic SQL to delete rows from the specified table, and returns a count representing the number of rows that are deleted after successful execution of the statement.

To process a DML statement dynamically, perform the following steps:
1. Use `OPEN_CURSOR` to establish an area in memory to process a SQL statement.
2. Use `PARSE` to establish the validity of the SQL statement.
3. Use the `EXECUTE` function to run the SQL statement. This function returns the number of rows processed.
4. Use `CLOSE_CURSOR` to close the cursor.

The steps to execute a DDL statement are similar; but step 3 is optional because a DDL statement is immediately executed when the `PARSE` is successfully done—that is, the statement syntax and semantics are correct. If you use the `EXECUTE` function with a DDL statement, then it does not do anything and returns a value of `0` for the number of rows processed because DDL statements do not process rows.

# Using DBMS_SQL with a
# Parameterized DML Statement

```
CREATE PROCEDURE insert_row (table_name VARCHAR2,
 id VARCHAR2, name VARCHAR2, region NUMBER) IS
  csr_id      INTEGER;
  stmt        VARCHAR2(200);
  rows_added NUMBER;
BEGIN
  stmt := 'INSERT INTO '||table_name||
          ' VALUES (:cid, :cname, :rid)';
  csr_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cid', id);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':cname', name);
  DBMS_SQL.BIND_VARIABLE(csr_id, ':rid', region);
  rows_added := DBMS_SQL.EXECUTE(csr_id);
  DBMS_SQL.CLOSE_CURSOR(csr_id);
  DBMS_OUTPUT.PUT_LINE(rows_added||' row added');
END;
/
```

ORACLE

### Using DBMS_SQL with a Parameterized DML Statement

The example in the slide performs the DML operation to insert a row into a specified table.
The example demonstrates the extra step required to associate values to bind variables that
exist in the SQL statement. For example, a call to the procedure shown in the slide is:

```
EXECUTE insert_row('countries', 'ZA', 'South Africa', 4)
```

After the statement is parsed, you must call the DBMS_SQL.BIND_VARIABLE
procedure to assign values for each bind variable that exists in the statement. The binding
of values must be done before executing the code. To process a SELECT statement
dynamically, perform the following steps after opening and before closing the cursor:
1. Execute DBMS_SQL.DEFINE_COLUMN for each column selected.
2. Execute DBMS_SQL.BIND_VARIABLE for each bind variable in the query.
3. For each row, perform the following steps:
   a. Execute DBMS_SQL.FETCH_ROWS to retrieve a row and return the number of
      rows fetched. Stop additional processing when a zero value is returned.
   b. Execute DBMS_SQL.COLUMN_VALUE  to retrieve each selected column
      value into each PL/SQL variable for processing.

Although this coding process is not complex, it is more time consuming to write and is
prone to error compared with using the Native Dynamic SQL approach.

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-16**

# Comparison of Native Dynamic SQL and the `DBMS_SQL` Package

**Native Dynamic SQL:**

- **Is easier to use than `DBMS_SQL`**
- **Requires less code than `DBMS_SQL`**
- **Enhances performance because the PL/SQL interpreter provides native support for it**
- **Supports all types supported by static SQL in PL/SQL, including user-defined types**
- **Can fetch rows directly into PL/SQL records**

ORACLE

**Comparison of Native Dynamic SQL and the `DBMS_SQL` Package**

Native Dynamic SQL provides the following advantages over the `DBMS_SQL` package.

**Ease of use:** Because Native Dynamic SQL is integrated with SQL, you can use it in the same way that you currently use static SQL within PL/SQL code. The code is typically more compact and readable compared with the code written with the `DBMS_SQL` package.

**Performance improvement:** Native Dynamic SQL performs significantly better than `DBMS_SQL`, in most circumstances, due to native support provided by the PL/SQL interpreter. The `DBMS_SQL` approach uses a procedural API and suffers from high procedure call and data copy overhead.

**Support for user-defined types:** Native Dynamic SQL supports all the types supported by static SQL in PL/SQL. Therefore, Native Dynamic SQL provides support for user-defined types such as user-defined objects, collections, and REFs. The `DBMS_SQL` package does not support these user-defined types. However, it has limited support for arrays.

**Support for fetching into records:** With Native Dynamic SQL, the rows resulting from a query can be directly fetched into PL/SQL records. The `DBMS_SQL` package does not support fetching into records structures.

# DBMS_METADATA Package

**The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.**
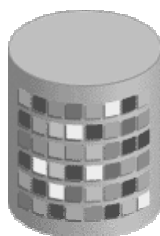
### DBMS_METADATA Package

You can invoke DBMS_METADATA to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

You can use DBMS_METADATA for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column data types, and so on), and then converting the metadata to data definition language (DDL) so that the object can be re-created on the same or another database. In the past, you needed to do this programmatically with problems resulting in each new release.

The DBMS_METADATA functionality is used for the Oracle 10*g* Export/Import replacement, commonly called "the Data Pump."

This package was introduced in Oracle9*i* and is further enhanced in Oracle Database 10*g*.

**Note:** For more information about the DBMS_DATAPUMP package, refer to the Online Course titled *Oracle Database 10g: Reduce Management - Tools and Utilities*.

**Metadata API**

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.
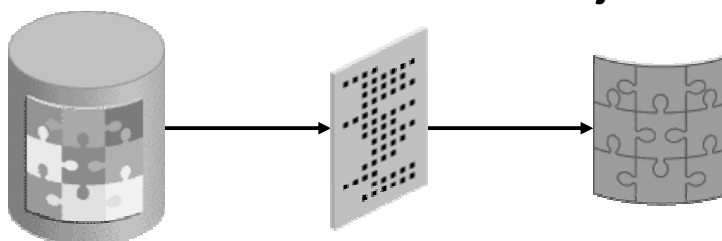
**Metadata API**

Every entity in the database is modeled as an object that belongs to an object type. For example, the EMPLOYEES table is an object; its object type is TABLE. When you fetch an object's metadata, you must specify the object type.

Every object type is implemented by using three entities:
- A user-defined type (UDT) whose attributes comprise all the metadata for objects of the type. An object's XML representation is a translation of a type instance into XML, with the XML tag names derived from the type attribute names. (In the case of tables, several UDTs are needed to represent the different varieties of the object type.)
- An object view of the UDT that populates instances of the object type
- An Extensible Style Sheet Language (XSL) script that converts the XML representation of an object into SQL DDL

# Subprograms in DBMS_METADATA

| Name | Description |
|------|-------------|
| OPEN | Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects. |
| SET_FILTER | Specifies restrictions on the objects to be retrieved such as the object name or schema |
| SET_COUNT | Specifies the maximum number of objects to be retrieved in a single FETCH_xxx call |
| GET_QUERY | Returns the text of the queries that will be used by FETCH_xxx |
| SET_PARSE_ITEM | Enables output parsing and specifies an object attribute to be parsed and returned |
| ADD_TRANSFORM | Specifies a transform that FETCH_xxx applies to the XML representation of the retrieved objects |
| SET_TRANSFORM_PARAM, SET_REMAP_PARAM | Specifies parameters to the XSLT stylesheet identified by transform_handle |
| FETCH_XXX | Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER |
| CLOSE | Invalidates the handle returned by OPEN and cleans up the associated state |

ORACLE

### Subprograms in DBMS_METADATA

The table provides an overview of the procedures and functions available in the DBMS_METADATA package. To retrieve metadata, you can specify:
- The kind of object retrieved, either an object type (a table, index, procedure) or a heterogeneous collection of object types forming a logical unit (such as database export and schema export)
- Selection criteria (owner, name, and so on)
- "parse items" attributes of objects to be parsed and returned separately
- Transformations on the output, implemented by XSLT scripts

The package provides two types of retrieval interface for two types of usage:
- **For programmatic use:** OPEN, SET_FILTER, SET_COUNT, GET_QUERY, SET_PARSE_ITEM, ADD_TRANSFORM, SET_TRANSFORM_PARAM, SET_REMAP_PARAM (new in Oracle Database 10*g*), FETCH_xxx, and CLOSE. These enable flexible selection criteria and the extraction of a stream of objects.
- **For use in SQL queries and for ad hoc browsing:** The GET_*xxx* interfaces (GET_XML and GET_DDL) return metadata for a single named object. The GET_DEPENDENT_xxx and GET_GRANTED_xxx interfaces return metadata for one or more dependent or granted objects. None of these APIs support heterogeneous object types.

# FETCH_xxx Subprograms

| Name | Description |
|------|-------------|
| FETCH_XML | **This function returns the XML metadata for an object as an XMLType.** |
| FETCH_DDL | **This function returns the DDL (either to create or to drop the object) into a predefined nested table.** |
| FETCH_CLOB | **This function returns the objects (transformed or not) as a CLOB.** |
| FETCH_XML_CLOB | **This procedure returns the XML metadata for the objects as a CLOB in an** IN OUT NOCOPY **parameter to avoid expensive LOB copies.** |

## FETCH_xxx Subprograms

These functions and procedures return metadata for objects meeting the criteria established by the call to the OPEN function that returned the handle, and the subsequent calls to SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on. Each call to FETCH_xxx returns the number of objects specified by SET_COUNT (or a smaller number, if fewer objects remain in the current cursor) until all objects have been returned.

# SET_FILTER Procedure

- **Syntax:**

```
PROCEDURE set_filter
( handle IN NUMBER,
  name    IN VARCHAR2,
  value   IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- **Example:**

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME',
'HR');
...
```

ORACLE

## SET_FILTER Procedure

You use the SET_FILTER procedure to identify restrictions on objects that are to be retrieved. For example, you can specify restrictions on an object or schema that is being retrieved. This procedure is overloaded with the parameters having the following meanings:

- handle is the handle returned from the OPEN function.
- name is the name of the filter. For each filter, the object type applies to its name, data type (text or Boolean), and meaning or effect (including its default value, if there is one).
- value is the value of the filter. It can be text, Boolean, or a numeric value.
- object_type_path is a path name designating the object types to which the filter applies. By default, the filter applies to the object type of the OPEN handle.

If you use an expression filter, then it is placed to the right of a SQL comparison, and the value is compared with it. The value must contain parentheses and quotation marks where appropriate. A filter value is combined with a particular object attribute to produce a WHERE condition in the query that fetches the objects.

# Filters

**There are over 70 filters, which are organized into object type categories such as:**
- **Named objects**
- **Tables**
- **Objects dependent on tables**
- **Index**
- **Dependent objects**
- **Granted objects**
- **Table data**
- **Index statistics**
- **Constraints**
- **All object types**
- **Database export**

ORACLE

**Filters**

There are over 70 filters that you can specify when using the SET_FILTER procedure. These filters are organized into object type categories. Some of the new object type categories in Oracle Database 10*g* are listed in the slide.

When using the SET_FILTER procedure, you specify the name of the filter and its respective value.

For example, you can use the SCHEMA filter with a value to identify the schema whose objects are selected. Then use a second call to the SET_FILTER procedure and use a filter named INCLUDE_USER that has a Boolean data type for its value. If it is set to TRUE, then objects containing privileged information about the user are retrieved.

```
DBMS_METADATA.SET_FILTER(handle, SCHEMA, 'HR');
DBMS_METADATA. SET_FILTER(handle, INCLUDE_USER, TRUE);
```

Each call to SET_FILTER causes a WHERE condition to be added to the underlying query that fetches the set of objects. The WHERE conditions are combined using an AND operator, so you can use multiple SET_FILTER calls to refine the set of objects to be returned.

# Examples of Setting Filters

**Set up the filter to fetch the `HR` schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain `PAYROLL` in the start of the view name:**

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
  'IN (''PAYROLL'', ''HR'')');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''FUNCTION''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PROCEDURE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
  '=''PACKAGE''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
  'LIKE ''PAYROLL%''', 'VIEW');
```

ORACLE

### Examples of Setting Filters

The example shown in the slide calls the SET_FILTER procedure several times to create a WHERE condition that identifies which object types are to be fetched. First, the objects in the PAYROLL and HR schemas are identified as object types to be fetched. Subsequently, the SET_FILTER procedure identifies certain object types (functions, procedures, and packages) and view object names to be excluded.

# Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
  h    NUMBER; th1  NUMBER; th2  NUMBER;
  doc  sys.ku$_ddls;  ←─ (1)
BEGIN
  h := DBMS_METADATA.OPEN('SCHEMA_EXPORT');  ←─ (2)
  DBMS_METADATA.SET_FILTER (h,'SCHEMA','HR');  ←─ (3)
  th1 := DBMS_METADATA.ADD_TRANSFORM (h,←─ (4)
     'MODIFY', NULL, 'TABLE');
  DBMS_METADATA.SET_REMAP_PARAM(th1, ←─ (5)
     'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
  th2 :=DBMS_METADATA.ADD_TRANSFORM(h, 'DDL');←─
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,←─
     'SQLTERMINATOR', TRUE);
  DBMS_METADATA.SET_TRANSFORM_PARAM(th2,←─ (6)
     'REF_CONSTRAINTS', FALSE, 'TABLE');
  LOOP
    doc := DBMS_METADATA.FETCH_DDL(h); ←─ (7)
    EXIT WHEN doc IS NULL;
  END LOOP;
  DBMS_METADATA.CLOSE(h); ←─ (8)
END;
```

ORACLE

## Programmatic Use: Example 1

In this example, all objects are retrieved from the HR schema as creation DDL. The MODIFY transform is used to change the tablespaces for the tables.

1. The DBMS_METADATA package has several predefined types that are owned by SYS. The sys.ku_$ddls type is defined in the DBMS_METADATA package. It is a table type that holds the CLOB type of data.
2. You use the OPEN function to specify the type of object to be retrieved, the version of its metadata, and the object model. It returns a context handle for the set of objects.
   In this example, 'SCHEMA_EXPORT' is the object type, and it indicates all metadata objects in a schema. There are 85 predefined types of objects for the model that you can specify for this parameter. Both the version of metadata and the object model parameters are not identified in this example. The version of metadata parameter defaults to 'COMPATIBLE'. You can also specify 'LATEST' or a specific database version. Currently, the model parameter supports only the Oracle model in Oracle Database 10g. This is the default.
3. The SET_FILTER procedure identifies restrictions on the objects to be retrieved.

### Programmatic Use: Example 1 (continued)

4. The `ADD_TRANSFORM` function specifies a transform that `FETCH_XXX` applies to the XML representation of the retrieved objects. You can have more than one transform. In the example, two transforms occur, one for each of the `th1` and `th2` program variables. The `ADD_TRANSFORM` function accepts four parameters and returns a number representing the opaque handle to the transform. The parameters are the handle returned from the `OPEN` statement, the name of the transform (`DDL`, `DROP`, or `MODIFY`), the encoding name (which is the name of the NLS [national language support] character set in which the style sheet pointed to by the name is encoded), and the object type. If the object type is omitted, the transform applies to all objects; otherwise, it applies only to the object type specified.
   The first transform shown in the program code is the handle returned from the `OPEN` function. The second transform shown in the code has two parameter values specified. The first parameter is the handle identified from the `OPEN` function. The second parameter value is DDL, which means the document is transformed to DDL that creates the object. The output of this transform is not an XML document. The third and fourth parameters are not specified. Both take the default values for the encoding and object type parameters.

5. The `SET_REMAP_PARAM` procedure identifies the parameters to the XSLT style sheet identified by the transform handle, which is the first parameter passed to the procedure. In the example, the second parameter value `'REMAP_TABLESPACE'` means that the objects have their tablespaces renamed from an old value to a new value. In the `ADD_TRANSFORM` function, the choices are `DDL`, `DROP`, or `MODIFY`. For each of these values, the `SET_REMAP_PARAM` identifies the name of the parameter. `REMAP_TABLESPACE` means the objects in the document will have their tablespaces renamed from an old value to a new value. The third and fourth parameters identify the old value and new value. In this example, the old tablespace name is `SYSTEM`, and the new tablespace name is `TBS1`.

6. `SET_TRANSFORM_PARAM` works similarly to `SET_REMAP_PARAM`. In the code shown, the first call to `SET_TRANSFORM_PARAM` identifies parameters for the `th2` variable. The `SQLTERMINATOR` and `TRUE` parameter values cause the SQL terminator (`;` or `/`) to be appended to each DDL statement.
   The second call to `SET_TRANSFORM_PARAM` identifies more characteristics for the `th2` variable. `REF_CONSTRAINTS`, `FALSE`, `TABLE` means that referential constraints on the tables are not copied to the document.

7. The `FETCH_DDL` function returns metadata for objects meeting the criteria established by the `OPEN`, `SET_FILTER`, `ADD_TRANSFORM`, `SET_REMAP_PARAM`, and `SET_TRANSFORM_PARAM` subroutines.

8. The `CLOSE` function invalidates the handle returned by the `OPEN` function and cleans up the associated state. Use this function to terminate the stream of objects established by the `OPEN` function.

# Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
 h    NUMBER; -- returned by 'OPEN'
 th   NUMBER; -- returned by 'ADD_TRANSFORM'
 doc  CLOB;
BEGIN
 -- specify the OBJECT TYPE
 h := DBMS_METADATA.OPEN('TABLE');
 -- use FILTERS to specify the objects desired
 DBMS_METADATA.SET_FILTER(h,'SCHEMA','HR');
 DBMS_METADATA.SET_FILTER(h,'NAME','EMPLOYEES');
 -- request to be TRANSFORMED into creation DDL
 th := DBMS_METADATA.ADD_TRANSFORM(h,'DDL');
 -- FETCH the object
 doc := DBMS_METADATA.FETCH_CLOB(h);
 -- release resources
 DBMS_METADATA.CLOSE(h);
 RETURN doc;
END;
/
```

**Programmatic Use: Example 2**

This example returns the metadata for the EMPLOYEES table. The result is:

```
set pagesize 0
set long  1000000
SELECT get_table_md FROM dual;

CREATE TABLE "HR"."EMPLOYEES"
    (  "EMPLOYEE_ID" NUMBER(6,0),
        "FIRST_NAME" VARCHAR2(20),
        "LAST_NAME" VARCHAR2(25) CONSTRAINT
"EMP_LAST_NAME_NN"
            NOT NULL ENABLE,
        "e-mail" VARCHAR2(25) CONSTRAINT "EMP_e-mail_NN"
            NOT NULL ENABLE,
        "PHONE_NUMBER" VARCHAR2(20),
        "HIRE_DATE" DATE CONSTRAINT "EMP_HIRE_DATE_NN"
            NOT NULL ENABLE,
        "JOB_ID" VARCHAR2(10) CONSTRAINT "EMP_JOB_NN"
            NOT NULL ENABLE,
        "SALARY" NUMBER(8,2),
    ...
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-27**

**Programmatic Use: Example 2 (continued)**

```
            "COMMISSION_PCT" NUMBER(2,2),
                 "MANAGER_ID" NUMBER(6,0),
                 "DEPARTMENT_ID" NUMBER(4,0),
                  CONSTRAINT "EMP_SALARY_MIN" CHECK (salary > 0)
        ENABLE,
                 CONSTRAINT "EMP_e-mail_UK" UNIQUE ("e-mail")
            USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
            STORAGE(INITIAL 65536 NEXT 65536 MINEXTENTS 1
              MAXEXTENTS 2147483645
            PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
               BUFFER_POOL DEFAULT)
            TABLESPACE "EXAMPLE"  ENABLE,
                 CONSTRAINT "EMP_EMP_ID_PK" PRIMARY KEY
    ("EMPLOYEE_ID")
            USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
            STORAGE(INITIAL 65536 NEXT 65536 MINEXTENTS 1
              MAXEXTENTS 2147483645
            PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
               BUFFER_POOL DEFAULT)
            TABLESPACE "EXAMPLE"  ENABLE,
                 CONSTRAINT "EMP_DEPT_FK" FOREIGN KEY
    ("DEPARTMENT_ID")
                   REFERENCES "HR"."DEPARTMENTS"
    ("DEPARTMENT_ID") ENABLE,
                 CONSTRAINT "EMP_JOB_FK" FOREIGN KEY ("JOB_ID")
                  REFERENCES "HR"."JOBS" ("JOB_ID") ENABLE,
                 CONSTRAINT "EMP_MANAGER_FK" FOREIGN KEY
    ("MANAGER_ID")
                   REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID")
        ENABLE
            ) PCTFREE 0 PCTUSED 40 INITRANS 1 MAXTRANS 255 C
              OMPRESS LOGGING
            STORAGE(INITIAL 65536 NEXT 65536 MINEXTENTS 1
               MAXEXTENTS 2147483645
            PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
               BUFFER_POOL DEFAULT)
            TABLESPACE "EXAMPLE"
You can accomplish the same effect with the browsing
   interface:
  SELECT dbms_metadata.get_ddl
         ('TABLE','EMPLOYEES','HR')
     FROM dual;
```

# Browsing APIs

| Name | Description |
|------|-------------|
| GET_*XXX* | The GET_XML and GET_DDL functions return metadata for a single named object. |
| GET_DEPENDENT_*XXX* | This function returns metadata for a dependent object. |
| GET_GRANTED_*XXX* | This function returns metadata for a granted object. |

| Where *xxx* is: | DDL or XML |
|------|-------------|

**ORACLE**

## Browsing APIs

The browsing APIs are designed for use in SQL queries and ad hoc browsing. These functions allow you to fetch metadata for objects with a single call. They encapsulate calls to OPEN, SET_FILTER, and so on. Which function you use depends on the characteristics of the object type and whether you want XML or DDL.

For some object types, you can use more than one function. You can use GET_XXX to fetch an index by name, or GET_DEPENDENT_XXX to fetch the same index by specifying the table on which it is defined.

GET_XXX returns a single object name.

For GET_DEPENDENT_XXX and GET_GRANTED_XXX, an arbitrary number of granted or dependent objects may match the input criteria. You can specify an object count when fetching these objects.

