
Oracle9i: Develop PL/SQL Program Units

Student Guide • Volume 1

40056GC10
Production 1.0
July 2001
D33490

ORACLE®

Author

Nagavalli Pataballa

Technical Contributors and Reviewers

Anna Atkinson
Bryan Roberts
Caroline Pereda
Cesljas Zarco
Coley William
Daniel Gabel
Dr. Christoph Burandt
Hakan Lindfors
Helen Robertson
John Hoff
Lachlan Williams
Laszlo Czinkoczki
Laura Pezzini
Linda Boldt
Marco Verbeek
Natarajan Senthil
Priya Vennapusa
Roger Abuzalaf
Ruediger Steffan
Sarah Jones
Stefan Lindblad
Susan Dee

Publisher

Sheryl Domingue

Copyright © Oracle Corporation, 1999, 2000, 2001. All rights reserved.

This documentation contains proprietary information of Oracle Corporation. It is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited. If this documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions for commercial computer software and shall be deemed to be Restricted Rights software under Federal law, as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

This material or any portion of it may not be copied in any form or by any means without the express prior written permission of Oracle Corporation. Any other copying is a violation of copyright law and may result in civil and/or criminal penalties.

If this documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights," as defined in FAR 52.227-14, Rights in Data-General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them in writing to Education Products, Oracle Corporation, 500 Oracle Parkway, Box SB-6, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

All references to Oracle and Oracle products are trademarks or registered trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

Preface

Curriculum Map

1 Overview of PL/SQL Subprograms

- Course Objectives 1-2
- Lesson Objectives 1-3
- Oracle Internet Platform 1-4
- PL/SQL Program Constructs 1-5
- Overview of Subprograms 1-6
- Block Structure for Anonymous PL/SQL Blocks 1-7
- Block Structure for PL/SQL Subprograms 1-8
- PL/SQL Subprograms 1-9
- Benefits of Subprograms 1-10
- Developing Subprograms by Using iSQL*Plus 1-11
- Invoking Stored Procedures and Functions 1-12
- Summary 1-13

2 Creating Procedures

- Objectives 2-2
- What Is a Procedure? 2-3
- Syntax for Creating Procedures 2-4
- Developing Procedures 2-5
- Formal Versus Actual Parameters 2-6
- Procedural Parameter Modes 2-7
- Creating Procedures with Parameters 2-8
- IN Parameters: Example 2-9
- OUT Parameters: Example 2-10
- Viewing OUT Parameters 2-12
- IN OUT Parameters 2-13
- Viewing IN OUT Parameters 2-14
- Methods for Passing Parameters 2-15
- DEFAULT Option for Parameters 2-16
- Examples of Passing Parameters 2-17
- Declaring Subprograms 2-18
- Invoking a Procedure from an Anonymous PL/SQL Block 2-19
- Invoking a Procedure from Another Procedure 2-20
- Handled Exceptions 2-21
- Unhandled Exceptions 2-23
- Removing Procedures 2-25
- Benefits of Subprograms 2-26
- Summary 2-27
- Practice 2 Overview 2-29

3 Creating Functions

- Objectives 3-2
- Overview of Stored Functions 3-3
- Syntax for Creating Functions 3-4
- Creating a Function 3-5
- Creating a Stored Function by Using iSQL*Plus 3-6
- Creating a Stored Function by Using iSQL*Plus: Example 3-7
- Executing Functions 3-8
- Executing Functions: Example 3-9
- Advantages of User-Defined Functions in SQL Expressions 3-10
- Invoking Functions in SQL Expressions: Example 3-11
- Locations to Call User-Defined Functions 3-12
- Restrictions on Calling Functions from SQL Expressions 3-13
- Restrictions on Calling from SQL 3-15
- Removing Functions 3-16
- Procedure or Function? 3-17
- Comparing Procedures and Functions 3-18
- Benefits of Stored Procedures and Functions 3-19
- Summary 3-20
- Practice 3 Overview 3-21

4 Managing Subprograms

- Objectives 4-2
- Required Privileges 4-3
- Granting Access to Data 4-4
- Using Invoker's-Rights 4-5
- Managing Stored PL/SQL Objects 4-6
- USER_OBJECTS 4-7
- List All Procedures and Functions 4-8
- USER_SOURCE Data Dictionary View 4-9
- List the Code of Procedures and Functions 4-10
- USER_ERRORS 4-11
- Detecting Compilation Errors: Example 4-12
- List Compilation Errors by Using USER_ERRORS 4-13
- List Compilation Errors by Using SHOW ERRORS 4-14
- DESCRIBE in iSQL*Plus 4-15
- Debugging PL/SQL Program Units 4-16
- Summary 4-17
- Practice 4 Overview 4-19

5 Creating Packages

- Objectives 5-2
- Overview of Packages 5-3
- Components of a Package 5-4
- Referencing Package Objects 5-5
- Developing a Package 5-6
- Creating the Package Specification 5-8
- Declaring Public Constructs 5-9
- Creating a Package Specification: Example 5-10
- Creating the Package Body 5-11
- Public and Private Constructs 5-12
- Creating a Package Body: Example 5-13
- Invoking Package Constructs 5-15
- Declaring a Bodiless Package 5-17
- Referencing a Public Variable from a Stand-Alone Procedure 5-18
- Removing Packages 5-19
- Guidelines for Developing Packages 5-20
- Advantages of Packages 5-21
- Summary 5-23
- Practice 5 Overview 5-26

6 More Package Concepts

- Objectives 6-2
- Overloading 6-3
- Overloading: Example 6-4
- Using Forward Declarations 6-7
- Creating a One-Time-Only Procedure 6-9
- Restrictions on Package Functions Used in SQL 6-10
- User Defined Package: taxes_pack 6-11
- Invoking a User-Defined Package Function from a SQL Statement 6-12
- Persistent State of Package Variables: Example 6-13
- Persistent State of Package Variables 6-15
- Controlling the Persistent State of a Package Cursor 6-15
- Executing PACK_CUR 6-17
- PL/SQL Tables and Records in Packages 6-18
- Summary 6-19
- Practice 6 Overview 6-20

7 Oracle Supplied Packages

- Objectives 7-2
- Using Supplied Packages 7-3
- Using Native Dynamic SQL 7-4
- Execution Flow 7-5
- Using the DBMS_SQL Package 7-6
- Using DBMS_SQL 7-8
- Using the EXECUTE IMMEDIATE Statement 7-9
- Dynamic SQL Using EXECUTE IMMEDIATE 7-11
- Using the DBMS_DDL Package 7-12
- Using DBMS_JOB for Scheduling 7-13
- DBMS_JOB Subprograms 7-14
- Submitting Jobs 7-15
- Changing Job Characteristics 7-17
- Running, Removing, and Breaking Jobs 7-18
- Viewing Information on Submitted Jobs 7-19
- Using the DBMS_OUTPUT Package 7-20
- Interacting with Operating System Files 7-21
- What Is the UTL_FILE Package? 7-22
- File Processing Using the UTL_FILE Package 7-23
- UTL_FILE Procedures and Functions 7-24
- Exceptions Specific to the UTL_FILE Package 7-25
- The FOPEN and IS_OPEN Functions 7-26
- Using UTL_FILE 7-27
- The UTL_HTTP Package 7-29
- Using the UTL_HTTP Package 7-30
- Using the UTL_TCP Package 7-31
- Oracle-Supplied Packages 7-32
- Summary 7-37
- Practice 7 Overview 7-38

8 Manipulating Large Objects

- Objectives 8-2
- What Is a LOB? 8-3
- Contrasting LONG and LOB Data Types 8-4
- Anatomy of a LOB 8-5
- Internal LOBs 8-6
- Managing Internal LOBs 8-7
- What Are BFILES? 8-8
- Securing BFILES 8-9
- A New Database Object: DIRECTORY 8-10
- Guidelines for Creating DIRECTORY Objects 8-11

- Managing BFILES 8-12
- Preparing to Use BFILES 8-13
- The BFILENAME Function 8-14
- Loading BFILES 8-15
- Migrating from LONG to LOB 8-17
- The DBMS_LOB Package 8-19
- DBMS_LOB.READ and DBMS_LOB.WRITE 8-22
- Adding LOB Columns to a Table 8-23
- Populating LOB Columns 8-24
- Updating LOB by Using SQL 8-26
- Updating LOB by Using DBMS_LOB in PL/SQL 8-27
- Selecting CLOB Values by Using SQL 8-28
- Selecting CLOB Values by Using DBMS_LOB 8-29
- Selecting CLOB Values in PL/SQL 8-30
- Removing LOBs 8-31
- Temporary LOBs 8-32
- Creating a Temporary LOB 8-33
- Summary 8-34
- Practice 8 Overview 8-35

9 Creating Database Triggers

- Objectives 9-2
- Types of Triggers 9-3
- Guidelines for Designing Triggers 9-4
- Database Trigger: Example 9-5
- Creating DML Triggers 9-6
- DML Trigger Components 9-7
- Firing Sequence 9-11
- Syntax for Creating DML Statement Triggers 9-13
- Creating DML Statement Triggers 9-14
- Testing SECURE_EMP 9-15
- Using Conditional Predicates 9-16
- Creating a DML Row Trigger 9-17
- Creating DML Row Triggers 9-18
- Using OLD and NEW Qualifiers 9-19
- Using OLD and NEW Qualifiers: Example Using Audit_Emp_Table 9-20
- Restricting a Row Trigger 9-21
- INSTEAD OF Triggers 9-22
- Creating an INSTEAD OF Trigger 9-23
- Creating an INSTEAD OF Trigger 9-26
- Differentiating Between Database Triggers and Stored Procedures 9-27
- Differentiating Between Database Triggers and Form Builder Triggers 9-28
- Managing Triggers 9-29
- DROP TRIGGER Syntax 9-30

Trigger Test Cases 9-31
Trigger Execution Model and Constraint Checking 9-32
Trigger Execution Model and Constraint Checking: Example 9-33
A Sample Demonstration for Triggers Using Package Constructs 9-34
After Row and After Statement Triggers 9-35
Demonstration: VAR_PACK Package Specification 9-36
Demonstration: Using the AUDIT_EMP Procedure 9-38
Summary 9-39
Practice 9 Overview 9-40

10 More Trigger Concepts

Objectives 10-2
Creating Database Triggers 10-3
Creating Triggers on DDL Statements 10-4
Creating Triggers on System Events 10-5
LOGON and LOGOFF Trigger Example 10-6
CALL Statements 10-7
Reading Data from a Mutating Table 10-8
Mutating Table: Example 10-9
Implementing Triggers 10-11
Controlling Security Within the Server 10-12
Controlling Security with a Database Trigger 10-13
Using the Server Facility to Audit Data Operations 10-14
Auditing by Using a Trigger 10-15
Enforcing Data Integrity Within the Server 10-16
Protecting Data Integrity with a Trigger 10-17
Enforcing Referential Integrity Within the Server 10-18
Protecting Referential Integrity with a Trigger 10-19
Replicating a Table Within the Server 10-20
Replicating a Table with a Trigger 10-21
Computing Derived Data within the Server 10-22
Computing Derived Values with a Trigger 10-23
Logging Events with a Trigger 10-24
Benefits of Database Triggers 10-26
Managing Triggers 10-27
Viewing Trigger Information 10-28
Using USER_TRIGGERS 10-29
Listing the Code of Triggers 10-30
Summary 10-31
Practice 10 Overview 10-32

11 Managing Dependencies

Objectives	11-2
Understanding Dependencies	11-3
Dependencies	11-4
Local Dependencies	11-5
A Scenario of Local Dependencies	11-6
Displaying Direct Dependencies by Using USER_DEPENDENCIES	11-7
Displaying Direct and Indirect Dependencies	11-8
Displaying Dependencies	11-9
Another Scenario of Local Dependencies	11-10
A Scenario of Local Naming Dependencies	11-11
Understanding Remote Dependencies	11-12
Concepts of Remote Dependencies	11-13
REMOTE_DEPENDENCIES_MODE Parameter	11-14
Remote Dependencies and Time Stamp Mode	11-15
Remote Procedure B Compiles at 8:00 a.m.	11-16
Local Procedure A Compiles at 9:00 a.m.	11-17
Execute Procedure A	11-18
Remote Procedure B Recompiled at 11:00 a.m.	11-19
Execute Procedure A	11-20
Signature Mode	11-21
Recompiling a PL/SQL Program Unit	11-22
Unsuccessful Recompile	11-23
Successful Recompile	11-24
Recompilation of Procedures	11-25
Packages and Dependencies	11-26
Summary	11-28
Practice 11 Overview	11-29

A PL/SQL Fundamentals Quiz

B PL/SQL Fundamentals Quiz Answers

C Practice Solutions

D Table Descriptions and Data

E Review of PL/SQL

F Creating Program Units by Using Procedure Builder

Index

Additional Practices

Additional Practice Solutions

Additional Practices: Table Descriptions and Data

Preface

Profile

Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL, *iSQL*Plus*, and working experience developing applications. Required prerequisites are *Introduction to Oracle9i: SQL*, or *Introduction to Oracle9i for Experienced SQL Users*, and *PL/SQL Fundamentals*.

How This Course Is Organized

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
<i>Oracle9i Application Developer's Guide-Fundamentals</i>	<i>A86797-01</i>
<i>Oracle9i Application Developer's Guide-Large Objects</i>	<i>A86800-01</i>
<i>Oracle9i Supplied PL/SQL Packages Reference</i>	<i>A86815-01</i>
<i>PL/SQL User's Guide and Reference, Release 8.1.6</i>	<i>A86811-01</i>

Additional Publications

- System release bulletins
- Installation and user's guides
- read.me files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

Following are two lists of typographical conventions that are used specifically within text or within code.

Typographic Conventions Within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, table names, PL/SQL objects, schemas	Use the <code>SELECT</code> command to view information stored in the <code>LAST_NAME</code> column of the <code>EMPLOYEES</code> table.
Lowercase,	Filenames, syntax variables, usernames, passwords	where: <i>role</i> is the name of the role italic to be created.
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject, see <i>Oracle8 Server SQL Language Reference Manual</i> . Do <i>not</i> save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

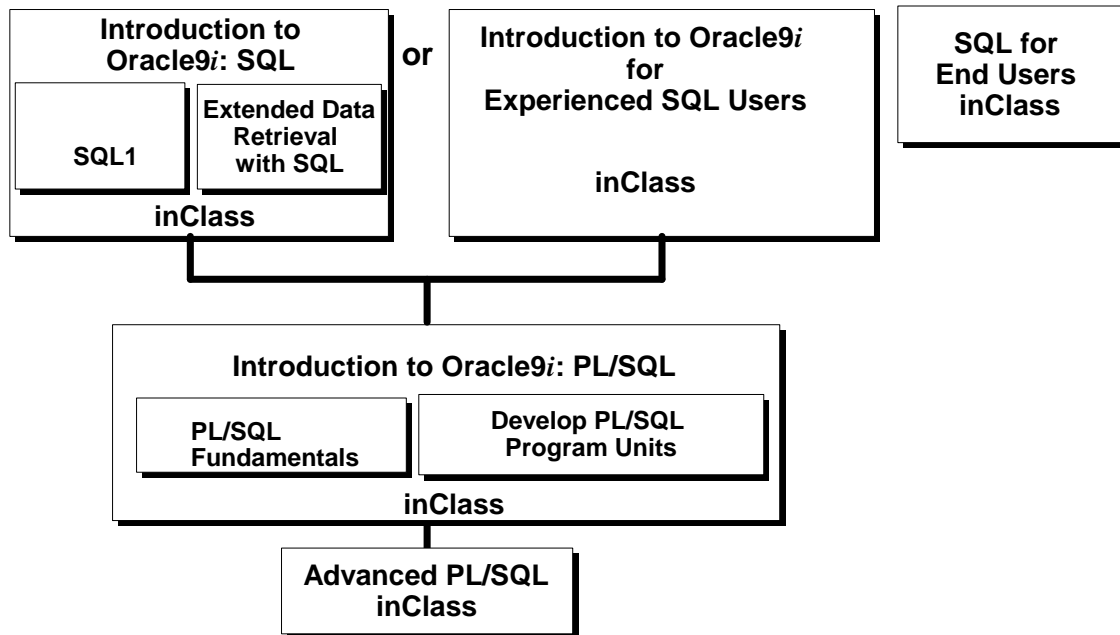
Typographic Conventions (continued)

Typographic Conventions Within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	SQL> SELECT userid 2 FROM emp;
Lowercase, italic	Syntax variables	SQL> CREATE ROLE <i>role</i> ;
Initial cap	Forms triggers	Form module: ORD Trigger level: S_ITEM.QUANTITY item Trigger name: When-Validate-Item . . .
Lowercase	Column names, table names, filenames, PL/SQL objects	. . . OG_ACTIVATE_LAYER (OG_GET_LAYER ('prod_pie_layer')) . . . SQL> SELECT last_name 2 FROM emp;
Bold	Text that must be entered by a user	SQLDBA> DROP USER scott 2> IDENTIFIED BY tiger;

Curriculum Map

Languages Curriculum for Oracle9i



ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Integrated Languages Curriculum

Introduction to Oracle9i: SQL consists of two modules, *SQL1* and *Extended Data Retrieval with SQL*. *SQL1* covers creating database structures and storing, retrieving, and manipulating data in a relational database. *Extended Data Retrieval with SQL* covers advanced SELECT statements, Oracle SQL and iSQL*Plus Reporting.

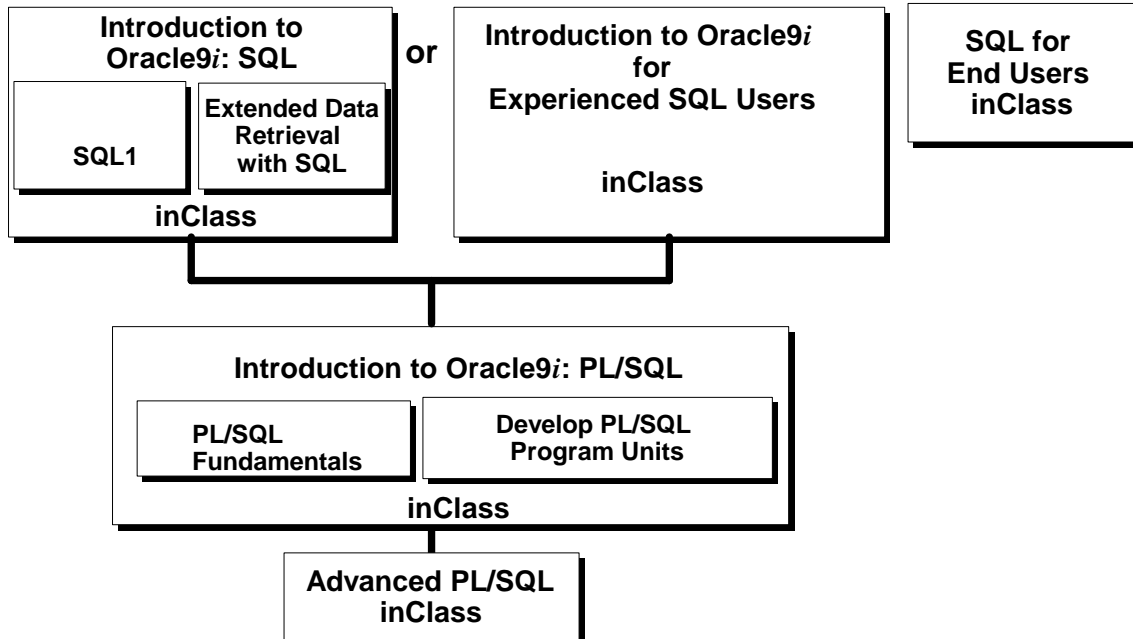
For people who have worked with other relational databases and have knowledge of SQL, another course, called *Introduction to Oracle9i for Experienced SQL Users* is offered. This course covers the SQL statements that are not part of ANSI SQL but are specific to Oracle.

Introduction to Oracle9i: PL/SQL consists of two modules, *PL/SQL Fundamentals* and *Develop PL/SQL Program Units*. *PL/SQL Fundamentals* covers PL/SQL basics including the PL/SQL language structure, flow of execution and interface with SQL. *Develop PL/SQL Program Units* covers how to create stored procedures, functions, packages, and triggers as well as maintain and debug program code.

SQL for End Users is directed towards individuals with little programming background and covers basic SQL statements. This course is for end users who need to know some basic SQL programming.

Advanced PL/SQL is appropriate for individuals who have experience in PL/SQL programming and covers coding efficiency topics, object-oriented programming, working with external code, and the advanced features of the Oracle supplied packages.

Languages Curriculum for Oracle9i



ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Integrated Languages Curriculum

The slide lists various modules and courses that are available in the languages curriculum. The following table lists the modules and courses with their equivalent TBTs.

Course or Module	Equivalent TBT
<i>SQL1</i>	<i>Oracle SQL: Basic SELECT Statements</i> <i>Oracle SQL: Data Retrieval Techniques</i> <i>Oracle SQL: DML and DDL</i>
<i>Extended Data Retrieval with SQL</i>	<i>Oracle SQL and SQL*Plus: Advanced SELECT Statements</i> <i>Oracle SQL and SQL*Plus: SQL*Plus and Reporting</i>
<i>Introduction to Oracle9i for Experienced SQL Users</i>	<i>Oracle SQL Specifics: Retrieving and Formatting Data</i> <i>Oracle SQL Specifics: Creating and Managing Database Objects</i>
<i>PL/SQL Fundamentals</i>	<i>PL/SQL: Basics</i>
<i>Develop PL/SQL Program Units</i>	<i>PL/SQL: Procedures, Functions, and Packages</i> <i>PL/SQL: Database Programming</i>
<i>SQL for End Users</i>	<i>SQL for End Users: Part 1</i> <i>SQL for End Users: Part 2</i>
<i>Advanced PL/SQL</i>	<i>Advanced PL/SQL: Implementation and Advanced Features</i> <i>Advanced PL/SQL: Design Considerations and Object Types</i>

1

Overview of PL/SQL Subprograms

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Course Objectives

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

- **Create, execute, and maintain procedures, functions, packages, and database triggers**
- **Manage PL/SQL subprograms and triggers**
- **Describe Oracle-supplied packages**
- **Manipulate large objects (LOBs)**

ORACLE



Course Overview

You can develop modularized applications with database procedures, using database objects such as the following:

- Procedures and functions
- Packages
- Database triggers

Modular applications improve:

- Functionality
- Security
- Overall performance

Lesson Objectives

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

- **Distinguish anonymous PL/SQL blocks from named PL/SQL blocks (subprograms)**
- **Describe subprograms**
- **List the benefits of using subprograms**
- **List the different environments from which subprograms can be invoked**

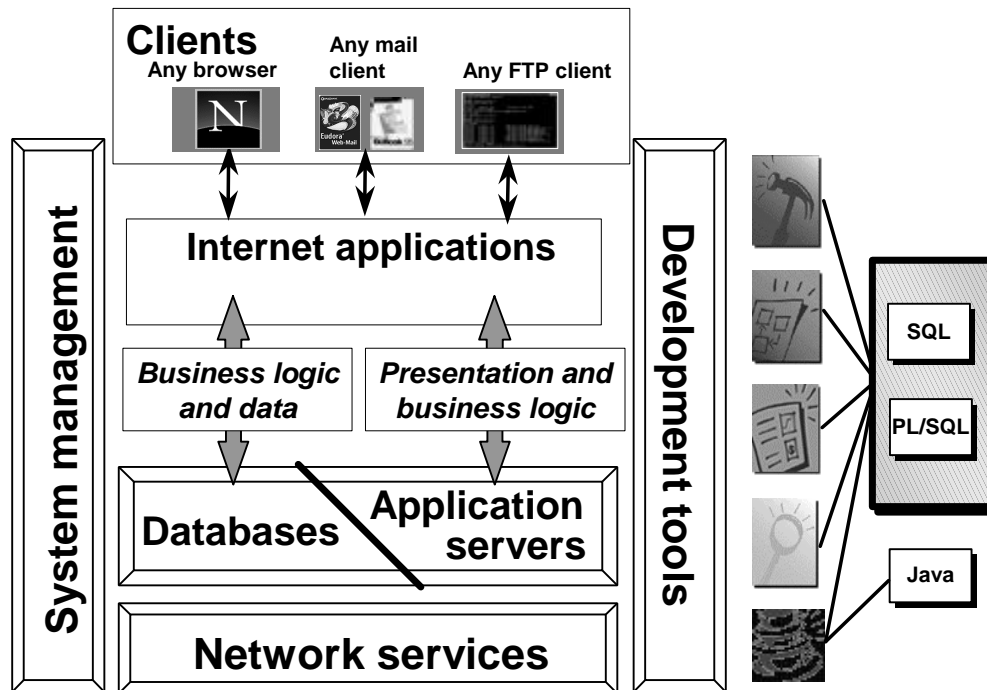
ORACLE



Lesson Aim

PL/SQL supports many different program constructs. In this lesson you learn the difference between anonymous blocks and named PL/SQL blocks. Named PL/SQL blocks are also referred to as subprograms or program units.