If you invoke these functions from *i*SQL*Plus, then you should use the SET LONG and SET PAGESIZE commands to retrieve complete, uninterrupted output.

```
SET LONG 2000000
SET PAGESIZE 300
```

# Browsing APIs: Examples

1. **Get the XML representation of `HR.EMPLOYEES`:**

```
SELECT DBMS_METADATA.GET_XML
                        ('TABLE', 'EMPLOYEES', 'HR')
FROM    dual;
```

2. **Fetch the DDL for all object grants on `HR.EMPLOYEES`:**

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
                        ('OBJECT_GRANT', 'EMPLOYEES', 'HR')
FROM    dual;
```

3. **Fetch the DDL for all system grants granted to `HR`:**

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
                    ('SYSTEM_GRANT', 'HR')
FROM    dual;
```

**Browsing APIs: Examples**

1. Results for fetching the XML representation of `HR.EMPLOYEES`:

```
DBMS_METADATA.GET_XML('TABLE','EMPLOYEES','HR')
------------------------------------------------
<?xml version="1.0"?>
<ROWSET>
 <ROW>
  <TABLE_T>
   <VERS_MAJOR>1</VERS_MAJOR>
```

2. Results for fetching the DDL for all object grants on `HR.EMPLOYEES`:

```
DBMS_METADATA.GET_DEPENDENT_DDL
('OBJECT_GRANT','EMPLOYEES','HR')
------------------------------------------------
GRANT SELECT ON "HR"."EMPLOYEES" TO "OE"
GRANT REFERENCES ON "HR"."EMPLOY
```

3. Results for fetching the DDL for all system grants granted to `HR`:

```
DBMS_METADATA.GET_GRANTED_DDL('SYSTEM_GRANT','HR')
------------------------------------------------------
 -
GRANT UNLIMITED TABLESPACE TO "HR"
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-30**

# Browsing APIs: Examples

```
BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(
   DBMS_METADATA.SESSION_TRANSFORM,                     1
   'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
FROM   user_all_tables u                                2
WHERE  u.nested = 'NO'
AND    (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
 DBMS_METADATA.SET_TRANSFORM_PARAM(                     3
   DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT'):
END;
/
```

ORACLE

## Browsing APIs: Examples (continued)

The example in the slide shows how to fetch creation DDL for all "complete" tables in the current schema, filtering out nested tables and overflow segments. The steps shown in the slide are as follows:

1. The SET_TRANSFORM_PARAM function specifies that the storage clauses are not to be returned in the SQL DDL. The SESSION_TRANSFORM function is interpreted to mean "for the current session."

2. Use the GET_DDL function to retrieve DDL on all non-nested and non-IOT (index-organized table) tables.

```
CREATE TABLE "HR"."COUNTRIES"
  ( "COUNTRY_ID" CHAR(2)
      CONSTRAINT "COUNTRY_ID_NN" NOT NULL  ENABLE,
    "COUNTRY_NAME" VARCHAR2(40),
    "REGION_ID" NUMBER,
      CONSTRAINT "COUNTRY_C_ID_PK"
      PRIMARY KEY ("COUNTRY_ID") ENABLE,
      CONSTRAINT "COUNTR_REG_FK" FOREIGN KEY
   ...
```

3. Reset the session-level parameters to their defaults.

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-31**

# Summary

**In this lesson, you should have learned how to:**

- **Explain the execution flow of SQL statements**
- **Create SQL statements dynamically and execute them using either Native Dynamic SQL statements or the `DBMS_SQL` package**
- **Recognize the advantages of using Native Dynamic SQL compared to the `DBMS_SQL` package**
- **Use `DBMS_METADATA` subprograms to programmatically obtain metadata from the data dictionary**

**Summary**

In this lesson, you discovered how to dynamically create any SQL statement and execute it using the Native Dynamic SQL statements. Dynamically executing SQL and PL/SQL code extends the capabilities of PL/SQL beyond query and transactional operations. For earlier releases of the database, you could achieve similar results with the DBMS_SQL package.

The lesson explored some differences and compared using Native Dynamic SQL to the DBMS_SQL package. If you are using Oracle8*i* or later releases, you should use Native Dynamic SQL for new projects.

The lesson also discussed using the DBMS_METADATA package to retrieve metadata from the database dictionary with results presented in XML or creational DDL format. The resulting XML data can be used for re-creating the object.

# Practice 6: Overview

This practice covers the following topics:
- **Creating a package that uses Native Dynamic SQL to create or drop a table and to populate, modify, and delete rows from a table**
- **Creating a package that compiles the PL/SQL code in your schema**
- **Using `DBMS_METADATA` to display the statement to regenerate a PL/SQL subprogram**

**Practice 6: Overview**

In this practice, you write code to perform the following tasks:
- Create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.
- Create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an INVALID status in the USER_OBJECTS table.
- Use DBMS_METADATA to regenerate PL/SQL code for any procedure that you have in your schema.

## Practice 6

1. Create a package called `TABLE_PKG` that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.

   a. Create a package specification with the following procedures:
   ```
   PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2)
   PROCEDURE add_row(table_name VARCHAR2, col_values
   VARCHAR2,
     cols VARCHAR2 := NULL)
   PROCEDURE upd_row(table_name VARCHAR2, set_values
   VARCHAR2,
     conditions VARCHAR2 := NULL)
   PROCEDURE del_row(table_name VARCHAR2,
    conditions VARCHAR2 := NULL);
   PROCEDURE remove(table_name VARCHAR2)
   ```
   Ensure that subprograms manage optional default parameters with `NULL` values.

   b. Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the `remove` procedure that should be written using the `DBMS_SQL` package.

   c. Execute the package `MAKE` procedure to create a table as follows:
   ```
   make('my_contacts', 'id number(4), name varchar2(40)');
   ```

   d. Describe the `MY_CONTACTS` table structure.

   e. Execute the `ADD_ROW` package procedure to add the following rows:
   ```
   add_row('my_contacts','1,''Geoff Gallus''','id, name');
   add_row('my_contacts','2,''Nancy''','id, name');
   add_row('my_contacts','3,''Sunitha Patel''','id,name');
   add_row('my_contacts','4,''Valli Pataballa''','id,name');
   ```

   f. Query the `MY_CONTACTS` table contents.

   g. Execute the `DEL_ROW` package procedure to delete a contact with `ID` value 1.

   h. Execute the `UPD_ROW` procedure with the following row data:
   ```
   upd_row('my_contacts','name=''Nancy Greenberg''','id=2');
   ```

   i. Select the data from the `MY_CONTACTS` table again to view the changes.

   j. Drop the table by using the `remove` procedure and describe the `MY_CONTACTS` table.

2. Create a `COMPILE_PKG` package that compiles the PL/SQL code in your schema.

   a. In the specification, create a package procedure called `MAKE` that accepts the `name` of a PL/SQL program unit to be compiled.

   b. In the body, the `MAKE` procedure should call a private function named `GET_TYPE` to determine the PL/SQL object type from the data dictionary, and return the type name (use `PACKAGE` for a package with a body) if the object exists; otherwise, it should return a `NULL`. If the object exists, `MAKE` dynamically compiles it with the `ALTER` statement.

   c. Use the `COMPILE_PKG.MAKE` procedure to compile the `EMPLOYEE_REPORT` procedure, the `EMP_PKG` package, and a nonexistent object called `EMP_DATA`.

**Oracle Database 10*g*: Develop PL/SQL Program Units   6-34**

**Practice 6 (continued)**

3. Add a procedure to the COMPILE_PKG that uses the DBMS_METADATA to obtain a DDL statement that can regenerate a named PL/SQL subprogram, and writes the DDL statement to a file by using the UTL_FILE package.

    a. In the package specification, create a procedure called REGENERATE that accepts the name of a PL/SQL component to be regenerated. Declare a public VARCHAR2 variable called dir initialized with the directory alias value 'UTL_FILE'. Compile the specification.

    b. In the package body, implement the REGENERATE procedure so that it uses the GET_TYPE function to determine the PL/SQL object type from the supplied name. If the object exists, then obtain the DDL statement used to create the component using the DBMS_METADATA.GET_DDL procedure, which must be provided with the object name in uppercase text.
Save the DDL statement in a file by using the UTL_FILE.PUT procedure. Write the file in the directory path stored in the public variable called dir (from the specification). Construct a file name (in lowercase characters) by concatenating the USER function, an underscore, and the object name with a .sql extension. For example: ora1_myobject.sql. Compile the body.

    c. Execute the COMPILE_PKG.REGENERATE procedure by using the name of the TABLE_PKG created in the first task of this practice.

    d. Use Putty FTP to get the generated file from the server to your local directory. Edit the file to insert a / terminator character at the end of a CREATE statement (if required). Cut and paste the results into the *i*SQL*Plus buffer and execute the statement.

# Design Considerations for PL/SQL Code

**7**

# Objectives

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

- **Use package specifications to create standard constants and exceptions**
- **Write and call local subprograms**
- **Set the** `AUTHID` **directive to control the run-time privileges of a subprogram**
- **Execute subprograms to perform autonomous transactions**
- **Use bulk binding and the** `RETURNING` **clause with DML**
- **Pass parameters by reference using a** `NOCOPY` **hint**
- **Use the** `PARALLEL ENABLE` **hint for optimization**

ORACLE

**Lesson Aim**

In this lesson, you learn to use package specifications to standardize names for constant values and exceptions. You learn how to create subprograms in the `Declaration` section of any PL/SQL block for using locally in the block. The `AUTHID` compiler directive is discussed to show how you can manage run-time privileges of the PL/SQL code, and create independent transactions by using the `AUTONOMOUS TRANSACTION` directive for subprograms.

This lesson also covers some performance considerations that can be applied to PL/SQL applications, such as bulk binding operations with a single SQL statement, the `RETURNING` clause, and the `NOCOPY` and `PARALLEL ENABLE` hints.

# Standardizing Constants and Exceptions

**Constants and exceptions are typically implemented using a bodiless package (that is, in a package specification).**

- **Standardizing helps to:**
  - **Develop programs that are consistent**
  - **Promote a higher degree of code reuse**
  - **Ease code maintenance**
  - **Implement company standards across entire applications**
- **Start with standardization of:**
  - **Exception names**
  - **Constant definitions**

ORACLE

## Standardizing Constants and Exceptions

When several developers are writing their own exception handlers in an application, there could be inconsistencies in the handling of error situations. Unless certain standards are adhered to, the situation can become confusing because of the different approaches followed in handling the same error or because of the display of conflicting error messages that confuse users. To overcome these, you can:

- Implement company standards that use a consistent approach to error handling across the entire application
- Create predefined, generic exception handlers that produce consistency in the application
- Write and call programs that produce consistent error messages

All good programming environments promote naming and coding standards. In PL/SQL, a good place to start implementing naming and coding standards is with commonly used constants and exceptions that occur in the application domain.

The PL/SQL package specification construct is an excellent component to support standardization because all identifiers declared in the package specification are public. They are visible to the subprograms that are developed by the owner of the package and all code with EXECUTE rights to the package specification.

# Standardizing Exceptions

**Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.**

```
CREATE OR REPLACE PACKAGE error_pkg IS
  fk_err        EXCEPTION;
  seq_nbr_err   EXCEPTION;
  PRAGMA EXCEPTION_INIT (fk_err, -2292);
  PRAGMA EXCEPTION_INIT (seq_nbr_err, -2277);
  ...
END error_pkg;
/
```

**Standardizing Exceptions**

In the example in the slide, the error_pkg package is a standardized exception package. It declares a set of programmer-defined exception identifiers. Because many of the Oracle database predefined exceptions do not have identifying names, the example package shown in the slide uses the PRAGMA EXCEPTION_INIT directive to associate selected exception names with an Oracle database error number. This enables you to refer to any of the exceptions in a standard way in your applications, as in the following example:

```
BEGIN
  DELETE FROM departments
WHERE department_id = deptno;
  ...
EXCEPTION
  WHEN error_pkg.fk_err THEN
  ...
  WHEN OTHERS THEN
  ...
END;
/
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   7-4**

# Standardizing Exception Handling

**Consider writing a subprogram for common exception handling to:**

- **Display errors based on `SQLCODE` and `SQLERRM` values for exceptions**
- **Track run-time errors easily by using parameters in your code to identify:**
  - **The procedure in which the error occurred**
  - **The location (line number) of the error**
  - **`RAISE_APPLICATION_ERROR` using stack trace capabilities, with the third argument set to `TRUE`**

ORACLE

### Standardizing Exception Handling

Standardized exception handling can be implemented either as a stand-alone subprogram or a subprogram added to the package that defines the standard exceptions. Consider creating a package with:

- Every named exception that is to be used in the application
- All unnamed, programmer-defined exceptions that are used in the application. These are error numbers –20000 through –20999.
- A program to call `RAISE_APPLICATION_ERROR` based on package exceptions
- A program to display an error based on the values of `SQLCODE` and `SQLERRM`
- Additional objects, such as error log tables, and programs to access the tables

A common practice is to use parameters that identify the name of the procedure and the location in which the error has occurred. This enables you to keep track of run-time errors more easily. An alternative is to use the `RAISE_APPLICATION_ERROR` built-in procedure to keep a stack trace of exceptions that can be used to track the call sequence leading to the error. To do this, set the third optional argument to `TRUE`. For example:

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

This is meaningful when more than one exception is raised in this manner.

# Standardizing Constants

**For programs that use local variables whose values should not change:**

- **Convert the variables to constants to reduce maintenance and debugging**
- **Create one central package specification and place all constants in it**

```
CREATE OR REPLACE PACKAGE constant_pkg IS
  c_order_received CONSTANT VARCHAR(2) := 'OR';
  c_order_shipped  CONSTANT VARCHAR(2) := 'OS';
  c_min_sal        CONSTANT NUMBER(3)  := 900;
  ...
END constant_pkg;
```

ORACLE

**Standardizing Constants**

By definition, a variable's value changes, whereas a constant's value cannot be changed. If you have programs that use local variables whose values should not and do not change, then convert the variables to constants. This can help with the maintenance and debugging of your code.

Consider creating a single shared package with all your constants in it. This makes maintenance and change of the constants much easier. This procedure or package can be loaded on system startup for better performance.

The example in the slide shows the constant_pkg package containing a few constants. Refer to any of the package constants in your application as required. Here is an example:

```
BEGIN
  UPDATE employees
     SET salary  = salary + 200
  WHERE salary <= constant_pkg.c_min_sal;
...
END;
/
```

**Oracle Database 10*g*: Develop PL/SQL Program Units   7-6**

# Local Subprograms

- **A local subprogram is a `PROCEDURE` or `FUNCTION` defined in the declarative section.**

```
CREATE PROCEDURE employee_sal(id NUMBER) IS
    emp employees%ROWTYPE;
    FUNCTION tax(salary VARCHAR2) RETURN NUMBER IS
    BEGIN
      RETURN salary * 0.825;
    END tax;
BEGIN
    SELECT * INTO emp
    FROM EMPLOYEES WHERE employee_id = id;
    DBMS_OUTPUT.PUT_LINE('Tax: '|| tax(emp.salary));
END;
```

- **The local subprogram must be defined at the end of the declarative section.**

ORACLE

### Local Subprograms

Local subprograms can drive top-down design. They reduce the size of a module by removing redundant code. This is one of the main reasons for creating a local subprogram. If a module needs the same routine several times, but only this module needs the routine, then define it as a local subprogram.

You can define a named PL/SQL block in the declarative section of any PL/SQL program, procedure, function, or anonymous block provided that it is declared at the end of the Declaration section. Local subprograms have the following characteristics:
- They are only accessible to the block in which they are defined.
- They are compiled as part of their enclosing blocks.

The benefits of local subprograms are:
- Reduction of repetitive code
- Improved code readability and ease of maintenance
- Less administration because there is one program to maintain instead of two

The concept is simple. The example shown in the slide illustrates this with a basic example of an income tax calculation of an employee's salary.

# Definer's Rights Versus Invoker's Rights

**Definer's rights:**

- **Used prior to Oracle8*i***
- **Programs execute with the privileges of the creating user.**
- **User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.**

**Invoker's rights:**

- **Introduced in Oracle8*i***
- **Programs execute with the privileges of the calling user.**
- **User requires privileges on the underlying objects that the procedure accesses.**

ORACLE

## Definer's Rights Versus Invoker's Rights

### Definer's Rights Model

Prior to Oracle8*i*, all programs executed with the privileges of the user who created the subprogram. This is known as the definer's rights model, which:
- Allows a caller of the program the privilege to execute the procedure, but no privileges on the underlying objects that the procedure accesses
- Requires the owner to have all the necessary object privileges for the objects that the procedure references

For example, if user Scott creates a PL/SQL subprogram `get_employees` that is subsequently invoked by Sarah, then the `get_employees` procedure runs with the privileges of the definer Scott.

### Invoker's Rights Model

In the invoker's rights model, which was introduced in Oracle8*i*, programs are executed with the privileges of the calling user. A user of a procedure running with invoker's rights requires privileges on the underlying objects that the procedure references.

For example, if Scott's PL/SQL subprogram `get_employees` is invoked by Sarah, then the `get_employees` procedure runs with the privileges of the invoker Sarah.

# Specifying Invoker's Rights

**Set** `AUTHID` **to** `CURRENT_USER`:

```
CREATE OR REPLACE PROCEDURE add_dept(
   id NUMBER, name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
   INSERT INTO departments
   VALUES (id,name,NULL,NULL);
END;
```

**When used with stand-alone functions, procedures, or packages:**

- **Names used in queries, DML, Native Dynamic SQL, and `DBMS_SQL` package are resolved in the invoker's schema**
- **Calls to other packages, functions, and procedures are resolved in the definer's schema**

## Specifying Invoker's Rights

### Setting Invoker's Rights

The following is the syntax for setting invoker's rights for different PL/SQL subprogram constructs:

```
CREATE FUNCTION name RETURN type AUTHID CURRENT_USER
IS...
CREATE PROCEDURE name AUTHID CURRENT_USER IS…
CREATE PACKAGE name AUTHID CURRENT_USER IS…
CREATE TYPE name AUTHID CURRENT_USER IS OBJECT…
```

In these statements, set `AUTHID` to `DEFINER`, or do not use it for default behavior.

### Name Resolution

For a definer's rights procedure, all external references are resolved in the definer's schema. For an invoker's rights procedure, the resolution of external references depends on the kind of statement in which they appear:

- Names used in queries, data manipulation language (DML) statements, dynamic SQL, and `DBMS_SQL` are resolved in the invoker's schema.
- All other statements, such as calls to packages, functions, and procedures, are resolved in the definer's schema.

# Autonomous Transactions

- **Are independent transactions started by another main transaction.**
- **Are specified with `PRAGMA AUTONOMOUS_TRANSACTION`**

```
PROCEDURE proc1 IS
  emp_id  NUMBER;
BEGIN
  emp_id := 1234;
  COMMIT;
① INSERT ...
② proc2;
  DELETE ...
⑦ COMMIT;
END proc1;
```

```
PROCEDURE proc2 IS
 PRAGMA
  AUTONOMOUS_TRANSACTION;
  dept_id  NUMBER := 90;
③ BEGIN
④  UPDATE ...
   INSERT ...
⑤  COMMIT;   -- Required
⑥ END proc2;
```
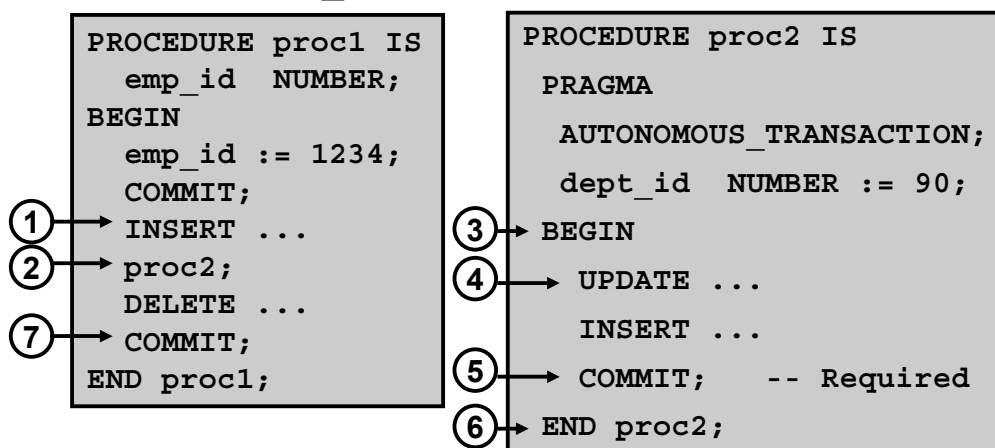
ORACLE

## Autonomous Transactions

A transaction is a series of statements that does a logical unit of work that completes or fails as an integrated unit. Often, one transaction starts another that may need to operate outside the scope of the transaction that started it. That is, in an existing transaction, a required independent transaction may need to commit or roll back changes without affecting the outcome of the starting transaction. For example, in a stock purchase transaction, the customer's information must be committed regardless of whether the overall stock purchase completes. Or, while running that same transaction, you want to log messages to a table even if the overall transaction rolls back.

Since Oracle8*i*, the autonomous transactions were added to make it possible to create an independent transaction. An autonomous transaction (AT) is an independent transaction started by another main transaction (MT). The slide depicts the behavior of an AT:

1. The main transaction begins.
2. A `proc2` procedure is called to start the autonomous transaction.
3. The main transaction is suspended.
4. The autonomous transactional operation begins.
5. The autonomous transaction ends with a commit or roll back operation.
6. The main transaction is resumed.
7. The main transaction ends.

# Features of Autonomous Transactions

**Autonomous transactions:**

- **Are independent of the main transaction**
- **Suspend the calling transaction until it is completed**
- **Are not nested transactions**
- **Do not roll back if the main transaction rolls back**
- **Enable the changes to become visible to other transactions upon a commit**
- **Are demarcated (started and ended) by individual subprograms and not by nested or anonymous PL/SQL blocks**

ORACLE

**Features of Autonomous Transactions**

Autonomous transactions exhibit the following features:

- Although called within a transaction, autonomous transactions are independent of that transaction. That is, they are not nested transactions.
- If the main transaction rolls back, autonomous transactions do not.
- Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits.
- With their stack-like functionality, only the "top" transaction is accessible at any given time. After completion, the autonomous transaction is popped, and the calling transaction is resumed.
- There are no limits other than resource limits on how many autonomous transactions can be recursively called.
- Autonomous transactions must be explicitly committed or rolled back; otherwise, an error is returned when attempting to return from the autonomous block.
- You cannot use PRAGMA to mark all subprograms in a package as autonomous. Only individual routines can be marked autonomous.
- You cannot mark a nested or anonymous PL/SQL block as autonomous.

# Using Autonomous Transactions

**Example:**

```
PROCEDURE bank_trans(cardnbr NUMBER,loc NUMBER) IS
BEGIN
  log_usage (cardnbr, loc);
  INSERT INTO txn VALUES (9001, 1000,...);
END bank_trans;
```

```
PROCEDURE log_usage (card_id NUMBER, loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage
  VALUES (card_id, loc);
  COMMIT;
END log_usage;
```

## Using Autonomous Transactions

To define autonomous transactions, you use PRAGMA AUTONOMOUS_TRANSACTION.
PRAGMA instructs the PL/SQL compiler to mark a routine as autonomous (independent).
In this context, the term "routine" includes top-level (not nested) anonymous PL/SQL
blocks; local, stand-alone, and packaged functions and procedures; methods of a SQL
object type; and database triggers. You can code PRAGMA anywhere in the declarative
section of a routine. However, for readability, it is best placed at the top of the
Declaration section.

In the example in the slide, you track where the bankcard is used, regardless of whether
the transaction is successful. The following are the benefits of autonomous transactions:
- After starting, an autonomous transaction is fully independent. It shares no locks,
  resources, or commit dependencies with the main transaction, so you can log events,
  increment retry counters, and so on even if the main transaction rolls back.
- More important, autonomous transactions help you build modular, reusable software
  components. For example, stored procedures can start and finish autonomous
  transactions on their own. A calling application need not know about a procedure's
  autonomous operations, and the procedure need not know about the application's
  transaction context. That makes autonomous transactions less error-prone than
  regular transactions and easier to use.

## RETURNING Clause

**The RETURNING clause:**

- **Improves performance by returning column values with INSERT, UPDATE, and DELETE statements**
- **Eliminates the need for a SELECT statement**

```
CREATE PROCEDURE update_salary(emp_id NUMBER) IS
  name     employees.last_name%TYPE;
  new_sal employees.salary%TYPE;
BEGIN
  UPDATE employees
    SET salary = salary * 1.1
  WHERE employee_id = emp_id
  RETURNING last_name, salary INTO name, new_sal;
END update_salary;
/
```

### RETURNING Clause

Often, applications need information about the row affected by a SQL operation—for example, to generate a report or to take a subsequent action. The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to SELECT the row after an INSERT or UPDATE, or before a DELETE. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

The example in the slide shows how to update the salary of an employee and, at the same time, retrieve the employee's last name and new salary into a local PL/SQL variable.

# Bulk Binding

**Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times**

| PL/SQL run-time engine | | SQL engine |
|---|---|---|
| **PL/SQL block**<br><br>`FORALL j IN 1..1000`<br>` INSERT (id,`<br>`        dates)`<br>` VALUES (ids(j),`<br>`        dates(j));`<br>`...` | **Procedural statement executor** | **SQL statement executor** |
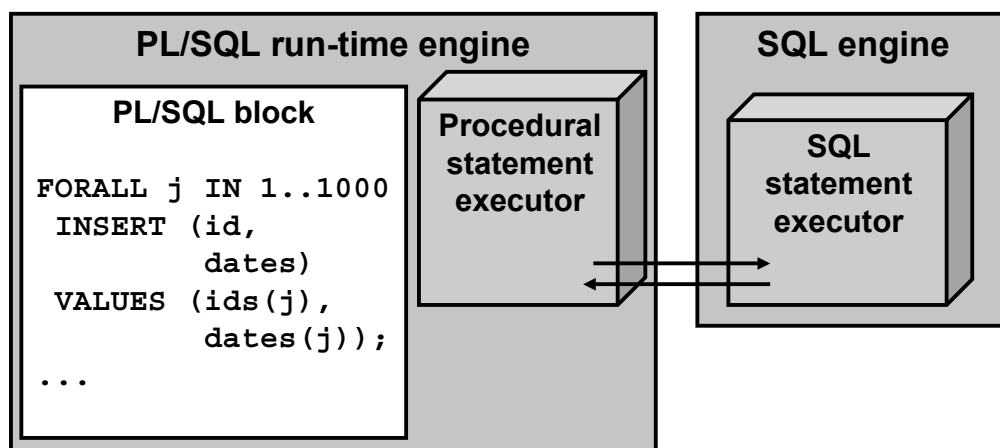
### Bulk Binding

The Oracle server uses two engines to run PL/SQL blocks and subprograms:
- The PL/SQL run-time engine, which runs procedural statements but passes the SQL statements to the SQL engine
- The SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine

During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty for excessive amounts of SQL processing. This is typical of applications that have a SQL statement in a loop that uses indexed collection elements values. Collections include nested tables, varying arrays, index-by tables, and host arrays.

Performance can be substantially improved by minimizing the number of context switches through the use of bulk binding. Bulk binding causes an entire collection to be bound in one call, a context switch, to the SQL engine. That is, a bulk bind process passes the entire collection of values back and forth between the two engines in a single context switch, compared with incurring a context switch for each collection element in an iteration of a loop. The more rows affected by a SQL statement, the greater is the performance gain with bulk binding.

**Oracle Database 10*g*: Develop PL/SQL Program Units   7-14**

# Using Bulk Binding

**Keywords to support bulk binding:**

- **The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine.**

```
FORALL index IN lower_bound .. upper_bound
   [SAVE EXCEPTIONS]
   sql_statement;
```

- **The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections before returning them to the PL/SQL engine.**

```
... BULK COLLECT INTO
       collection_name[,collection_name] ...
```

ORACLE

**Using Bulk Binding**

Use bulk binds to improve the performance of:
- DML statements that reference collection elements
- `SELECT` statements that reference collection elements
- Cursor `FOR` loops that reference collections and the `RETURNING INTO` clause

**Keywords to Support Bulk Binding**

The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine. Although the `FORALL` statement contains an iteration scheme, it is not a `FOR` loop.

The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections, before returning them to the PL/SQL engine. This allows you to bind locations into which SQL can return the retrieved values in bulk. Thus, you can use these keywords in the `SELECT INTO`, `FETCH INTO`, and `RETURNING INTO` clauses.

The `SAVE EXCEPTIONS` keyword is optional. However, if some of the DML operations succeed and some fail, you will want to track or report on those that fail. Using the `SAVE EXCEPTIONS` keyword causes failed operations to be stored in a cursor attribute called `%BULK_EXCEPTIONS`, which is a collection of records indicating the bulk DML iteration number and corresponding error code.

**Oracle Database 10*g*: Develop PL/SQL Program Units   7-15**

# Bulk Binding FORALL: Example

```
CREATE PROCEDURE raise_salary(percent NUMBER) IS
  TYPE numlist IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  id  numlist;
BEGIN
 id(1) := 100; id(2) := 102;
 id(3) := 104; id(4) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN id.FIRST .. id.LAST
    UPDATE employees
      SET salary = (1 + percent/100) * salary
      WHERE manager_id = id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

ORACLE

**Bulk Binding FORALL: Example**

In the example in the slide, the PL/SQL block increases the salary for employees whose manager's ID is 100, 102, 104, or 110. It uses the FORALL keyword to bulk bind the collection. Without bulk binding, the PL/SQL block would have sent a SQL statement to the SQL engine for each employee that is updated. If there are many employees to update, then the large number of context switches between the PL/SQL engine and the SQL engine can affect performance. However, the FORALL keyword bulk binds the collection to improve performance.

**Note:** A looping construct is no longer required when using this feature.

To manage exceptions and have the bulk bind complete despite errors, add the SAVE EXCEPTIONS keyword to your FORALL statement after the bounds, before the DML statement. Use the new cursor attribute %BULK_EXCEPTIONS, which stores a collection of records with two fields:

- %BULK_EXCEPTIONS(i).ERROR_INDEX holds the "iteration" of the statement during which the exception was raised.
- %BULK_EXCEPTIONS(i).ERROR_CODE holds the corresponding error code.

Values stored in %BULK_EXCEPTIONS refer to the most recently executed FORALL statement. Its subscripts range from 1 to %BULK_EXCEPTIONS.COUNT.

**Oracle Database 10*g*: Develop PL/SQL Program Units   7-16**

## Bulk Binding FORALL: Example (continued)

### An Additional Cursor Attribute for DML Operations

Another cursor attribute added to support bulk operations is %BULK_ROWCOUNT. The %BULK_ROWCOUNT attribute is a composite structure designed for use with the FORALL statement. This attribute acts like an index-by table. Its $i$th element stores the number of rows processed by the $i$th execution of an UPDATE or DELETE statement. If the $i$th execution affects no rows, then %BULK_ROWCOUNT(i) returns zero.

Here is an example:

```
CREATE TABLE num_table (n NUMBER);
DECLARE
  TYPE NumList IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  nums NumList;
BEGIN
  nums(1) := 1;
  nums(2) := 3;
  nums(3) := 5;
  nums(4) := 7;
  nums(5) := 11;
  FORALL i IN nums.FIRST .. nums.LAST
    INSERT INTO num_table (n) VALUES (nums(i));
  FOR i IN nums.FIRST .. nums.LAST
  LOOP
    dbms_output.put_line('Inserted ' ||
      SQL%BULK_ROWCOUNT(i) || ' row(s)'
      || ' on iteration ' || i);
  END LOOP;
END;
/
DROP TABLE num_table;
```

The following results are produced by this example:

```
Inserted 1 row(s) on iteration 1
Inserted 1 row(s) on iteration 2
Inserted 1 row(s) on iteration 3
Inserted 1 row(s) on iteration 4
Inserted 1 row(s) on iteration 5

PL/SQL procedure successfully completed.
```

# Using BULK COLLECT INTO with Queries

**The SELECT statement has been enhanced to support the BULK COLLECT INTO syntax.**

**Example:**

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  TYPE dept_tabtype IS
    TABLE OF departments%ROWTYPE;
  depts dept_tabtype;
BEGIN
  SELECT * BULK COLLECT INTO depts
  FROM departments
  WHERE location_id = loc;
  FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_id
      ||' '||depts(i).department_name);
  END LOOP;
END;
```

**Using BULK COLLECT INTO with Queries**

In Oracle Database 10*g*, when using a SELECT statement in PL/SQL, you can use the bulk collection syntax shown in the example in the slide. Thus, you can quickly obtain a set of rows without using a cursor mechanism.

The example reads all the department rows for a specified region into a PL/SQL table, whose contents are displayed with the FOR loop that follows the SELECT statement.

# Using BULK COLLECT INTO with Cursors

**The FETCH statement has been enhanced to support the BULK COLLECT INTO syntax.**

**Example:**

```
CREATE PROCEDURE get_departments(loc NUMBER) IS
  CURSOR dept_csr IS SELECT * FROM departments
                     WHERE location_id = loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts;
  CLOSE dept_csr;
 FOR I IN 1 .. depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(depts(i).department_id
     ||' '||depts(i).department_name);
  END LOOP;
END;
```

## Using BULK COLLECT INTO with Cursors

In Oracle Database 10*g*, when using cursors in PL/SQL, you can use a form of the FETCH statement that supports the bulk collection syntax shown in the example in the slide.

This example shows how BULK COLLECT INTO can be used with cursors.

You can also add a LIMIT clause to control the number of rows fetched in each operation. The code example in the slide could be modified as follows:

```
CREATE PROCEDURE get_departments(loc NUMBER,
  nrows NUMBER) IS
  CURSOR dept_csr IS SELECT * FROM departments
                     WHERE location_id = loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts LIMIT nrows;
  CLOSE dept_csr;
  DBMS_OUTPUT.PUT_LINE(depts.COUNT||' rows read');
END;
```

# Using BULK COLLECT INTO with a RETURNING Clause

**Example:**

```
CREATE PROCEDURE raise_salary(rate NUMBER) IS
   TYPE emplist IS TABLE OF NUMBER;
   TYPE numlist IS TABLE OF employees.salary%TYPE
     INDEX BY BINARY_INTEGER;
   emp_ids  emplist := emplist(100,101,102,104);
   new_sals numlist;
BEGIN
  FORALL i IN emp_ids.FIRST .. emp_ids.LAST
    UPDATE employees
      SET commission_pct = rate * salary
    WHERE employee_id = emp_ids(i)
    RETURNING salary BULK COLLECT INTO new_sals;
  FOR i IN 1 .. new_sals.COUNT LOOP ...
END;
```

ORACLE

**Using BULK COLLECT INTO with a RETURNING Clause**

Bulk binds can be used to improve the performance of FOR loops that reference collections and return DML. If you have, or plan to have, PL/SQL code that does this, then you can use the FORALL keyword along with the RETURNING and BULK COLLECT INTO keywords to improve performance.

In the example shown in the slide, the salary information is retrieved from the EMPLOYEES table and collected into the new_sals array. The new_sals collection is returned in bulk to the PL/SQL engine.

The example in the slide shows an incomplete FOR loop that is used to iterate through the new salary data received from the UPDATE operation and then process the results.

# Using the NOCOPY Hint

**The NOCOPY hint:**

- **Is a request to the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value**
- **Enhances performance by reducing overhead when passing parameters**

```
DECLARE
  TYPE emptabtype IS TABLE OF employees%ROWTYPE;
  emp_tab emptabtype;
  PROCEDURE populate(tab IN OUT NOCOPY emptabtype)
  IS BEGIN ... END;
BEGIN
  populate(emp_tab);
END;
/
```

## Using the NOCOPY Hint

Note that PL/SQL subprograms support three parameter-passing modes: IN, OUT, and IN OUT. By default:

- The IN parameter is passed by reference. A pointer to the IN actual parameter is passed to the corresponding formal parameter. So both parameters reference the same memory location, which holds the value of the actual parameter.
- The OUT and IN OUT parameters are passed by value. The value of the OUT or IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

Copying parameters that represent large data structures (such as collections, records, and instances of object types) with OUT and IN OUT parameters slows down execution and uses up memory. To prevent this overhead, you can specify the NOCOPY hint, which enables the PL/SQL compiler to pass OUT and IN OUT parameters by reference.

The slide shows an example of declaring an IN OUT parameter with the NOCOPY hint.

# Effects of the NOCOPY Hint

- **If the subprogram exits with an exception that is not handled:**
  - **You cannot rely on the values of the actual parameters passed to a NOCOPY parameter**
  - **Any incomplete modifications are not "rolled back"**
- **The remote procedure call (RPC) protocol enables you to pass parameters only by value.**

### Effects of the NOCOPY Hint

As a trade-off for better performance, the NOCOPY hint enables you to trade well-defined exception semantics for better performance. Its use affects exception handling in the following ways:
- Because NOCOPY is a hint, not a directive, the compiler can pass NOCOPY parameters to a subprogram by value or by reference. So, if the subprogram exits with an unhandled exception, you cannot rely on the values of the NOCOPY actual parameters.
- By default, if a subprogram exits with an unhandled exception, the values assigned to its OUT and IN OUT formal parameters are not copied to the corresponding actual parameters, and changes appear to roll back. However, when you specify NOCOPY, assignments to the formal parameters immediately affect the actual parameters as well. So, if the subprogram exits with an unhandled exception, the (possibly unfinished) changes are not "rolled back."
- Currently, the RPC protocol enables you to pass parameters only by value. So exception semantics can change without notification when you partition applications. For example, if you move a local procedure with NOCOPY parameters to a remote site, those parameters are no longer passed by reference.

# NOCOPY Hint Can Be Ignored

**The NOCOPY hint has no effect if:**

- **The actual parameter:**
    - **Is an element of an index-by table**
    - **Is constrained (for example, by scale or NOT NULL)**
    - **And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ**
    - **Requires an implicit data type conversion**
- **The subprogram is involved in an external or remote procedure call**

ORACLE

**NOCOPY Hint Can Be Ignored**

In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method (with no error generated):

- The actual parameter is an element of an index-by table. This restriction does not apply to entire index-by tables.
- The actual parameter is constrained (by scale or NOT NULL). This restriction does not extend to constrained elements or attributes. Also, it does not apply to size-constrained character strings.
- The actual and formal parameters are records; one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records; the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit data type conversion.
- The subprogram is involved in an external or remote procedure call.

# PARALLEL_ENABLE Hint

**The `PARALLEL_ENABLE` hint:**

• **Can be used in functions as an optimization hint**

```
CREATE OR REPLACE FUNCTION f2 (p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p1 * 2;
END f2;
```

• **Indicates that a function can be used in a parallelized query or parallelized DML statement**

ORACLE

**`PARALLEL_ENABLE` Hint**

The `PARALLEL_ENABLE` keyword can be used in the syntax for declaring a function. It is an optimization hint that indicates that the function can be used in a parallelized query or parallelized DML statement. Oracle's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement that is run in parallel can have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process.

For DML statements, prior to Oracle8*i*, the parallelization optimization looked to see whether a function was noted as having all four of `RNDS`, `WNDS`, `RNPS`, and `WNPS` specified in a `PRAGMA RESTRICT_REFERENCES` declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a `CREATE FUNCTION` statement had their code implicitly examined to determine whether they were actually pure enough; parallelized execution might occur even though a `PRAGMA` cannot be specified on these functions.

The `PARALLEL_ENABLE` keyword is placed after the return value type in the declaration of the function, as shown in the example in the slide.

**Note:** The function should not use session state, such as package variables, because those variables may not be shared among the parallel execution servers.

**Oracle Database 10*g*: Develop PL/SQL Program Units   7-24**

# Summary

**In this lesson, you should have learned how to:**
- **Create standardized constants and exceptions using packages**
- **Develop and invoke local subprograms**
- **Control the run-time privileges of a subprogram by setting the `AUTHID` directive**
- **Execute autonomous transactions**
- **Use the `RETURNING` clause with DML statements, and bulk binding collections with the `FORALL` and `BULK COLLECT INTO` clauses**
- **Pass parameters by reference using a `NOCOPY` hint**
- **Enable optimization with `PARALLEL ENABLE` hints**

ORACLE

**Summary**

The lesson provides insights into managing your PL/SQL code by defining constants and exceptions in a package specification. This enables a high degree of reuse and standardization of code.

Local subprograms can be used to simplify and modularize a block of code where the subprogram functionality is repeatedly used in the local block.

The run-time security privileges of a subprogram can be altered by using definer's or invoker's rights.

Autonomous transactions can be executed without affecting an existing main transaction.

You should understand how to obtain performance gains by using the `NOCOPY` hint, bulk binding and the `RETURNING` clauses in SQL statements, and the `PARALLEL_ENABLE` hint for optimization of functions.

**Practice 7: Overview**

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table.

You add an `add_employee` procedure to the package that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the `add_employee` procedure is called, whether it successfully adds a record or not.

Finally, you make the package use `AUTHID` of `CURRENT_USER` and test the behavior with any other student. You test the code first with definer's rights and then with invoker's rights.

## Practice 7

1. Update EMP_PKG with a new procedure to query employees in a specified department.
    a. In the specification, declare a get_employees procedure, with its parameter called dept_id based on the employees.department_id column type. Define an index-by PL/SQL type as a TABLE OF EMPLOYEES%ROWTYPE.
    b. In the body of the package, define a private variable called emp_table based on the type defined in the specification to hold employee records. Implement the get_employees procedure to bulk fetch the data into the table.
    c. Create a new procedure in the specification and body, called show_employees, that does not take arguments and displays the contents of the private PL/SQL table variable (if any data exists).
    **Hint:** Use the print_employee procedure.
    d. Invoke the emp_pkg.get_employees procedure for department 30, and then invoke emp_pkg.show_employees. Repeat this for department 60.
2. Your manager wants to keep a log whenever the add_employee procedure in the package is invoked to insert a new employee into the EMPLOYEES table.
    a. First, load and execute the E:\labs\PLPU\labs\lab_07_02_a.sql script to create a log table called LOG_NEWEMP, and a sequence called log_newemp_seq.
    b. In the package body, modify the add_employee procedure, which performs the actual INSERT operation, to have a local procedure called audit_newemp. The audit_newemp procedure must use an autonomous transaction to insert a log record into the LOG_NEWEMP table. Store the USER, the current time, and the new employee name in the log table row. Use log_newemp_seq to set the entry_id column.
    **Note:** Remember to perform a COMMIT operation in a procedure with an autonomous transaction.
    c. Modify the add_employee procedure to invoke audit_emp before it performs the insert operation.
    d. Invoke the add_employee procedure for these new employees: Max Smart in department 20 and Clark Kent in department 10. What happens?
    e. Query the two EMPLOYEES records added, and the records in LOG_NEWEMP table. How many log records are present?
    f. Execute a ROLLBACK statement to undo the insert operations that have not been committed. Use the same queries from Exercise 2e: the first to check whether the employee rows for Smart and Kent have been removed, and the second to check the log records in the LOG_NEWEMP table. How many log records are present? Why?

**Practice 7 (continued)**

**If you have time, complete the following exercise:**

3.  Modify the EMP_PKG package to use AUTHID of CURRENT_USER and test the behavior with any other student.
    **Note:** Verify whether the LOG_NEWEMP table exists from Exercise 2 in this practice.

    a.  Grant the EXECUTE privilege on your EMP_PKG package to another student.
    b.  Ask the other student to invoke your add_employee procedure to insert employee Jaco Pastorius in department 10. Remember to prefix the package name with the owner of the package. The call should operate with definer's rights.
    c.  Now, execute a query of the employees in department 10. In which user's employee table did the new record get inserted?
    d.  Modify your package EMP_PKG specification to use an AUTHID CURRENT_USER. Compile the body of EMP_PKG.
    e.  Ask the same student to execute the add_employee procedure again, to add employee Joe Zawinal in department 10.
    f.  Query your employees in department 10. In which table was the new employee added?
    g.  Write a query to display the records added in the LOG_NEWEMP tables. Ask the other student to query his or her own copy of the table.

# Managing Dependencies

8

# Objectives

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

- **Track procedural dependencies**
- **Predict the effect of changing a database object on stored procedures and functions**
- **Manage procedural dependencies**

ORACLE

**Lesson Aim**

This lesson introduces you to object dependencies and implicit and explicit recompilation of invalid objects.