Oracle Internet Platform



ORACLE

1-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle Internet Platform

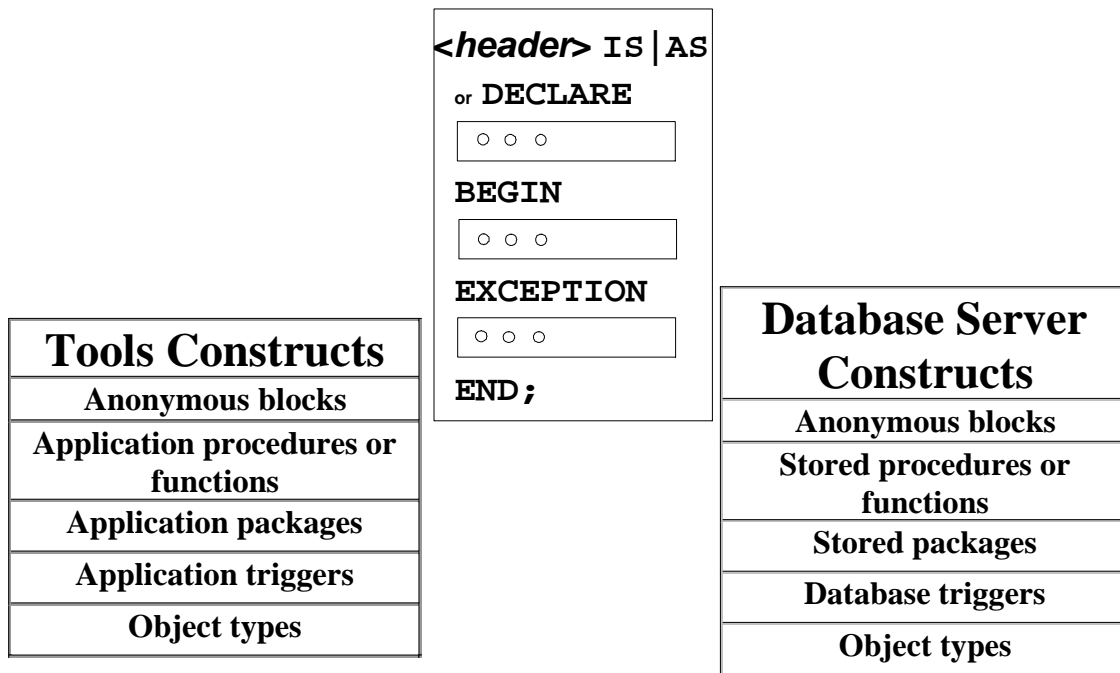
Oracle offers a comprehensive high-performance Internet platform for e-commerce and data warehousing. This integrated platform includes everything needed to develop, deploy, and manage Internet applications. The Oracle Internet Platform is built on three core pieces:

- Browser-based clients to process presentation
- Application servers to execute business logic and serve presentation logic to browser-based clients
- Databases to execute database-intensive business logic and serve data

Oracle offers a wide variety of the most advanced graphical user interface (GUI) driven development tools to build business applications, as well as a large suite of software applications for many areas of business and industry. Stored procedures, functions, and packages can be written by using SQL, PL/SQL, or Java.

*iSQL*Plus* is an Oracle tool that recognizes and submits SQL and PL/SQL statements to the server for execution and contains its own command language.

PL/SQL Program Constructs



ORACLE

1-5

Copyright © Oracle Corporation, 2001. All rights reserved.

PL/SQL Program Constructs

The diagram above displays a variety of different PL/SQL program constructs using the basic PL/SQL block. In general, a block is either an anonymous block or a named block (subprogram or program unit).

PL/SQL Block Structure

Every PL/SQL construct is composed of one or more blocks. These blocks can be entirely separate or nested within one another. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Note: In the slide, the word "or" prior to the keyword `DECLARE` is not part of the syntax. It is used in the diagram to differentiate between starting of subprograms and anonymous blocks.

The PL/SQL blocks can be constructed on and use the Oracle server (stored PL/SQL program units). They can also be constructed using the Oracle Developer tools such as Oracle Forms Developer, Oracle Report Developer, and so on (application or client-side PL/SQL program units).

Object types are user-defined composite data types that encapsulates a data structure along with the functions and procedures needed to manipulate the data. You can create object types either on the Oracle server or using the Oracle Developer tools.

In this course, you will learn writing and managing stored procedures and functions, database triggers, and packages. Creating object types is not covered in this course.

Overview of Subprograms

A subprogram:

- **Is a named PL/SQL block that can accept parameters and be invoked from a calling environment**
- **Is of two types:**
 - A procedure that performs an action
 - A function that computes a value
- **Is based on standard PL/SQL block structure**
- **Provides modularity, reusability, extensibility, and maintainability**
- **Provides easy maintenance, improved data security and integrity, improved performance, and improved code clarity**

ORACLE

Overview of Subprogram

A subprogram is based on standard PL/SQL structure that contains a declarative section, an executable section, and an optional exception-handling section.

A subprogram can be compiled and stored in the database. It provides modularity, extensibility, reusability, and maintainability.

Modularization is the process of breaking up large blocks of code into smaller groups of code called modules. After code is modularized, the modules can be reused by the same program or shared by other programs. It is easier to maintain and debug code of smaller modules than a single large program. Also, the 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 hence any modifications required to the subprogram can 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 appropriate access privilege is granted to the user.

Block Structure for Anonymous PL/SQL Blocks

DECLARE (optional)
 Declare PL/SQL objects to be used
 within this block

BEGIN (mandatory)
 Define the executable statements

EXCEPTION (optional)
 Define the actions that take place if
 an error or exception arises

END ; (mandatory)

ORACLE



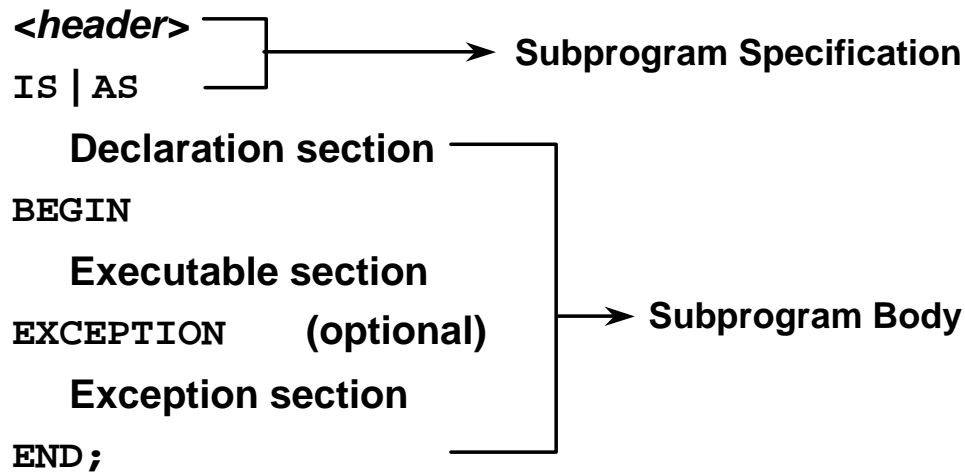
Anonymous Blocks

Anonymous blocks do not have names. You declare them at the point in an application where they are to be run, and they are passed to the PL/SQL engine for execution at run time.

- The section between the keywords **DECLARE** and **BEGIN** is referred to as the declaration section. In the declaration section, you define the PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block. The **DECLARE** keyword is optional if you do not declare any PL/SQL objects.
- The **BEGIN** and **END** keywords are mandatory and enclose the body of actions to be performed. This section is referred to as the executable section of the block.
- The section between **EXCEPTION** and **END** is referred to as the exception section. The exception section traps error conditions. In it, you define actions to take if the specified condition arises. The exception section is optional.

The keywords **DECLARE**, **BEGIN**, and **EXCEPTION** are not followed by semicolons, but **END** and all other PL/SQL statements do require semicolons.

Block Structure for PL/SQL Subprograms



ORACLE

Subprograms

Subprograms are named PL/SQL blocks that can accept parameters and be invoked from a calling environment. PL/SQL has two types of subprograms, *procedures* and *functions*.

Subprogram Specification

- The header is relevant for named blocks only and determines the way that the program unit is called or invoked.

The header determines:

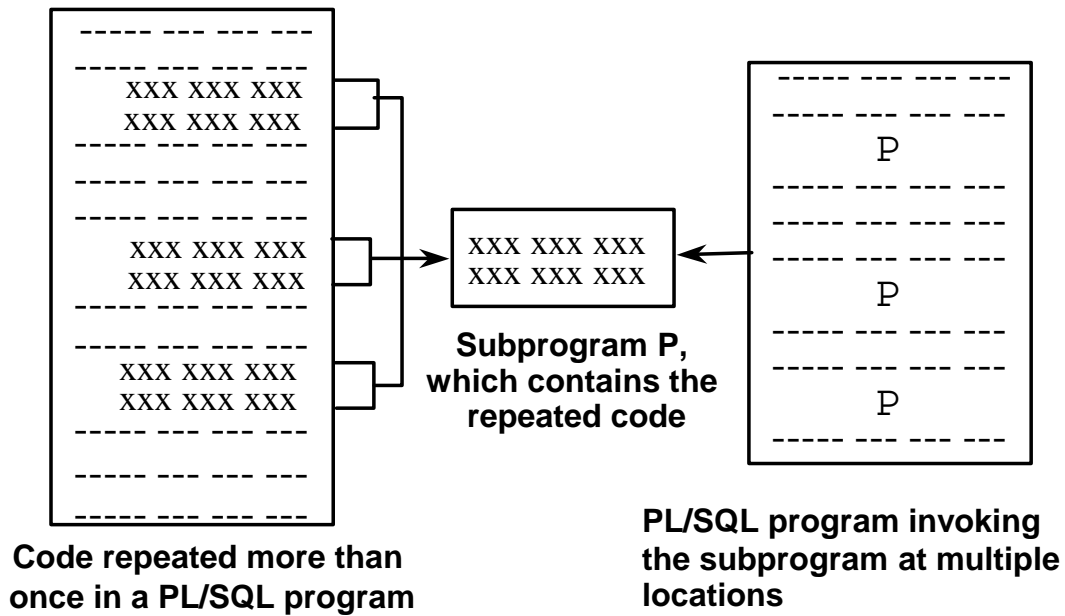
- The PL/SQL subprogram type, that is, either a procedure or a function
- The name of the subprogram
- The parameter list, if one exists
- The RETURN clause, which applies only to functions

- The IS or AS keyword is mandatory.

Subprogram Body

- The declaration section of the block between IS | AS and BEGIN. The keyword DECLARE that is used to indicate the starting of the declaration section in anonymous blocks is not used here.
- The executable section between the BEGIN and END keywords is mandatory, enclosing the body of actions to be performed. There must be at least one statement existing in this section. There should be at least a NULL ; statement, that is considered an executable statement.
- The exception section between EXCEPTION and END is optional. This section traps predefined error conditions. In this section, you define actions to take if the specified error condition arises.

PL/SQL Subprograms



ORACLE

1-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Subprograms

The diagram in the slide explains how you can replace a sequence of PL/SQL statements repeated in a PL/SQL block with a subprogram.

When a sequence of statements is repeated more than once in a PL/SQL subprogram, you can create a subprogram with the repeated code. You can invoke the subprogram at multiple locations in a PL/SQL block. After the subprogram is created and stored in the database, it can be invoked any number of times and from multiple applications.

Benefits of Subprograms

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

ORACLE



1-10

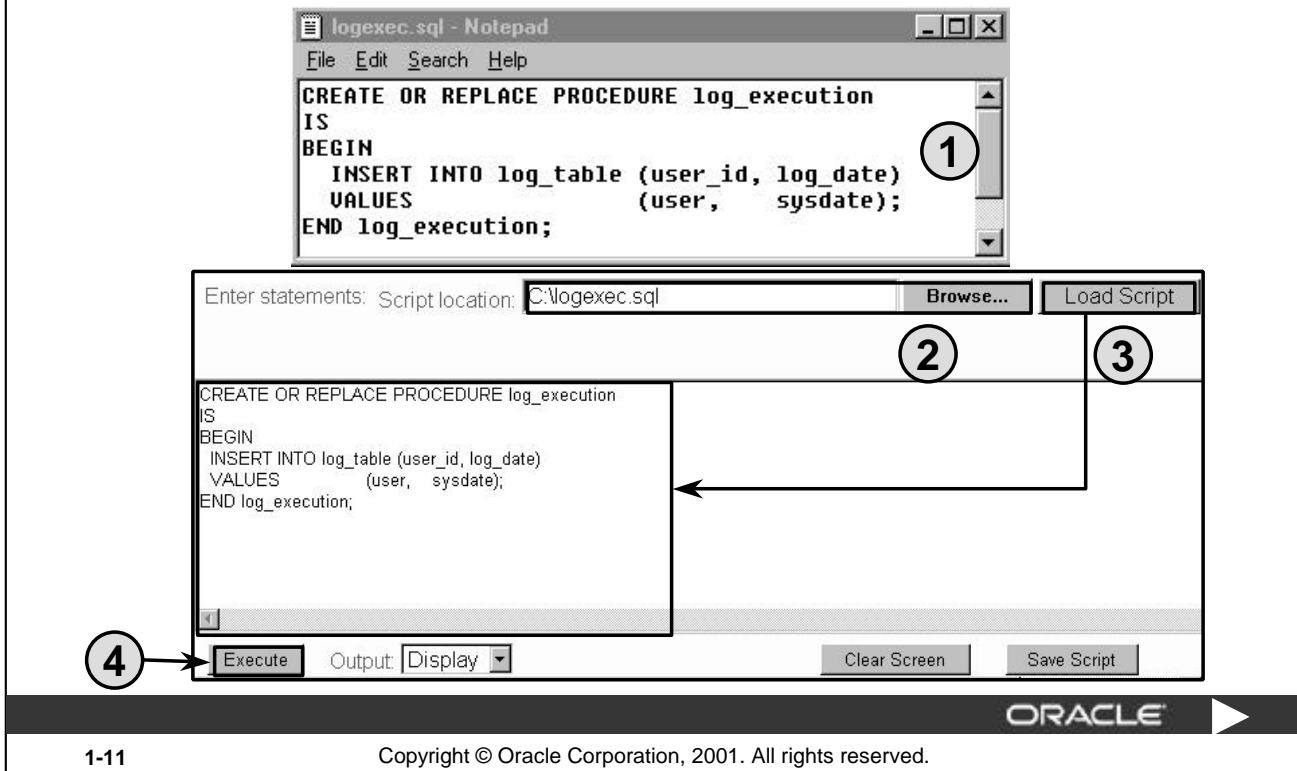
Copyright © Oracle Corporation, 2001. All rights reserved.

Benefits of Subprograms

Stored procedures and functions have many benefits in addition to modularizing application development:

- **Easy maintenance**
 - Modify routines online without interfering with other users
 - Modify one routine to affect multiple applications
 - Modify one routine to eliminate duplicate testing
- **Improved data security and integrity**
 - Control indirect access to database objects from nonprivileged users with security privileges
 - Ensure that related actions are performed together, or not at all, by funneling activity for related tables through a single path
- **Improved performance**
 - Avoid reparsing for multiple users by exploiting the shared SQL area
 - Avoid PL/SQL parsing at run time by parsing at compile time
 - Reduce the number of calls to the database and decrease network traffic by bundling commands
- **Improved code clarity:** Using appropriate identifier names to describe the action of the routines reduces the need for comments and enhances the clarity of the code.

Developing Subprograms by Using iSQL*Plus



Developing Subprograms by Using iSQL*Plus

iSQL*Plus is an Internet-enabled interface to SQL*Plus. You can use a Web browser to connect to an Oracle database and perform the same actions as you would through other SQL*Plus interfaces.

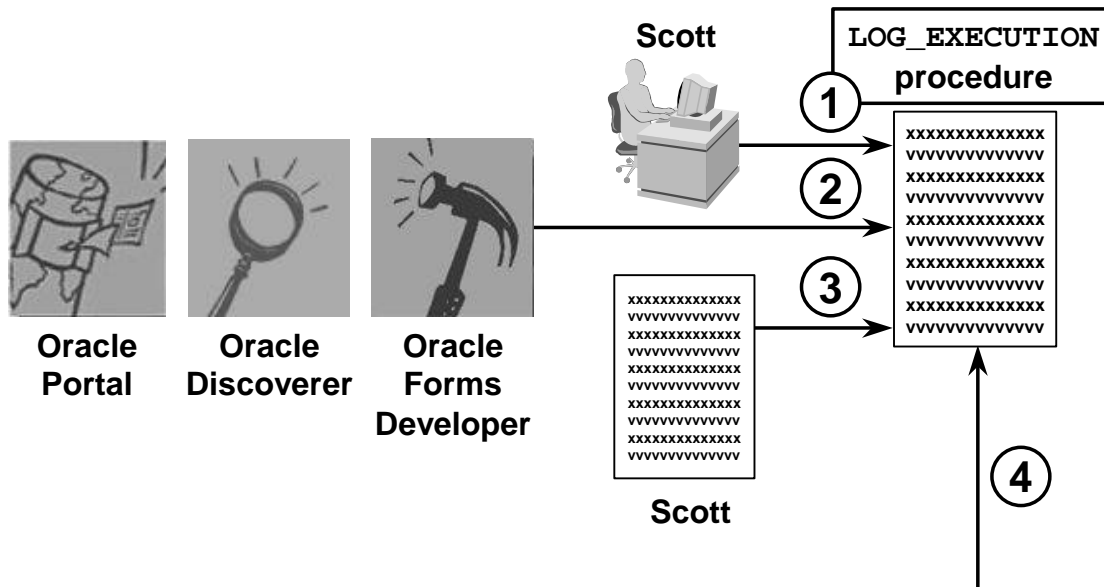
1. Use a text editor to create a SQL script file to define your subprogram. The example in the slide creates the stored procedure LOG_EXECUTION without any parameters. The procedure records the username and current date in a database table called LOG_TABLE.

From iSQL*Plus browser window:

2. Use the Browse button to locate the SQL script file.
3. Use the Load Script button to load the script into the iSQL*Plus buffer.
4. Use the Execute button to run the code. By default, the output from the code is displayed on the screen.

PL/SQL subprograms can also be created by using the Oracle development tools such as Oracle Forms Developer.

Invoking Stored Procedures and Functions



How to Invoke Stored Procedures and Functions

You can invoke a previously created procedure or function from a variety of environments such as *iSQL*Plus*, Oracle Forms Developer, Oracle Discoverer, Oracle Portal, another stored procedure, and many other Oracle tools and precompiler applications. The table below describes how you can invoke a previously created procedure, `log_execution`, from a variety of environments.

<i>iSQL*Plus</i>	<code>EXECUTE log_execution</code>
Oracle development tools such as Oracle Forms Developer	<code>log_execution;</code>
Another procedure	<pre>CREATE OR REPLACE PROCEDURE leave_emp (p_id IN employees.employee_id%TYPE) IS BEGIN DELETE FROM employees WHERE employee_id = p_id; log_execution; END leave_emp;</pre>
Other environments such as Pro*C	

Summary

In this lesson, you should have learned that:

- **Anonymous blocks are unnamed PL/SQL blocks.**
- **Subprograms are named PL/SQL blocks, declared as either procedures or functions.**
- **You can create subprograms in *iSQL*Plus* using a text editor.**
- **You can invoke subprograms from different environments.**

ORACLE

1-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

PL/SQL supports many different program constructs. Anonymous blocks are unnamed PL/SQL blocks. Named PL/SQL blocks are also known as subprograms or program units.

Procedures, functions, packages, and triggers are different PL/SQL constructs.

You can invoke subprograms from different environments.

2

Creating Procedures

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe a procedure**
- **Create a procedure**
- **Differentiate between formal and actual parameters**
- **List the features of different parameter modes**
- **Create procedures with parameters**
- **Invoke a procedure**
- **Handle exceptions in procedures**
- **Remove a procedure**

ORACLE

2-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn to create, execute, and remove procedures.

What Is a Procedure?

- **A procedure is a type of subprogram that performs an action.**
- **A procedure can be stored in the database, as a schema object, for repeated execution.**

ORACLE

2-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Definition of a Procedure

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as arguments), and be invoked. Generally speaking, you use a procedure to perform an action. A procedure has a header, a declaration section, an executable section, and an optional exception-handling section.

A procedure can be compiled and stored in the database as a schema object.

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

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter1 [mode1] datatype1,
    parameter2 [mode2] datatype2,
    . . .)]
IS | AS
PL/SQL Block;
```

- The **REPLACE** option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
- PL/SQL block starts with either **BEGIN** or the declaration of local variables and ends with either **END** or **END *procedure_name***.

ORACLE

Syntax for Creating Procedures

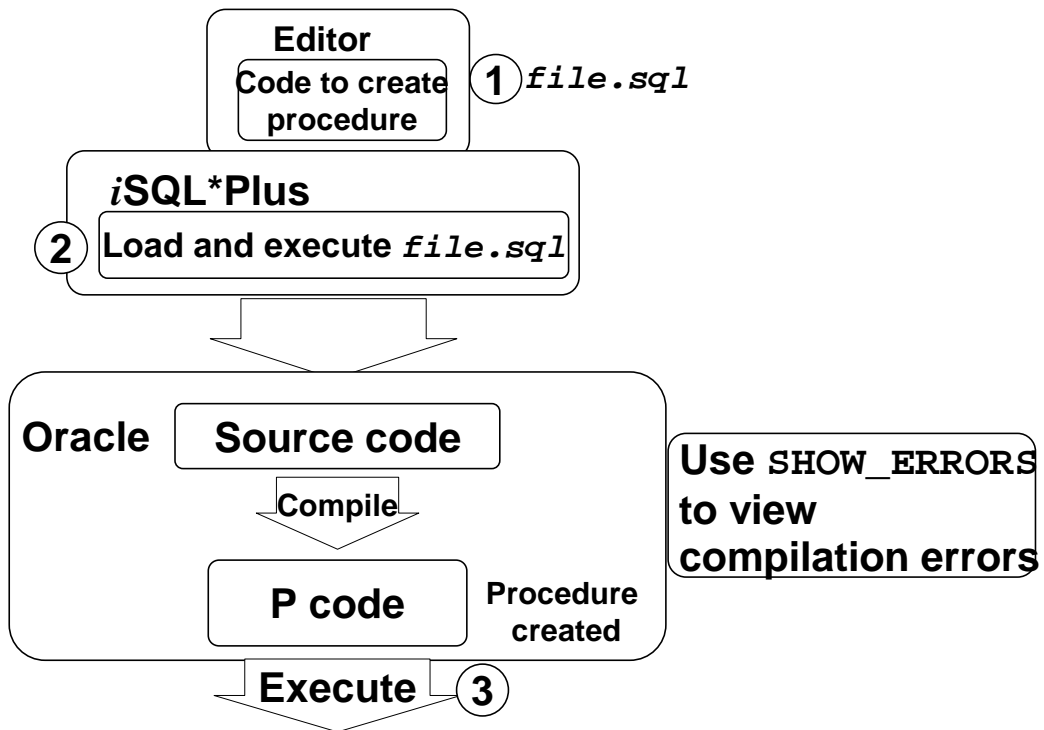
Syntax Definitions

Parameter	Description
<i>procedure_name</i>	Name of the procedure
<i>parameter</i>	Name of a PL/SQL variable whose value is passed to or populated by the calling environment, or both, depending on the <i>mode</i> being used
<i>mode</i>	Type of argument: IN (default) OUT IN OUT
<i>Data type</i>	Data type of the argument—can be any SQL / PLSQL data type. Can be of %TYPE, %ROWTYPE, or any scalar or composite data type.
<i>PL/SQL block</i>	Procedural body that defines the action performed by the procedure

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, which are stored in an Oracle database.

- PL/SQL blocks start with either **BEGIN** or the declaration of local variables and end with either **END** or **END *procedure_name***. You cannot reference host or bind variables in the PL/SQL block of a stored procedure.
- The **REPLACE** option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
- You cannot restrict the size of the data type in the parameters.

Developing Procedures



Developing Procedures

Following are the main steps for developing a stored procedure. The next two pages provide more detail about creating procedures.

1. Write the syntax: Enter the code to create a procedure (CREATE PROCEDURE statement) in a system editor or word processor and save it as a SQL script file (.sql extension).
2. Compile the code: Using iSQL*Plus, load and run the SQL script file. The source code is compiled into P code and the procedure is created.

A script file with the CREATE PROCEDURE (or CREATE OR REPLACE PROCEDURE) statement enables you to change the statement if there are any compilation or run-time errors, or to make subsequent changes to the statement. You cannot successfully invoke a procedure that contains any compilation or run-time errors. In iSQL*Plus, use SHOW ERRORS to see any compilation errors. Running the CREATE PROCEDURE statement stores the source code in the data dictionary even if the procedure contains compilation errors. Fix the errors in the code using the editor and recompile the code.

3. Execute the procedure to perform the desired action. After the source code is compiled and the procedure is successfully created, the procedure can be executed any number of times using the EXECUTE command from iSQL*Plus. The PL/SQL compiler generates the pseudocode or P code, based on the parsed code. The PL/SQL engine executes this when the procedure is invoked.

Note: If there are any compilation errors, and you make subsequent changes to the CREATE PROCEDURE statement, you must either DROP the procedure first, or use the OR REPLACE syntax.

You can create client side procedures that are used with client-side applications using tools such as the Oracle Forms and Reports of Oracle integrated development environment (IDE). Refer to Appendix F to see how the client side subprograms can be created using the Oracle Procedure Builder tool.

Formal Versus Actual Parameters

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

Example:

```
CREATE PROCEDURE raise_sal(p_id NUMBER, p_amount NUMBER)
...
END raise_sal;
```

- **Actual parameters: variables or expressions referenced in the parameter list of a subprogram call**

Example:

```
raise_sal(v_id, 2000)
```

ORACLE

Formal Versus Actual Parameters

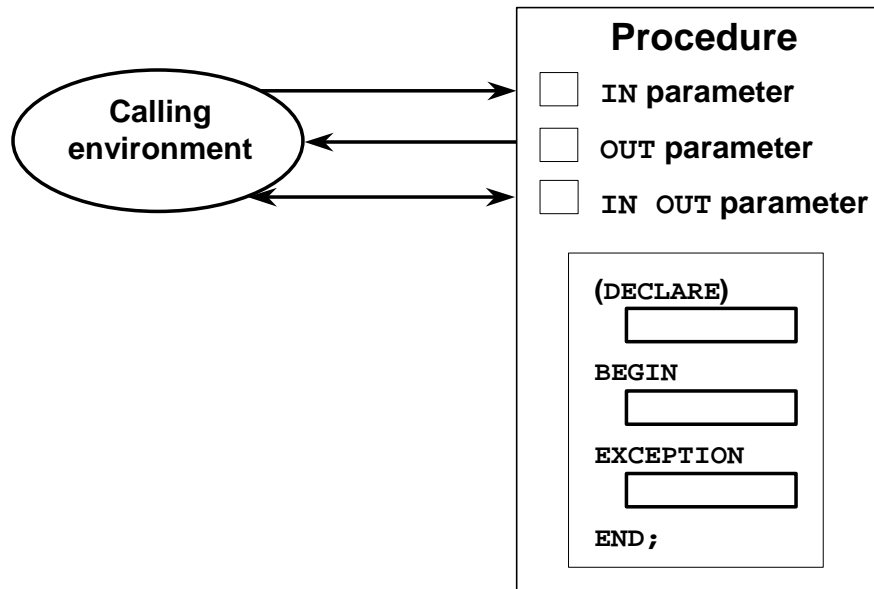
Formal parameters are variables declared in the parameter list of a subprogram specification. For example, in the procedure `RAISE_SAL`, the variables `P_ID` and `P_AMOUNT` are formal parameters.

Actual parameters are variables or expressions referenced in the parameter list of a subprogram call. For example, in the call `raise_sal(v_id, 2000)` to the procedure `RAISE_SAL`, the variable `V_ID` and `2000` are actual parameters.

- Actual parameters are evaluated and results are assigned to formal parameters during the subprogram call.
- Actual parameters can also be expressions such as in the following:

```
raise_sal(v_id, raise+100);
```
- It is good practice to use different names for formal and actual parameters. Formal parameters have the prefix `p_` in this course.
- 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



ORACLE

2-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Procedural Parameter Modes

You can transfer values to and from the calling environment through parameters. Select one of the three modes for each parameter: IN, OUT, or IN OUT.

Attempts to change the value of an IN parameter will result in an error.

Note: DATATYPE can be only the %TYPE definition, the %ROWTYPE definition, or an explicit data type with no size specification.

Type of Parameter	Description
IN (default)	Passes a constant value from the calling environment into the procedure
OUT	Passes a value from the procedure to the calling environment
IN OUT	Passes a value from the calling environment into the procedure and a possibly different value from the procedure back to the calling environment using the same parameter

Creating Procedures with Parameters

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

ORACLE

2-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Procedures with Parameters

When you create the procedure, the formal parameter defines the value used in the executable section of the PL/SQL block, whereas the actual parameter is referenced when invoking the procedure.

The parameter mode **IN** is the default parameter mode. That is, no mode is specified with a parameter, the parameter is considered an **IN** parameter. The parameter modes **OUT** and **IN OUT** must be explicitly specified in front of such parameters.

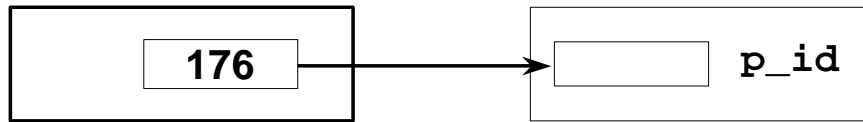
A formal parameter of **IN** mode cannot be assigned a value. That is, an **IN** parameter cannot be modified in the body of the procedure.

An **OUT** or **IN OUT** parameter must be assigned a value before returning to the calling environment.

IN parameters can be assigned a default value in the parameter list. **OUT** and **IN OUT** parameters cannot be assigned default values.

By default, the **IN** parameter is passed by reference and the **OUT** and **IN OUT** parameters are passed by value. To improve performance with **OUT** and **IN OUT** parameters, the compiler hint **NOCOPY** can be used to request to pass by reference. Using **NOCOPY** is discussed in detail in the *Advanced PL/SQL* course.

IN Parameters: Example



```
CREATE OR REPLACE PROCEDURE raise_salary
  (p_id IN employees.employee_id%TYPE)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * 1.10
  WHERE  employee_id = p_id;
END raise_salary;
/
```

Procedure created.

ORACLE

2-9

Copyright © Oracle Corporation, 2001. All rights reserved.

IN Parameters: Example

The example in the slide shows a procedure with one IN parameter. Running this statement in *iSQL*Plus* creates the RAISE_SALARY procedure. When invoked, RAISE_SALARY accepts the parameter for the employee ID and updates the employee's record with a salary increase of 10 percent. To invoke a procedure in *iSQL*Plus*, use the EXECUTE command.

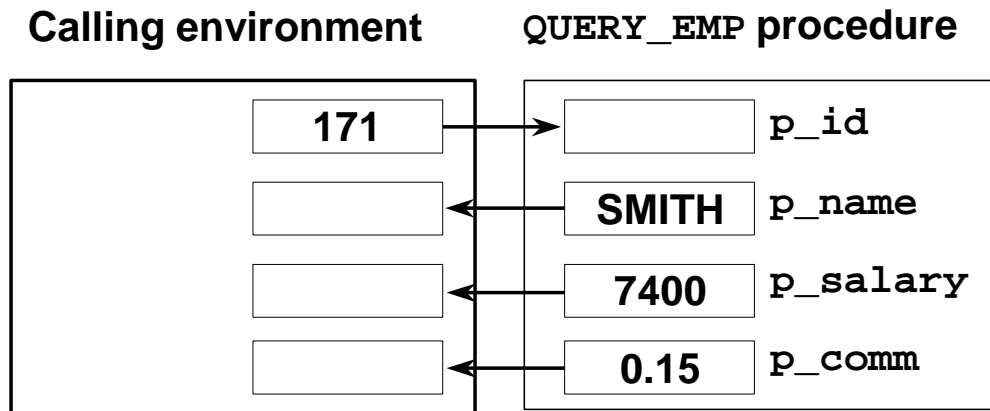
```
EXECUTE raise_salary (176)
```

To invoke a procedure from another procedure, use a direct call. At the location of calling the new procedure, enter the procedure name and actual parameters.

```
raise_salary (176);
```

IN parameters are passed as constants from the calling environment into the procedure. Attempts to change the value of an IN parameter result in an error.

OUT Parameters: Example



ORACLE

2-10

Copyright © Oracle Corporation, 2001. All rights reserved.

OUT Parameters: Example

In this example, you create a procedure with OUT parameters to retrieve information about an employee. The procedure accepts a value 171 for employee ID and retrieves the name, salary, and commission percentage of the employee with ID 171 into the three output parameters. The code to create the `QUERY_EMP` procedure is shown in the next slide.

OUT Parameters: Example

emp_query.sql

```
CREATE OR REPLACE PROCEDURE query_emp
(p_id      IN   employees.employee_id%TYPE,
p_name     OUT  employees.last_name%TYPE,
p_salary   OUT  employees.salary%TYPE,
p_comm     OUT  employees.commission_pct%TYPE)
IS
BEGIN
  SELECT  last_name, salary, commission_pct
  INTO    p_name, p_salary, p_comm
  FROM    employees
  WHERE   employee_id = p_id;
END query_emp;
/
```

ORACLE

2-11

Copyright © Oracle Corporation, 2001. All rights reserved.

OUT Parameters: Example (continued)

Run the script file shown in the slide to create the `QUERY_EMP` procedure. This procedure has four formal parameters. Three of them are OUT parameters that return values to the calling environment.

The procedure accepts an `EMPLOYEE_ID` value for the parameter `P_ID`. The name, salary, and commission percentage values corresponding to the employee ID are retrieved into the three OUT parameters whose values are returned to the calling environment.

Notice that the name of the script file need not be the same as the procedure name. (The script file is on the client side and the procedure is being stored on the database schema.)

Viewing OUT Parameters

- Load and run the `emp_query.sql` script file to create the `QUERY_EMP` procedure.
- Declare host variables, execute the `QUERY_EMP` procedure, and print the value of the global `G_NAME` variable.

```
VARIABLE g_name      VARCHAR2(25)
VARIABLE g_sal        NUMBER
VARIABLE g_comm       NUMBER

EXECUTE query_emp(171, :g_name, :g_sal, :g_comm)

PRINT g_name
```

PL/SQL procedure successfully completed.

G_NAME
Smith

ORACLE

2-12

Copyright © Oracle Corporation, 2001. All rights reserved.

How to View the Value of OUT Parameters with *iSQL*Plus*

1. Run the SQL script file to generate and compile the source code.
2. Create host variables in *iSQL*Plus*, using the `VARIABLE` command.
3. Invoke the `QUERY_EMP` procedure, supplying these host variables as the OUT parameters. Note the use of the colon (`:`) to reference the host variables in the `EXECUTE` command.
4. To view the values passed from the procedure to the calling environment, use the `PRINT` command.

The example in the slide shows the value of the `G_NAME` variable passed back to the the calling environment. The other variables can be viewed, either individually, as above, or with a single `PRINT` command.

```
PRINT g_name g_sal g_comm
```

Do not specify a size for a host variable of data type `NUMBER` when using the `VARIABLE` command. A host variable of data type `CHAR` or `VARCHAR2` defaults to a length of one, unless a value is supplied in parentheses.

`PRINT` and `VARIABLE` are *iSQL*Plus* commands.

Note: Passing a constant or expression as an actual parameter to the OUT variable causes compilation errors. For example:

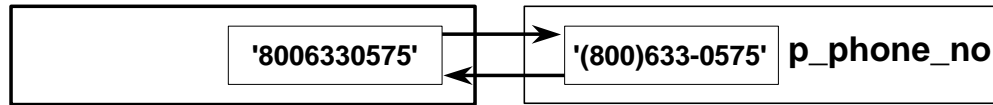
```
EXECUTE query_emp(171, :g_name, raise+100, :g_comm)
```

causes a compilation error.

IN OUT Parameters

Calling environment

FORMAT_PHONE procedure



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

ORACLE

2-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Using IN OUT Parameters

With an IN OUT parameter, you can pass values into a procedure and return a value to the calling environment. The value that is returned is either the original, an unchanged value, or a new value set within the procedure.

An IN OUT parameter acts as an initialized variable.

Example

Create a procedure with an IN OUT parameter to accept a character string containing 10 digits and return a phone number formatted as (800) 633-0575.

Run the statement to create the FORMAT_PHONE procedure.

Viewing IN OUT Parameters

```
VARIABLE g_phone_no VARCHAR2(15)
BEGIN
    :g_phone_no := '8006330575';
END;
/
PRINT g_phone_no
EXECUTE format_phone (:g_phone_no)
PRINT g_phone_no
```

PL/SQL procedure successfully completed.

G_PHONE_NO
8006330575

PL/SQL procedure successfully completed.

G_PHONE_NO
(800)633-0575

ORACLE

How to View IN OUT Parameters with iSQL*Plus

1. Create a host variable, using the VARIABLE command.
2. Populate the host variable with a value, using an anonymous PL/SQL block.
3. Invoke the FORMAT_PHONE procedure, supplying the host variable as the IN OUT parameter. Note the use of the colon (:) to reference the host variable in the EXECUTE command.
4. To view the value passed back to the calling environment, use the PRINT command.

Methods for Passing Parameters

- **Positional:** List actual parameters in the same order as formal parameters.
- **Named:** List actual parameters in arbitrary order by associating each with its corresponding formal parameter.
- **Combination:** List some of the actual parameters as positional and some as named.

ORACLE

2-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Parameter Passing Methods

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

Method	Description
Positional	Lists values in the order in which the parameters are declared
Named association	Lists values in arbitrary order by associating each one with its parameter name, using special syntax ($=>$)
Combination	Lists the first values positionally, and the remainder using the special syntax of the named method

DEFAULT Option for Parameters

```
CREATE OR REPLACE PROCEDURE add_dept
  (p_name  IN departments.department_name%TYPE
   p_loc   IN departments.location_id%TYPE
   DEFAULT 'unknown',
   DEFAULT 1700)
IS
BEGIN
  INSERT INTO departments(department_id,
                          department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
/
```

Procedure created.

ORACLE

2-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Default Values for Parameters

You can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.

Execute the statement in the slide to create the ADD_DEPT procedure. Note the use of the DEFAULT clause in the declaration of the formal parameter.

You can assign default values only to parameters of the IN mode. OUT and IN OUT parameters are not permitted to have default values. If default values are passed to these types of parameters, you get the following compilation error:

```
PLS-00230: OUT and IN OUT formal parameters may not have default
expressions
```

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the calls to the above procedure that are depicted in the next page.

Examples of Passing Parameters

```
BEGIN
  add_dept;
  add_dept ('TRAINING', 2500);
  add_dept ( p_loc => 2400, p_name =>'EDUCATION');
  add_dept ( p_loc => 1200) ;
END;
/
SELECT department_id, department_name, location_id
FROM departments;
```

PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
40	Human Resources	2400
90	Ev	1200

31 rows selected.

ORACLE

Example of Default Values for Parameters (continued)

The anonymous block above shows the different ways the ADD_DEPT procedure can be invoked, and the output of each way the procedure is invoked.

Usually, you can use positional notation to override the default values of formal parameters. However, you cannot skip a formal parameter by leaving out its actual parameter.

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

```
EXECUTE add_dept(p_name=>'new dept', 'new location')
BEGIN add_dept(p_name=>'new dept', 'new location'); END;
*
```

ERROR at line 1:

ORA-06550: line 1, column 31:

PLS-00312: a positional parameter association may not follow a named association

ORA-06550: line 1, column 7:

PL/SQL: Statement ignored

Declaring Subprograms

leave_emp2.sql

```
CREATE OR REPLACE PROCEDURE leave_emp2
  (p_id IN employees.employee_id%TYPE)
IS
  PROCEDURE log_exec
  IS
  BEGIN
    INSERT INTO log_table (user_id, log_date)
      VALUES (USER, SYSDATE);
  END log_exec;
BEGIN
  DELETE FROM employees
    WHERE employee_id = p_id;
  log_exec;
END leave_emp2;
/
```

ORACLE

2-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Declaring Subprograms

You can declare subprograms in any PL/SQL block. This is an alternative to creating the stand-alone procedure LOG_EXEC. Subprograms declared in this manner are called local subprograms (or local modules). Because they are defined within a declaration section of another program, the scope of local subprograms is limited to the parent (enclosing) block in which they are defined. This means that local subprograms cannot be called from outside the block in which they are declared. Declaring local subprograms enhances the clarity of the code by assigning appropriate business-rule identifiers to blocks of code.

Note: You must declare the subprogram in the declaration section of the block, and it must be the last item, after all the other program items. For example, a variable declared after the end of the subprogram, before the BEGIN of the procedure, will cause a compilation error.

If the code must be accessed by multiple applications, place the subprogram in a package or create a stand-alone subprogram with the code. Packages are discussed later in this course.

Invoking a Procedure from an Anonymous PL/SQL Block

```
DECLARE
  v_id NUMBER := 163;
BEGIN
  raise_salary(v_id);    --invoke procedure
  COMMIT;
  ...
END;
```

ORACLE

Invoking a Procedure from an Anonymous PL/SQL Block

Invoke the RAISE_SALARY procedure from an anonymous PL/SQL block, as shown in the slide.

Procedures are callable from *any* tool or language that supports PL/SQL.

You have already seen how to invoke an independent procedure from iSQL*Plus.

Invoking a Procedure from Another Procedure

process_emps.sql

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

ORACLE

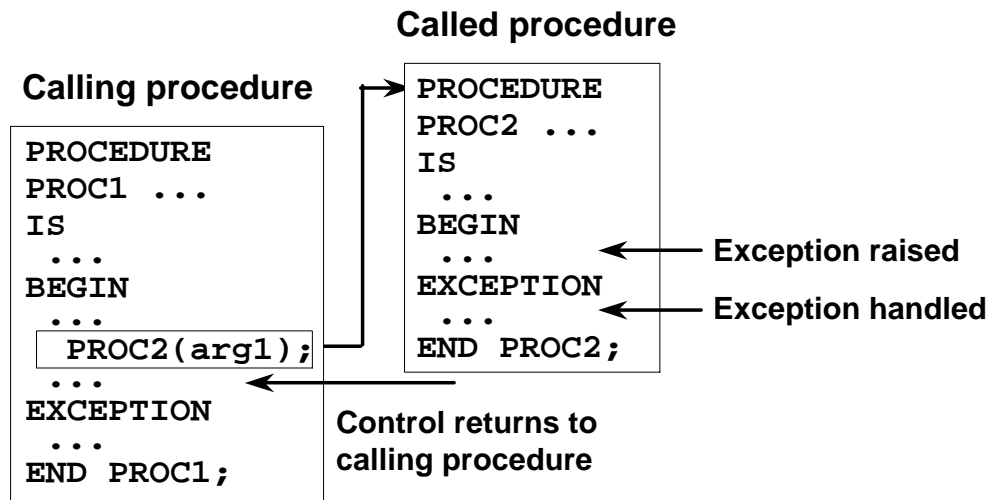
2-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Invoking a Procedure from Another Procedure

This example shows you how to invoke a procedure from another stored procedure. The `PROCESS_EMPS` 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 percent salary increase across the company.

Handled Exceptions



ORACLE

How Handled Exceptions Affect the Calling Procedure

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, control immediately goes to the exception section of that block. If the exception is handled, the block terminates, and control goes to the calling program. Any data manipulation language (DML) statements issued before the exception was raised remain as part of the transaction.

Handled Exceptions

```
CREATE PROCEDURE p2_ins_dept(p_locid NUMBER) IS
  v_did NUMBER(4);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Procedure p2_ins_dept started');
  INSERT INTO departments VALUES (5, 'Dept 5', 145, p_locid);
  SELECT department_id INTO v_did FROM employees
    WHERE employee_id = 999;
END;
```

```
CREATE PROCEDURE p1_ins_loc(p_lid NUMBER, p_city VARCHAR2)
IS
  v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Main Procedure p1_ins_loc');
  INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
  SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
  DBMS_OUTPUT.PUT_LINE('Inserted city '||v_city);
  DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_ins_dept ...');
  p2_ins_dept(p_lid);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such dept/loc for any employee');
END;
```

ORACLE

2-22

Copyright © Oracle Corporation, 2001. All rights reserved.

How Handled Exceptions Affect the Calling Procedure (continued)

The example in the slide shows two procedures. Procedure P1_INS_LOC inserts a new location (supplied through the parameters) into the LOCATIONS table. Procedure P2_INS_DEPT inserts a new department (with department ID 5) at the new location inserted through the P1_INS_LOC procedure. The P1_INS_LOC procedure invokes the P2_INS_DEPT procedure.

The P2_INS_DEPT procedure has a SELECT statement that selects DEPARTMENT_ID for a nonexisting employee and raises a NO_DATA_FOUND exception. Because this exception is not handled in the P2_INS_DEPT procedure, the control returns to the calling procedure P1_INS_LOC where the exception is handled. As the exception is handled, the DML in the P2_INS_DEPT procedure is not rolled back and is part of the transaction of the P1_INS_LOC procedure.

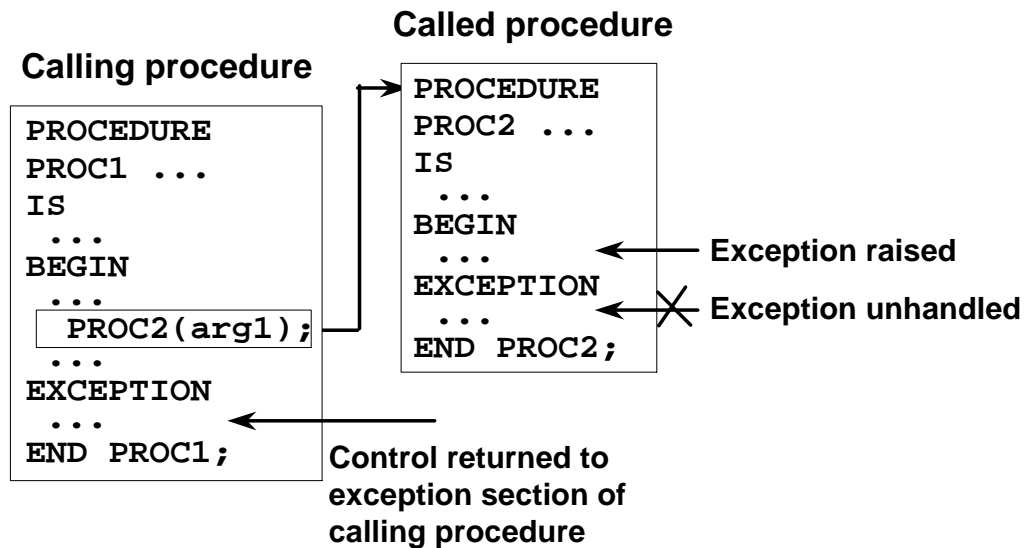
The following code shows that the INSERT statements from both the procedures are successful:

```
EXECUTE p1_ins_loc(1, 'Redwood Shores')
SELECT location_id, city FROM locations
  WHERE location_id = 1;
SELECT * FROM departments WHERE department_id = 5;
```

PL/SQL procedure successfully completed.

LOCATION_ID		CITY	
1		Redwood Shores	
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
5	Dept 5	145	1

Unhandled Exceptions



ORACLE

How Unhandled Exceptions Affect the Calling Procedure

When an exception is raised in a called procedure, control immediately goes to the exception section of that block. If the exception is unhandled, the block terminates, and control goes to the exception section of the calling procedure. PL/SQL does not roll back database work that is done by the subprogram.

If the exception is handled in the calling procedure, all DML statements in the calling procedure and in the called procedure remain as part of the transaction.

If the exception is unhandled in the calling procedure, the calling procedure terminates and the exception propagates to the calling environment. All the DML statements in the calling procedure and the called procedure are rolled back along with any changes to any host variables. The host environment determines the outcome for the unhandled exception.

Unhandled Exceptions

```
CREATE PROCEDURE p2_noexcep(p_locid NUMBER) IS
  v_did NUMBER(4);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Procedure p2_noexcep started');
  INSERT INTO departments VALUES (6, 'Dept 6', 145, p_locid);
  SELECT department_id INTO v_did FROM employees
    WHERE employee_id = 999;
END;
```

```
CREATE PROCEDURE p1_noexcep(p_lid NUMBER, p_city VARCHAR2)
IS
  v_city VARCHAR2(30); v_dname VARCHAR2(30);
BEGIN
  DBMS_OUTPUT.PUT_LINE(' Main Procedure p1_noexcep');
  INSERT INTO locations (location_id, city) VALUES (p_lid, p_city);
  SELECT city INTO v_city FROM locations WHERE location_id = p_lid;
  DBMS_OUTPUT.PUT_LINE('Inserted new city '||v_city);
  DBMS_OUTPUT.PUT_LINE('Invoking the procedure p2_noexcep ...');
  p2_noexcep(p_lid);
END;
```

ORACLE

2-24

Copyright © Oracle Corporation, 2001. All rights reserved.

How Unhandled Exceptions Affect the Calling Procedure (continued)

The example in the slide shows two procedures. Procedure P1_NOEXCEP inserts a new location (supplied through the parameters) into the LOCATIONS table. Procedure P2_NOEXCEP inserts a new department (with department ID 5) at the new location inserted through the P1_NOEXCEP procedure. Procedure P1_NOEXCEP invokes the P2_NOEXCEP procedure.

The P2_NOEXCEP procedure has a SELECT statement that selects DEPARTMENT_ID for a nonexisting employee and raises a NO_DATA_FOUND exception. Because this exception is not handled in the P2_NOEXCEP procedure, the control returns to the calling procedure P1_NOEXCEP. The exception is not handled. Because the exception is not handled, the DML in the P2_NOEXCEP procedure is rolled back along with the transaction of the P1_NOEXCEP procedure.

The following code shows that the DML statements from both the procedures are unsuccessful.