# Understanding Dependencies

**Dependent objects**

> **Database trigger**
>
> **Function**
>
> **Procedure**
>
> **Package body**
>
> **Package specification**
>
> **User-defined object and collection types**
>
> **View**

**Referenced objects**

> **Function**
>
> **Package specification**
>
> **Procedure**
>
> **Sequence**
>
> **Synonym**
>
> **Table**
>
> **User-defined object and collection types**
>
> **View**
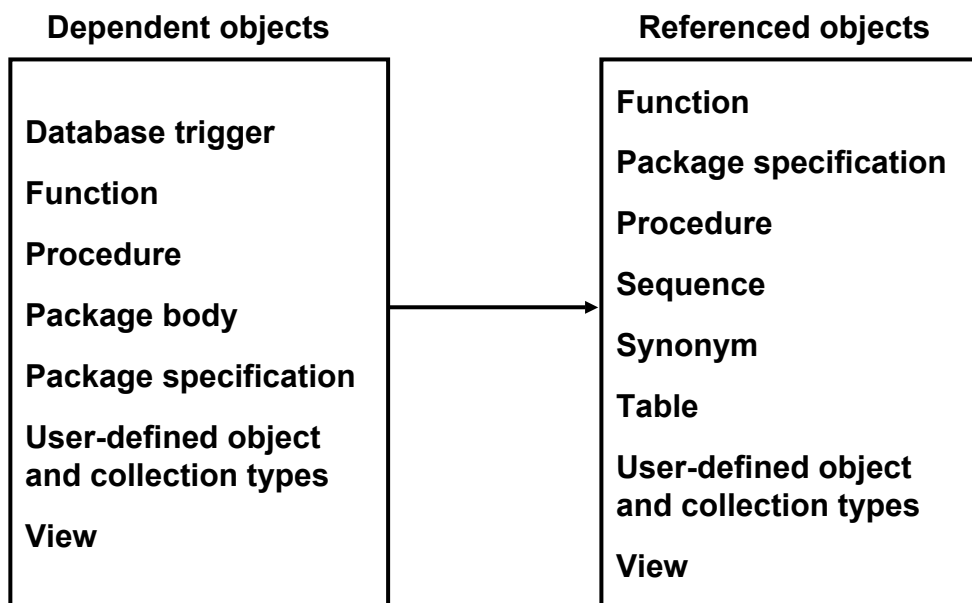
ORACLE

## Dependent and Referenced Objects

Some objects reference other objects as part of their definitions. For example, a stored procedure could contain a SELECT statement that selects columns from a table. For this reason, the stored procedure is called a dependent object, whereas the table is called a referenced object.

### Dependency Issues

If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, the procedure may or may not continue to work without error.

The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary, and you can view the status in the USER_OBJECTS data dictionary view.

| Status | Significance |
|--------|--------------|
| VALID | The schema object has been compiled and can be immediately used when referenced. |
| INVALID | The schema object must be compiled before it can be used. |

**Oracle Database 10*g*: Develop PL/SQL Program Units   8-3**

# Dependencies



**Procedure**

```
xxxxxxxxxxxxx
vvvvvvvvvvvvv
xxxxxxxxxxxxx
vvvvvvvvvvvvv
xxxxxxxxxxxxx
vvvvvvvvvvvvv
xxxxxxxxxxxxx
vvvvvvvvvvvvv
xxxxxxxxxxxxx
vvvvvvvvvvvvv
```

**View or procedure**

**Table**

**Direct dependency**

**Direct dependency**

**Referenced**
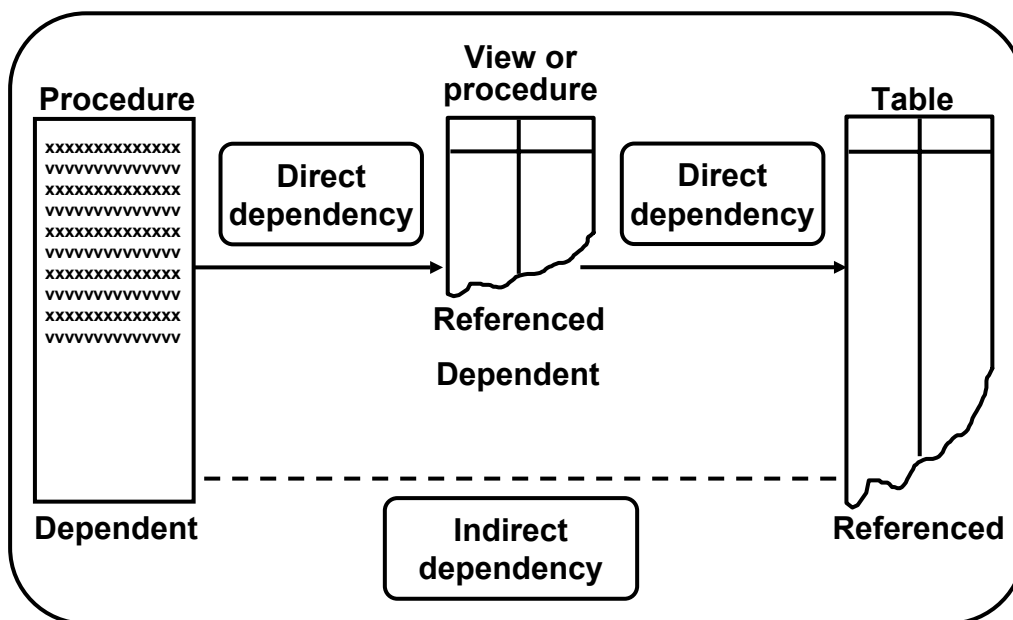
**Dependent**

**Dependent**

**Indirect dependency**

**Referenced**

ORACLE

## Dependent and Referenced Objects (continued)

A procedure or function can directly or indirectly (through an intermediate view, procedure, function, or packaged procedure or function) reference the following objects:

- Tables
- Views
- Sequences
- Procedures
- Functions
- Packaged procedures or functions

# Local Dependencies



| Procedure | Procedure | View | Table |
|-----------|-----------|------|-------|
| xxxxxxxxxxxxxx<br>vvvvvvvvvvvvvv<br>xxxxxxxxxxxxxx<br>vvvvvvvvvvvvvv<br>xxxxxxxxxxxxxx<br>vvvvvvvvvvvvvv<br>vvvvvvvvvvvvvv | vvvvvvvvvvvvvv<br>xxxxxxxxxxxxxx<br>vvvvvvvvvvvvvv<br>xxxxxxxxxxxxxx<br>vvvvvvvvvvvvvv<br>xxxxxxxxxxxxxx<br>vvvvvvvvvvvvvv | | |

**Local references**
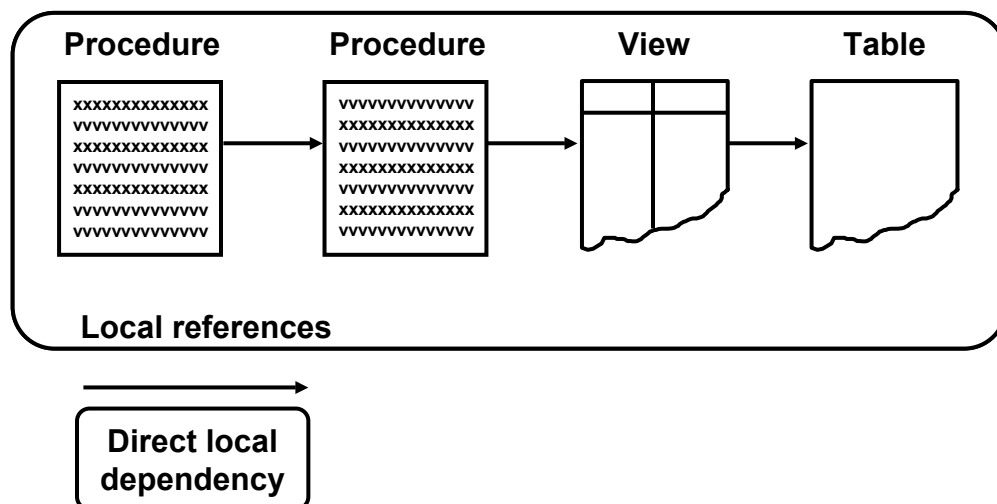
→ **Direct local dependency**

ORACLE

## Managing Local Dependencies

In the case of local dependencies, the objects are on the same node in the same database. The Oracle server automatically manages all local dependencies, using the database's internal "depends-on" table. When a referenced object is modified, the dependent objects are invalidated. The next time an invalidated object is called, the Oracle server automatically recompiles it.

## Local Dependencies

**Procedure**　**Procedure**　**View**　**Table**

```
xxxxxxxxxxxxxx          vvvvvvvvvvvvvv
vvvvvvvvvvvvvv          xxxxxxxxxxxxxx
xxxxxxxxxxxxxx          vvvvvvvvvvvvvv
vvvvvvvvvvvvvv          xxxxxxxxxxxxxx
xxxxxxxxxxxxxx          vvvvvvvvvvvvvv
vvvvvvvvvvvvvv          xxxxxxxxxxxxxx
vvvvvvvvvvvvvv          vvvvvvvvvvvvvv
```

`INVALID`　`INVALID`　`INVALID`

**Local references**

**Direct local dependency**

**Definition change**
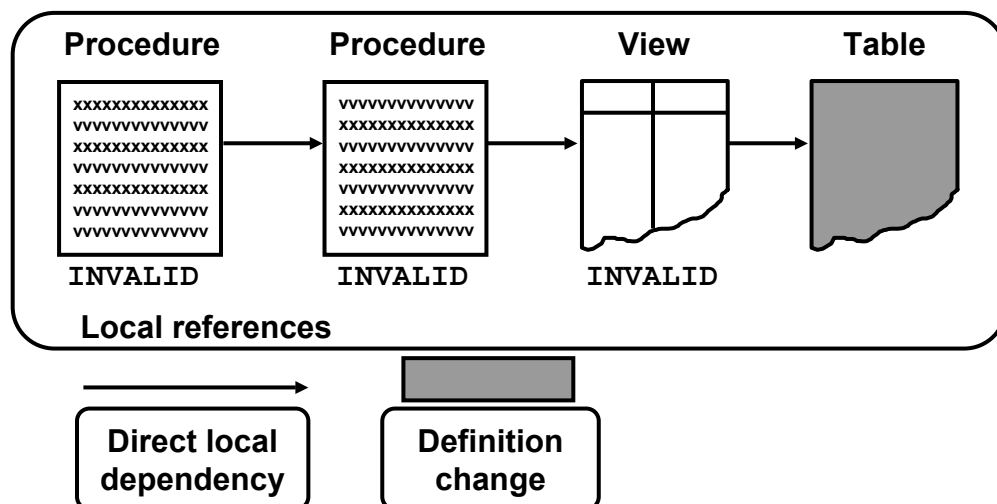
**The Oracle server implicitly recompiles any `INVALID` object when the object is next called.**

ORACLE

### Managing Local Dependencies (continued)

Assume that the structure of the table on which a view is based is modified. When you describe the view by using the *i*SQL*Plus `DESCRIBE` command, you get an error message that states that the object is invalid to describe. This is because the command is not a SQL command; at this stage, the view is invalid because the structure of its base table is changed. If you query the view now, then the view is recompiled automatically and you can see the result if it is successfully recompiled.

# A Scenario of Local Dependencies

**ADD_EMP procedure**

```
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv
```

**EMP_VW view**

| EMPLOYEE_ID | LAST_NAME | FIRST_NAME | EMAIL | DEPARTMEN |
|---|---|---|---|---|
| 100 | King | Steven | SKING | |
| 101 | Kochhar | Neena | NKOCHHAR | |
| 102 | De Haan | Lex | LDEHAAN | |
| 105 | Austin | David | DAUSTIN | |
| 108 | Greenberg | Nancy | NGREENBE | |

...

**QUERY_EMP procedure**

```
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvvv
vvvvvvxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvvv
```

**EMPLOYEES table**

| EMPLOYEE_ID | FIRST_NAME | LAST_NAME | EMAIL | PHONE_NI |
|---|---|---|---|---|
| 100 | Steven | King | SKING | 515.123.456 |
| 101 | Neena | Kochhar | NKOCHHAR | 515.123.456 |
| 102 | Lex | De Haan | LDEHAAN | 515.123.456 |
| 105 | David | Austin | DAUSTIN | 590.423.456 |
| 108 | Nancy | Greenberg | NGREENBE | 515.124.456 |

...

## Example

The QUERY_EMP procedure directly references the EMPLOYEES table. The ADD_EMP procedure updates the EMPLOYEES table indirectly by using the EMP_VW view.

In each of the following cases, is the ADD_EMP procedure invalidated and does it successfully recompile?

1. The internal logic of the QUERY_EMP procedure is modified.
2. A new column is added to the EMPLOYEES table.
3. The EMP_VW view is dropped.

# Displaying Direct Dependencies by Using
## USER_DEPENDENCIES

```
SELECT name, type, referenced_name, referenced_type
FROM   user_dependencies
WHERE  referenced_name IN ('EMPLOYEES','EMP_VW' );
```

| NAME | TYPE | REFERENCED_NAME | REFERENCED_T |
|------|------|-----------------|--------------|
| EMP_DETAILS_VIEW | VIEW | EMPLOYEES | TABLE |
| ... | | | |
| EMP_VW | VIEW | EMPLOYEES | TABLE |
| ... | | | |
| QUERY_EMP | PROCEDURE | EMPLOYEES | TABLE |
| ADD_EMP | PROCEDURE | EMP_VW | VIEW |

**Displaying Direct Dependencies by Using** USER_DEPENDENCIES

Determine which database objects to recompile manually by displaying direct dependencies from the USER_DEPENDENCIES data dictionary view.

Examine the ALL_DEPENDENCIES and DBA_DEPENDENCIES views, each of which contains the additional column OWNER, which references the owner of the object.

| Column | Column Description |
|--------|--------------------|
| NAME | The name of the dependent object |
| TYPE | The type of the dependent object (PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, or VIEW) |
| REFERENCED_OWNER | The schema of the referenced object |
| REFERENCED_NAME | The name of the referenced object |
| REFERENCED_TYPE | The type of the referenced object |
| REFERENCED_LINK_NAME | The database link used to access the referenced object |

# Displaying Direct and Indirect Dependencies

1. **Run the `utldtree.sql` script that creates the objects that enable you to display the direct and indirect dependencies.**

2. **Execute the `DEPTREE_FILL` procedure.**

```
EXECUTE deptree_fill('TABLE','SCOTT','EMPLOYEES')
```

**Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle**

Display direct and indirect dependencies from additional user views called DEPTREE and IDEPTREE; these views are provided by Oracle.

**Example**

1. Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/rdbms/admin` folder. (This script is supplied in the `lab` folder of your class files.)

2. Populate the DEPTREE_TEMPTAB table with information for a particular referenced object by invoking the DEPTREE_FILL procedure. There are three parameters for this procedure:

| | |
|---|---|
| *object_type* | Type of the referenced object |
| *object_owner* | Schema of the referenced object |
| *object_name* | Name of the referenced object |

# Displaying Dependencies

**The `DEPTREE` view:**

```
SELECT   nested_level, type, name
FROM     deptree
ORDER BY seq#;
```

| NESTED_LEVEL | TYPE | NAME |
|---|---|---|
| 0 | TABLE | EMPLOYEES |
| 1 | VIEW | EMP_DETAILS_VIEW |
| ... | | |
| 1 | TRIGGER | CHECK_SALARY |
| 1 | VIEW | EMP_VW |
| 2 | PROCEDURE | ADD_EMP |
| 1 | PACKAGE | MGR_CONSTRAINTS_PKG |
| 2 | TRIGGER | CHECK_PRES_TITLE |
| ... | | |

**Displaying Dependencies**

**Example**

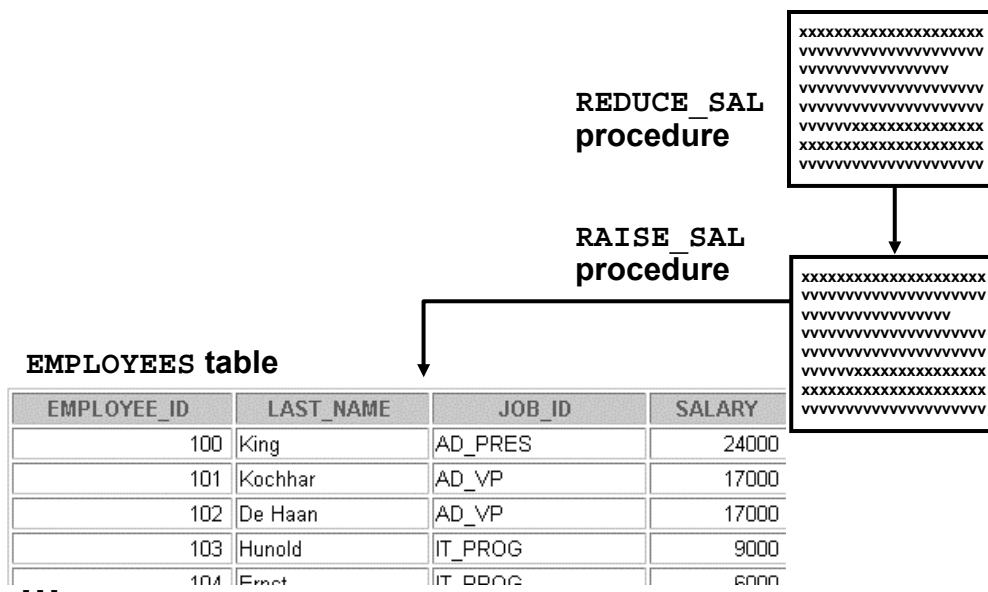Display a tabular representation of all dependent objects by querying the DEPTREE view.

Display an indented representation of the same information by querying the IDEPTREE view, which consists of a single column named DEPENDENCIES.

For example,

```
SELECT *
FROM   ideptree;
```

provides a single column of indented output of the dependencies in a hierarchical structure.

# Another Scenario of Local Dependencies



REDUCE_SAL
procedure

RAISE_SAL
procedure

EMPLOYEES **table**

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 103 | Hunold | IT_PROG | 9000 |
| 104 | Ernst | IT_PROG | 6000 |

...

## Another Scenario of Local Dependencies

**Example 1**

Predict the effect that a change in the definition of a procedure has on the recompilation of a dependent procedure.

Suppose that the RAISE_SAL procedure updates the EMPLOYEES table directly, and that the REDUCE_SAL procedure updates the EMPLOYEES table indirectly by way of RAISE_SAL.

In each of the following cases, does the REDUCE_SAL procedure successfully recompile?

1. The internal logic of the RAISE_SAL procedure is modified.
2. One of the formal parameters to the RAISE_SAL procedure is eliminated.

# A Scenario of Local Naming Dependencies
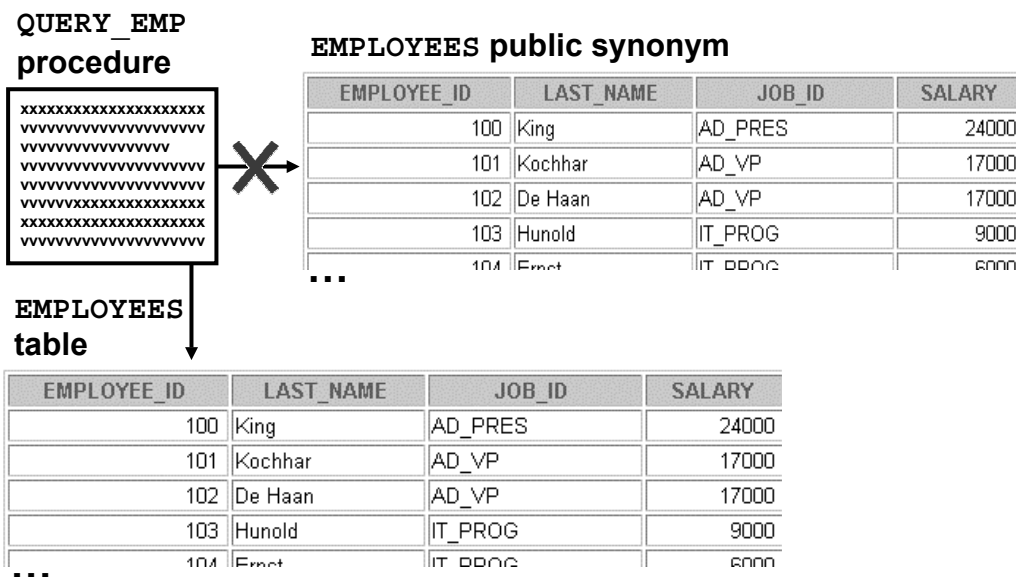
`QUERY_EMP`
**procedure**

`EMPLOYEES` **public synonym**

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 103 | Hunold | IT_PROG | 9000 |
| 104 | Ernst | IT_PROG | 6000 |

...

`EMPLOYEES`
**table**

| EMPLOYEE_ID | LAST_NAME | JOB_ID | SALARY |
|---|---|---|---|
| 100 | King | AD_PRES | 24000 |
| 101 | Kochhar | AD_VP | 17000 |
| 102 | De Haan | AD_VP | 17000 |
| 103 | Hunold | IT_PROG | 9000 |
| 104 | Ernst | IT_PROG | 6000 |

...

ORACLE

## A Scenario of Local Naming Dependencies

**Example 2**

Be aware of the subtle case in which the creation of a table, view, or synonym may unexpectedly invalidate a dependent object because it interferes with the Oracle server hierarchy for resolving name references.

Predict the effect that the name of a new object has upon a dependent procedure.

Suppose that your `QUERY_EMP` procedure originally referenced a public synonym called `EMPLOYEES`. However, you have just created a new table called `EMPLOYEES` within your own schema. Does this change invalidate the procedure? Which of the two `EMPLOYEES` objects does `QUERY_EMP` reference when the procedure recompiles?

Now suppose that you drop your private `EMPLOYEES` table. Does this invalidate the procedure? What happens when the procedure recompiles?

You can track security dependencies in the `USER_TAB_PRIVS` data dictionary view.