```
EXECUTE p1_noexcep(3, 'New Delhi')
SELECT location_id, city FROM locations
  WHERE location_id = 3;
SELECT * FROM departments WHERE department_id = 6;
```

Removing Procedures

Drop a procedure stored in the database.

Syntax:

```
DROP PROCEDURE procedure_name
```

Example:

```
DROP PROCEDURE raise_salary;
```

Procedure dropped.

ORACLE

Removing Procedures

When a stored procedure is no longer required, you can use a SQL statement to drop it.

To remove a server-side procedure by using *iSQL*Plus*, execute the SQL command `DROP PROCEDURE`.

Issuing rollback does not have an effect after executing a data definition language (DDL) command such as `DROP PROCEDURE`, which commits any pending transactions.

Benefits of Subprograms

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

ORACLE

2-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Benefits of Subprograms

Procedures and functions have many benefits in addition to modularizing application development:

- **Easy maintenance:** Subprograms are located in one location and hence it is easy to:
 - Modify routines online without interfering with other users.
 - Modify one routine to affect multiple applications.
 - Modify one routine to eliminate duplicate testing.
- **Improved data security and integrity**
 - Controls indirect access to database objects from nonprivileged users with security privileges. As a subprogram is executed with its definer's right by default, it is easy to restrict the access privilege by granting a privilege only to execute the subprogram to a user.
 - Ensures that related actions are performed together, or not at all, by funneling activity for related tables through a single path.
- **Improved performance**
 - After a subprogram is compiled, the parsed code is available in the shared SQL area of the server and subsequent calls to the subprogram use this parsed code. This avoids reparsing for multiple users.
 - Avoids PL/SQL parsing at run time by parsing at compile time.
 - Reduces the number of calls to the database and decreases network traffic by bundling commands.
- **Improves code clarity:** Using appropriate identifier names to describe the action of the routines reduces the need for comments and enhances the clarity of the code.

Summary

In this lesson, you should have learned that:

- **A procedure is a subprogram that performs an action.**
- **You create procedures by using the `CREATE PROCEDURE` command.**
- **You can compile and save a procedure in the database.**
- **Parameters are used to pass data from the calling environment to the procedure.**
- **There are three parameter modes: `IN`, `OUT`, and `IN OUT`.**

ORACLE

2-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

A procedure is a subprogram that performs a specified action. You can compile and save a procedure as 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`.

Summary

- **Local subprograms are programs that are defined within the declaration section of another program.**
- **Procedures can be invoked from any tool or language that supports PL/SQL.**
- **You should be aware of the effect of handled and unhandled exceptions on transactions and calling procedures.**
- **You can remove procedures from the database by using the `DROP PROCEDURE` command.**
- **Procedures can serve as building blocks for an application.**

ORACLE

2-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary (continued)

Subprograms that are defined within the declaration section of another program are called local subprograms. The scope of the local subprograms is the program unit within which it is defined.

You should be aware of the effect of handled and unhandled exceptions on transactions and calling procedures. The exceptions are handled in the exception section of a subprogram.

You can modify and remove procedures. You can also create client-side procedures that can be used by client-side applications.

Practice 2 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 matching with 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**

ORACLE

Practice 2 Overview

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

If you encounter compilation errors when you are using *iSQL*Plus*, use the `SHOW ERRORS` command. Using the `SHOW ERRORS` command is discussed in detail in the *Managing Subprograms* lesson.

If you correct any compilation errors in *iSQL*Plus*, do so in the original script file, not in the buffer, and then rerun the new version of the file. This will save a new version of the procedure to the data dictionary.

Practice 2

Note: You can find table descriptions and sample data in Appendix D “Table Descriptions and Data.”

Save your subprograms as .sql files, using the Save Script button.

Remember to set the SERVEROUTPUT on if you set it off previously.

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, and 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 job ID IT_WEB and 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 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 job ID IT_WEB). You should get the message you used in the exception-handling section of the procedure as output.

Practice 2 (continued)

4. Create a procedure called QUERY_EMP 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.

Use host variables for the two OUT parameters salary and job ID.

- b. Compile the code, invoke the procedure to display the salary and job ID for employee ID 120.

G_SAL	
	8000

G_JOB	
ST_MAN	

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

3

Creating Functions

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

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®

3-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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

Overview of Stored Functions

- **A function is a named PL/SQL block that returns a value.**
- **A function can be stored in the database as a schema object for repeated execution.**
- **A function is called as part of an expression.**

ORACLE

Stored Functions

A function is a named PL/SQL block that can accept parameters and be invoked. Generally speaking, 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 part, an executable part, and an optional exception-handling part. 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 a schema object for repeated execution. A function stored in the database is referred to as a stored function. Functions can also be created at client side applications. This lesson discusses creating stored functions. Refer to appendix “Creating Program Units by Using Procedure Builder” for creating 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.

Function is called as part of a SQL expression or as part of a PL/SQL expression. In 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.

Syntax for Creating Functions

```
CREATE [OR REPLACE] FUNCTION function_name
  [(parameter1 [mode1] datatype1,
    parameter2 [mode2] datatype2,
    . . .)]
RETURN datatype
IS | AS
PL/SQL Block;
```

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

ORACLE

3-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Functions Syntax

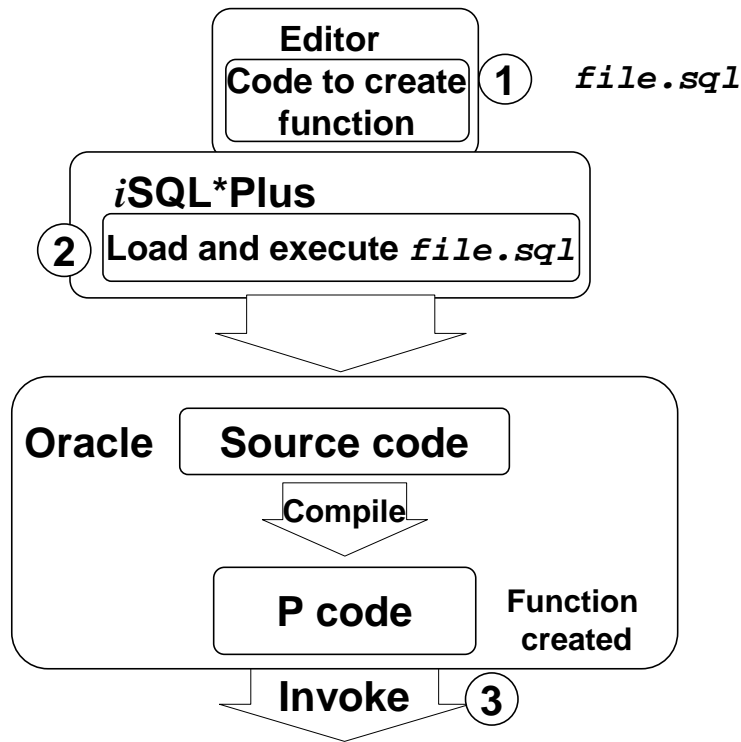
A function is a PL/SQL block that returns a value. You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

- The REPLACE option indicates that if the function exists, it will be dropped and replaced with the new version created by the statement.
- The RETURN data type must not include a size specification.
- PL/SQL blocks start with either BEGIN or the declaration of local variables and end with either END or END *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.

Syntax Definitions

Parameter	Description
<i>function_name</i>	Name of the function
<i>parameter</i>	Name of a PL/SQL variable whose value is passed into the function
<i>mode</i>	The type of the parameter; only IN parameters should be declared
<i>datatype</i>	Data type of the parameter
RETURN <i>datatype</i>	Data type of the RETURN value that must be output by the function
PL/SQL block	Procedural body that defines the action performed by the function

Creating a Function



ORACLE

How to Develop Stored Functions

The following are the basic steps you use to develop a stored function. The next two pages provide further details about creating functions.

1. Write the syntax: Enter the code to create a function in a text editor and save it as a SQL script file.
2. Compile the code: Using *iSQL*Plus*, upload and run the SQL script file. The source code is compiled into P code. The function is created.
3. Invoke the function from a PL/SQL block.

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.

Although 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.

Note: The PL/SQL compiler generates the *pseudocode* or P code, based on the parsed code. The PL/SQL engine executes this when the procedure is invoked.

Creating a Stored Function by Using *iSQL*Plus*

1. Enter the text of the `CREATE FUNCTION` statement in an editor and save it as a SQL script file.
2. Run the script file to store the source code and compile the function.
3. Use `SHOW ERRORS` to see compilation errors.
4. When successfully compiled, invoke the function.

ORACLE

How to Create a Stored Function

1. Enter the text of the `CREATE FUNCTION` statement in a system editor or word processor and save it as a script file (`.sql` extension).
2. From *iSQL*Plus*, load and run the script file to store the source code and compile the source code into P-code.
3. Use `SHOW ERRORS` to see any compilation errors.
4. When the code is successfully compiled, the function is ready for execution. Invoke the function from an Oracle server environment.

A script file with the `CREATE FUNCTION` statement enables you to change the statement if compilation or run-time errors occur, or to make subsequent changes to the statement. You cannot successfully invoke a function that contains any compilation or run-time errors. In *iSQL*Plus*, use `SHOW ERRORS` to see any compilation errors.

Running the `CREATE FUNCTION` statement stores the source code in the data dictionary even if the function contains compilation errors.

Note: If there are any compilation errors and you make subsequent changes to the `CREATE FUNCTION` statement, you either have to drop the function first or use the `OR REPLACE` syntax.

Creating a Stored Function by Using iSQL*Plus: Example

get_salary.sql

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

ORACLE

3-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

Create a function with one IN parameter to return a number.

Run the script file to create the GET_SAL function. Invoke a function as part of a PL/SQL expression, because the function will return a value to the calling environment.

It is a good programming practice to assign a returning value to a variable and use a single RETURN statement in the executable section of the code. There can be a RETURN statement in the exception section of the program also.

Executing Functions

- **Invoke a function as part of a PL/SQL expression.**
- **Create a variable to hold the returned value.**
- **Execute the function. The variable will be populated by the value returned through a RETURN statement.**

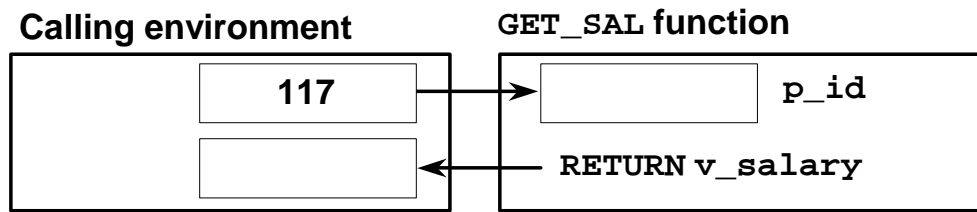
ORACLE

Function Execution

A function may accept one or many parameters, but must return a single value. You invoke functions as part of PL/SQL expressions, using variables to hold the returned value.

Although the three parameter modes, IN (the default), OUT, and IN OUT, can be used with any subprogram, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to accept zero or more arguments (actual parameters) and return a single value. To have a function return multiple values is poor programming practice. Also, functions should be free from side effects, which change the values of variables that are not local to the subprogram. Side effects are discussed later in this lesson.

Executing Functions: Example



1. Load and run the `get_salary.sql` file to create the function

```
② → VARIABLE g_salary NUMBER
③ → EXECUTE :g_salary := get_sal(117)
④ → PRINT g_salary
```

PL/SQL procedure successfully completed

G_SALARY
2800

Example

Execute the `GET_SAL` function from *iSQL*Plus*:

1. Load and run the script file `get_salary.sql` to create the stored function `GET_SAL`.
2. Create a host variable that will be populated by the `RETURN (variable)` statement within the function.
3. Using the `EXECUTE` command in *iSQL*Plus*, invoke the `GET_SAL` function by creating a PL/SQL expression. Supply a value for the parameter (employee ID in this example). The value returned from the function will be held by the host variable, `G_SALARY`. Note the use of the colon (`:`) to reference the host variable.
4. View the result of the function call by using the `PRINT` command. Employee Tobias, with employee ID 117, earns a monthly salary of 2800.

In a function, there must be at least one execution path that leads to a `RETURN` statement. Otherwise, you get a `Function returned without value` error at run time.

Advantages of User-Defined Functions in SQL Expressions

- **Extend SQL where activities are too complex, too awkward, or unavailable with SQL**
- **Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application**
- **Can manipulate character strings**

ORACLE

3-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Invoking User-Defined Functions from SQL Expressions

SQL expressions can reference PL/SQL user-defined functions. Anywhere a built-in SQL function can be placed, a user-defined function can be placed as well.

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

Invoking Functions in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
RETURN NUMBER IS
BEGIN
    RETURN (p_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	7600	608
113	Popp	6900	552

6 rows selected.

ORACLE

Example

The slide shows how to create a function `tax` that is invoked from a `SELECT` statement. The function accepts a `NUMBER` parameter and returns the tax after multiplying the parameter value with 0.08.

In *iSQL*Plus*, invoke the `TAX` function inside a query displaying employee ID, name, salary, and tax.

Locations to Call User-Defined Functions

- **Select list of a SELECT command**
- **Condition of the WHERE and HAVING clauses**
- **CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses**
- **VALUES clause of the INSERT command**
- **SET clause of the UPDATE command**

ORACLE

Usage of User-Defined Functions

PL/SQL user-defined functions can be called from any SQL expression where a built-in 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
108	960
147	960
205	960
168	920

10 rows selected.

Restrictions on Calling Functions from SQL Expressions

To be callable from SQL expressions, a user-defined function must:

- Be a stored function
- Accept only `IN` parameters
- Accept only valid SQL data types, not PL/SQL specific types, as parameters
- Return data types that are valid SQL data types, not PL/SQL specific types

ORACLE

3-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Restrictions When Calling Functions from SQL Expressions

To be callable from SQL expressions, a user-defined PL/SQL function must meet certain requirements.

- Parameters to a PL/SQL function called from a SQL statement must use positional notation. Named notation is not supported.
- Stored PL/SQL functions cannot be called from the `CHECK` constraint clause of a `CREATE` or `ALTER TABLE` command or be used to specify a default value for a column.
- You must own or have the `EXECUTE` privilege on the function to call it from a SQL statement.
- 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 parameters of the function.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called.

The ability to use a user-defined PL/SQL function in a SQL expression is available with PL/SQL 2.1 and later. Tools using earlier versions of PL/SQL do not support this functionality. Prior to Oracle9i, user-defined functions can be only single-row functions. Starting with Oracle9i, user-defined functions can also be defined as aggregate functions.

Note: Functions that are callable from SQL expressions cannot contain `OUT` and `IN OUT` parameters. Other functions can contain parameters with these modes, but it is not recommended.

Restrictions on Calling Functions from SQL Expressions

- Functions called from SQL expressions cannot contain DML statements.
- Functions called from UPDATE/DELETE statements on a table T cannot contain DML on the same table T.
- Functions called from a DML statement on a table T cannot query the same table.
- Functions called from SQL statements cannot contain statements that end the transactions.
- Calls to subprograms that break the previous restriction are not allowed in the function.

ORACLE

3-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling Side Effects

To execute a SQL statement that calls a stored function, the Oracle server must know whether the function is free of side effects. Side effects are unacceptable changes to database tables. Therefore, restrictions apply to stored functions that are called from SQL expressions.

Restrictions

- When called from a SELECT statement or a parallelized UPDATE or DELETE statement, the function cannot modify any database tables.
- When called from an UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.
- When called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot execute DDL statements (such as CREATE) because they are followed by an automatic commit.
- The function cannot call another subprogram that breaks one of the above restrictions.

Restrictions on Calling from SQL

```
CREATE OR REPLACE FUNCTION dml_call_sql (p_sal NUMBER)
RETURN NUMBER IS
BEGIN
    INSERT INTO employees(employee_id, last_name, email,
                           hire_date, job_id, salary)
    VALUES(1, 'employee 1', 'empl@company.com',
            SYSDATE, 'SA_MAN', 1000);
    RETURN (p_sal + 100);
END;
```

Function created.

```
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 HR.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "HR.DML_CALL_SQL", line 4
ORA-06512: at line 1
```

ORACLE

3-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Restrictions on Calling Functions from SQL: Example

The code example in the slide shows an example of having a DML statement in a function. The function DML_CALL_SQL contains a DML statement that inserts a new record into the EMPLOYEES table. 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 returns an error saying that the table is mutating.

Consider the following example where the function QUERY_CALL_SQL queries the SALARY column of the EMPLOYEE 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;
```

The above function, when invoked from the following UPDATE statement, returns the error message as shown in the slide.

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

Removing Functions

Drop a stored function.

Syntax:

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

Function dropped.

- All the privileges granted on a function are revoked when the function is dropped.
- The **CREATE OR REPLACE** syntax is equivalent to dropping a function and recreating it. Privileges granted on the function remain the same when this syntax is used.

ORACLE

Removing Functions

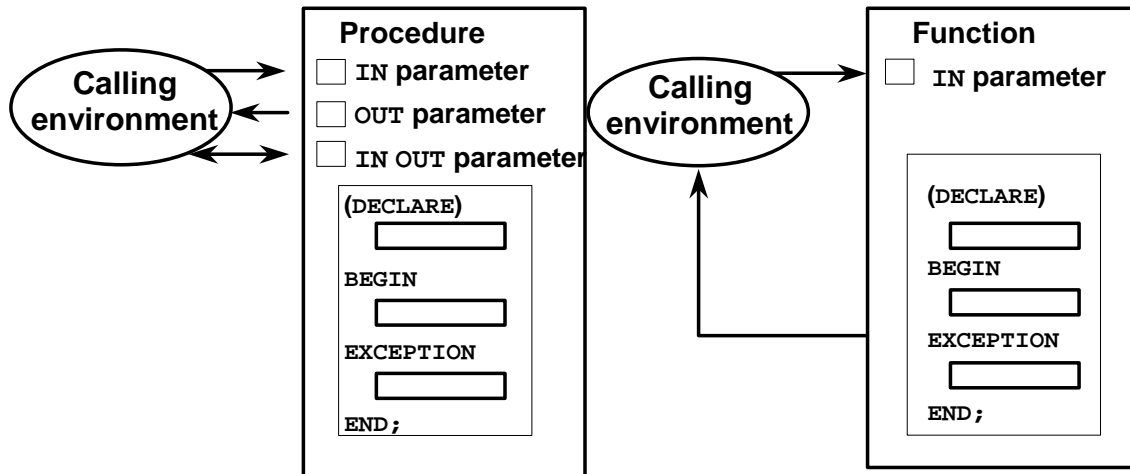
When a stored function is no longer required, you can use a SQL statement in *iSQL*Plus* to drop it.

To remove a stored function by using *iSQL*Plus*, execute the SQL command **DROP FUNCTION**.

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 create it again, all the privileges granted on this function are automatically revoked.

Procedure or Function?



ORACLE

3-17

Copyright © Oracle Corporation, 2001. All rights reserved.

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.

You create a function when you want to compute a value, which must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions should return only a single value, and the value is returned through a RETURN statement. Functions used in SQL statements cannot have OUT or IN OUT mode parameters.

Comparing Procedures and Functions

Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No RETURN clause in the header	Must contain a RETURN clause in the header
Can return none, one, or many values	Must return a single value
Can contain a RETURN statement	Must contain at least one RETURN statement

ORACLE

How Procedures and Functions Differ (continued)

A procedure containing one OUT parameter can be rewritten as a function containing a RETURN statement.

Benefits of Stored Procedures and Functions

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

ORACLE

Benefits

In addition to modularizing application development, stored procedures and functions have the following benefits:

- Improved performance
 - Avoid reparsing for multiple users by exploiting the shared SQL area
 - Avoid PL/SQL parsing at run time by parsing at compile time
 - Reduce the number of calls to the database and decrease network traffic by bundling commands
- Easy maintenance
 - Modify routines online without interfering with other users
 - Modify one routine to affect multiple applications
 - Modify one routine to eliminate duplicate testing
- Improved data security and integrity
 - Control indirect access to database objects from nonprivileged users with security privileges
 - Ensure that related actions are performed together, or not at all, by funneling activity for related tables through a single path
- Improved code clarity: By using appropriate identifier names to describe the actions of the routine, you reduce the need for comments and enhance clarity.

Summary

In this lesson, you should have learned that:

- **A function is a named PL/SQL block that must return a value.**
- **A function is created by using the `CREATE FUNCTION` syntax.**
- **A function is invoked as part of an expression.**
- **A function stored in the database can be called in SQL statements.**
- **A function can be removed from the database by using the `DROP FUNCTION` syntax.**
- **Generally, you use a procedure to perform an action and a function to compute a value.**

ORACLE

3-20

Copyright © Oracle Corporation, 2001. All rights reserved.

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 a procedure to perform an action.

A function can be created or dropped.

A function is invoked as a part of an expression.

Practice 3 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

3-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 3 Overview

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

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

Practice 3

1. Create and invoke the Q_JOB function to return a job title.
 - a. Create a function called Q_JOB to return a job title to a host variable.
 - b. Compile the code; create a host variable G_TITLE and invoke the function with job ID SA_REP. Query the host variable to view the result.

G_TITLE
Sales Representative

2. Create a function called ANNUAL_COMP to return the annual salary by accepting two parameters: an employee's monthly salary and commission. The function should address NULL values.
 - a. Create and invoke the function ANNUAL_COMP, passing in values for monthly salary and commission. Either or both values passed can be NULL, but the function should still return an annual salary, which is not NULL. The annual salary is defined by the basic formula:
$$(\text{salary} * 12) + (\text{commission_pct} * \text{salary} * 12)$$
 - b. Use the function in a SELECT statement against the EMPLOYEES table for department 80.

EMPLOYEE_ID	LAST_NAME	Annual Compensation
145	Russell	235200
146	Partners	210600
147	Errazuriz	187200
148		
177	Livingston	120960
179	Johnson	81840

34 rows selected.

3. Create a procedure, NEW_EMP, to insert a new employee into the EMPLOYEES table. The procedure should contain a call to the VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.
 - a. Create the function VALID_DEPTID to validate a specified department ID. The function should return a BOOLEAN value.
 - b. Create the procedure NEW_EMP to add an employee to the EMPLOYEES table. A new row should be added to the EMPLOYEES table if the function returns TRUE. If the function returns FALSE, the procedure should alert the user with an appropriate message.

Define default values for most parameters. The default commission is 0, the default salary is 1000, the default department number is 30, the default job is SA_REP, and the default manager number is 145. For the employee's ID number, use the sequence EMPLOYEES_SEQ. Provide the last name, first name, and e-mail address for the employee.
 - c. Test your NEW_EMP procedure by adding a new employee named Jane Harris to department 15. Allow all other parameters to default. What was the result?
 - d. Test your NEW_EMP procedure by adding a new employee named Joe Harris to department 80. Allow all other parameters to default. What was the result?

4

Managing Subprograms

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Contrast system privileges with object privileges**
- **Contrast invokers rights with definers rights**
- **Identify views in the data dictionary to manage stored objects**
- **Describe how to debug subprograms by using the DBMS_OUTPUT package**

ORACLE

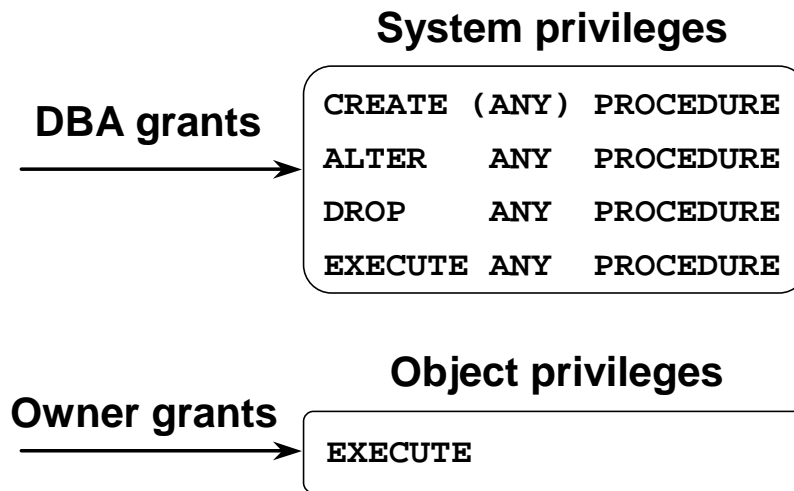
4-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

This lesson introduces you to system and object privilege requirements. You learn how to use the data dictionary to gain information about stored objects. You also learn how to debug subprograms.

Required Privileges



To be able to refer and access objects from a different schema in a subprogram, you must be granted access to the referred objects explicitly, not through a role.

ORACLE

System and Object Privileges

There are more than 80 system privileges. Privileges that use the word CREATE or ANY are system privileges; for example, `GRANT ALTER ANY TABLE TO green;`. System privileges are assigned by user SYSTEM or SYS.

Object privileges are rights assigned to a specific object within a schema and always include the name of the object. For example, Scott can assign privileges to Green to alter his EMPLOYEES table as follows:

```
GRANT ALTER ON employees TO green;
```

To create a PL/SQL subprogram, you must have the system privilege CREATE PROCEDURE. You can alter, drop, or execute PL/SQL subprograms without any further privileges being required.

If a PL/SQL subprogram refers to any objects that are not in the same schema, you must be granted access to these explicitly, not through a role.

If the ANY keyword is used, you can create, alter, drop, or execute your own subprograms and those in another schema. Note that the ANY keyword is optional only for the CREATE PROCEDURE privilege.

You must have the EXECUTE object privilege to invoke the PL/SQL subprogram if you are not the owner and do not have the EXECUTE ANY system privilege.

By default the PL/SQL subprogram executes under the security domain of the owner.

Note: The keyword PROCEDURE is used for stored procedures, functions, and packages.

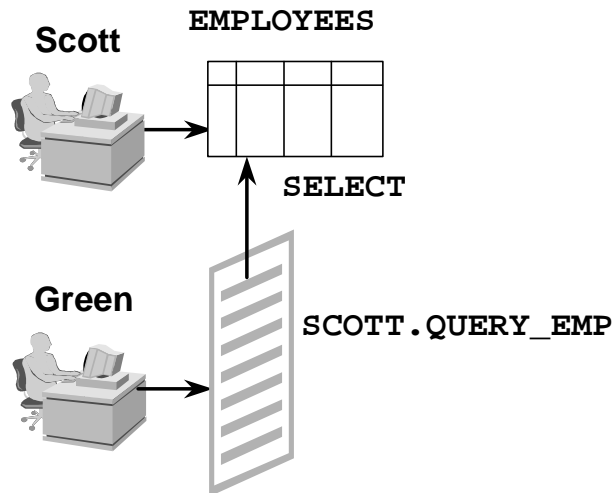
Granting Access to Data

Direct access:

```
GRANT SELECT
ON   employees
TO   scott;
Grant Succeeded.
```

Indirect access:

```
GRANT EXECUTE
ON   query_emp
TO   green;
Grant Succeeded.
```



The procedure executes with the privileges of the owner (default).