# Understanding Remote Dependencies
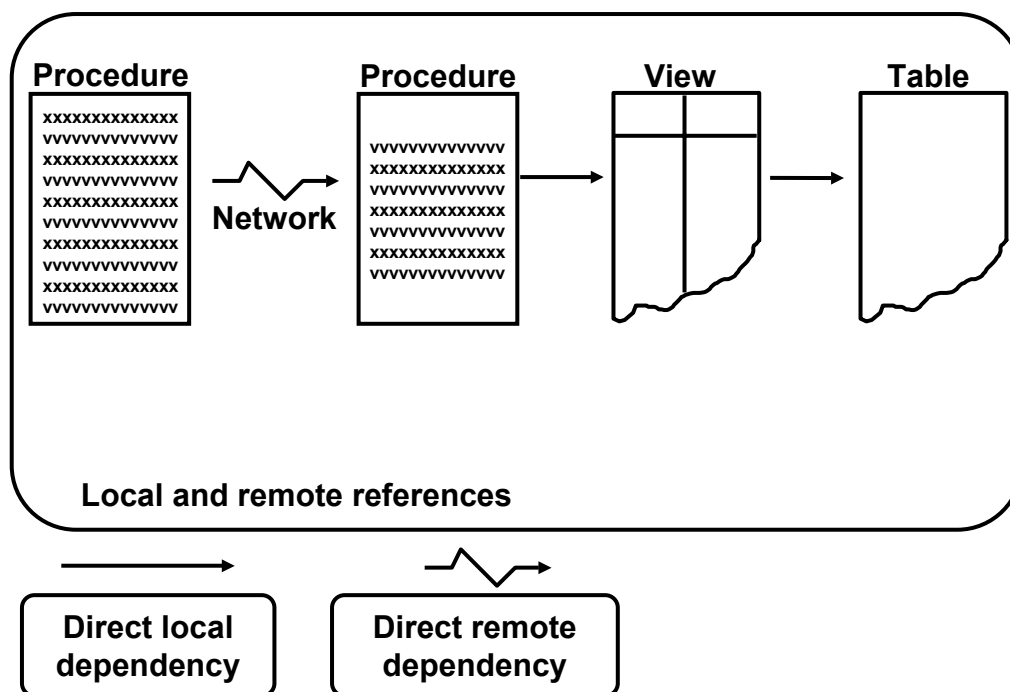
## Understanding Remote Dependencies

In the case of remote dependencies, the objects are on separate nodes. The Oracle server does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies (including functions, packages, and triggers). The local stored procedure and all its dependent objects are invalidated but do not automatically recompile when called for the first time.

**Understanding Remote Dependencies**

Procedure · Procedure · View · Table

VALID · INVALID · INVALID

**Local and remote references**

| Direct local dependency | Direct remote dependency | Definition change |
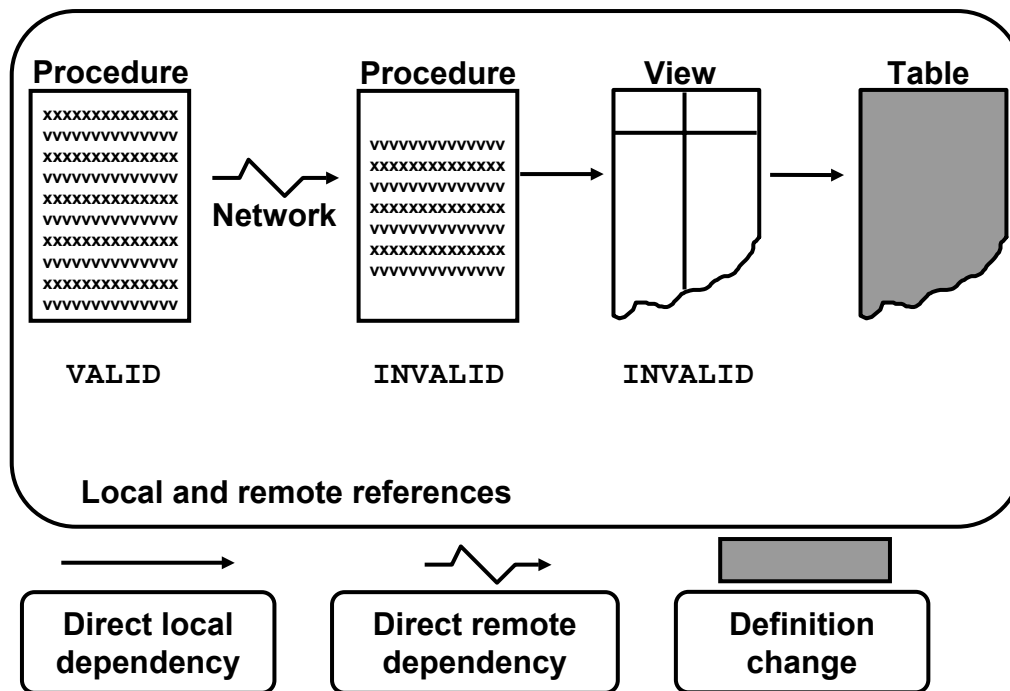
**Understanding Remote Dependencies (continued)**

**Recompilation of Dependent Objects: Local and Remote**

- Verify successful explicit recompilation of the dependent remote procedures and implicit recompilation of the dependent local procedures by checking the status of these procedures within the USER_OBJECTS view.
- If an automatic implicit recompilation of the dependent local procedures fails, the status remains invalid and the Oracle server issues a run-time error. Therefore, to avoid disrupting production, it is strongly recommended that you recompile local dependent objects manually, rather than relying on an automatic mechanism.

# Concepts of Remote Dependencies

**Remote dependencies are governed by the mode that is chosen by the user:**

- `TIMESTAMP` **checking**
- `SIGNATURE` **checking**

ORACLE

**Concepts of Remote Dependencies**

**`TIMESTAMP` Checking**

Each PL/SQL program unit carries a time stamp that is set when it is created or recompiled. Whenever you alter a PL/SQL program unit or a relevant schema object, all its dependent program units are marked as invalid and must be recompiled before they can execute. The actual time stamp comparison occurs when a statement in the body of a local procedure calls a remote procedure.

**`SIGNATURE` Checking**

For each PL/SQL program unit, both the time stamp and the signature are recorded. The signature of a PL/SQL construct contains information about the following:
- The name of the construct (procedure, function, or package)
- The base types of the parameters of the construct
- The modes of the parameters (`IN`, `OUT`, or `IN OUT`)
- The number of the parameters

The recorded time stamp in the calling program unit is compared with the current time stamp in the called remote program unit. If the time stamps match, the call proceeds. If they do not match, the remote procedure call (RPC) layer performs a simple comparison of the signature to determine whether the call is safe or not. If the signature has not been changed in an incompatible manner, execution continues; otherwise, an error is returned.

**Oracle Database 10*g*: Develop PL/SQL Program Units   8-15**

# REMOTE_DEPENDENCIES_MODE Parameter

**Setting REMOTE_DEPENDENCIES_MODE:**

- **As an init.ora parameter**
  REMOTE_DEPENDENCIES_MODE = value

- **At the system level**
  ALTER SYSTEM SET
  REMOTE_DEPENDENCIES_MODE = value

- **At the session level**
  ALTER SESSION SET
  REMOTE_DEPENDENCIES_MODE = value

ORACLE

**REMOTE_DEPENDENCIES_MODE Parameter**

**Setting the REMOTE_DEPENDENCIES_MODE**

*value*    TIMESTAMP
          SIGNATURE

Specify the value of the REMOTE_DEPENDENCIES_MODE parameter using one of the three methods described in the slide.

**Note:** The calling site determines the dependency model.

# Remote Dependencies and
# Time Stamp Mode
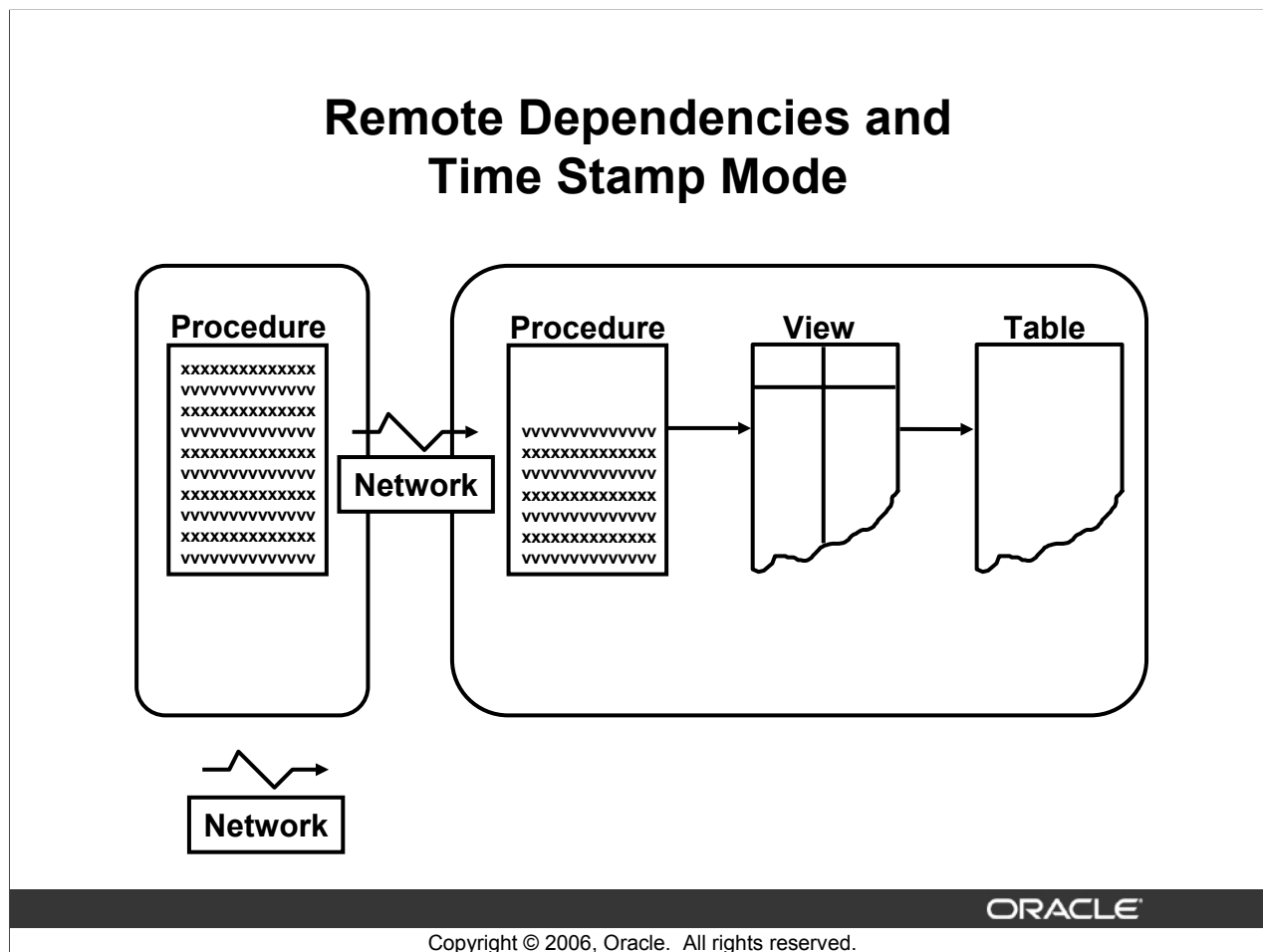
**Remote Dependencies and Time Stamp Mode**

If time stamps are used to handle dependencies among PL/SQL program units, then whenever you alter a program unit or a relevant schema object, all its dependent units are marked as invalid and must be recompiled before they can be run.

# Remote Dependencies and Time Stamp Mode

Procedure — VALID
Network
Procedure — INVALID
View — INVALID
Table

Network
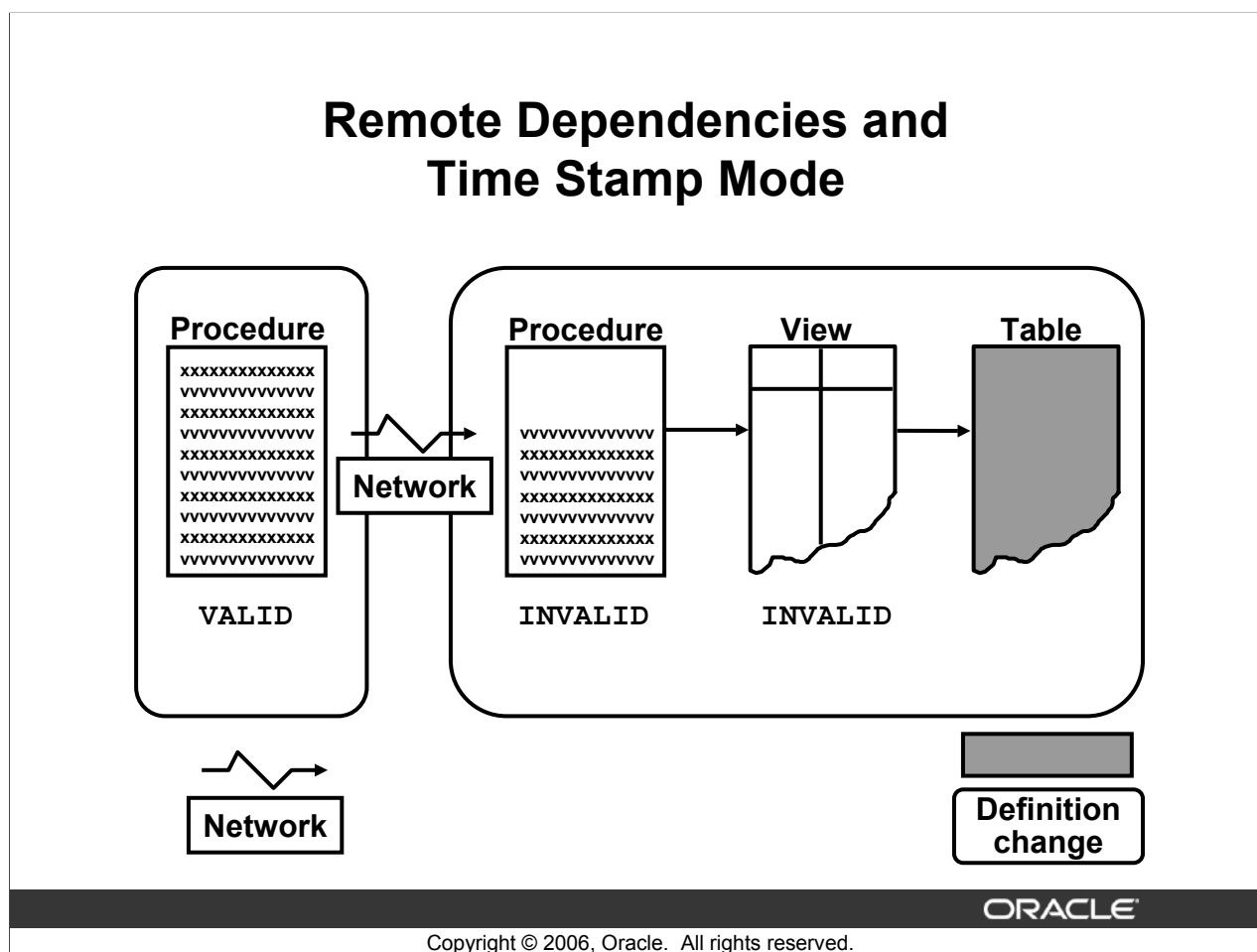
Definition change

### Remote Dependencies and Time Stamp Mode (continued)

In the example in the slide, the definition of the table changes. Therefore, all its dependent units are marked as invalid and must be recompiled before they can be run.

- When remote objects change, it is strongly recommended that you recompile local dependent objects manually in order to avoid disrupting production.
- The remote dependency mechanism is different from the automatic local dependency mechanism already discussed. The first time a recompiled remote subprogram is invoked by a local subprogram, you get an execution error and the local subprogram is invalidated; the second time it is invoked, implicit automatic recompilation takes place.

**Remote Procedure B
Compiles at 8:00 a.m.**

**Remote procedure B**
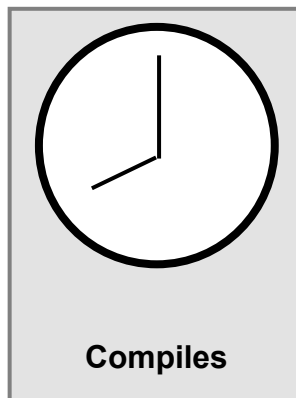
**Compiles**

**Valid**

**Local Procedures Referencing Remote Procedures**

A local procedure that references a remote procedure is invalidated by the Oracle server if the remote procedure is recompiled after the local procedure is compiled.

**Automatic Remote Dependency Mechanism**

When a procedure compiles, the Oracle server records the time stamp of that compilation within the P code of the procedure.

In the slide, when the remote procedure B is successfully compiled at 8:00 a.m., this time is recorded as its time stamp.

**Local Procedure A
Compiles at 9:00 a.m.**
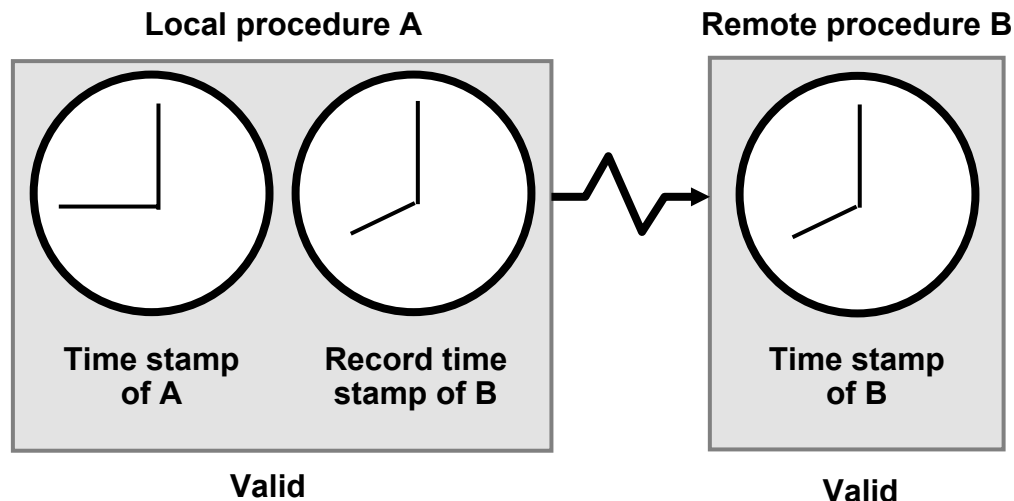
Local procedure A

Remote procedure B

Time stamp
of A

Record time
stamp of B

Time stamp
of B

Valid

Valid

### Local Procedures Referencing Remote Procedures (continued)

#### Automatic Remote Dependency Mechanism (continued)

When a local procedure referencing a remote procedure compiles, the Oracle server also records the time stamp of the remote procedure in the P code of the local procedure.

In the slide, local procedure A (which is dependent on remote procedure B) is compiled at 9:00 a.m. The time stamps of both procedure A and remote procedure B are recorded in the P code of procedure A.

**Execute Procedure A**

Local procedure A · Remote procedure B

Time stamp comparison

Time stamp of A · Time stamp of B · Execute B · Time stamp of B

Valid · Valid
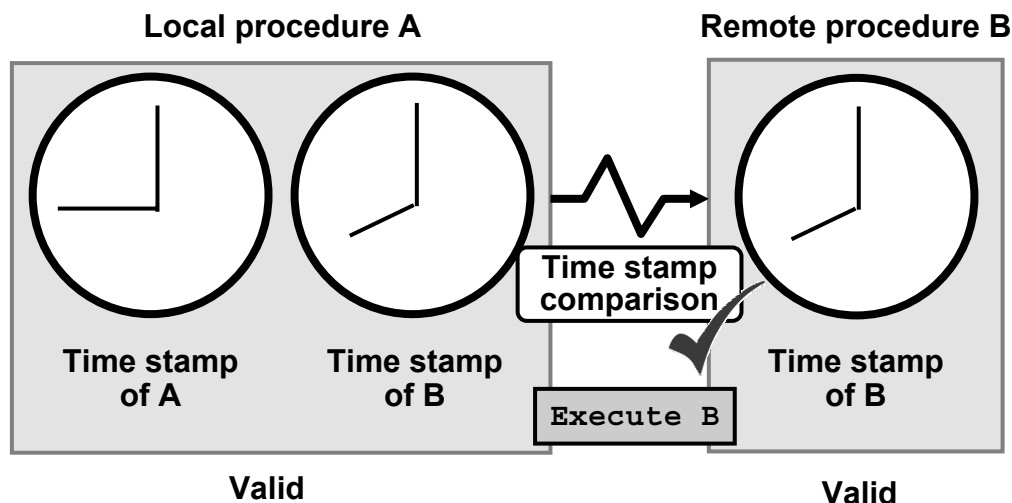
**Automatic Remote Dependency**

When the local procedure is invoked at run time, the Oracle server compares the two time stamps of the referenced remote procedure.

If the time stamps are equal (indicating that the remote procedure has not recompiled), then the Oracle server executes the local procedure.

In the example in the slide, the time stamp recorded with the P code of remote procedure B is the same as that recorded with local procedure A. Therefore, local procedure A is valid.

# Remote Procedure B Recompiled at 11:00 a.m.

**Remote procedure B**
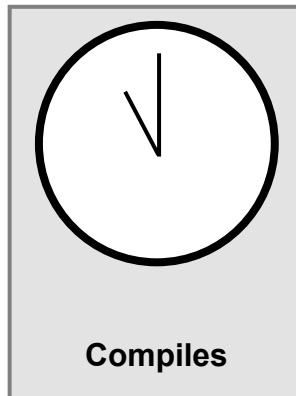
**Compiles**

**Valid**

**Local Procedures Referencing Remote Procedures**

Assume that remote procedure B is successfully recompiled at 11:00 a.m. The new time stamp is recorded along with its P code.