ORACLE

4-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Providing Indirect Access to Data

Suppose the **EMPLOYEES** table is located within the **PERSONNEL** schema, and there is a developer named **Scott** and an end user named **Green**. Ensure that **Green** can access the **EMPLOYEES** table only by way of the **QUERY_EMP** procedure that **Scott** created, which queries employee records.

Direct Access

- From the **PERSONNEL** schema, provide object privileges on the **EMPLOYEES** table to **Scott**.
- **Scott** creates the **QUERY_EMP** procedure that queries the **EMPLOYEES** table.

Indirect Access

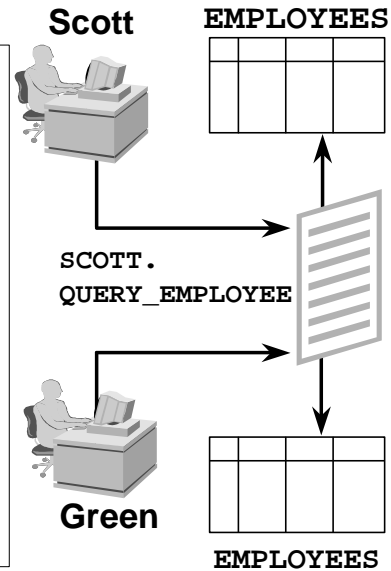
Scott provides the **EXECUTE** object privilege to **Green** on the **QUERY_EMP** procedure.

By default the PL/SQL subprogram executes under the security domain of the owner. This is referred to as definer's-rights. Because **Scott** has direct privileges to **EMPLOYEES** and has created a procedure called **QUERY_EMP**, **Green** can retrieve information from the **EMPLOYEES** table by using the **QUERY_EMP** procedure.

Using Invoker's-Rights

The procedure executes with the privileges of the user.

```
CREATE PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE,
p_name OUT employees.last_name%TYPE,
p_salary OUT employees.salary%TYPE,
p_comm OUT
  employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
  SELECT last_name, salary,
         commission_pct
  INTO   p_name, p_salary, p_comm
  FROM   employees
  WHERE  employee_id=p_id;
END query_employee;
```



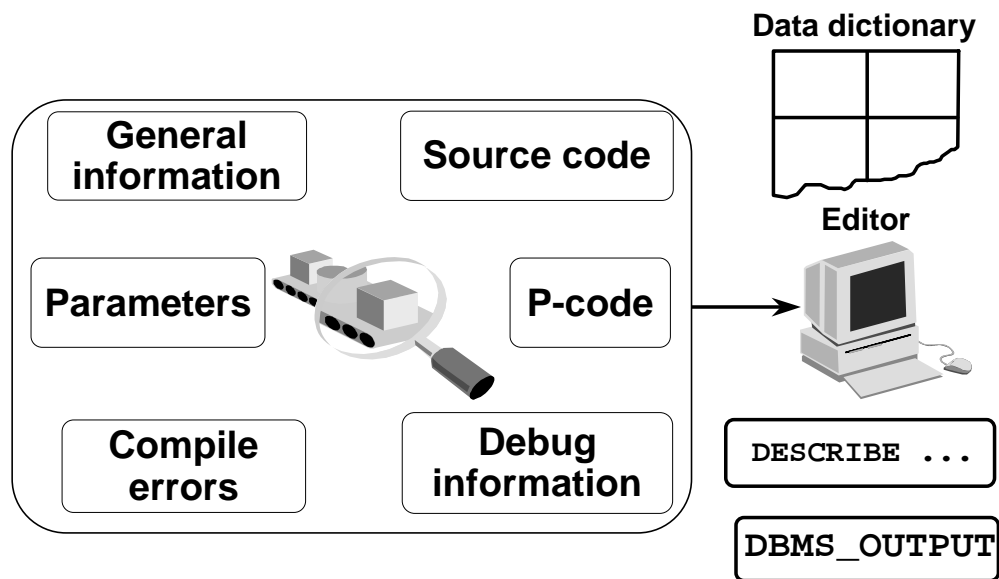
ORACLE

Invoker's-Rights

To ensure that the procedure executes using the security of the executing user, and not the owner, use `AUTHID CURRENT_USER`. This ensures that the procedure executes with the privileges and schema context of its current user.

Default behavior, as shown on the previous page, is when the procedure executes under the security domain of the owner; but if you wanted to explicitly state that the procedure should execute using the owner's privileges, then use `AUTHID DEFINER`.

Managing Stored PL/SQL Objects



Stored Information	Description	Access Method
General	Object information	The USER_OBJECTS data dictionary view
Source code	Text of the procedure	The USER_SOURCE data dictionary view
Parameters	Mode: IN/ OUT/ IN OUT, datatype	iSQL*Plus: DESCRIBE command
P-code	Compiled object code	Not accessible
Compile errors	PL/SQL syntax errors	The USER_ERRORS data dictionary view iSQL*Plus: SHOW ERRORS command
Run-time debug information	User-specified debug variables and messages	The DBMS_OUTPUT Oracle-supplied package

USER_OBJECTS

Column	Column Description
OBJECT_NAME	Name of the object
OBJECT_ID	Internal identifier for the object
OBJECT_TYPE	Type of object, for example, TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
CREATED	Date when the object was created
LAST_DDL_TIME	Date when the object was last modified
TIMESTAMP	Date and time when the object was last recompiled
STATUS	VALID or INVALID

*Abridged column list

ORACLE

Using USER_OBJECTS

To obtain the names of all PL/SQL stored objects within a schema, query the USER_OBJECTS data dictionary view.

You can also examine the ALL_OBJECTS and DBA_OBJECTS views, each of which contains the additional OWNER column, for the owner of the object.

List All Procedures and Functions

```
SELECT object_name, object_type
FROM   user_objects
WHERE  object_type in ('PROCEDURE','FUNCTION')
ORDER BY object_name;
```

OBJECT_NAME	OBJECT_TYPE
ADD_DEPT	PROCEDURE
ADD_JOB	PROCEDURE
ADD_JOB_HISTORY	PROCEDURE
ANNUAL_COMP	FUNCTION
DEL_JOB	PROCEDURE
FORMAT_PHONE	PROCEDURE
LEAVE_EMP	PROCEDURE
LEAVE_EMP2	PROCEDURE
LOG_SESSION	PROCEDURE

20 rows selected.

ORACLE

Example

The example in the slide displays the names of all the procedures and functions that you have created.

USER_SOURCE Data Dictionary View

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
LINE	Line number of the source code
TEXT	Text of the source code line

ORACLE

Using USER_SOURCE

To obtain the text of a stored procedure or function, use the USER_SOURCE data dictionary view.

Also examine the ALL_SOURCE and DBA_SOURCE views, each of which contains the additional OWNER column, for the owner of the object.

If the source file is unavailable, you can use *iSQL*Plus* to regenerate it from USER_SOURCE.

List the Code of Procedures and Functions

```
SELECT text
FROM user_source
WHERE name = 'QUERY_EMPLOYEE'
ORDER BY line;
```

TEXT
PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE, p_name OUT employees.last_name%TYPE,
p_salary OUT employees.salary%TYPE, p_comm OUT employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
SELECT last_name, salary, commission_pct
INTO p_name, p_salary, p_comm
FROM employees
WHERE employee_id=p_id;
END query_employee;

11 rows selected.

ORACLE

Example

Use the USER_SOURCE data dictionary view to display the complete text for the QUERY_EMPLOYEE procedure.

USER_ERRORS

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
SEQUENCE	Sequence number, for ordering
LINE	Line number of the source code at which the error occurs
POSITION	Position in the line at which the error occurs
TEXT	Text of the error message

ORACLE

Obtaining Compile Errors

To obtain the text for compile errors, use the USER_ERRORS data dictionary view or the SHOW ERRORS *iSQL**Plus command.

Also examine the ALL_ERRORS and DBA_ERRORS views, each of which contains the additional OWNER column, for the owner of the object.

Detecting Compilation Errors: Example

```
CREATE OR REPLACE PROCEDURE log_execution
IS
BEGIN
  INPUT INTO log_table (user_id, log_date)
                                -- wrong
VALUES (USER, SYSDATE);
END;
/
```

Warning: Procedure created with compilation errors.

ORACLE

Example

Given the above code for LOG_EXECUTION, there will be a compile error when you run the script for compilation.

List Compilation Errors by Using USER_ERRORS

```
SELECT line || '/' || position POS, text
FROM   user_errors
WHERE  name = 'LOG_EXECUTION'
ORDER BY line;
```

POS	TEXT
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . [@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignored.
6/1	PLS-00103: Encountered the symbol "END"

ORACLE

Listing Compilation Errors, Using USER_ERRORS

The SQL statement above is a SELECT statement from the USER_ERRORS data dictionary view, which you use to see compilation errors.

List Compilation Errors by Using SHOW ERRORS

```
SHOW ERRORS PROCEDURE log_execution
```

Errors for PROCEDURE LOG_EXECUTION:

LINE/COL	ERROR
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: (, % ; limit The symbol "VALUES" was ignored.
6/1	PLS-00103: Encountered the symbol "END"

ORACLE

SHOW ERRORS

Use SHOW ERRORS without any arguments at the SQL prompt to obtain compilation errors for the last object you compiled.

You can also use the command with a specific program unit. The syntax is as follows:

```
SHOW ERRORS [ { FUNCTION | PROCEDURE | PACKAGE | PACKAGE  
              BODY | TRIGGER | VIEW } [ schema . ] name ]
```

Using the SHOW ERRORS command, you can view only the compilation errors that are generated by the latest statement that is used to create a subprogram. The USER_ERRORS data dictionary view stores all the compilation errors generated previously while creating subprograms.

DESCRIBE in iSQL*Plus

```
DESCRIBE query_employee  
DESCRIBE add_dept  
DESCRIBE tax
```

PROCEDURE query_employee

Argument Name	Type	In/Out	Default?
P_ID	NUMBER(5)	IN	
P_NAME	VARCHAR2(25)	OUT	
P_SALARY	NUMBER(8,2)	OUT	
P_COMM	NUMBER(2,2)	OUT	

PROCEDURE add_dept

Argument Name	Type	In/Out	Default?
P_NAME	VARCHAR2(30)	IN	DEFAULT
P_LOC	NUMBER(4)	IN	DEFAULT

FUNCTION tax RETURNS NUMBER

Argument Name	Type	In/Out	Default?
P_VALUE	NUMBER	IN	

ORACLE

Describing Procedures and Functions

To display a procedure or function and its parameter list, use the *iSQL*Plus* DESCRIBE command.

Example

The code in the slide displays the parameter list for the QUERY_EMPLOYEE and ADD_DEPT procedures and the TAX function.

Consider the displayed parameter list for the ADD_DEPT procedure, which has defaults. The DEFAULT column indicates only that there is a default value; it does not give the actual value itself.

Debugging PL/SQL Program Units

- **The DBMS_OUTPUT package:**
 - Accumulates information into a buffer
 - Allows retrieval of the information from the buffer
- **Autonomous procedure calls (for example, writing the output to a log table)**
- **Software that uses DBMS_DEBUG**
 - Procedure Builder
 - Third-party debugging software

ORACLE

4-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Debugging PL/SQL Program Units

Different packages that can be used for debugging PL/SQL program units are shown in the slide. You can use DBMS_OUTPUT packaged procedures to output values and messages from a PL/SQL block. This is done by accumulating information into a buffer and then allowing the retrieval of the information from the buffer. DBMS_OUTPUT is an Oracle-supplied package. You qualify every reference to these procedures with the DBMS_OUTPUT prefix.

Benefits of Using DBMS_OUTPUT Package

This package enables developers to follow closely the execution of a function or procedure by sending messages and values to the output buffer. Within *iSQL*Plus* use SET SERVEROUTPUT ON or OFF instead of using the ENABLE or DISABLE procedure.

Suggested Diagnostic Information

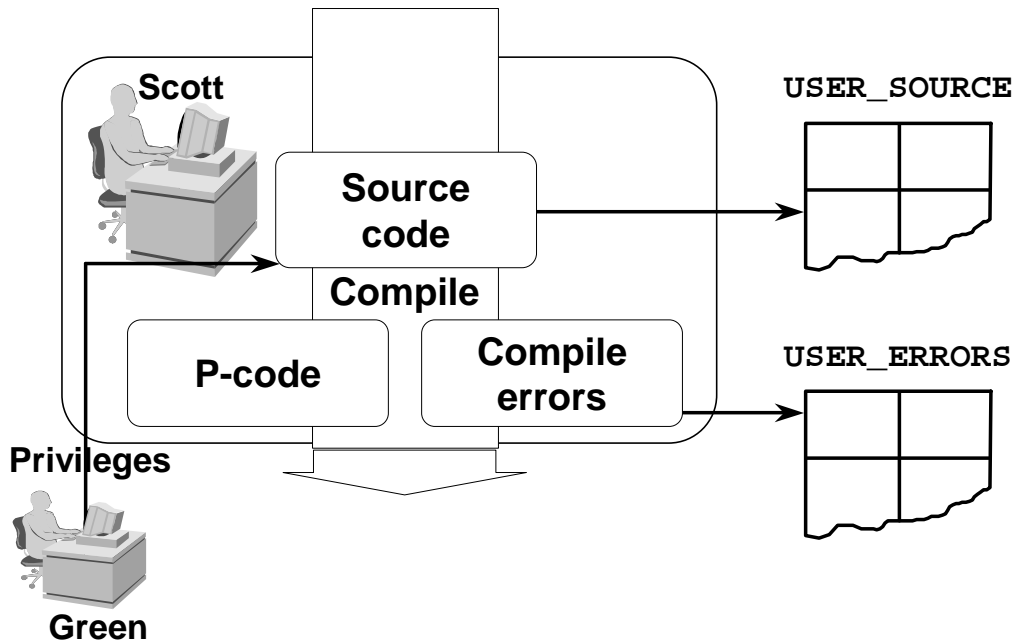
- Message upon entering, leaving a procedure, or indicating that an operation has occurred
- Counter for a loop
- Value for a variable before and after an assignment

Note: The buffer is not emptied until the block terminates.

You can debug subprograms by specifying autonomous procedure calls and store the output as values of columns into a log table.

Debugging using Oracle Procedure Builder is discussed in Appendix F. Procedure Builder uses a Oracle-specified debugging package called DBMS_DEBUG.

Summary



ORACLE

4-17

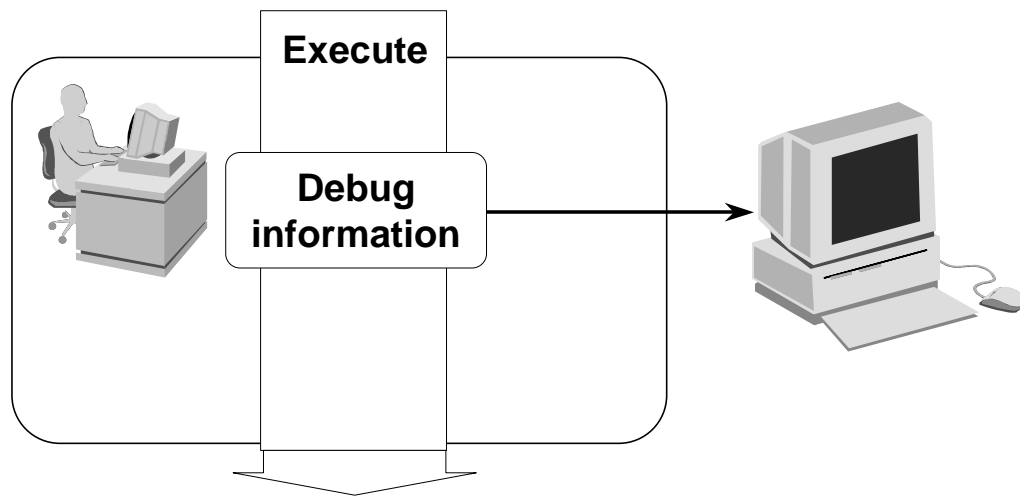
Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

A user must be granted the necessary privileges to access database objects through a subprogram. Take advantage of various data dictionary views, SQL commands, *iSQL*Plus* commands, and Oracle-supplied procedures to manage a stored procedure or function during its development cycle.

Name	Data Dictionary View or Command	Description
USER_OBJECTS	Data dictionary view	Provides general information about the object
USER_SOURCE	Data dictionary view	Provides the text of the object, (that is, the PL/SQL block)
DESCRIBE	<i>iSQL*Plus</i> command	Provides the declaration of the object
USER_ERRORS	Data dictionary view	Shows compilation errors
SHOW ERRORS	<i>iSQL*Plus</i> command	Shows compilation errors, per procedure or function
DBMS_OUTPUT	Oracle-supplied package	Provides user-specified debugging, giving variable values and messages
GRANT	<i>iSQL</i> command	Provides the security privileges for the owner who creates the procedure and the user who runs it, enabling them to perform their respective operations

Summary



ORACLE

4-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary (continued)

- Query the data dictionary.
 - List all your procedures and functions, using the `USER_OBJECTS` view.
 - List the text of certain procedures or functions, using the `USER_SOURCE` view.
- Prepare a procedure: Recreate a procedure and display any compile errors automatically.
- Test a procedure: Test a procedure by supplying input values; test a procedure or function by displaying output or return values.

Practice 4 Overview

This practice covers the following topics:

- Re-creating the source file for a procedure
- Re-creating the source file for a function

ORACLE

4-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 4 Overview

In this practice you will re-create the source code for a procedure and a function.

Practice 4

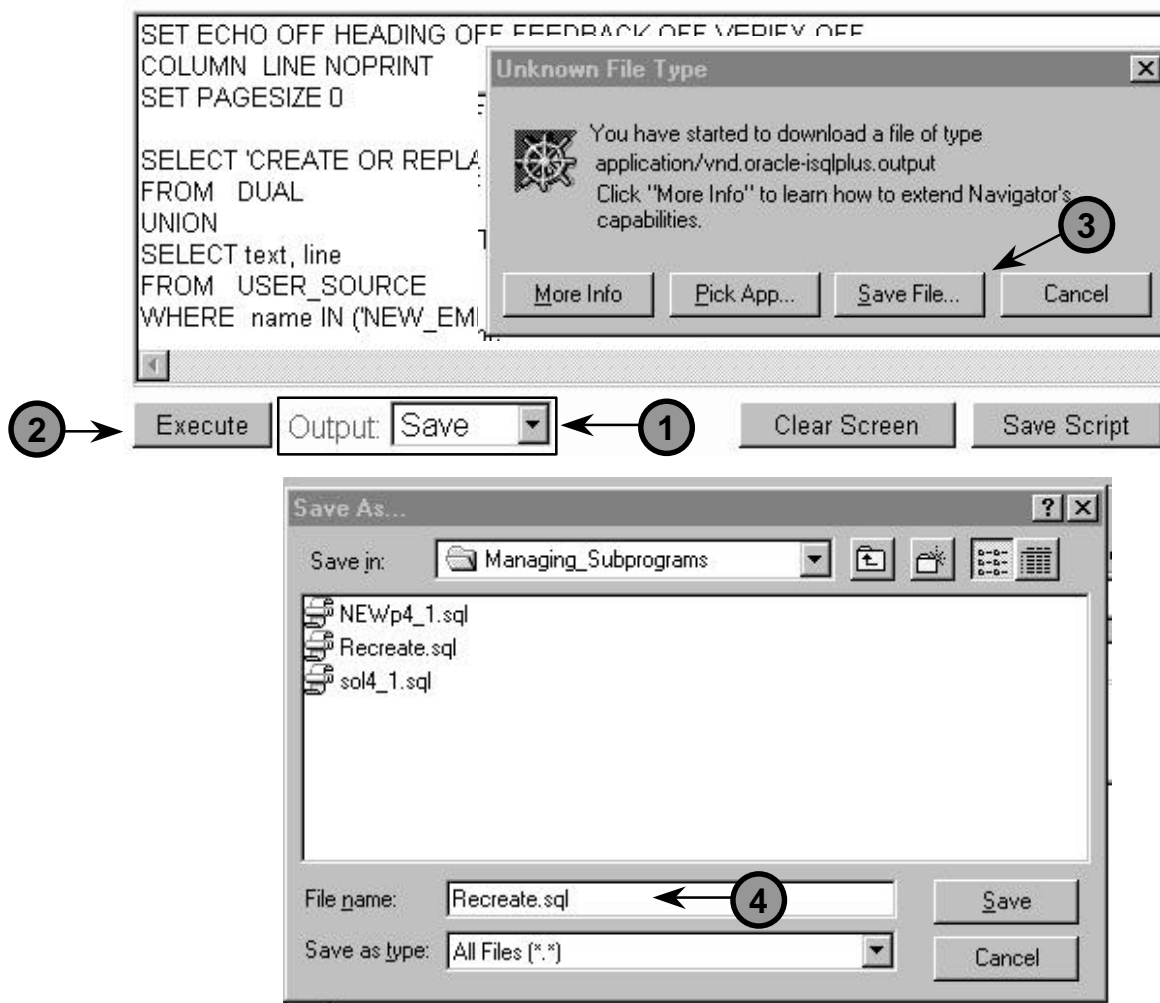
Suppose you have lost the code for the NEW_EMP procedure and the VALID_DEPTNO function that you created in lesson 3. (If you did not complete the practices in lesson 3, you can run the solution scripts to create the procedure and function.)

Create a iSQL*Plus spool file to query the appropriate data dictionary view to regenerate the code.

Hint:

```
SET                                -- options ON|OFF
SELECT                            -- statement(s) to extract the code
SET                                -- reset options ON|OFF
```

To spool the output of the file to a .sql file from iSQL*Plus, select the Save option for the Output and execute the code.



5

Creating Packages

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe packages and list their possible 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 a use for a bodiless package**

ORACLE

5-2

Copyright © Oracle Corporation, 2001. All rights reserved.

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.

Overview of Packages

Packages:

- **Group logically related PL/SQL types, items, and subprograms**
- **Consist of two parts:**
 - **Specification**
 - **Body**
- **Cannot be invoked, parameterized, or nested**
- **Allow the Oracle server to read multiple objects into memory at once**

ORACLE

5-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Packages Overview

Packages bundle related PL/SQL types, items, 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 has a specification and a body, stored separately in the database.

The specification is the interface to your applications. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGMA's, which are directives to the compiler.

The body fully defines cursors and subprograms, and so implements the specification.

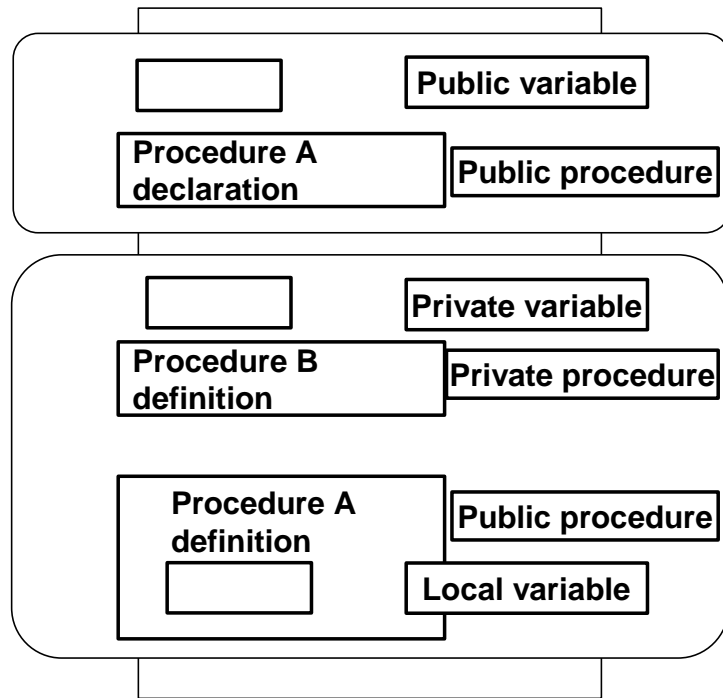
The package itself cannot be called, parameterized, or nested. Still, the format of a package is similar to that of a subprogram. Once written and compiled, the contents can be shared by many applications.

When you call a packaged PL/SQL construct for the first time, the whole package is loaded into memory. Thus, later calls to constructs in the same package require no disk input/output (I/O).

Components of a Package

**Package
specification**

**Package
body**



ORACLE

5-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Package Development

You create a package in two parts: first the package specification, and then the package body. Public package constructs are those that are declared in the package specification and defined in the package body. Private package constructs are those that are defined solely within the package body.

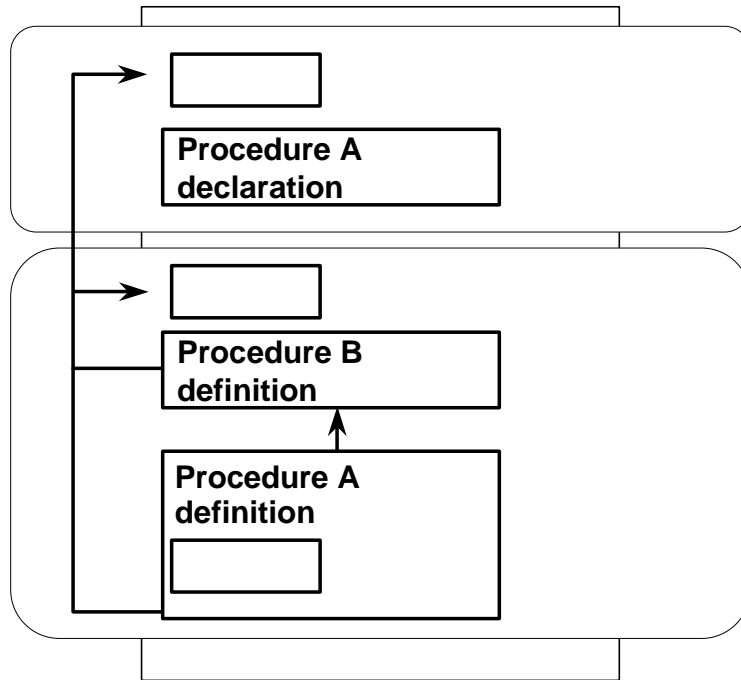
Scope of the Construct	Description	Placement within the Package
Public	Can be referenced from any Oracle server environment	Declared within the package specification and may be defined within the package body
Private	Can be referenced only by other constructs which are part of the same package	Declared and defined within the package body

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

Referencing Package Objects

**Package
specification**

**Package
body**



ORACLE

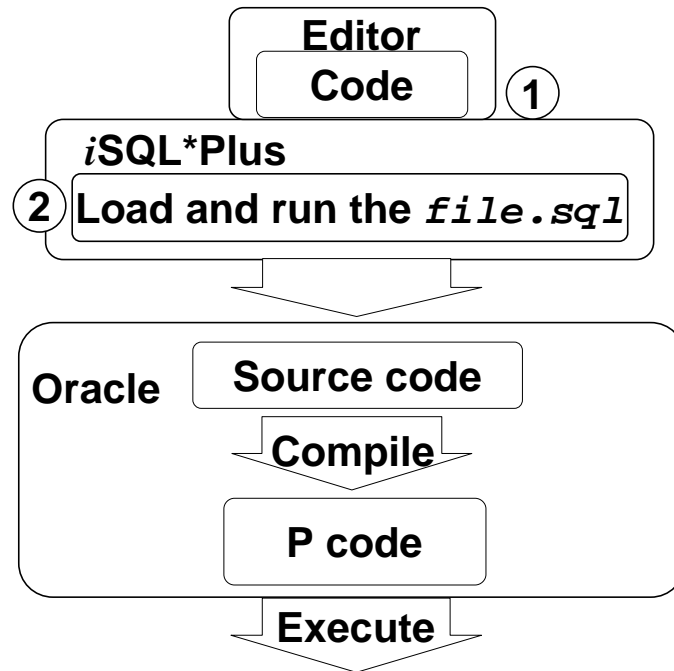
5-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Package Development (continued)

Visibility of the Construct	Description
Local	<p>A variable defined within a subprogram that is not visible to external users.</p> <p>Private (local to the package) variable: You can define variables in a package body. These variables can be accessed only by other objects in the same package. They are not visible to any subprograms or objects outside of the package.</p>
Global	<p>A variable or subprogram that can be referenced (and changed) outside the package and is visible to external users. Global package items must be declared in the package specification.</p>

Developing a Package



ORACLE

How to Develop a Package

1. Write the syntax: Enter the code in a text editor and save it as a SQL script file.
2. Compile the code: Run the SQL script file to generate and compile the source code. The source code is compiled into P code.

Developing a Package

- **Saving the text of the `CREATE PACKAGE` statement in two different SQL files facilitates later modifications to the package.**
- **A package specification can exist without a package body, but a package body cannot exist without a package specification.**

ORACLE

5-7

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Develop a Package

There are three basic steps to developing a package, similar to those steps that are used to develop a stand-alone procedure.

1. Write the text of the `CREATE PACKAGE` statement within a SQL script file to create the package specification and run the script file. The source code is compiled into P code and is stored within the data dictionary.
2. Write the text of the `CREATE PACKAGE BODY` statement within a SQL script file to create the package body and run the script file.
The source code is compiled into P code and is also stored within the data dictionary.
3. Invoke any public construct within the package from an Oracle server environment.

Creating the Package Specification

Syntax:

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

- The **REPLACE** option drops and recreates 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

5-8

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Create a Package Specification

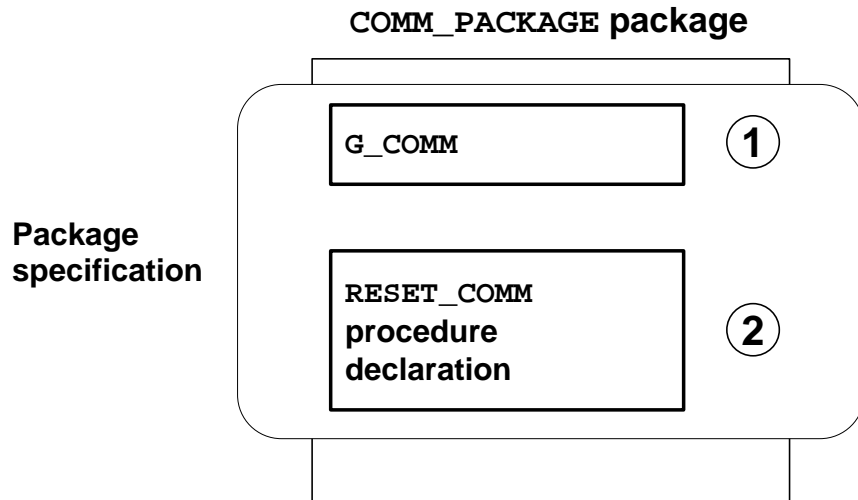
To create packages, you declare all public constructs within the package specification.

- Specify the **REPLACE** option when the package specification already exists.
- Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to **NULL**.

Syntax Definition

Parameter	Description
<i>package_name</i>	Name the package
<i>public type and item declarations</i>	Declare variables, constants, cursors, exceptions, or types
<i>subprogram specifications</i>	Declare the PL/SQL subprograms

Declaring Public Constructs



ORACLE

5-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a Package Specification

In the slide, G_COMM is a public (global) variable, and RESET_COMM is a public procedure.

In the package specification, you declare public variables, public procedures, and public functions.

The public procedures or functions are routines that can be invoked repeatedly by other constructs in the same package or from outside the package.

Creating a Package Specification: Example

```
CREATE OR REPLACE PACKAGE comm_package IS
  g_comm NUMBER := 0.10; --initialized to 0.10
  PROCEDURE reset_comm
    (p_comm IN NUMBER);
END comm_package;
/
```

Package created.

- **G_COMM** is a global variable and is initialized to 0.10.
- **RESET_COMM** is a public procedure that is implemented in the package body.

ORACLE

5-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Package Specification for COMM_PACKAGE

In the slide, the variable G_COMM and the procedure RESET_COMM are public constructs.

Creating the Package Body

Syntax:

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

- The REPLACE option drops and recreates the package body.
- Identifiers defined only in the package body are private constructs. These are not visible outside the package body.
- All private constructs must be declared before they are used in the public constructs.

ORACLE

5-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating the Package Body

To create packages, define all public and private constructs within the package body.

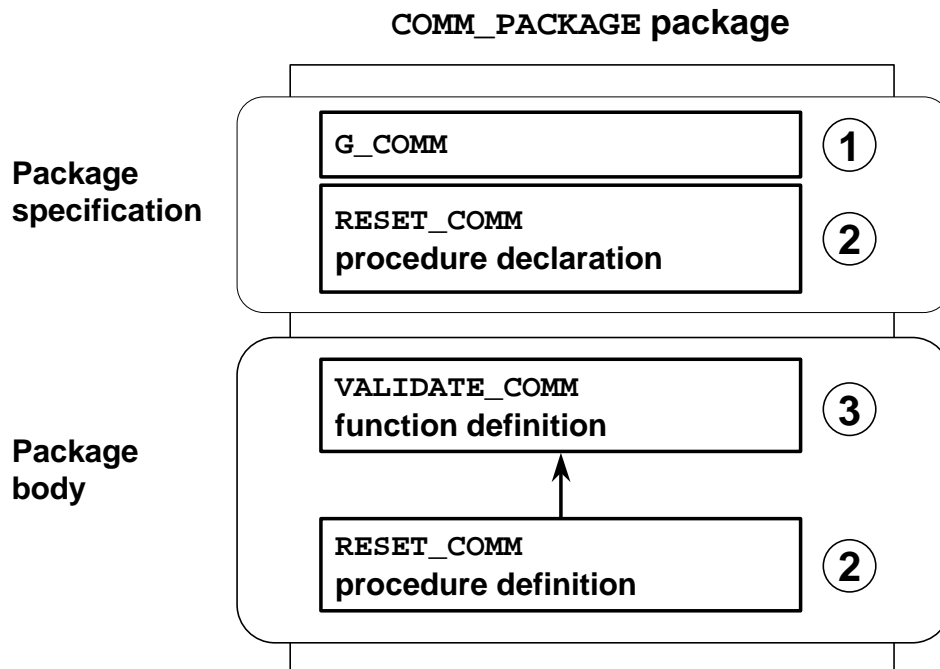
- Specify the REPLACE option when the package body already exists.
- The order in which subprograms are defined within the package body is important: you must declare a variable before another variable or subprogram can refer to it, and you must declare or define private subprograms before calling them from other subprograms. It is quite common in the package body to see all private variables and subprograms defined first and the public subprograms defined last.

Syntax Definition

Define all public and private procedures and functions in the package body.

Parameter	Description
<i>package_name</i>	Is the name of the package
<i>private type and item declarations</i>	Declares variables, constants, cursors, exceptions, or types
<i>subprogram bodies</i>	Defines the PL/SQL subprograms, public and private

Public and Private Constructs



ORACLE

5-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Create a Package Body Example

In the slide on this page:

- 1 is a public (global) variable
- 2 is a public procedure
- 3 is a private function

You can define a private procedure or function to modularize and clarify the code of public procedures and functions.

Note: In the slide, the private function is shown above the public procedure. When you are coding the package body, the definition of the private function has to be above the definition of the public procedure.

Only subprograms and cursors declarations without body in a package specification have an underlying implementation in the package body. So if a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. However, the body can still be used to initialize items declared in the package specification.

Creating a Package Body: Example

comm_pack.sql

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
    FUNCTION validate_comm (p_comm IN NUMBER)
        RETURN BOOLEAN
    IS
        v_max_comm    NUMBER;
    BEGIN
        SELECT      MAX(commission_pct)
          INTO      v_max_comm
        FROM        employees;
        IF    p_comm > v_max_comm THEN RETURN(FALSE);
        ELSE    RETURN(TRUE);
        END IF;
    END validate_comm;
    ...

```

ORACLE

Package Body for COMM_PACKAGE

Define a function to validate the commission. The commission may not be greater than the highest commission among all existing employees.

Creating a Package Body: Example

`comm_pack.sql`

```
PROCEDURE reset_comm (p_comm IN NUMBER)
IS
BEGIN
  IF validate_comm(p_comm)
  THEN g_comm:=p_comm; --reset global variable
  ELSE
    RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
  END IF;
END reset_comm;
END comm_package;
/
```

Package body created.

ORACLE

Package Body for `COMM_PACKAGE` (continued)

Define a procedure that enables you to reset and validate the prevailing commission.

Invoking Package Constructs

Example 1: Invoke a function from a procedure within the same package.

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
    . . .
    PROCEDURE reset_comm
        (p_comm IN NUMBER)
    IS
    BEGIN
        IF validate_comm(p_comm)
        THEN g_comm := p_comm;
        ELSE
            RAISE_APPLICATION_ERROR
                (-20210, 'Invalid commission');
        END IF;
    END reset_comm;
END comm_package;
```

ORACLE

Invoking Package Constructs

After the package is stored in the database, you can invoke a package construct within the package or from outside the package, depending on whether the construct is private or public.

When you invoke a package procedure or function from within the same package, you do not need to qualify its name.

Example 1

Call the `VALIDATE_COMM` function from the `RESET_COMM` procedure. Both subprograms are in the `COMM_PACKAGE` package.

Invoking Package Constructs

Example 2: Invoke a package procedure from *iSQL*Plus*.

```
EXECUTE comm_package.reset_comm(0.15)
```

Example 3: Invoke a package procedure in a different schema.

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

Example 4: Invoke a package procedure in a remote database.

```
EXECUTE comm_package.reset_comm@ny(0.15)
```

ORACLE

Invoking Package Constructs (continued)

When you invoke a package procedure or function from outside the package, you must qualify its name with the name of the package.

Example 2

Call the `RESET_COMM` procedure from *iSQL*Plus*, making the prevailing commission 0.15 for the user session.

Example 3

Call the `RESET_COMM` procedure that is located in the `SCOTT` schema from *iSQL*Plus*, making the prevailing commission 0.15 for the user session.

Example 4

Call the `RESET_COMM` procedure that is located in a remote database that is determined by the database link named `NY` from *iSQL*Plus*, making the prevailing commission 0.15 for the user session.

Adhere to normal naming conventions for invoking a procedure in a different schema, or in a different database on another node.

Declaring a Bodiless Package

```
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;
/

EXECUTE DBMS_OUTPUT.PUT_LINE('20 miles = ' || 20 *
                             global_consts.mile_2_kilo || ' km')
```

Package created.
20 miles = 32.186 km
PL/SQL procedure successfully completed.

ORACLE

5-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Declaring a Bodiless Package

You can declare public (global) variables that exist for the duration of the user session. You can create a package specification that does not need a package body. As discussed earlier in this lesson, if a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary.

Example

In the example in the slide, a package specification containing several conversion rates is defined. All the global identifiers are declared as constants.

A package body is not required to support this package specification because implementation details are not required for any of the constructs of the package specification.

Referencing a Public Variable from a Stand-Alone Procedure

Example:

```
CREATE OR REPLACE PROCEDURE meter_to_yard
    (p_meter IN NUMBER, p_yard OUT NUMBER)
IS
BEGIN
    p_yard := p_meter * global_consts.meter_2_yard;
END meter_to_yard;
/
VARIABLE yard NUMBER
EXECUTE meter_to_yard (1, :yard)
```

Procedure created.
PL/SQL procedure successfully completed.

YARD
1.0936

ORACLE

Example

Use the METER_TO_YARD procedure to convert meters to yards, using the conversion rate packaged in GLOBAL_CONSTS.

When you reference a variable, cursor, constant, or exception from outside the package, you must qualify its name with the name of the package.

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;
```

ORACLE

Removing a Package

When a package is no longer required, you can use a SQL statement in *iSQL*Plus* to drop it. A package has two parts, so you can drop the whole package or just the package body and retain the package specification.

Guidelines for Developing 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

5-20

Copyright © Oracle Corporation, 2001. All rights reserved.

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 users of the package. That way 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 gets 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 Packages

- **Modularity: Encapsulate related constructs.**
- **Easier application design: Code and compile 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

5-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Advantages of Using Packages

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

Modularity

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 can decide which constructs are public (visible and accessible) or private (hidden and inaccessible). Only the 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 calling programs. Also, by hiding implementation details from users, you protect the integrity of the package.

Advantages of 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**

ORACLE

5-22

Copyright © Oracle Corporation, 2001. All rights reserved.

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 only be accessed within the package.

Better Performance

When you call a packaged subprogram the first time, the entire package is loaded into memory. This way, later calls to related subprograms in the package require no further disk I/O. Packaged subprograms also stop cascading dependencies and so 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 datatype.

Summary

In this lesson, you should have learned how to:

- **Improve organization, management, security, and performance by using packages**
- **Group related procedures and functions together in a package**
- **Change a package body without affecting a package specification**
- **Grant security access to the entire package**

ORACLE

5-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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

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

Summary

In this lesson, you should have learned how to:

- **Hide the source code from users**
- **Load the entire package into memory on the first call**
- **Reduce disk access for subsequent calls**
- **Provide identifiers for the user session**

ORACLE

5-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary (continued)

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 the package body
DROP PACKAGE BODY	Remove the package body only

ORACLE

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 5 Overview

This practice covers the following topics:

- **Creating packages**
- **Invoking package program units**

ORACLE

5-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 5 Overview

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

Practice 5

1. Create a package specification and body called `JOB_PACK`. (You can save the package body and specification in two separate files.) This package contains your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures, as well as your `Q_JOB` function.

Note: Use the code in your previously saved script files when creating the package.

- a. Make all the constructs public.

Note: Consider whether you still need the stand-alone procedures and functions you just packaged.

- b. Invoke your `ADD_JOB` procedure by passing values `IT_SYSAN` and `SYSTEMS ANALYST` as parameters.
- c. 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_PACK` that contains your `NEW_EMP` procedure as a public construct, and your `VALID_DEPTID` function as a private construct. (You can save the specification and body into separate files.)
 - b. Invoke the `NEW_EMP` procedure, using 15 as a department number. Because the department ID 15 does not exist in the `DEPARTMENTS` table, you should get an error message as specified in the exception handler of your procedure.
 - c. Invoke the `NEW_EMP` procedure, using an existing department ID 80.

If you have time:

3.
 - a. Create a package called `CHK_PACK` that contains the procedures `CHK_HIREDATE` and `CHK_DEPT_MGR`. Make both constructs public. (You can save the specification and body into separate files.)

The procedure `CHK_HIREDATE` checks whether an employee's hire date is within the following range: `[SYSDATE - 50 years, SYSDATE + 3 months]`.

Note:

- If the date is invalid, you should raise an application error with an appropriate message indicating why the date value is not acceptable.
- Make sure the time component in the date value is ignored.
- Use a constant to refer to the 50 years boundary.
- A null value for the hire date should be treated as an invalid hire date.

The procedure `CHK_DEPT_MGR` checks the department and manager combination for a given employee. The `CHK_DEPT_MGR` procedure accepts an employee ID and a manager ID. The procedure checks that the manager and employee work in the same department. The procedure also checks that the job title of the manager ID provided is `MANAGER`.

Note: If the department ID and manager combination is invalid, you should raise an application error with an appropriate message.

Practice 5 (continued)

- b. Test the CHK_HIREDATE procedure with the following command:

```
EXECUTE chk_pack.chk_hiredate('01-JAN-47')
```

What happens, and why?

- c. Test the CHK_HIREDATE procedure with the following command:

```
EXECUTE chk_pack.chk_hiredate(NULL)
```

What happens, and why?

- d. Test the CHK_DEPT_MGR procedure with the following command:

```
EXECUTE chk_pack.chk_dept_mgr(117,100)
```

What happens, and why?

6

More Package Concepts

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Write packages that use the overloading feature**
- **Describe errors with mutually referential subprograms**
- **Initialize variables with a one-time-only procedure**
- **List the four purity levels of a function**
- **Identify persistent states**

ORACLE

6-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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

Overloading

- Enables you to use the same name for different subprograms inside a PL/SQL block, a subprogram, or a package
- Requires the formal parameters of the subprograms to differ in number, order, or data type family
- Enables you to build more flexibility because a user or application is not restricted by the specific data type or number of formal parameters

Note: Only local or packaged subprograms can be overloaded. You cannot overload stand-alone subprograms.

ORACLE

6-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Overloading

This feature enables you to define different subprograms with the same name. You can distinguish the subprograms both by name and by parameters. Sometimes the processing in two subprograms is the same, but the parameters passed to them varies. In that case it is logical to give them the same name. PL/SQL determines which subprogram is called by checking its formal parameters. Only local or packaged subprograms can be overloaded. Stand-alone subprograms cannot be overloaded.

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 above features.

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

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 like-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

over_pack.sql

```
CREATE OR REPLACE PACKAGE over_pack
IS
  PROCEDURE add_dept
    (p_deptno IN departments.department_id%TYPE,
     p_name IN departments.department_name%TYPE
                                     DEFAULT 'unknown',
     p_loc IN departments.location_id%TYPE DEFAULT 0);
  PROCEDURE add_dept
    (p_name IN departments.department_name%TYPE
                                     DEFAULT 'unknown',
     p_loc IN departments.location_id%TYPE DEFAULT 0);
END over_pack;
/
```

Package created.

ORACLE

6-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Overloading: Example

The slide shows the package specification of a package with overloaded procedures.

The package contains ADD_DEPT as the name of two overloaded procedures. The first definition takes three parameters to be able to insert a new department to the department table. The second definition takes only two parameters, because the department ID is populated through a sequence.

Overloading: Example

over_pack_body.sql

```
CREATE OR REPLACE PACKAGE BODY over_pack IS
  PROCEDURE add_dept
    (p_deptno IN departments.department_id%TYPE,
     p_name IN departments.department_name%TYPE DEFAULT 'unknown',
     p_loc IN departments.location_id%TYPE DEFAULT 0)
  IS
  BEGIN
    INSERT INTO departments (department_id,
                             department_name, location_id)
    VALUES (p_deptno, p_name, p_loc);
  END add_dept;
  PROCEDURE add_dept
    (p_name IN departments.department_name%TYPE DEFAULT 'unknown',
     p_loc IN departments.location_id%TYPE DEFAULT 0)
  IS
  BEGIN
    INSERT INTO departments (department_id,
                             department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
  END add_dept;
END over_pack;
/
```

ORACLE

6-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Overloading Example (continued)

If you call ADD_DEPT with an explicitly provided department ID, PL/SQL uses the first version of the procedure. If you call ADD_DEPT with no department ID, PL/SQL uses the second version.

```
EXECUTE over_pack.add_dept (980,'Education',2500)
```

```
EXECUTE over_pack.add_dept ('Training', 2400)
```

```
SELECT * FROM departments
WHERE department_id = 980;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
980	Education		2500

```
SELECT * FROM departments
WHERE department_name = 'Training';
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Training		2400

Overloading: Example

- **Most built-in functions are overloaded.**
- **For example, see the TO_CHAR function of the STANDARD package.**

```
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;
```

- **If you redeclare a built-in subprogram in a PL/SQL program, your local declaration overrides the global declaration.**

ORACLE

6-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Overloading Example (continued)

Most built-in functions are overloaded. For example, the function TO_CHAR in the package STANDARD has four different declarations, as shown in the slide. The function can take either the DATE or the NUMBER data type and convert it to the character data type. The format into 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, your local declaration overrides the standard or built-in subprogram. To be able to access the built-in subprogram, you need to qualify it with its package name. For example, if you redeclare the TO_CHAR function, to access the built-in function you refer it as: STANDARD.TO_CHAR.

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

Using Forward Declarations

You must declare identifiers before referencing them.

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
  PROCEDURE award_bonus(. . .)
  IS
  BEGIN
    calc_rating(. . .);      --illegal reference

  END;

  PROCEDURE calc_rating(. . .)
  IS
  BEGIN
    ...
  END;

END forward_pack;
/
```

ORACLE

6-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Forward Declarations

PL/SQL does not allow forward references. You must declare an identifier before using it. Therefore, a subprogram must be declared before calling it.

In the example in the slide, the procedure `CALC_RATING` 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 always work. Suppose the procedures call each other or you absolutely want to define them in alphabetical order.

PL/SQL enables for a special subprogram declaration called a forward declaration. It consists of the subprogram specification terminated by a semicolon. You can use forward declarations to do the following:

- Define subprograms in logical or alphabetical order
- Define mutually recursive subprograms
- Group subprograms in a package

Mutually recursive programs are programs that call each other directly or indirectly.

Using Forward Declarations

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
    PROCEDURE calc_rating(. . .);    -- forward declaration
    PROCEDURE award_bonus(. . .)
    IS                                -- subprograms defined
    BEGIN                            -- in alphabetical order
        calc_rating(. . .);
        . . .
    END;

    PROCEDURE calc_rating(. . .)
    IS
    BEGIN
        . . .
    END;

END forward_pack;
/
```

ORACLE

6-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Forward Declarations (continued)

- The formal parameter list 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

Forward declarations typically let you group related subprograms in a package. The subprogram specifications go in the package specification, and the subprogram bodies go in the package body, where they are invisible to the applications. In this way, packages enable you to hide implementation details.

Creating a One-Time-Only Procedure

```
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

Define an Automatic, One-Time-Only Procedure

A one-time-only procedure is executed only once, when the package is first invoked within the user session. In the preceding slide, the current value for TAX is set to the value in the TAX_RATES table the first time the TAXES package is referenced.

Note: Initialize public or private variables with an automatic, one-time-only procedure when the derivation is too complex to embed within the variable declaration. In this case, do not initialize the variable in the declaration, because the value is reset by the one-time-only procedure.

The keyword END is not used at the end of a one-time-only procedure. Observe that in the example in the slide, there is no END at the end of the one-time-only procedure.

Restrictions on Package Functions Used in SQL

A function called from:

- **A query or DML statement can not end the current transaction, create or roll back to a savepoint, or ALTER the system or session.**
- **A query statement or a parallelized DML statement can not execute a DML statement or modify the database.**
- **A DML statement can not read or modify the particular table being modified by that DML statement.**

Note: Calls to subprograms that break the above restrictions are not allowed.

ORACLE

6-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling Side Effects

For the Oracle server to execute a SQL statement that calls a stored function, it must know the purity level of a stored functions, that is, whether the functions are free of side effects. Side effects are changes to database tables or public packaged variables (those declared in a package specification). Side effects could 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 side effects are not allowed when a function is called from a SQL query or DML statement. Therefore, the following restrictions apply to stored functions called from SQL expressions:

- A function called from a query or DML statement can not 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 can not execute a DML statement or otherwise modify the database
- A function called from a DML statement can not read or modify the particular table being modified by that DML statement

Note: In releases prior to Oracle8i, the purity checking used to be performed during compilation time, by including the `PRAGMA RESTRICT_REFERENCES` compiler directive in the package specification. But from Oracle8i, a user-written function can be called from a SQL statement without any compile-time checking of its purity. You can use `PRAGMA RESTRICT_REFERENCES` to ask the PL/SQL compiler to verify that a function has only the side effects that you expect. SQL statements, package variable accesses, or calls to functions that violate the declared restrictions continue to raise PL/SQL compilation errors to help you isolate the code that has unintended effects.