# Execute Procedure A

**Local procedure A** · **Remote procedure B**

Time stamp comparison

**Time stamp of A** · **Time stamp of B** · ERROR · **Time stamp of B**

Valid · **Invalid** · **Valid**

ORACLE

## Automatic Remote Dependency

If the time stamps are not equal (indicating that the remote procedure has recompiled), then the Oracle server invalidates the local procedure and returns a 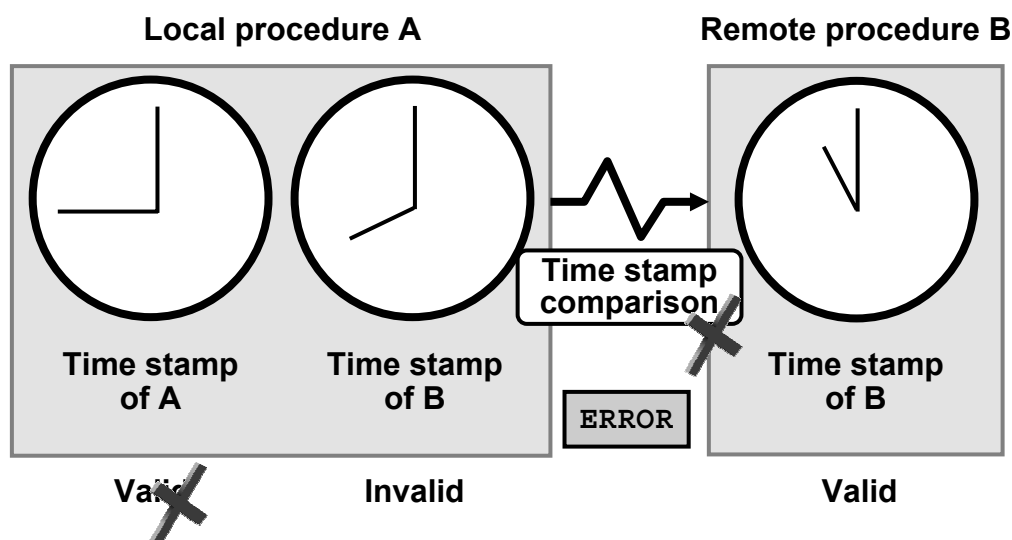run-time error. If the local procedure (which is now tagged as invalid) is invoked a second time, then the Oracle server recompiles it before executing, in accordance with the automatic local dependency mechanism.

**Note:** If a local procedure returns a run-time error the first time it is invoked (indicating that the remote procedure's time stamp has changed), then you should develop a strategy to reinvoke the local procedure.

In the example in the slide, the remote procedure is recompiled at 11:00 a.m. and this time is recorded as its time stamp in the P code. The P code of local procedure A still has 8:00 a.m. as the time stamp for remote procedure B. Because the time stamp recorded with the P code of local procedure A is different from that recorded with the remote procedure B, the local procedure is marked invalid. When the local procedure is invoked for the second time, it can be successfully compiled and marked valid.

A disadvantage of time stamp mode is that it is unnecessarily restrictive. Recompilation of dependent objects across the network is often performed when not strictly necessary, leading to performance degradation.

**Oracle Database 10*g*: Develop PL/SQL Program Units   8-23**

# Signature Mode

- **The signature of a procedure is:**
  - **The name of the procedure**
  - **The data types of the parameters**
  - **The modes of the parameters**
- **The signature of the remote procedure is saved in the local procedure.**
- **When executing a dependent procedure, the signature of the referenced remote procedure is compared.**

**Signatures**

To alleviate some of the problems with the time stamp–only dependency model, you can use the signature model. This allows the remote procedure to be recompiled without affecting the local procedures. This is important if the database is distributed.

The signature of a subprogram contains the following information:
- The name of the subprogram
- The data types of the parameters
- The modes of the parameters
- The number of parameters
- The data type of the return value for a function

If a remote program is changed and recompiled but the signature does not change, then the local procedure can execute the remote procedure. With the time stamp method, an error would have been raised because the time stamps would not have matched.

# Recompiling a PL/SQL
# Program Unit

**Recompilation:**

- **Is handled automatically through implicit run-time recompilation**
- **Is handled through explicit recompilation with the `ALTER` statement**

```
ALTER PROCEDURE [SCHEMA.]procedure_name COMPILE;
```

```
ALTER FUNCTION   [SCHEMA.]function_name  COMPILE;
```

```
ALTER PACKAGE [SCHEMA.]package_name
   COMPILE [PACKAGE | SPECIFICATION | BODY];
```

```
ALTER TRIGGER trigger_name [COMPILE[DEBUG]];
```

ORACLE

### Recompiling PL/SQL Objects

If the recompilation is successful, the object becomes valid. If not, the Oracle server returns an error and the object remains invalid. When you recompile a PL/SQL object, the Oracle server first recompiles any invalid object on which it depends.

**Procedure:** Any local objects that depend on a procedure (such as procedures that call the recompiled procedure or package bodies that define the procedures that call the recompiled procedure) are also invalidated.

**Packages:** The COMPILE PACKAGE option recompiles both the package specification and the body, regardless of whether it is invalid. The COMPILE SPECIFICATION option recompiles the package specification. Recompiling a package specification invalidates any local objects that depend on the specification, such as subprograms that use the package. Note that the body of a package also depends on its specification. The COMPILE BODY option recompiles only the package body.

**Triggers:** Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The DEBUG option instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

**Oracle Database 10*g*: Develop PL/SQL Program Units   8-25**

# Unsuccessful Recompilation

**Recompiling dependent procedures and functions is unsuccessful when:**

- **The referenced object is dropped or renamed**
- **The data type of the referenced column is changed**
- **The referenced column is dropped**
- **A referenced view is replaced by a view with different columns**
- **The parameter list of a referenced procedure is modified**

ORACLE

**Unsuccessful Recompilation**

Sometimes a recompilation of dependent procedures is unsuccessful (for example, when a referenced table is dropped or renamed).

The success of any recompilation is based on the exact dependency. If a referenced view is re-created, any object that is dependent on the view needs to be recompiled. The success of the recompilation depends on the columns that the view now contains, as well as the columns that the dependent objects require for their execution. If the required columns are not part of the new view, then the object remains invalid.

# Successful Recompilation

**Recompiling dependent procedures and functions is successful if:**

- **The referenced table has new columns**
- **The data type of referenced columns has not changed**
- **A private table is dropped, but a public table that has the same name and structure exists**
- **The PL/SQL body of a referenced procedure has been modified and recompiled successfully**

### Successful Recompilation

The recompilation of dependent objects is successful if:
- New columns are added to a referenced table
- All INSERT statements include a column list
- No new column is defined as NOT NULL

When a private table is referenced by a dependent procedure and the private table is dropped, the status of the dependent procedure becomes invalid. When the procedure is recompiled (either explicitly or implicitly) and a public table exists, the procedure can recompile successfully but is now dependent on the public table. The recompilation is successful only if the public table contains the columns that the procedure requires; otherwise, the status of the procedure remains invalid.

# Recompilation of Procedures

**Minimize dependency failures by:**

- **Declaring records with the `%ROWTYPE` attribute**
- **Declaring variables with the `%TYPE` attribute**
- **Querying with the `SELECT *` notation**
- **Including a column list with `INSERT` statements**

**Recompilation of Procedures**

You can minimize recompilation failure by following the guidelines that are shown in the slide.

# Packages and Dependencies

**Package specification**

**Stand-alone procedure**

**Procedure A declaration**

**Valid**

**Valid**

**Package body**

**Procedure A definition**

**Definition changed**
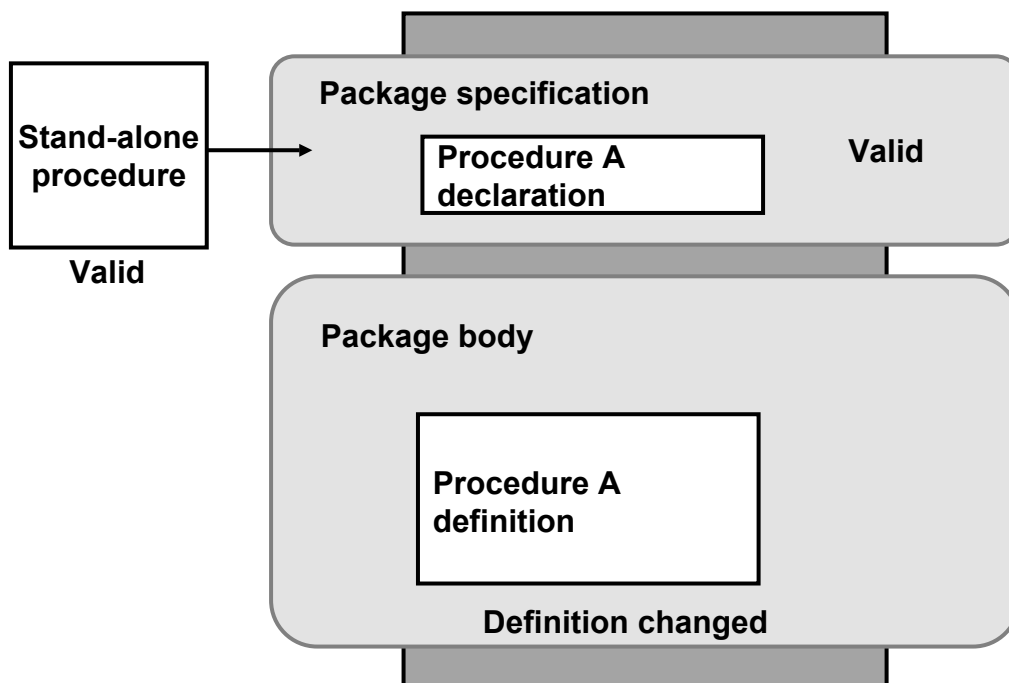
## Managing Dependencies

You can simplify dependency management with packages when referencing a package procedure or function from a stand-alone procedure or function.

- If the package body changes and the package specification does not change, then the stand-alone procedure that references a package construct remains valid.
- If the package specification changes, then the outside procedure referencing a package construct is invalidated, as is the package body.

# Packages and Dependencies

**Package specification**

| Procedure A declaration |

**Valid**

**Package body**

**Stand-alone procedure**

**Definition changed**

| Procedure A definition |
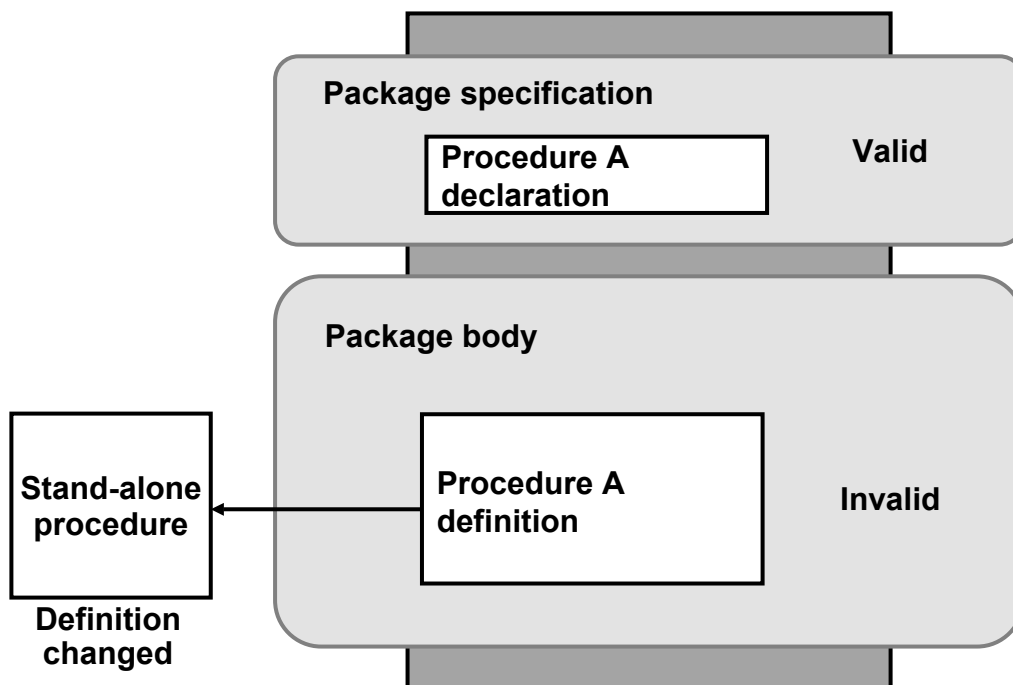
**Invalid**

### Managing Dependencies (continued)

If a stand-alone procedure that is referenced within the package changes, then the entire package body is invalidated, but the package specification remains valid. Therefore, it is recommended that you bring the procedure into the package.

# Summary

**In this lesson, you should have learned how to:**

- **Keep track of dependent procedures**
- **Recompile procedures manually as soon as possible after the definition of a database object changes**

## Summary

Avoid disrupting production by keeping track of dependent procedures and recompiling them manually as soon as possible after the definition of a database object changes.

| Situation | Automatic Recompilation |
|---|---|
| Procedure depends on a local object. | Yes, at first reexecution |
| Procedure depends on a remote procedure. | Yes, but at second reexecution. Use manual recompilation for first reexecution, or reinvoke it a second time. |
| Procedure depends on a remote object other than a procedure. | No |

# Practice 8: Overview

**This practice covers the following topics:**

- **Using `DEPTREE_FILL` and `IDEPTREE` to view dependencies**
- **Recompiling procedures, functions, and packages**

**Practice 8: Overview**

In this practice, you use the `DEPTREE_FILL` procedure and the `IDEPTREE` view to investigate dependencies in your schema. In addition, you recompile invalid procedures, functions, packages, and views.

**Practice 8**

1. Answer the following questions:
   a. Can a table or a synonym be invalidated?
   b. Consider the following dependency example:

   The stand-alone procedure MY_PROC depends on the MY_PROC_PACK
   package procedure. The MY_PROC_PACK procedure's definition is changed by
   recompiling the package body. The MY_PROC_PACK procedure's declaration
   is not altered in the package specification.

   In this scenario, is the stand-alone procedure MY_PROC invalidated?

2. Create a tree structure showing all dependencies involving your add_employee
   procedure and your valid_deptid function.
   **Note:** add_employee and valid_deptid were created in the lesson titled
   "Creating Stored Functions." You can run the solution scripts for Practice 2 if you
   need to create the procedure and function.
   a. Load and execute the utldtree.sql script, which is located in the
      E:\lab\PLPU\labs folder.
   b. Execute the deptree_fill procedure for the add_employee procedure.
   c. Query the IDEPTREE view to see your results.
   d. Execute the deptree_fill procedure for the valid_deptid function.
   e. Query the IDEPTREE view to see your results.

**If you have time, complete the following exercise:**

3. Dynamically validate invalid objects.
   a. Make a copy of your EMPLOYEES table, called EMPS.
   b. Alter your EMPLOYEES table and add the column TOTSAL with data type
      NUMBER(9,2).
   c. Create and save a query to display the name, type, and status of all invalid
      objects.
   d. In the compile_pkg (created in Practice 6 in the lesson titled "Dynamic SQL
      and Metadata"), add a procedure called recompile that recompiles all invalid
      procedures, functions, and packages in your schema. Use Native Dynamic SQL
      to alter the invalid object type and compile it.
   e. Execute the compile_pkg.recompile procedure.
   f. Run the script file that you created in step 3c to check the status column value.
      Do you still have objects with an INVALID status?

**Oracle Database 10*g*: Develop PL/SQL Program Units   8-33**

**9**

# Manipulating Large Objects

# Objectives

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

- **Compare and contrast** LONG **and** LOB **(large object) data types**
- **Create and maintain** LOB **data types**
- **Differentiate between internal and external** LOBs
- **Use the** DBMS_LOB **PL/SQL package**
- **Describe the use of temporary** LOBs

**Lesson Aim**

Databases have long been used to store large objects. However, the mechanisms built into databases have never been as useful as the large object (LOB) data types that have been provided since Oracle8. This lesson describes the characteristics of the new data types, comparing and contrasting them with earlier data types. Examples, syntax, and issues regarding the LOB types are also presented.

**Note:** A LOB is a data type and should not be confused with an object type.

# What Is a LOB?

**LOBs are used to store large unstructured data such as text, graphic images, films, and sound waveforms.**

"Four score and seven years ago, our forefathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."

**Text (CLOB)**  **Photo (BLOB)**  **Movie (BFILE)**
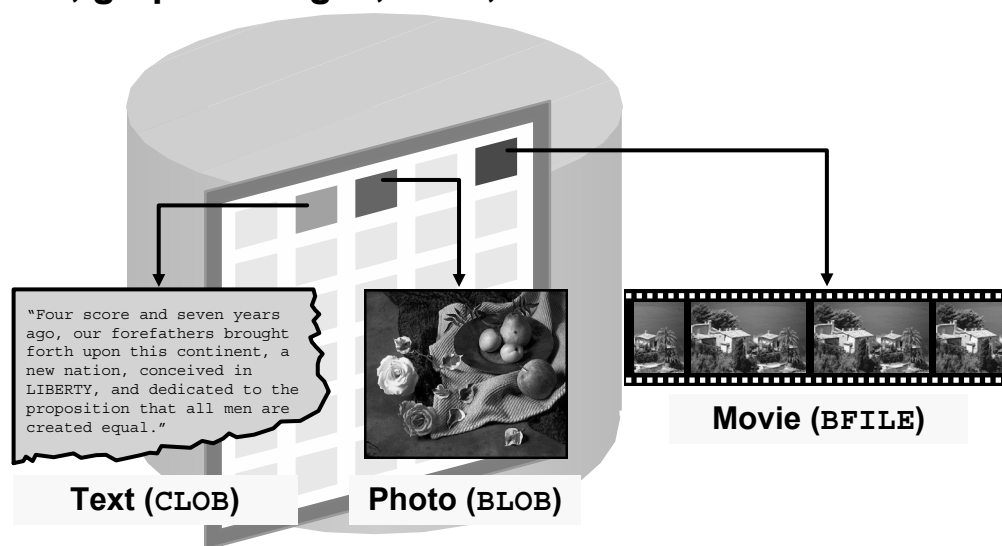
ORACLE

## LOB: Overview

A LOB is a data type that is used to store large, unstructured data such as text, graphic images, video clippings, and so on. Structured data, such as a customer record, may be a few hundred bytes, but even small amounts of multimedia data can be thousands of times larger. Also, multimedia data may reside in operating system (OS) files, which may need to be accessed from a database.

There are four large object data types:

- BLOB represents a binary large object, such as a video clip.
- CLOB represents a character large object.
- NCLOB represents a multibyte character large object.
- BFILE represents a binary file stored in an OS binary file outside the database. The BFILE column or attribute stores a file locator that points to the external file.

LOBs are characterized in two ways, according to their interpretations by the Oracle server (binary or character) and their storage aspects. LOBs can be stored internally (inside the database) or in host files. There are two categories of LOBs:

- **Internal LOBs (CLOB, NCLOB, BLOB):** Stored in the database
- **External files (BFILE):** Stored outside the database

**LOB: Overview (continued)**

Oracle Database 10*g* performs implicit conversion between `CLOB` and `VARCHAR2` data types. The other implicit conversions between `LOB`s are not possible. For example, if the user creates a table `T` with a `CLOB` column and a table `S` with a `BLOB` column, the data is not directly transferable between these two columns.

`BFILE`s can be accessed only in read-only mode from an Oracle server.

# Contrasting LONG and LOB Data Types

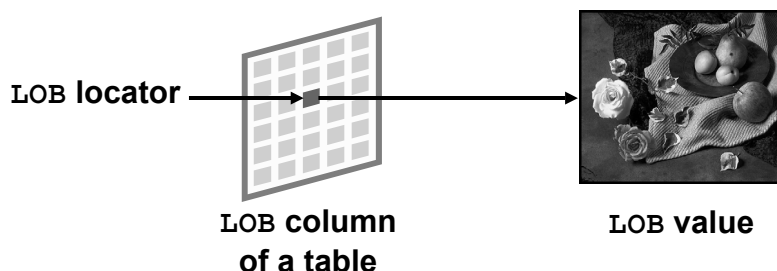| LONG and LONG RAW | LOB |
|---|---|
| Single LONG column per table | Multiple LOB columns per table |
| Up to 2 GB | Up to 4 GB |
| SELECT returns data | SELECT returns locator |
| Data stored in-line | Data stored in-line or out-of-line |
| Sequential access to data | Random access to data |

### LONG and LOB Data Types

LONG and LONG RAW data types were previously used for unstructured data such as binary images, documents, or geographical information. These data types are superseded by the LOB data types. Oracle Database 10*g* provides a LONG-to-LOB application programming interface (API) to migrate from LONG columns to LOB columns. The following bulleted list compares the LOB functionality with the older types, where LONGs refer to LONG and LONG RAW, and LOBs refer to all LOB data types:

- A table can have multiple LOB columns and object type attributes. A table can have only one LONG column.
- The maximum size of LONGs is 2 GB; LOBs can be up to 4 GB.
- LOBs return the locator; LONGs return the data.
- LOBs store a locator in the table and the data in a different segment, unless the data is less than 4,000 bytes; LONGs store all data in the same data block. In addition, LOBs allow data to be stored in a separate segment and tablespace, or in a host file.
- LOBs can be object type attributes; LONGs cannot be object type attributes.
- LOBs support random piecewise access to the data through a file-like interface; LONGs are restricted to sequential piecewise access.

The TO_LOB function can be used to convert LONG and LONG RAW values in a column to LOB values. You use this in the SELECT list of a subquery in an INSERT statement.

**Oracle Database 10*g*: Develop PL/SQL Program Units   9-5**

**Anatomy of a LOB**

**The LOB column stores a locator to the LOB's value.**

LOB locator →

LOB column
of a table

LOB value

**Components of a LOB**

There are two distinct parts to a LOB:
- **LOB value:** The data that constitutes the real object being stored
- **LOB locator:** A pointer to the location of the LOB value stored in the database

Regardless of where the value of LOB is stored, a locator is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value.

A LOB column does not contain the data; it contains the locator of the LOB value.

When a user creates an internal LOB, the value is stored in the LOB segment and a locator to the out-of-line LOB value is placed in the LOB column of the corresponding row in the table. External LOBs store the data outside the database, so only a locator to the LOB value is stored in the table.

To access and manipulate LOBs without SQL data manipulation language (DML), you must create a LOB locator. The programmatic interfaces operate on the LOB values, using these locators in a manner similar to OS file handles.