Note: The restrictions on functions discussed above are the same as those discussed in the lesson “*Creating Functions.*”

User Defined Package: taxes_pack

```
CREATE OR REPLACE PACKAGE taxes_pack
IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER;
END taxes_pack;
/
```

Package created.

```
CREATE OR REPLACE PACKAGE BODY taxes_pack
IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER
    IS
        v_rate NUMBER := 0.08;
    BEGIN
        RETURN (p_value * v_rate);
    END tax;
END taxes_pack;
/
```

Package body created.

ORACLE

Example

Encapsulate the function TAX in the package TAXES_PACK. The function is called from SQL statements on remote databases.

Invoking a User-Defined Package Function from a SQL Statement

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

TAXES_PACK.TAX(SALARY)	SALARY	LAST_NAME
1920	24000	King
1360	17000	Kochhar
1360	17000	De Haan
720	9000	Hunold
480	6000	Ernst
384	4800	Austin
384	4800	Pataballa

109 rows selected.

ORACLE

Calling Package Functions

You call PL/SQL functions the same way that you call built-in SQL functions.

Example

Call the TAX function (in the TAXES_PACK package) from a SELECT statement.

Note: If you are using Oracle versions prior to 8i, you need to 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 function TAX does not guarantee that it will not update the database while invoking the package function in a query.

Persistent State of Package Variables: Example

```
CREATE OR REPLACE PACKAGE comm_package IS
  g_comm NUMBER := 10;           --initialized to 10
  PROCEDURE reset_comm (p_comm IN NUMBER);
END comm_package;
/
```

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
  FUNCTION validate_comm (p_comm IN NUMBER)
    RETURN BOOLEAN
  IS v_max_comm NUMBER;
  BEGIN
    ...      -- validates commission to be less than maximum
              -- commission in the table
  END validate_comm;
  PROCEDURE reset_comm (p_comm IN NUMBER)
  IS BEGIN
    ...      -- calls validate_comm with specified value
  END reset_comm;
END comm_package;
/
```

ORACLE

Persistent State of Package Variables

This sample package illustrates the persistent state of package variables. The `VALIDATE_COMM` function validates commission to be no more than maximum currently earned. The `RESET_COMM` procedure invokes the `VALIDATE_COMM` function. If you try to reset the commission to be higher than the maximum, the exception `RAISE_APPLICATION_ERROR` is raised. On the next page, the `RESET_COMM` procedure is used in the example.

Note: Refer to page 13 of lesson 5 for the code of the `VALIDATE_COMM` function and the `RESET_COMM` procedure. In the `VALIDATE_COMM` function, the maximum salary from the `EMPLOYEES` table is selected into the variable `V_MAXSAL`. Once the variable is assigned a value, the value persists in the session until it is modified again. The example in the following slide shows how the value of a global package variable persists for a session.

Persistent State of Package Variables

Time	Scott	Jones
9:00	EXECUTE comm_package.reset_comm (0.25) max_comm=0.4 > 0.25 g_comm = 0.25	INSERT INTO employees (last_name, commission_pct) VALUES ('Madonna', 0.8); max_comm=0.8
9:30		
9:35		EXECUTE comm_package.reset_comm(0.5) max_comm=0.8 > 0.5 g_comm = 0.5
10:00	EXECUTE comm_package.reset_comm (0.6) max_comm=0.4 < 0.6 INVALID	
11:00		ROLLBACK;
11:01		EXIT
11:45		Logged In again. g_comm = 10, max_comm=0.4
12:00	VALID	EXECUTE comm_package.reset_comm(0.25)

ORACLE

Controlling the Persistent State of a Package Variable

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.

1. Initialize the variable within its declaration or within an automatic, one-time-only procedure.
2. Change the value of the variable by means of package procedures.
3. The value of the variable is released when the user disconnects.

The sequence of steps in the slide shows how the state of a package variable persists.

9:00: When Scott invoked the procedure RESET_COMM with a commission percentage value 0.25, the global variable G_COMM was initialized to 10 in his session. The value 0.25 was validated with the maximum commission percentage value 0.4 (obtained from the EMPLOYEES table). Because 0.25 is less than 0.4, the global variable was set to 0.25. 9:30: Jones inserted a new row into EMPLOYEES table with commission percentage value 0.8.

9:35: Jones invoked the procedure RESET_COMM with a commission percentage value 0.5. The global variable G_COMM was initialized to 10 in his session. The value 0.5 was validated with the maximum commission percentage value 0.8 (because the new row has 0.8). Because 0.5 is less than 0.8, the global variable was set to 0.5.

10:00: Scott invoked the procedure with commission percentage value of 0.6. This value is more than the maximum commission percentage 0.4 (Scott could not see new value because Jones did not complete the transaction). Hence, it was invalid.

11:00 to 12:00: Jones rolled back the transaction and exited the session. The global value was initialized to 10 when he logged in at 11:45. The procedure was successful because the new value 0.25 is less than the maximum value 0.4.

Controlling the Persistent State of a Package Cursor

Example:

```
CREATE OR REPLACE PACKAGE pack_cur
IS
  CURSOR c1 IS SELECT employee_id
                FROM employees
                ORDER BY employee_id DESC;
  PROCEDURE proc1_3rows;
  PROCEDURE proc4_6rows;
END pack_cur;
/
```

Package created.

ORACLE

6-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling the Persistent State of a Package Cursor

Example

Use the following steps to control a public cursor:

1. Declare the public (global) cursor in the package specification.
2. Open the cursor and fetch successive rows from the cursor, using one (public) packaged procedure, PROC1_3ROWS.
3. Continue to fetch successive rows from the cursor, and then close the cursor by using another (public) packaged procedure, PROC4_6ROWS.

The slide shows the package specification for PACK_CUR.

Controlling the Persistent State of a Package Cursor

```
CREATE OR REPLACE PACKAGE BODY pack_cur IS
  v_empno NUMBER;
  PROCEDURE proc1_3rows IS
  BEGIN
    OPEN c1;
    LOOP
      FETCH c1 INTO v_empno;
      DBMS_OUTPUT.PUT_LINE('Id : ' || (v_empno));
      EXIT WHEN c1%ROWCOUNT >= 3;
    END LOOP
  END proc1_3rows;
  PROCEDURE proc4_6rows IS
  BEGIN
    LOOP
      FETCH c1 INTO v_empno;
      DBMS_OUTPUT.PUT_LINE('Id : ' || (v_empno));
      EXIT WHEN c1%ROWCOUNT >= 6;
    END LOOP;
    CLOSE c1;
  END proc4_6rows;
END pack_cur;
/
```

ORACLE

6-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling the Persistent State of a Package Cursor (continued)

Example

The slide on this page shows the package body for PACK_CUR to support the package specification. In the package body:

1. Open the cursor and fetch successive rows from the cursor by using one packaged procedure, PROC1_3ROWS.
2. Continue to fetch successive rows from the cursor and close the cursor, using another packaged procedure, PROC4_6ROWS.

Executing PACK_CUR

```
SET SERVEROUTPUT ON
EXECUTE pack_cur.proc1_3rows
EXECUTE pack_cur.proc4_6rows
```

```
Id 207
Id 206
Id 205
PL/SQL procedure successfully completed.
Id 204
Id 203
Id 202
PL/SQL procedure successfully completed.
```

ORACLE

Result of Executing PACK_CUR

The state of a package variable or cursor persists across transactions within a session. The state does not persist from session to session for the same user, nor does it persist from user to user.

PL/SQL Tables and Records in Packages

```
CREATE OR REPLACE PACKAGE emp_package IS
  TYPE emp_table_type IS TABLE OF employees%ROWTYPE
    INDEX BY BINARY_INTEGER;
  PROCEDURE read_emp_table
    (p_emp_table OUT emp_table_type);
END emp_package;
/
```

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

ORACLE

6-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Passing Tables of Records to Procedures or Functions Inside a Package

Invoke the READ_EMP_TABLE procedure from an anonymous PL/SQL block, using *iSQL*Plus*.

```
DECLARE
  v_emp_table emp_package.emp_table_type;
BEGIN
  emp_package.read_emp_table(v_emp_table);
  DBMS_OUTPUT.PUT_LINE('An example: ' || v_emp_table(4).last_name);
END;
/
```

```
An example: Ernst
PL/SQL procedure successfully completed.
```

To invoke the READ_EMP_TABLE procedure from another procedure or any PL/SQL block outside the package, the actual parameter referring to the OUT parameter P_EMP_TABLE must be prefixed with its package name. In the example above, the V_EMP_TABLE variable is declared of the EMP_TABLE_TYPE type with the package name added as a prefix.

Summary

In this lesson, you should have learned how to:

- **Overload subprograms**
- **Use forward referencing**
- **Use one-time-only procedures**
- **Describe the purity level of package functions**
- **Identify the persistent state of packaged objects**

ORACLE

6-19

Copyright © Oracle Corporation, 2001. All rights reserved.

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 in situations when the processing in both the subprograms is the same, but the parameters passed to them varies.

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

A one-time-only procedure 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 that the user disconnects.

Practice 6 Overview

This practice covers the following topics:

- **Using overloaded subprograms**
- **Creating a one-time-only procedure**

ORACLE

6-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 6 Overview

In this practice you create a package containing an overloaded function. You also create a one-time-only procedure within a package to populate a PL/SQL table.

Practice 6

1. Create a package called OVER_LOAD. Create two functions in this package; name each function PRINT_IT. The function accepts a date or character string and prints a date or a number, depending on how the function is invoked.

Note:

- To print the date value, use DD-MON-YY as the input format, and FmMonth,dd yyyy as the output format. Make sure you handle invalid input.
 - To print out the number, use 999,999.00 as the input format.
- a. Test the first version of PRINT_IT with the following set of commands:

```
VARIABLE display_date VARCHAR2(20)
EXECUTE :display_date := over_load.print_it('08-MAR-01')
PRINT display_date
```

PL/SQL procedure successfully completed.

TODAYS_DATE
March,8 2001

- b. Test the second version of PRINT_IT with the following set of commands:

```
VARIABLE g_emp_sal NUMBER
EXECUTE :g_emp_sal := over_load.print_it('33,600')
PRINT g_emp_sal
```

PL/SQL procedure successfully completed.

G_EMP_SAL
33600

2. Create a new package, called CHECK_PACK, to implement a new business rule.
 - a. Create a procedure called CHK_DEPT_JOB to verify whether a given combination of department ID and job is a valid one. In this case *valid* means that it must be a combination that currently exists in the EMPLOYEES table.

Note:

- Use a PL/SQL table to store the valid department and job combination.
 - The PL/SQL table needs to be populated only once.
 - Raise an application error with an appropriate message if the combination is not valid.
- b. Test your CHK_DEPT_JOB package procedure by executing the following command:
EXECUTE check_pack.chk_dept_job(50,'ST_CLERK')
What happens, and why?
 - c. Test your CHK_DEPT_JOB package procedure by executing the following command:
EXECUTE check_pack.chk_dept_job(20,'ST_CLERK')
What happens, and why?



Oracle Supplied Packages

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Write dynamic SQL statements using `DBMS_SQL` and `EXECUTE IMMEDIATE`**
- **Describe the use and application of some Oracle server-supplied packages:**
 - `DBMS_DDL`
 - `DBMS_JOB`
 - `DBMS_OUTPUT`
 - `UTL_FILE`
 - `UTL_HTTP` and `UTL_TCP`

ORACLE

7-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to use some of the Oracle server supplied packages and to take advantage of their capabilities.

Using Supplied Packages

Oracle-supplied packages:

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

ORACLE

7-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Supplied Packages

Packages are provided with the Oracle server to allow either PL/SQL access to certain SQL features, or to extend the functionality of the database.

You can take advantage of 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`.

Using Native Dynamic SQL

Dynamic SQL:

- **Is a SQL statement that contains variables that can change during runtime**
- **Is a SQL statement with placeholders and is stored as a character string**
- **Enables general-purpose code to be written**
- **Enables data-definition, data-control, or session-control statements to be written and executed from PL/SQL**
- **Is written using either DBMS_SQL or native dynamic SQL**

ORACLE

7-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Native Dynamic SQL (Dynamic SQL)

You can write PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program but rather are stored in character strings that are input to, or built by, the program. That is, the SQL statements can be created dynamically at run time by using variables. For example, you use dynamic SQL to create a procedure that operates on a table whose name is not known until run time, or to write and execute a data definition language (DDL) statement (such as `CREATE TABLE`), a data control statement (such as `GRANT`), or a session control statement (such as `ALTER SESSION`). In PL/SQL, such statements cannot be executed statically.

In Oracle8, and earlier, you have to use `DBMS_SQL` to write dynamic SQL.

In Oracle 8i, you can use `DBMS_SQL` or native dynamic SQL. The `EXECUTE IMMEDIATE` statement can perform dynamic single-row queries. Also, this is used for functionality such as objects and collections, which are not supported by `DBMS_SQL`. If the statement is a multirow `SELECT` statement, you use `OPEN-FOR`, `FETCH`, and `CLOSE` statements.

Execution Flow

SQL statements go through various stages:

- **Parse**
- **Bind**
- **Execute**
- **Fetch**

Note: Some stages may be skipped.

ORACLE

7-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Steps to Process SQL Statements

All SQL statements have to go through various stages. Some stages may be skipped.

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 ensuring that the relevant privileges to those objects exist.

Bind

After parsing, the Oracle server knows the meaning of the Oracle statement but still may not have enough information to execute the statement. The Oracle server may need values for any bind variable in the statement. The process of obtaining these values is called binding variables.

Execute

At this point, the Oracle server has all necessary information and resources, and the statement is executed.

Fetch

In the fetch stage, 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. You can fetch queries, but not the DML statements.

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

7-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the DBMS_SQL Package

Using **DBMS_SQL**, you can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL.

DBMS_SQL can issue data definition language statements in PL/SQL. For example, you can choose to issue a **DROP TABLE** statement from within a stored procedure.

The operations provided by this package are performed under the current user, not under the package owner **SYS**. Therefore, if the caller is an anonymous PL/SQL block, the operations are performed according to the privileges of the current user; if the caller is a stored procedure, the operations are performed according to the owner of the stored procedure.

Using this package to execute DDL statements can result in a deadlock. The most likely reason for this is that the package is being used to drop a procedure that you are still using.

Components of the DBMS_SQL Package

The DBMS_SQL package uses dynamic SQL to access the database.

Function or Procedure	Description
OPEN_CURSOR	Opens a new cursor and assigns a cursor ID number
PARSE	Parses the DDL or DML statement: that is, checks the statement's syntax and associates it with the opened cursor (DDL statements are immediately executed when parsed)
BIND_VARIABLE	Binds the given value to the variable identified by its name in the parsed statement in the given cursor
EXECUTE	Executes the SQL statement and returns the number of rows processed
FETCH_ROWS	Retrieves a row for the specified cursor (for multiple rows, call in a loop)
CLOSE_CURSOR	Closes the specified cursor

Using DBMS_SQL

```
CREATE OR REPLACE PROCEDURE delete_all_rows
(p_tab_name IN VARCHAR2, p_rows_del OUT NUMBER)
IS
  cursor_name  INTEGER;
BEGIN
  cursor_name := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cursor_name, 'DELETE FROM ' || p_tab_name,
    DBMS_SQL.NATIVE );
  p_rows_del := DBMS_SQL.EXECUTE (cursor_name);
  DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

Use dynamic SQL to delete rows

```
VARIABLE deleted NUMBER
EXECUTE delete_all_rows('employees', :deleted)
PRINT deleted
```

PL/SQL procedure successfully completed.

DELETED
109

ORACLE

Example of a DBMS_SQL Package

In the slide, the table name is passed into the DELETE_ALL_ROWS procedure by using an IN parameter. The procedure uses dynamic SQL to delete rows from the specified table. The number of rows that are deleted as a result of the successful execution of the dynamic SQL are passed to the calling environment through an OUT parameter.

How to Process Dynamic DML

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 row processed.
4. Use CLOSE_CURSOR to close the cursor.

Using the EXECUTE IMMEDIATE Statement

Use the EXECUTE IMMEDIATE statement for native dynamic SQL with better performance.

```
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.

ORACLE

7-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the EXECUTE IMMEDIATE Statement

Syntax Definition

Parameter	Description
<i>dynamic_string</i>	A string expression that represents a dynamic SQL statement (without terminator) or a PL/SQL block (with terminator)
<i>define_variable</i>	A variable that stores the selected column value
<i>record</i>	A user-defined or %ROWTYPE record that stores a selected row
<i>bind_argument</i>	An expression whose value is passed to the dynamic SQL statement or PL/SQL block

You can use the INTO clause for a single-row query, but you must use OPEN-FOR, FETCH, and CLOSE for a multirow query.

Note: The syntax shown in the slide is not complete. The other clauses of the statement are discussed in the *Advanced PL/SQL* course.

Using the EXECUTE IMMEDIATE Statement (continued)

In the EXECUTE IMMEDIATE statement:

- 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 RETURNING INTO clause specifies the variables into which column values are returned. It is used only for DML statements that have a RETURNING clause (without a BULK COLLECT clause). For each value returned by the DML statement, there must be a corresponding, type-compatible variable in the RETURNING INTO clause.
- The USING clause holds all bind arguments. The default parameter mode is IN. For DML statements that have a RETURNING clause, you can place OUT arguments in the RETURNING INTO clause without specifying the parameter mode, which, by definition, is OUT. If you use both the USING clause and the RETURNING INTO clause, the USING clause can contain only IN arguments.

At run time, bind arguments replace corresponding placeholders in the dynamic string. Thus, every placeholder must be associated with a bind argument in the USING clause or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL).

Dynamic SQL supports all the SQL data types. For example, define variables and bind arguments can be collections, LOBs, instances of an object type, and REFs. As a rule, dynamic SQL does not support PL/SQL-specific types. For example, define variables and bind arguments cannot be Booleans or index-by tables. The only exception is that a PL/SQL record can appear in the INTO clause.

You can execute a dynamic SQL statement repeatedly, using new values for the bind arguments. However, you incur some overhead because EXECUTE IMMEDIATE reprepares the dynamic string before every execution.

Dynamic SQL Using EXECUTE IMMEDIATE

```
CREATE PROCEDURE del_rows
  (p_table_name IN VARCHAR2,
   p_rows_deld  OUT NUMBER)
IS
BEGIN
  EXECUTE IMMEDIATE 'delete from ' || p_table_name;
  p_rows_deld := SQL%ROWCOUNT;
END;
/
```

PL/SQL procedure successfully completed.

```
VARIABLE deleted NUMBER
EXECUTE del_rows('test_employees',:deleted)
PRINT deleted
```

DELETED
109

ORACLE

Dynamic SQL Using EXECUTE IMMEDIATE

This is the same dynamic SQL as seen with DBMS_SQL, using the Oracle8i statement EXECUTE IMMEDIATE. The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes the dynamic SQL statement.

Using the DBMS_DDL Package

The DBMS_DDL Package:

- Provides access to some SQL DDL statements from stored procedures
- Includes some procedures:

- ALTER_COMPILE (object_type, owner, object_name)

```
DBMS_DDL.ALTER_COMPILE('PROCEDURE','A_USER','QUERY_EMP')
```

- ANALYZE_OBJECT (object_type, owner, name, method)

```
DBMS_DDL.ANALYZE_OBJECT('TABLE','A_USER','JOBS','COMPUTE')
```

Note: This package runs with the privileges of calling user, rather than the package owner SYS.

ORACLE

7-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the DBMS_DDL package

This package provides access to some SQL DDL statements, which you can use in PL/SQL programs. DBMS_DDL is not allowed in triggers, in procedures called from Forms Builder, or in remote sessions. This package runs with the privileges of calling user, rather than the package owner SYS.

Practical Uses

- You can recompile your modified PL/SQL program units by using DBMS_DDL.ALTER_COMPILE. The object type must be either procedure, function, package, package body, or trigger.
- You can analyze a single object, using DBMS_DDL.ANALYZE_OBJECT. (There is a way of analyzing more than one object at a time, using DBMS_UTILITY.) The object type should be TABLE, CLUSTER, or INDEX. The method must be COMPUTE, ESTIMATE, or DELETE.
- This package gives developers access to ALTER and ANALYZE SQL statements through PL/SQL environments.

Using DBMS_JOB for Scheduling

DBMS_JOB Enables the scheduling and execution of PL/SQL programs:

- **Submitting jobs**
- **Executing jobs**
- **Changing execution parameters of jobs**
- **Removing jobs**
- **Suspending Jobs**

ORACLE

7-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Scheduling Jobs by Using DBMS_JOB

The package DBMS_JOB is used to schedule PL/SQL programs to run. Using DBMS_JOB, you can submit PL/SQL programs for execution, execute PL/SQL programs on a schedule, identify when PL/SQL programs should run, remove PL/SQL programs from the schedule, and suspend PL/SQL programs from running.

It can be used to schedule batch jobs during nonpeak hours or to run maintenance programs during times of low usage.

DBMS_JOB Subprograms

Available subprograms include:

- SUBMIT
- REMOVE
- CHANGE
- WHAT
- NEXT_DATE
- INTERVAL
- BROKEN
- RUN

ORACLE

7-14

Copyright © Oracle Corporation, 2001. All rights reserved.

DBMS_JOB Subprograms

Subprogram	Description
SUBMIT	Submits a job to the job queue
REMOVE	Removes a specified job from the job queue
CHANGE	Alters a specified job that has already been submitted to the job queue (you can alter the job description, the time at which the job will be run, or the interval between executions of the job)
WHAT	Alters the job description for a specified job
NEXT_DATE	Alters the next execution time for a specified job
INTERVAL	Alters the interval between executions for a specified job
BROKEN	Disables job execution (if a job is marked as broken, the Oracle server does not attempt to execute it)
RUN	Forces a specified job to run

Submitting Jobs

You can submit jobs by using `DBMS_JOB.SUBMIT`.

Available parameters include:

- `JOB OUT BINARY_INTEGER`
- `WHAT IN VARCHAR2`
- `NEXT_DATE IN DATE DEFAULT SYSDATE`
- `INTERVAL IN VARCHAR2 DEFAULT 'NULL'`
- `NO_PARSE IN BOOLEAN DEFAULT FALSE`

ORACLE

7-15

Copyright © Oracle Corporation, 2001. All rights reserved.

`DBMS_JOB.SUBMIT` Parameters

The `DBMS_JOB.SUBMIT` procedure adds a new job to the job queue. It accepts five parameters and returns the number of a job submitted through the `OUT` parameter `JOB`. The descriptions of the parameters are listed below.

Parameter	Mode	Description
<code>JOB</code>	<code>OUT</code>	Unique identifier of the job
<code>WHAT</code>	<code>IN</code>	PL/SQL code to execute as a job
<code>NEXT_DATE</code>	<code>IN</code>	Next execution date of the job
<code>INTERVAL</code>	<code>IN</code>	Date function to compute the next execution date of a job
<code>NO_PARSE</code>	<code>IN</code>	Boolean flag that indicates whether to parse the job at job submission (the default is false)

Note: An exception is raised if the interval does not evaluate to a time in the future.

Submitting Jobs

Use `DBMS_JOB.SUBMIT` to place a job to be executed in the job queue.

```
VARIABLE jobno NUMBER
BEGIN
  DBMS_JOB.SUBMIT (
    job => :jobno,
    what => 'OVER_PACK.ADD_DEPT(''EDUCATION'',2710);',
    next_date => TRUNC(SYSDATE + 1),
    interval => 'TRUNC(SYSDATE + 1)'
  );
  COMMIT;
END;
/
PRINT jobno
```

PL/SQL procedure successfully completed.

JOBNO
121

ORACLE

Example

The block of code in the slide submits the `ADD_DEPT` procedure of the `OVER_PACK` package to the job queue. The job number is returned through the `JOB` parameter. The `WHAT` parameter must be enclosed in single quotation marks and must include a semicolon at the end of the text string. This job is submitted to run every day at midnight.

Note: In the example, the parameters are passed using named notation.

The transactions in the submitted job are not committed until either `COMMIT` is issued, or `DBMS_JOB.RUN` is executed to run the job. `COMMIT` in the slide commits the transaction.

Changing Job Characteristics

- **DBMS_JOB.CHANGE:** Changes the **WHAT**, **NEXT_DATE**, and **INTERVAL** parameters
- **DBMS_JOB.INTERVAL:** Changes the **INTERVAL** parameter
- **DBMS_JOB.NEXT_DATE:** Changes the next execution date
- **DBMS_JOB.WHAT:** Changes the **WHAT** parameter

ORACLE

7-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Changing Jobs After Being Submitted

The **CHANGE**, **INTERVAL**, **NEXT_DATE**, and **WHAT** procedures enable you to modify job characteristics after a job is submitted to the queue. Each of these procedures takes the **JOB** parameter as an **IN** parameter indicating which job is to be changed.

Example

The following code changes job number 121 to execute on the following day at 6:00 a.m. and every four hours after that.

```
BEGIN
    DBMS_JOB.CHANGE(121, NULL, TRUNC(SYSDATE+1)+6/24,
    'SYSDATE+4/24');
END;
/
```

PL/SQL Procedure successfully completed.

Note: Each of these procedures can be executed on jobs owned by the username to which the session is connected. If the parameter **what**, **next_date**, or **interval** is **NULL**, then the last values assigned to those parameters are used.

Running, Removing, and Breaking Jobs

- **DBMS_JOB.RUN:** Runs a submitted job immediately
- **DBMS_JOB.REMOVE:** Removes a submitted job from the job queue
- **DBMS_JOB.BROKEN:** Marks a submitted job as broken, and a broken job will not run

ORACLE

7-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Running, Removing, and Breaking Jobs

The `DBMS_JOB.RUN` procedure executes a job immediately. Pass the job number that you want to run immediately to the procedure.