# Internal **LOB**s

**The LOB value is stored in the database.**



```
"Four score and seven years ago,
our forefathers brought forth upon
this continent, a new nation,
conceived in LIBERTY, and dedicated
to the proposition that all men
are created equal."
```

**CLOB**                    **BLOB**
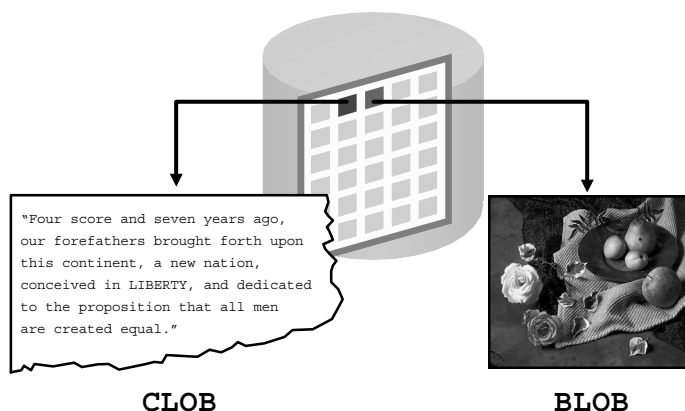
**Features of Internal LOBs**

The internal LOB is stored in the Oracle server. A BLOB, NCLOB, or CLOB can be one of the following:

- An attribute of a user-defined type
- A column in a table
- A bind or host variable
- A PL/SQL variable, parameter, or result

Internal LOBs can take advantage of Oracle features such as:

- Concurrency mechanisms
- Redo logging and recovery mechanisms
- Transactions with COMMIT or ROLLBACK

The BLOB data type is interpreted by the Oracle server as a bitstream, similar to the LONG RAW data type.

The CLOB data type is interpreted as a single-byte character stream.

The NCLOB data type is interpreted as a multiple-byte character stream, based on the byte length of the database national character set.

# Managing Internal LOBs

- **To interact fully with LOB, file-like interfaces are provided in:**
  - **PL/SQL package DBMS_LOB**
  - **Oracle Call Interface (OCI)**
  - **Oracle Objects for object linking and embedding (OLE)**
  - **Pro*C/C++ and Pro*COBOL precompilers**
  - **Java Database Connectivity (JDBC)**
- **The Oracle server provides some support for LOB management through SQL.**

ORACLE

## How to Manage LOBs

To manage an internal LOB, perform the following steps:

1. Create and populate the table containing the LOB data type.
2. Declare and initialize the LOB locator in the program.
3. Use SELECT FOR UPDATE to lock the row containing the LOB into the LOB locator.
4. Manipulate the LOB with DBMS_LOB package procedures, OCI calls, Oracle Objects for OLE, Oracle precompilers, or JDBC using the LOB locator as a reference to the LOB value.
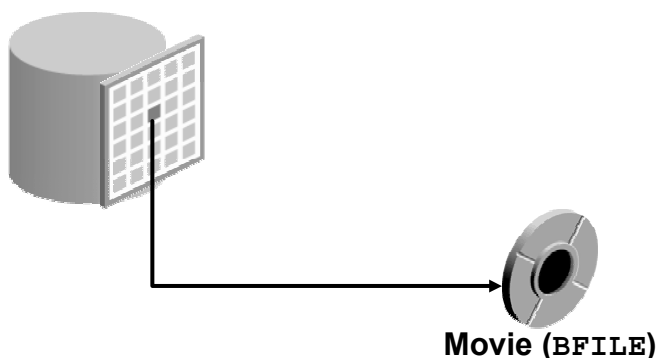
   You can also manage LOBs through SQL.
5. Use the COMMIT command to make any changes permanent.

**Oracle Database 10g: Develop PL/SQL Program Units 9-8**

# What Are BFILES?

**The BFILE data type supports an external or file-based large object as:**
- **Attributes in an object type**
- **Column values in a table**

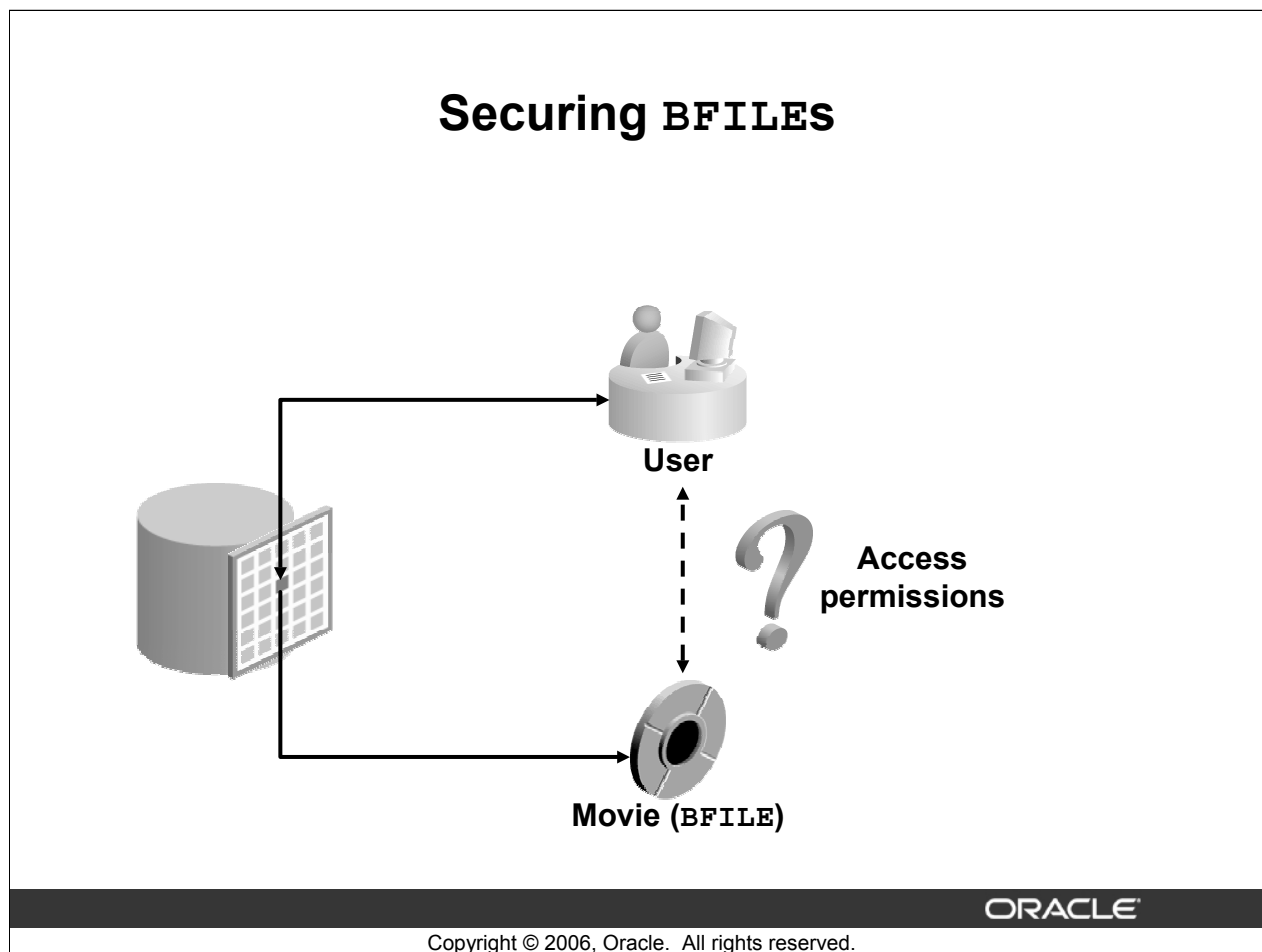**Movie (BFILE)**

## What Are BFILES?

BFILEs are external large objects (LOBs) stored in OS files that are external to database tables. The BFILE data type stores a locator to the physical file. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats. The external LOBs may be located on hard disks, CD-ROMs, photo CDs, or other media, but a single LOB cannot extend from one medium or device to another. The BFILE data type is available so that database users can access the external file system. Oracle Database 10*g* provides:

- Definition of BFILE objects
- Association of BFILE objects to corresponding external files
- Security for BFILEs

The remaining operations that are required for using BFILEs are possible through the DBMS_LOB package and OCI. BFILEs are read-only; they do not participate in transactions. Support for integrity and durability must be provided by the operating system. The file must be created and placed in the appropriate directory, giving the Oracle process privileges to read the file. When the LOB is deleted, the Oracle server does not delete the file. Administration of the files and the OS directory structures can be managed by the database administrator (DBA), system administrator, or user. The maximum size of an external large object depends on the operating system but cannot exceed 4 GB.

**Note:** BFILEs are available with the Oracle8 database and later releases.

# Securing BFILEs

Copyright © 2006, Oracle. All rights reserved.

**Securing BFILEs**

Unauthenticated access to files on a server presents a security risk. Oracle Database 10*g* can act as a security mechanism to shield the operating system from unsecured access while removing the need to manage additional user accounts on an enterprise computer system.

**File Location and Access Privileges**

The file must reside on the machine where the database exists. A timeout to read a nonexistent BFILE is based on the OS value.

You can read a BFILE in the same way as you read an internal LOB. However, there could be restrictions related to the file itself, such as:

- Access permissions
- File system space limits
- Non-Oracle manipulations of files
- OS maximum file size

Oracle Database 10*g* does not provide transactional support on BFILEs. Any support for integrity and durability must be provided by the underlying file system and the OS. Oracle backup and recovery methods support only the LOB locators, not the physical BFILEs.

# A New Database Object: DIRECTORY

**A New Database Object: DIRECTORY**

A DIRECTORY is a nonschema database object that enables the administration of access and usage of BFILEs in Oracle Database 10*g*.

A DIRECTORY specifies an alias for a directory on the file system of the server under which a BFILE is located. By granting suitable privileges for these items to users, you can provide secure access to files in the corresponding directories on a user-by-user basis (certain directories can be made read-only, inaccessible, and so on).

Furthermore, these directory aliases can be used while referring to files (open, close, read, and so on) in PL/SQL and OCI. This provides application abstraction from hard-coded path names and gives flexibility in portably managing file locations.

The DIRECTORY object is owned by SYS and created by the DBA (or a user with the CREATE ANY DIRECTORY privilege). The directory objects have object privileges, unlike any other nonschema object. Privileges to the DIRECTORY object can be granted and revoked. Logical path names are not supported.

The permissions for the actual directory depend on the operating system. They may differ from those defined for the DIRECTORY object and could change after the creation of the DIRECTORY object.

# Guidelines for Creating
## `DIRECTORY` Objects

- **Do not create `DIRECTORY` objects on paths with database files.**
- **Limit the number of people who are given the following system privileges:**
  - `CREATE ANY DIRECTORY`
  - `DROP ANY DIRECTORY`
- **All `DIRECTORY` objects are owned by `SYS`.**
- **Create directory paths and properly set permissions before using the `DIRECTORY` object so that the Oracle server can read the file.**

ORACLE

### Guidelines for Creating `DIRECTORY` Objects

To associate an OS file with a `BFILE`, you should first create a `DIRECTORY` object that is an alias for the full path name to the OS file.

Create `DIRECTORY` objects by using the following guidelines:
- Directories should point to paths that do not contain database files because tampering with these files could corrupt the database. Currently, only the `READ` privilege can be given for a `DIRECTORY` object.
- The `CREATE ANY DIRECTORY` and `DROP ANY DIRECTORY` system privileges should be used carefully and not granted to users indiscriminately.
- `DIRECTORY` objects are not schema objects; all are owned by `SYS`.
- Create the directory paths with appropriate permissions on the OS before creating the `DIRECTORY` object. Oracle does not create the OS path.

If you migrate the database to a different OS, then you may need to change the path value of the `DIRECTORY` object.

The `DIRECTORY` object information that you create by using the `CREATE DIRECTORY` command is stored in the `DBA_DIRECTORIES` and `ALL_DIRECTORIES` data dictionary views.

# Managing `BFILE`s

**The DBA or the system administrator:**
1. **Creates an OS directory and supplies files**
2. **Creates a `DIRECTORY` object in the database**
3. **Grants the `READ` privilege on the `DIRECTORY` object to appropriate database users**

**The developer or the user:**
4. **Creates an Oracle table with a column defined as a `BFILE` data type**
5. **Inserts rows into the table using the `BFILENAME` function to populate the `BFILE` column**
6. **Writes a PL/SQL subprogram that declares and initializes a `LOB` locator, and reads `BFILE`**

## How to Manage `BFILE`s

Managing `BFILE`s requires cooperation between the database administrator and the system administrator and then between the developer and the user of the files.

The database or system administrator should perform the following privileged tasks:
1. Create the operating system (OS) directory (as an Oracle user), and set permissions so that the Oracle server can read the contents of the OS directory. Load files into the OS directory.
2. Create a database `DIRECTORY` object that references the OS directory.
3. Grant the `READ` privilege on the database `DIRECTORY` object to database users requiring access to it.

The designer, application developer, or user should perform the following tasks:
4. Create a database table containing a column defined as the `BFILE` data type.
5. Insert rows into the table using the `BFILENAME` function to populate the `BFILE` column associating the field to an OS file in the named `DIRECTORY`.
6. Write PL/SQL subprograms that:
   a. Declare and initialize the `BFILE` `LOB` locator
   b. Select the row and column containing the `BFILE` into the `LOB` locator
   c. Read the `BFILE` with a `DBMS_LOB` function, using the locator file reference

**Oracle Database 10*g*: Develop PL/SQL Program Units   9-13**

# Preparing to Use BFILES

1. **Create an OS directory to store the physical data files:**

```
mkdir /temp/data_files
```

2. **Create a DIRECTORY object by using the CREATE DIRECTORY command:**

```
CREATE DIRECTORY data_files
           AS '/temp/data_files';
```

3. **Grant the READ privilege on the DIRECTORY object to appropriate users:**

```
GRANT READ ON DIRECTORY data_files
   TO SCOTT, MANAGER_ROLE, PUBLIC;
```

ORACLE

**Preparing to Use BFILES**

To use a BFILE within an Oracle table, you must have a table with a column of the BFILE type. For the Oracle server to access an external file, the server needs to know the physical location of the file in the OS directory structure.

The database DIRECTORY object provides the means to specify the location of the BFILES. Use the CREATE DIRECTORY command to specify the pointer to the location where your BFILEs are stored. You need the CREATE ANY DIRECTORY privilege.

**Syntax definition:** CREATE DIRECTORY *dir_name* AS *os_path*;

In this syntax, *dir_name* is the name of the directory database object, and *os_path* is the location of the BFILEs.

The slide examples show the commands to set up:
- The physical directory (for example /temp/data_files) in the OS
- A named DIRECTORY object, called data_files, that points to the physical directory in the OS
- The READ access right on the directory to be granted to users in the database, providing the privilege to read the BFILEs from the directory

**Note:** The value of the SESSION_MAX_OPEN_FILES database initialization parameter, which is set to 10 by default, limits the number of BFILEs that can be opened in a session.

**Oracle Database 10*g*: Develop PL/SQL Program Units   9-14**

# Populating BFILE Columns with SQL

- **Use the BFILENAME function to initialize a BFILE column. The function syntax is:**

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURN BFILE;
```

- **Example:**
  - **Add a BFILE column to a table:**

```
ALTER TABLE employees ADD video BFILE;
```

  - **Update the column using the BFILENAME function:**

```
UPDATE employees
  SET video = BFILENAME('DATA_FILES', 'King.avi')
WHERE employee_id = 100;
```

### Populating BFILE Columns with SQL

The BFILENAME function is a built-in function that you use to initialize a BFILE column, using the following two parameters:

- *directory_alias* for the name of the DIRECTORY database object that references the OS directory containing the files
- *filename* for the name of the BFILE to be read

The BFILENAME function creates a pointer (or LOB locator) to the external file stored in a physical directory, which is assigned a directory alias name that is used in the first parameter of the function. Populate the BFILE column using the BFILENAME function in either of the following:

- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

An UPDATE operation can be used to change the pointer reference target of the BFILE. A BFILE column can also be initialized to a NULL value and updated later with the BFILENAME function, as shown in the slide.

After the BFILE columns have been associated with a file, subsequent read operations on the BFILE can be performed by using the PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILEs. Therefore, they cannot be updated or deleted through BFILEs.

# Populating a `BFILE` Column with PL/SQL

```
CREATE PROCEDURE set_video(
  dir_alias VARCHAR2, dept_id NUMBER) IS
  filename VARCHAR2(40);
  file_ptr BFILE;
  CURSOR emp_csr IS
    SELECT first_name FROM employees
    WHERE department_id = dept_id FOR UPDATE;
BEGIN
  FOR rec IN emp_csr LOOP
    filename := rec.first_name || '.gif';
    file_ptr := BFILENAME(dir_alias, filename);
    DBMS_LOB.FILEOPEN(file_ptr);
    UPDATE employees SET video = file_ptr
      WHERE CURRENT OF emp_csr;
    DBMS_OUTPUT.PUT_LINE('FILE: ' || filename ||
    ' SIZE: ' || DBMS_LOB.GETLENGTH(file_ptr));
    DBMS_LOB.FILECLOSE(file_ptr);
  END LOOP;
END set_video;
```

**Populating a `BFILE` Column with PL/SQL**

The example shows a PL/SQL procedure called `set_video`, which accepts the name of the directory alias referencing the OS file system as a parameter, and a department ID. The procedure performs the following tasks:

- Uses a cursor `FOR` loop to obtain each employee record
- Sets the `filename` by appending `.gif` to the employee's `first_name`
- Creates an in-memory LOB locator for the `BFILE` in the `file_ptr` variable
- Calls the `DBMS_LOB.FILEOPEN` procedure to verify whether the file exists, and to determine the size of the file using the `DBMS_LOB.GETLENGTH` function
- Executes an `UPDATE` statement to write the `BFILE` locator value to the `video` `BFILE` column
- Displays the file size returned from the `DBMS_LOB.GETLENGTH` function
- Closes the file using the `DBMS_LOB.FILECLOSE` procedure

Suppose that you execute the following call:

```
EXECUTE set_video('DATA_FILE', 60)
```

Sample results are:

```
FILE: Alexander.gif SIZE: 5213
FILE: Bruce.gif SIZE: 26059
  :
```

# Using `DBMS_LOB` Routines with `BFILE`s

The `DBMS_LOB.FILEEXISTS` function can check whether the file exists in the OS. The function:
- **Returns 0 if the file does not exist**
- **Returns 1 if the file does exist**

```
CREATE FUNCTION get_filesize(file_ptr IN OUT BFILE)
RETURN NUMBER IS
  file_exists BOOLEAN;
  length NUMBER:= -1;
BEGIN
  file_exists := DBMS_LOB.FILEEXISTS(file_ptr)=1;
  IF file_exists THEN
    DBMS_LOB.FILEOPEN(file_ptr);
    length := DBMS_LOB.GETLENGTH(file_ptr);
    DBMS_LOB.FILECLOSE(file_ptr);
  END IF;
  RETURN length;
END;
/
```

## Using `DBMS_LOB` Routines with `BFILE`s

The `set_video` procedure on the previous page will terminate with an exception if a file does not exist. To prevent the loop from prematurely terminating, you could create a function, such as `get_filesize`, to determine whether a given `BFILE` locator references a file that actually exists on the server's file system. The `DBMS_LOB.FILEEXISTS` function expects the `BFILE` locator as a parameter and returns an `INTEGER` with:
- A value 0 if the physical file does not exist
- A value 1 if the physical file exists

If the `BFILE` parameter is invalid, one of the three exceptions may be raised:
- `NOEXIST_DIRECTORY` if the directory does not exist
- `NOPRIV_DIRECTORY` if database processes do not have privileges for the directory
- `INVALID_DIRECTORY` if the directory was invalidated after the file was opened

In the `get_filesize` function, the output of the `DBMS_LOB.FILEEXISTS` function is compared with value 1 and the result of the condition sets the `BOOLEAN` variable `file_exists`. The `DBMS_LOB.FILEOPEN` call is performed only if the file does exist, preventing unwanted exceptions from occurring. The `get_filesize` function returns a value of −1 if a file does not exist; otherwise, it returns the size of the file in bytes. The caller can take appropriate action with this information.

# Migrating from LONG to LOB

**Oracle Database 10*g* enables the migration of LONG columns to LOB columns.**

- **Data migration consists of the procedure to move existing tables containing LONG columns to use LOBs:**

```
ALTER TABLE [<schema>.] <table_name>
   MODIFY (<long_col_name> {CLOB | BLOB | NCLOB})
```

- **Application migration consists of changing existing LONG applications for using LOBs.**

### Migrating from LONG to LOB

Oracle Database 10*g* supports LONG-to-LOB migration using an API. In data migration, existing tables that contain LONG columns need to be moved to use LOB columns. This can be done by using the ALTER TABLE command. In Oracle8*i*, an operator named TO_LOB had to be used to copy a LONG to a LOB. In Oracle Database 10*g*, this operation can be performed using the syntax shown in the slide. You can use the syntax shown to:

- Modify a LONG column to a CLOB or an NCLOB column
- Modify a LONG RAW column to a BLOB column

The constraints of the LONG column (NULL and NOT NULL are the only allowed constraints) are maintained for the new LOB columns. The default value specified for the LONG column is also copied to the new LOB column. For example, suppose you have the following table:

```
CREATE TABLE long_tab (id NUMBER, long_col LONG);
```

To change the long_col column in the long_tab table to the CLOB data type, use:

```
ALTER TABLE long_tab MODIFY ( long_col CLOB );
```

For information about the limitations on LONG-to-LOB migration, refer to the *Oracle Database Application Developer's Guide - Large Objects*. In application migration, the existing LONG applications change for using LOBs. You can use SQL and PL/SQL to access LONGs and LOBs. The LONG-to-LOB migration API is provided for both OCI and PL/SQL.

**Oracle Database 10*g*: Develop PL/SQL Program Units   9-18**

# Migrating from LONG to LOB

- **Implicit conversion: From** LONG (LONG RAW) **or a** VARCHAR2(RAW) **variable to a** CLOB (BLOB) **variable, and vice versa**
- **Explicit conversion:**
  - TO_CLOB() **converts** LONG, VARCHAR2, **and** CHAR **to** CLOB.
  - TO_BLOB() **converts** LONG RAW **and** RAW **to** BLOB.
- **Function and procedure parameter passing:**
  - CLOB**s and** BLOB**s are passed as actual parameters**
  - VARCHAR2, LONG, RAW, **and** LONG RAW **are formal parameters, and vice versa.**
- LOB **data is acceptable in most of the SQL and PL/SQL operators and built-in functions.**

ORACLE

**Migrating from LONG to LOB (continued)**

With the new LONG-to-LOB API introduced in Oracle Database 10*g*, data from CLOB and BLOB columns can be referenced by regular SQL and PL/SQL statements.

**Implicit assignment and parameter passing:** The LONG-to-LOB migration API supports assigning a CLOB (BLOB) variable to a LONG (LONG RAW) or a VARCHAR2(RAW) variable, and vice versa.

**Explicit conversion functions:** In PL/SQL, the following two new explicit conversion functions have been added in Oracle Database 10*g* to convert other data types to CLOB and BLOB as part of the LONG-to-LOB migration:

- TO_CLOB() converts LONG, VARCHAR2, and CHAR to CLOB
- TO_BLOB() converts LONG RAW and RAW to BLOB

**Note:** TO_CHAR() is enabled to convert a CLOB to a CHAR type.

**Function and procedure parameter passing:** This enables the use of CLOBs and BLOBs as actual parameters where VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa. In SQL and PL/SQL built-in functions and operators, a CLOB can be passed to SQL and PL/SQL VARCHAR2 built-in functions, behaving exactly like a VARCHAR2. Or, the VARCHAR2 variable can be passed into DBMS_LOB APIs acting like a LOB locator.

# DBMS_LOB **Package**

- **Working with** LOBs **often requires the use of the Oracle-supplied** DBMS_LOB **package.**
- DBMS_LOB **provides routines to access and manipulate internal and external** LOBs.
- **Oracle Database 10***g* **enables retrieving** LOB **data directly using SQL without a special** LOB **API.**
- **In PL/SQL, you can define a** VARCHAR2 **for a** CLOB **and a** RAW **for a** BLOB.

ORACLE

## DBMS_LOB **Package**

In releases prior to Oracle9*i*, you must use the DBMS_LOB package for retrieving data from LOBs. To create the DBMS_LOB package, the dbmslob.sql and prvtlob.plb scripts must be executed as SYS. The catproc.sql script executes the scripts. Then users can be granted appropriate privileges to use the package.

The package does not support any concurrency control mechanism for BFILE operations. The user is responsible for locking the row containing the destination internal LOB before calling any subprograms that involve writing to the LOB value. These DBMS_LOB routines do not implicitly lock the row containing the LOB.

The two constants, LOBMAXSIZE and FILE_READONLY, defined in the package specification are also used in the procedures and functions of DBMS_LOB—for example, use them to achieve the maximum level of purity to be used in SQL expressions.

The DBMS_LOB functions and procedures can be broadly classified into two types: mutators and observers.

- The mutators can modify LOB values: APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL, and FILEOPEN.
- The observers can read LOB values: COMPARE, FILEGETNAME, INSTR, GETLENGTH, READ, SUBSTR, FILEEXISTS, and FILEISOPEN.

**Oracle Database 10***g*: **Develop PL/SQL Program Units   9-20**

# DBMS_LOB Package

- **Modify LOB values:**
  APPEND, COPY, ERASE, TRIM, WRITE,
  LOADFROMFILE
- **Read or examine LOB values:**
  GETLENGTH, INSTR, READ, SUBSTR
- **Specific to BFILEs:**
  FILECLOSE, FILECLOSEALL, FILEEXISTS,
  FILEGETNAME, FILEISOPEN, FILEOPEN

ORACLE

## DBMS_LOB Package (continued)

| APPEND | Appends the contents of the source LOB to the destination LOB |
|---|---|
| COPY | Copies all or part of the source LOB to the destination LOB |
| ERASE | Erases all or part of a LOB |
| LOADFROMFILE | Loads BFILE data into an internal LOB |
| TRIM | Trims the LOB value to a specified shorter length |
| WRITE | Writes data to the LOB from a specified offset |
| GETLENGTH | Gets the length of the LOB value |
| INSTR | Returns the matching position of the *n*th occurrence of the pattern in the LOB |
| READ | Reads data from the LOB starting at the specified offset |
| SUBSTR | Returns part of the LOB value starting at the specified offset |
| FILECLOSE | Closes the file |
| FILECLOSEALL | Closes all previously opened files |
| FILEEXISTS | Checks whether the file exists on the server |
| FILEGETNAME | Gets the directory alias and file name |
| FILEISOPEN | Checks whether the file was opened using the input BFILE locators |
| FILEOPEN | Opens a file |

**Oracle Database 10g: Develop PL/SQL Program Units   9-21**

# DBMS_LOB Package

- **NULL parameters get NULL returns.**
- **Offsets:**
  - **BLOB, BFILE: Measured in bytes**
  - **CLOB, NCLOB: Measured in characters**
- **There are no negative values for parameters.**

## Using the DBMS_LOB Routines

All functions in the DBMS_LOB package return NULL if any input parameters are NULL. All mutator procedures in the DBMS_LOB package raise an exception if the destination LOB/BFILE is input as NULL.

Only positive, absolute offsets are allowed. They represent the number of bytes or characters from the beginning of LOB data from which to start the operation. Negative offsets and ranges observed in SQL string functions and operators are not allowed. Corresponding exceptions are raised upon violation. The default value for an offset is 1, which indicates the first byte or character in the LOB value.

Similarly, only natural number values are allowed for the amount (BUFSIZ) parameter. Negative values are not allowed.

# DBMS_LOB.READ and DBMS_LOB.WRITE

```
PROCEDURE READ (
  lobsrc IN BFILE|BLOB|CLOB ,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER,
  buffer OUT RAW|VARCHAR2 )
```

```
PROCEDURE WRITE (
  lobdst IN OUT BLOB|CLOB,
  amount IN OUT BINARY_INTEGER,
  offset IN INTEGER := 1,
  buffer IN RAW|VARCHAR2 )   -- RAW for BLOB
```

ORACLE

### DBMS_LOB.READ

Call the READ procedure to read and return piecewise a specified AMOUNT of data from a given LOB, starting from OFFSET. An exception is raised when no more data remains to be read from the source LOB. The value returned in AMOUNT is less than the one specified if the end of the LOB is reached before the specified number of bytes or characters can be read. In the case of CLOBs, the character set of data in BUFFER is the same as that in the LOB.

PL/SQL allows a maximum length of 32,767 for RAW and VARCHAR2 parameters. Ensure that the allocated system resources are adequate to support buffer sizes for the given number of user sessions. Otherwise, the Oracle server raises the appropriate memory exceptions.

**Note:** BLOB and BFILE return RAW; the others return VARCHAR2.

### DBMS_LOB.WRITE

Call the WRITE procedure to write piecewise a specified AMOUNT of data into a given LOB, from the user-specified BUFFER, starting from an absolute OFFSET from the beginning of the LOB value.

Make sure (especially with multibyte characters) that the amount in bytes corresponds to the amount of buffer data. WRITE has no means of checking whether they match, and it will write AMOUNT bytes of the buffer contents into the LOB.

# Initializing LOB Columns Added to a Table

- **Create the table with columns using the LOB type, or add the LOB columns using ALTER TABLE.**

```
ALTER TABLE employees
    ADD (resume CLOB, picture BLOB);
```

- **Initialize the column LOB locator value with the DEFAULT option or DML statements using:**
  - **EMPTY_CLOB() function for a CLOB column**
  - **EMPTY_BLOB() function for a BLOB column**

```
CREATE TABLE emp_hiredata (
    employee_id   NUMBER(6),
    full_name     VARCHAR2(45),
    resume        CLOB DEFAULT EMPTY_CLOB(),
    picture       BLOB DEFAULT EMPTY_BLOB());
```

ORACLE

**Initializing LOB Columns Added to a Table**

LOB columns are defined by using SQL data definition language (DDL), as in the ALTER TABLE statement in the slide. The contents of a LOB column are stored in the LOB segment, whereas the column in the table contains only a reference to that specific storage area, called the LOB locator. In PL/SQL, you can define a variable of the LOB type, which contains only the value of the LOB locator. You can initialize the LOB locators using:

- EMPTY_CLOB() function to a LOB locator for a CLOB column
- EMPTY_BLOB() function to a LOB locator for a BLOB column

**Note:** These functions create the LOB locator value and not the LOB content. In general, you use the DBMS_LOB package subroutines to populate the content. The functions are available in Oracle SQL DML, and are not part of the DBMS_LOB package.

The last example in the slide shows how you can use the EMPTY_CLOB() and EMPTY_BLOB() functions in the DEFAULT option in a CREATE TABLE statement. In this way, the LOB locator values are populated in their respective columns when a row is inserted into the table and the LOB columns have not been specified in the INSERT statement.

The next page shows how to use the functions in INSERT and UPDATE statements to initialize the LOB locator values.

# Populating LOB Columns

- **Insert a row into a table with LOB columns:**

```
INSERT INTO emp_hiredata
  (employee_id, full_name, resume, picture)
VALUES (405, 'Marvin Ellis', EMPTY_CLOB(), NULL);
```

- **Initialize a LOB using the EMPTY_BLOB() function:**

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
 WHERE employee_id = 405;
```

- **Update a CLOB column:**

```
UPDATE emp_hiredata
 SET resume = 'Date of Birth: 1 June 1956'
 WHERE employee_id = 170;
```

## Populating LOB Columns

You can insert a value directly into a LOB column by using host variables in SQL or in PL/SQL, 3GL-embedded SQL, or OCI. You can use the special EMPTY_BLOB() and EMPTY_CLOB() functions in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NULL internal LOB to empty. To populate a LOB column, perform the following steps:

1. Initialize the LOB column to a non-NULL value—that is, set a LOB locator pointing to an empty or populated LOB value. This is done by using EMPTY_BLOB() and EMPTY_CLOB() functions.
2. Populate the LOB contents by using the DBMS_LOB package routines.

However, as shown in the slide examples, the two UPDATE statements initialize the resume LOB locator value and populate its contents by supplying a literal value. This can also be done in an INSERT statement. A LOB column can be updated to:

- Another LOB value
- A NULL value
- A LOB locator with empty contents by using the EMPTY_*LOB() built-in function

You can update the LOB by using a bind variable in embedded SQL. When assigning one LOB to another, a new copy of the LOB value is created. Use a SELECT FOR UPDATE statement to lock the row containing the LOB column before updating a piece of the LOB contents.

# Updating LOB by Using DBMS_LOB in PL/SQL

```
DECLARE
  lobloc CLOB;        -- serves as the LOB locator
  text   VARCHAR2(50) := 'Resigned = 5 June 2000';
  amount NUMBER ;     -- amount to be written
  offset INTEGER;     -- where to start writing
BEGIN
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 405 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 2;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text);
  text := ' Resigned = 30 September 2000';
  SELECT resume INTO lobloc FROM emp_hiredata
  WHERE employee_id = 170 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
  COMMIT;
END;
```

## Updating LOB by Using DBMS_LOB in PL/SQL

In the example in the slide, the LOBLOC variable serves as the LOB locator, and the AMOUNT variable is set to the length of the text you want to add. The SELECT FOR UPDATE statement locks the row and returns the LOB locator for the RESUME LOB column. Finally, the PL/SQL WRITE package procedure is called to write the text into the LOB value at the specified offset. WRITEAPPEND appends to the existing LOB value.

The example shows how to fetch a CLOB column in releases before Oracle9*i*. In those releases, it was not possible to fetch a CLOB column directly into a character column. The column value needed to be bound to a LOB locator, which is accessed by the DBMS_LOB package. An example later in this lesson shows that you can directly fetch a CLOB column by binding it to a character variable.

**Note:** Versions prior to Oracle9*i* did not allow LOBs in the WHERE clause of UPDATE and SELECT statements. Now SQL functions of LOBs are allowed in predicates of WHERE. An example is shown later in this lesson.

# Selecting CLOB Values by Using SQL

```
SELECT employee_id, full_name , resume -- CLOB
FROM emp_hiredata
WHERE employee_id IN (405, 170);
```

| EMPLOYEE_ID | FULL_NAME | RESUME |
|---|---|---|
| 405 | Marvin Ellis | Date of Birth: 8 February 1951 Resigned = 5 June 2000 |
| 170 | Joe Fox | Date of Birth: 1 June 1956 Resigned = 30 September 2000 |

### Selecting CLOB Values by Using SQL

It is possible to see the data in a CLOB column by using a SELECT statement. It is not possible to see the data in a BLOB or BFILE column by using a SELECT statement in *i*SQL*Plus. You have to use a tool that can display binary information for a BLOB, as well as the relevant software for a BFILE—for example, you can use Oracle Forms.

# Selecting CLOB Values by Using DBMS_LOB

- **DBMS_LOB.SUBSTR (lob, amount, start_pos)**
- **DBMS_LOB.INSTR (lob, pattern)**

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume,' = ')
FROM   emp_hiredata
WHERE  employee_id IN (170, 405);
```

| DBMS_LOB.SUBSTR(RESUME,5,18) | DBMS_LOB.INSTR(RESUME,'=') |
|---|---|
| Febru | 40 |
| June | 36 |

ORACLE

### Selecting CLOB Values by Using DBMS_LOB

**DBMS_LOB.SUBSTR**

Use DBMS_LOB.SUBSTR to display a part of a LOB. It is similar in functionality to the SUBSTR SQL function.

**DBMS_LOB.INSTR**

Use DBMS_LOB.INSTR to search for information within the LOB. This function returns the numerical position of the information.

**Note:** Starting with Oracle9*i*, you can also use the SUBSTR and INSTR SQL functions to perform the operations shown in the slide.

# Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT WORD_WRAP
DECLARE
  text VARCHAR2(4001);
BEGIN
 SELECT resume INTO text
 FROM emp_hiredata
 WHERE employee_id = 170;
 DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```

```
text is: Date of Birth: 1 June 1956 Resigned = 30
September 2000

PL/SQL procedure successfully completed.
```

### Selecting CLOB Values in PL/SQL

The slide shows the code for accessing CLOB values that can be implicitly converted to VARCHAR2 in Oracle10g. When selected, the RESUME column value is implicitly converted from a CLOB into a VARCHAR2 to be stored in the TEXT variable. Prior to Oracle9i, you first retrieved the CLOB locator value into a CLOB variable, and then read the LOB contents specifying the amount and offset in the DBMS_LOB.READ procedure:

```
DECLARE
   rlob   CLOB;
   text   VARCHAR2(4001);
   amt    NUMBER := 4001;
   offset NUMBER := 1;
BEGIN
 SELECT resume INTO rlob FROM emp_hirdata
 WHERE employee_id = 170;
 DBMS_LOB.READ(rlob, amt, offset, text);
 DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
text is: Date of Birth: 1 June 1956 Resigned = 30 September 2000
PL/SQL procedure successfully completed.
```

# Removing LOBs

- **Delete a row containing LOBs:**

```
DELETE
FROM   emp_hiredata
WHERE employee_id = 405;
```

- **Disassociate a LOB value from a row:**

```
UPDATE emp_hiredata
SET resume = EMPTY_CLOB()
WHERE employee_id = 170;
```

**Removing LOBs**

A LOB instance can be deleted (destroyed) using appropriate SQL DML statements. The SQL statement DELETE deletes a row and its associated internal LOB value. To preserve the row and destroy only the reference to the LOB, you must update the row by replacing the LOB column value with NULL or an empty string, or by using the EMPTY_B/CLOB() function.

**Note:** Replacing a column value with NULL and using EMPTY_B/CLOB are not the same. Using NULL sets the value to null; using EMPTY_B/CLOB ensures that there is nothing in the column.

A LOB is destroyed when the row containing the LOB column is deleted, when the table is dropped or truncated, or when all the LOB data is updated.

You must explicitly remove the file associated with a BFILE using OS commands.

To erase part of an internal LOB, you can use DBMS_LOB.ERASE.

# Temporary LOBs

- **Temporary LOBs:**
  - **Provide an interface to support creation of LOBs that act like local variables**
  - **Can be BLOBs, CLOBs, or NCLOBs**
  - **Are not associated with a specific table**
  - **Are created using the DBMS_LOB.CREATETEMPORARY procedure**
  - **Use DBMS_LOB routines**
- **The lifetime of a temporary LOB is a session.**
- **Temporary LOBs are useful for transforming data in permanent internal LOBs.**

**Temporary LOBs**

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables. Temporary LOBs can be BLOBs, CLOBs, or NCLOBs.

The following are the features of temporary LOBs:
- Data is stored in your temporary tablespace, not in tables.
- Temporary LOBs are faster than persistent LOBs because they do not generate any redo or rollback information.
- Temporary LOBs lookup is localized to each user's own session. Only the user who creates a temporary LOB can access it, and all temporary LOBs are deleted at the end of the session in which they were created.
- You can create a temporary LOB using DBMS_LOB.CREATETEMPORARY.

Temporary LOBs are useful when you want to perform some transformational operation on a LOB (for example, changing an image type from GIF to JPEG). A temporary LOB is empty when created and does not support the EMPTY_B/CLOB functions.

Use the DBMS_LOB package to use and manipulate temporary LOBs.

# Creating a Temporary LOB

**PL/SQL procedure to create and test a temporary LOB:**

```
CREATE OR REPLACE PROCEDURE is_templob_open(
  lob IN OUT BLOB, retval OUT INTEGER) IS
BEGIN
  -- create a temporary LOB
  DBMS_LOB.CREATETEMPORARY (lob, TRUE);
  -- see if the LOB is open: returns 1 if open
  retval := DBMS_LOB.ISOPEN (lob);
  DBMS_OUTPUT.PUT_LINE (
    'The file returned a value...' || retval);
  -- free the temporary LOB
  DBMS_LOB.FREETEMPORARY (lob);
END;
/
```

### Creating a Temporary LOB

The example in the slide shows a user-defined PL/SQL procedure, is_templob_open, which creates a temporary LOB. This procedure accepts a LOB locator as input, creates a temporary LOB, opens it, and tests whether the LOB is open.

The is_templob_open procedure uses the procedures and functions from the DBMS_LOB package as follows:

- The CREATETEMPORARY procedure is used to create the temporary LOB.
- The ISOPEN function is used to test whether a LOB is open: this function returns the value 1 if the LOB is open.
- The FREETEMPORARY procedure is used to free the temporary LOB. Memory increases incrementally as the number of temporary LOBs grows, and you can reuse temporary LOB space in your session by explicitly freeing temporary LOBs.

# Summary

**In this lesson, you should have learned how to:**

- **Identify four built-in types for large objects: BLOB, CLOB, NCLOB, and BFILE**
- **Describe how LOBs replace LONG and LONG RAW**
- **Describe two storage options for LOBs:**
  - **Oracle server (internal LOBs)**
  - **External host files (external LOBs)**
- **Use the DBMS_LOB PL/SQL package to provide routines for LOB management**
- **Use temporary LOBs in a session**

ORACLE

**Summary**

There are four LOB data types:
- A BLOB is a binary large object.
- A CLOB is a character large object.
- An NCLOB stores multibyte national character set data.
- A BFILE is a large object stored in a binary file outside the database.

LOBs can be stored internally (in the database) or externally (in an OS file). You can manage LOBs by using the DBMS_LOB package and its procedure.

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables.

# Practice 9: Overview

**This practice covers the following topics:**
- **Creating object types using the CLOB and BLOB data types**
- **Creating a table with LOB data types as columns**
- **Using the DBMS_LOB package to populate and interact with the LOB data**

**Practice 9: Overview**

In this practice, you create a table with both BLOB and CLOB columns. Then you use the DBMS_LOB package to populate the table and manipulate the data.

## Practice 9

1. Create a table called `PERSONNEL` by executing the script file
   `E:\labs\PLPU\labs\lab_09_01.sql`. The table contains the following attributes
   and data types:

| Column Name | Data Type | Length |
|---|---|---|
| ID | NUMBER | 6 |
| last_name | VARCHAR2 | 35 |
| review | CLOB | N/A |
| picture | BLOB | N/A |

2. Insert two rows into the `PERSONNEL` table, one each for employee `2034` (whose last
   name is `Allen`) and for employee `2035` (whose last name is `Bond`). Use the empty
   function for the `CLOB`, and provide `NULL` as the value for the `BLOB`.
3. Examine and execute the `E:\labs\PLPU\labs\lab_09_03.sql` script. The script
   creates a table named `REVIEW_TABLE`. This table contains annual review information
   for each employee. The script also contains two statements to insert review details for two
   employees.
4. Update the `PERSONNEL` table.
   a. Populate the `CLOB` for the first row, using this subquery in an `UPDATE` statement:
      ```
      SELECT  ann_review
      FROM    review_table
      WHERE   employee_id = 2034;
      ```
   b. Populate the `CLOB` for the second row, using PL/SQL and the `DBMS_LOB` package.
      Use the following `SELECT` statement to provide a value for the `LOB` locator.
      ```
      SELECT  ann_review
      FROM    review_table
      WHERE   employee_id = 2035;
      ```

**If you have time, complete the following exercise:**

5. Create a procedure that adds a locator to a binary file into the `PICTURE` column of the
   `COUNTRIES` table. The binary file is a picture of the country flag. The image files are
   named after the country IDs. You need to load an image file locator into all rows in the
   Europe region (`REGION_ID = 1`) in the `COUNTRIES` table. A `DIRECTORY` object called
   `COUNTRY_PIC` referencing the location of the binary files has to be created for you.
   a. Add the image column to the `COUNTRIES` table using:
      ```
      ALTER TABLE countries ADD (picture BFILE);
      ```
      Alternatively, use the `E:\labs\PLPU\labs\Lab_09_05_a.sql` file.
   b. Create a PL/SQL procedure called `load_country_image` that uses
      `DBMS_LOB.FILEEXISTS` to test whether the country picture file exists. If the file
      exists, then set the `BFILE` locator for the file in the `PICTURE` column; otherwise,
      display a message that the file does not exist. Use the `DBMS_OUTPUT` package to
      report file size information for each image associated with the `PICTURE` column.
   c. Invoke the procedure by passing the name of the directory object `COUNTRY_PIC` as
      a string literal parameter value.

**Oracle Database 10*g*: Develop PL/SQL Program Units   9-35**