```
EXECUTE DBMS_JOB.RUN(121)
```

The `DBMS_JOB.REMOVE` procedure removes a submitted job from the job queue. Pass the job number that you want to remove from the queue to the procedure.

```
EXECUTE DBMS_JOB.REMOVE(121)
```

The `DBMS_JOB.BROKEN` marks a job as broken or not broken. Jobs are not broken by default. You can change a job to the broken status. A broken job will not run. There are three parameters for this procedure. The `JOB` parameter identifies the job to be marked as broken or not broken. The `BROKEN` parameter is a Boolean parameter. Set this parameter to `FALSE` to indicate that a job is not broken, and set it to `TRUE` to indicate that it is broken. The `NEXT_DATE` parameter identifies the next execution date of the job.

```
EXECUTE DBMS_JOB.BROKEN(121, TRUE)
```

Viewing Information on Submitted Jobs

- Use the **DBA_JOBS** dictionary view to see the status of submitted jobs.

```
SELECT job, log_user, next_date, next_sec,  
       broken, what  
FROM DBA_JOBS;
```

JOB	LOG_USER	NEXT_DATE	NEXT_SEC	B	WHAT
121	PLPU	09-MAR-01	08:00:00	N	OVER_PACK.ADD_DEPT('EDUCATION',2710);

- Use the **DBA_JOBS_RUNNING** dictionary view to display jobs that are currently running.

ORACLE

Viewing Information on Submitted Jobs

The **DBA_JOBS** and **DBA_JOBS_RUNNING** dictionary views display information about jobs in the queue and jobs that have run. To be able to view the dictionary information, users should be granted the **SELECT** privilege on **SYS.DBA_JOBS**.

The query shown in the slide displays the job number, the user who submitted the job, the scheduled date for the job to run, the time for the job to run, and the PL/SQL block executed as a job.

Use the **USER_JOBS** data dictionary view to display information about jobs in the queue for you. This view has the same structure as the **DBA_JOBS** view.

Using the DBMS_OUTPUT Package

The DBMS_OUTPUT package enables you to output messages from PL/SQL blocks. Available procedures include:

- **PUT**
- **NEW_LINE**
- **PUT_LINE**
- **GET_LINE**
- **GET_LINES**
- **ENABLE/DISABLE**

ORACLE

7-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the DBMS_OUTPUT Package

The DBMS_OUTPUT package outputs values and messages from any PL/SQL block.

Function or Procedure	Description
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
GET_LINE	Retrieves the current line from the output buffer into the procedure
GET_LINES	Retrieves an array of lines from the output buffer into the procedure
ENABLE/DISABLE	Enables or disables calls to the DBMS_OUTPUT procedures

Practical Uses

- You can output intermediary results to the window for debugging purposes.
- This package enables developers to closely follow the execution of a function or procedure by sending messages and values to the output buffer.

Interacting with Operating System Files

- **UTL_FILE Oracle-supplied package:**
 - Provides text file I/O capabilities
 - Is available with version 7.3 and later
- **The DBMS_LOB Oracle-supplied package:**
 - Provides read-only operations on external **BFILES**
 - Is available with version 8 and later
 - Enables read and write operations on internal **LOBs**

ORACLE

Interacting with Operating System Files

Two Oracle-supplied packages are provided. You can use them to access operating system files.

With the Oracle-supplied UTL_FILE package, you can read from and write to operating system files. This package is available with database version 7.3 and later and the PL/SQL version 2.3 and later.

With the Oracle-supplied package DBMS_LOB, you can read from binary files on the operating system. This package is available from the database version 8.0 and later. This package is discussed later in the lesson “Manipulating Large Objects.”

What Is the UTL_FILE Package?

- **Extends I/O to text files within PL/SQL**
- **Provides security for directories on the server through the `init.ora` file**
- **Is similar to standard operating system I/O**
 - **Open files**
 - **Get text**
 - **Put text**
 - **Close files**
 - **Use the exceptions specific to the UTL_FILE package**

ORACLE

7-22

Copyright © Oracle Corporation, 2001. All rights reserved.

The UTL_FILE Package

The UTL_FILE package provides text file I/O from within PL/SQL. Client-side security implementation uses normal operating system file permission checking. Server-side security is implemented through restrictions on the directories that can be accessed. In the `init.ora` file, the initialization parameter `UTL_FILE_DIR` is set to the accessible directories desired.

```
UTL_FILE_DIR = directory_name
```

For example, the following initialization setting indicates that the directory `/usr/ngreenbe/my_app` is accessible to the `fopen` function, assuming that the directory is accessible to the database server processes. This parameter setting is case-sensitive on case-sensitive operating systems.

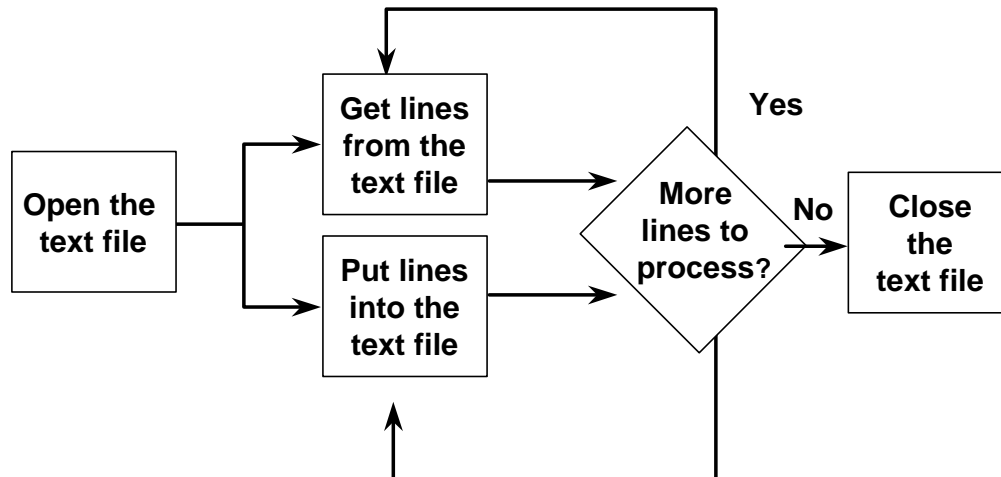
```
UTL_FILE_DIR = /user/ngreenbe/my_app
```

The directory should be on the same machine as the database server. Using the following setting turns off database permissions and makes all directories that are accessible to the database server processes also accessible to the UTL_FILE package.

```
UTL_FILE_DIR = *
```

Using the procedures and functions in the package, you can open files, get text from files, put text into files, and close files. There are seven exceptions declared in the package to account for possible errors raised during execution.

File Processing Using the UTL_FILE Package



ORACLE

File Processing Using the UTL_FILE Package

Before using the UTL_FILE package to read from or write to a text file, you must first check whether the text file is open by using the IS_OPEN function. If the file is not open, you open the file with the FOPEN function. You then either read the file or write to the file until processing is done. At the end of file processing, use the FCLOSE procedure to close the file.

Note: A summary of the procedures and functions within the UTL_FILE package is listed on the next page.

UTL_FILE Procedures and Functions

- Function FOPEN
- Function IS_OPEN
- Procedure GET_LINE
- Procedure PUT, PUT_LINE, PUTF
- Procedure NEW_LINE
- Procedure FFLUSH
- Procedure FCLOSE, FCLOSE_ALL

ORACLE

7-24

Copyright © Oracle Corporation, 2001. All rights reserved.

The UTL_FILE Package: Procedures and Functions

Function or Procedure	Description
FOPEN	A function that opens a file for input or output and returns a file handle used in subsequent I/O operations
IS_OPEN	A function that returns a Boolean value whenever a file handle refers to an open file
GET_LINE	A procedure that reads a line of text from the opened file and places the text in the output buffer parameter (the maximum size of an input record is 1,023 bytes unless you specify a larger size in the overloaded version of FOPEN)
PUT, PUT_LINE	A procedure that writes a text string stored in the buffer parameter to the opened file (no line terminator is appended by put; use new_line to terminate the line, or use PUT_LINE to write a complete line with a terminator)
PUTF	A formatted put procedure with two format specifiers: %s and \n (use %s to substitute a value into the output string. \n is a new line character)
NEW_LINE	Procedure that terminates a line in an output file
FFLUSH	Procedure that writes all data buffered in memory to a file
FCLOSE	Procedure that closes an opened file
FCLOSE_ALL	Procedure that closes all opened file handles for the session

Note: The maximum size of an input record is 1,023 bytes unless you specify a larger size in the overloaded version of FOPEN.

Oracle9i: Develop PL/SQL Program Units 7-24

Exceptions Specific to the UTL_FILE Package

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

ORACLE

7-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Exceptions to the UTL_FILE Package

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

Exception Name	Description
INVALID_PATH	The file location or filename was invalid.
INVALID_MODE	The OPEN_MODE parameter in FOPEN was invalid.
INVALID_FILEHANDLE	The file handle was invalid.
INVALID_OPERATION	The file could not be opened or operated on as requested.
READ_ERROR	An operating system error occurred during the read operation.
WRITE_ERROR	An operating system error occurred during the write operation.
INTERNAL_ERROR	An unspecified error occurred in PL/SQL.

Note: These exceptions must 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 FOPEN and IS_OPEN Functions

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

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

ORACLE

7-26

Copyright © Oracle Corporation, 2001. All rights reserved.

FOPEN Function Parameters

Syntax Definitions

Where	location	Is the operating-system-specific string that specifies the directory or area in which to open the file
	filename	Is the name of the file, including the extension, without any pathing information
	open_mode	Is string that specifies how the file is to be opened; Supported values are: 'r' read text (use GET_LINE) 'w' write text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH) 'a' append text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

The return value is the file handle that is passed to all subsequent routines that operate on the file.

IS_OPEN Function

The function IS_OPEN tests a file handle to see if it identifies an opened file. It returns a Boolean value indicating whether the file has been opened but not yet closed.

Note: For the full syntax, refer to *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Oracle9i: Develop PL/SQL Program Units 7-26

Using UTL_FILE

sal_status.sql

```
CREATE OR REPLACE PROCEDURE sal_status
(p_filedir IN VARCHAR2, p_filename IN VARCHAR2)
IS
  v_filehandle UTL_FILE.FILE_TYPE;
  CURSOR emp_info IS
    SELECT last_name, salary, department_id
    FROM employees
    ORDER BY department_id;
  v_newdeptno employees.department_id%TYPE;
  v_olddeptno employees.department_id%TYPE := 0;
BEGIN
  v_filehandle := UTL_FILE.FOPEN (p_filedir, p_filename, 'w');
  UTL_FILE.PUTF (v_filehandle, 'SALARY REPORT: GENERATED ON
                                %s\n', SYSDATE);
  UTL_FILE.NEW_LINE (v_filehandle);
  FOR v_emp_rec IN emp_info LOOP
    v_newdeptno := v_emp_rec.department_id;
    ...
  END LOOP;
END;
```

ORACLE

7-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Using UTL_FILE

Example

The SAL_STATUS procedure creates a report of employees for each department and their salaries. This information is sent to a text file by using the UTL_FILE procedures and functions.

The variable v_filehandle uses a type defined in the UTL_FILE package. This package defined type is a record with a field called ID of the BINARY_INTEGER datatype.

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

The contents of file_type are private to the UTL_FILE package. Users of the package should not reference or change components of this record.

The names of the text file and the location for the text file are provided as parameters to the program.

```
EXECUTE sal_status('C:\UTLFILE', 'SAL_RPT.TXT')
```

Note: The file location shown in the above example is defined as value of UTL_FILE_DIR in the init.ora file as follows: UTL_FILE_DIR = C:\UTLFILE.

When reading a complete file in a loop, you need to exit the loop using the NO_DATA_FOUND exception. UTL_FILE output is sent synchronously. DBMS_OUTPUT procedures do not produce output until the procedure is completed.

Using UTL_FILE

sal_status.sql

```
...
IF v_newdeptno <> v_olddeptno THEN
    UTL_FILE.PUTF(v_filehandle, 'DEPARTMENT: %s\n',
                  v_emp_rec.department_id);

    END IF;
    UTL_FILE.PUTF(v_filehandle, '  EMPLOYEE: %s earns: %s\n',
                  v_emp_rec.last_name, v_emp_rec.salary);
    v_olddeptno := v_newdeptno;
END LOOP;
UTL_FILE.PUT_LINE(v_filehandle, '*** END OF REPORT ***');
UTL_FILE.FCLOSE(v_filehandle);
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;
/
```

ORACLE

Using UTL_FILE (continued)

The output for this report in the sal_rpt.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 ***
```

The UTL_HTTP Package

The UTL_HTTP package:

- Enables HTTP callouts from PL/SQL and SQL to access data on the Internet
- Contains the functions `REQUEST` and `REQUEST_PIECES` which take the URL of a site as a parameter, contact that site, and return the data obtained from that site
- Requires a proxy parameter to be specified in the above functions, if the client is behind a firewall
- Raises `INIT_FAILED` or `REQUEST_FAILED` exceptions if HTTP call fails
- Reports an HTML error message if specified URL is not accessible

ORACLE

The UTL_HTTP Package

UTL_HTTP is a package that allows you to make HTTP requests directly from the database. The UTL_HTTP package makes hypertext transfer protocol (HTTP) callouts from PL/SQL and SQL. You can use it to access data on the Internet or to call Oracle Web Server Cartridges. By coupling UTL_HTTP with the DBMS_JOBS package, you can easily schedule reoccurring requests be made from your database server out to the Web.

This package contains two entry point functions: `REQUEST` and `REQUEST_PIECES`. Both functions take a string universal resource locator (URL) as a parameter, contact the site, and return the HTML data obtained from the site. The `REQUEST` function returns up to the first 2000 bytes of data retrieved from the given URL. The `REQUEST_PIECES` function returns a PL/SQL table of 2000-byte pieces of the data retrieved from the given URL.

If the HTTP call fails, for a reason such as that the URL is not properly specified in the HTTP syntax then the `REQUEST_FAILED` exception is raised. If initialization of the HTTP-callout subsystem fails, for a reason such as a lack of available memory, then the `INIT_FAILED` exception is raised.

If there is no response from the specified URL, then a formatted HTML error message may be returned.

If `REQUEST` or `REQUEST_PIECES` fails by returning either an exception or an error message, then verify the URL with a browser, to verify network availability from your machine. If you are behind a firewall, then you need to specify proxy as a parameter, in addition to the URL.

This package is covered in more detail in the course *Administering Oracle9i Application Server*.

For more information, refer to *Oracle9i Supplied PL/SQL Packages Reference*.

Using the UTL_HTTP Package

```
SELECT UTL_HTTP.REQUEST('http://www.oracle.com',  
                        'edu-proxy.us.oracle.com')  
  
FROM DUAL;
```

```
UTL_HTTP.REQUEST('HTTP://WWW.ORACLE.COM','EDU-PROXY.US.ORACLE.COM')  
-----  
<head>  
<title>Oracle Corporation</title>  
<meta http-equiv="Content-Type" content="text/html; charset=iso-  
8859-1">  
<meta name="description" content="Oracle Corporation provides the  
software that powers the Internet. For more information about  
Oracle, please call  
1 650/506-7000.">  
<meta name="keywords" content="Oracle, Oracle Corporation, Oracle  
Corp,  
Oracle8i, Oracle 9i, 8i, 9i">  
</head>  
...
```

ORACLE

7-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the UTL_HTTP Package

The SELECT statement and the output in the slide show how to use the REQUEST function of the UTL_HTTP package to retrieve contents from the URL `www.oracle.com`. The second parameter to the function indicates the proxy because the client being tested is behind a firewall.

The retrieved output is in HTML format.

You can use the function in a PL/SQL block as shown below. The function retrieves up to 100 pieces of data, each of a maximum 2000 bytes from the URL. The number of pieces and the total length of the data retrieved are printed.

```
DECLARE  
  x UTL_HTTP.HTML_PIECES;  
BEGIN  
  x := UTL_HTTP.REQUEST_PIECES('http://www.oracle.com/',100,  
                              'edu-proxy.us.oracle.com');  
  DBMS_OUTPUT.PUT_LINE(x.COUNT || ' pieces were retrieved.');
```

```
  DBMS_OUTPUT.PUT_LINE('with total length ');  
  IF x.COUNT < 1 THEN DBMS_OUTPUT.PUT_LINE('0');  
  ELSE DBMS_OUTPUT.PUT_LINE((2000*(x.COUNT - 1))+LENGTH(x(x.COUNT)));  
  END IF;  
END;
```

/

```
9 pieces were retrieved.  
with total length  
16575  
PL/SQL procedure successfully completed.
```


Using the UTL_TCP Package

The UTL_TCP Package:

- **Enables PL/SQL applications to communicate with external TCP/IP-based servers using TCP/IP**
- **Contains functions to open and close connections, to read or write binary or text data to or from a service on an open connection**
- **Requires remote host and port as well as local host and port as arguments to its functions**
- **Raises exceptions if the buffer size is too small, when no more data is available to read from a connection, when a generic network error occurs, or when bad arguments are passed to a function call**

ORACLE

Using the UTL_TCP Package

The UTL_TCP package enables PL/SQL applications to communicate with external TCP/IP-based servers using TCP/IP. Because many Internet application protocols are based on TCP/IP, this package is useful to PL/SQL applications that use Internet protocols.

The package contains functions such as:

OPEN_CONNECTION: This function opens a TCP/IP connection with the specified remote and local host and port details. The remote host is the host providing the service. The remote port is the port number on which the service is listening for connections. The local host and port numbers represent those of the host providing the service. The function returns a connection of PL/SQL record type.

CLOSE_CONNECTION: This procedure closes an open TCP/IP connection. It takes the connection details of a previously opened connection as parameter. The procedure **CLOSE_ALL_CONNECTIONS** closes all open connections.

READ_BINARY() / TEXT() / LINE() : This function receives binary, text, or text line data from a service on an open connection.

WRITE_BINARY() / TEXT() / LINE() : This function transmits binary, text, or text line message to a service on an open connection.

Exceptions are raised when buffer size for the input is too small, when generic network error occurs, when no more data is available to read from the connection, or when bad arguments are passed in a function call.

This package is discussed in detail in the course *Administering Oracle9i Application Server*. For more information, refer to *Oracle 9i Supplied PL/SQL Packages Reference*.

Oracle-Supplied Packages

Other Oracle-supplied packages include:

- DBMS_ALERT
- DBMS_APPLICATION_INFO
- DBMS_DESCRIBE
- DBMS_LOCK
- DBMS_SESSION
- DBMS_SHARED_POOL
- DBMS_TRANSACTION
- DBMS_UTILITY

ORACLE

7-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Oracle-Supplied Packages

Package	Description
DBMS_ALERT	Provides notification of database events
DBMS_APPLICATION_INFO	Allows application tools and application developers to inform the database of the high level of actions they are currently performing
DBMS_DESCRIBE	Returns a description of the arguments for a stored procedure
DBMS_LOCK	Requests, converts, and releases userlocks, which are managed by the RDBMS lock management services
DBMS_SESSION	Provides access to SQL session information
DBMS_SHARED_POOL	Keeps objects in shared memory
DBMS_TRANSACTION	Controls logical transactions and improves the performance of short, nondistributed transactions
DBMS_UTILITY	Analyzes objects in a particular schema, checks whether the server is running in parallel mode, and returns the time

Oracle-Supplied Packages

The following list summarizes and provides a brief description of the packages supplied with Oracle9i.

Built-in Name	Description
CALENDAR	Provides calendar maintenance functions
DBMS_ALERT	Supports asynchronous notification of database events. Messages or alerts are sent on a COMMIT command. Message transmittal is one way, but one sender can alert several receivers.
DBMS_APPLICATION_INFO	Is used to register an application name with the database for auditing or performance tracking purposes
DBMS_AQ	Provides message queuing as part of the Oracle server; is used to add a message (of a predefined object type) onto a queue or dequeue a message
DBMS_AQADM	Is used to perform administrative functions on a queue or queue table for messages of a predefined object type
DBMS_DDL	Is used to embed the equivalent of the SQL commands ALTER, COMPILER, and ANALYZE within your PL/SQL programs
DBMS_DEBUG	A PL/SQL API to the PL/SQL debugger layer, Probe, in the Oracle server
DBSM_DEFER DBMS_DEFER_QUERY DBMS_DEFER_SYS	Is used to build and administer deferred remote procedure calls (use of this feature requires the Replication Option)
DBMS_DESCRIBE	Is used to describe the arguments of a stored procedure
DBMS_DISTRIBRUTED_ TRUST_ADMIN	Is used to maintain the Trusted Servers list, which is used in conjunction with the list at the central authority to determine whether a privileged database link from a particular server can be accepted
DBMS_HS	Is used to administer heterogeneous services by registering or dropping distributed external procedures, remote libraries, and non-Oracle systems (you use dbms_hs to create or drop some initialization variables for non-Oracle systems)
DBMS_HS_EXTPROC	Enables heterogeneous services to establish security for distributed external procedures
DBMS_HS_PASSTHROUGH	Enables heterogeneous services to send pass-through SQL statements to non-Oracle systems
DBMS_IOT	Is used to schedule administrative procedures that you want performed at periodic intervals; is also the interface for the job queue
DBMS_JOB	Is used to schedule administrative procedures that you want performed at periodic intervals
DBMS_LOB	Provides general purpose routines for operations on Oracle large objects (LOBs) data types: BLOB, CLOB (read only) and BFILES (read-only)

Oracle Supplied Packages (continued)

Built-in Name	Description
DBMS_LOCK	Is used to request, convert, and release locks through Oracle Lock Management services
DBMS_LOGMNR	Provides functions to initialize and run the log reader
DBMS_LOGMNR_D	Queries the dictionary tables of the current database, and creates a text based file containing their contents
DBMS_OFFLINE_OG	Provides public APIs for offline instantiation of master groups
DBMS_OFFLINE_SNAPSHOT	Provides public APIs for offline instantiation of snapshots
DBMS_OLAP	Provides procedures for summaries, dimensions, and query rewrites
DBMS_ORACLE_TRACE_AGENT	Provides client callable interfaces to the Oracle TRACE instrumentation within the Oracle7 server
DBMS_ORACLE_TRACE_USER	Provides public access to the Oracle7 release server Oracle TRACE instrumentation for the calling user
DBMS_OUTPUT	Accumulates information in a buffer so that it can be retrieved out later
DBMS_PCLXUTIL	Provides intrapartition parallelism for creating partition-wise local indexes
DBMS_PIPE	Provides a DBMS pipe service that enables messages to be sent between sessions
DBMS_PROFILER	Provides a Probe Profiler API to profile existing PL/SQL applications and identify performance bottlenecks
DBMS_RANDOM	Provides a built-in random number generator
DBMS_RECTIFIER_DIFF	Provides APIs used to detect and resolve data inconsistencies between two replicated sites
DBMS_REFRESH	Is used to create groups of snapshots that can be refreshed together to a transactionally consistent point in time; requires the Distributed option
DBMS_REPAIR	Provides data corruption repair procedures
DBMS_REPCAT	Provides routines to administer and update the replication catalog and environment; requires the Replication option
DBMS_REPCAT_ADMIN	Is used to create users with the privileges needed by the symmetric replication facility; requires the Replication option
DBMS_REPCAT_INSTANTIATE	Instantiates deployment templates; requires the Replication option
DBMS_REPCAT_RGT	Controls the maintenance and definition of refresh group templates; requires the Replication option
DBMS_REPUTIL	Provides routines to generate shadow tables, triggers, and packages for table replication
DBMS_RESOURCE_MANAGER	Maintains plans, consumer groups, and plan directives; it also provides semantics so that you may group together changes to the plan schema

Oracle Supplied Packages (continued)

Built-in Name	Description
DBMS_RESOURCE_MANAGER_PRIVS	Maintains privileges associated with resource consumer groups
DBMS_RLS	Provides row-level security administrative interface
DBMS_ROWID	Is used to get information about ROWIDs, including the data block number, the object number, and other components
DBMS_SESSION	Enables programmatic use of the SQL ALTER SESSION statement as well as other session-level commands
DBMS_SHARED_POOL	Is used to keep objects in shared memory, so that they are not be aged out with the normal LRU mechanism
DBMS_SNAPSHOT	Is used to refresh one or more snapshots that are not part of the same refresh group and purge logs; use of this feature requires the Distributed option
DBMS_SPACE	Provides segment space information not available through standard views
DBMS_SPACE_ADMIN	Provides tablespace and segment space administration not available through standard SQL
DBMS_SQL	Is used to write stored procedure and anonymous PL/SQL blocks using dynamic SQL; also used to parse any DML or DDL statement
DBMS_STANDARD	Provides language facilities that help your application interact with the Oracle server
DBMS_STATS	Provides a mechanism for users to view and modify optimizer statistics gathered for database objects
DBMS_TRACE	Provides routines to start and stop PL/SQL tracing
DBMS_TRANSACTION	Provides procedures for a programmatic interface to transaction management
DBMS_TTS	Checks whether if the transportable set is self-contained
DBMS_UTILITY	Provides functionality for managing procedures, reporting errors, and other information
DEBUG_EXTPROC	Is used to debug external procedures on platforms with debuggers that can attach to a running process
OUTLN_PKG	Provides the interface for procedures and functions associated with management of stored outlines
PLITBLM	Handles index-table operations
SDO_ADMIN	Provides functions implementing spatial index creation and maintenance for spatial objects
SDO_GEOM	Provides functions implementing geometric operations on spatial objects
SDO_MIGRATE	Provides functions for migrating spatial data from release 7.3.3 and 7.3.4 to 8.1x
SDO_TUNE	Provides functions for selecting parameters that determine the behavior of the spatial indexing scheme used in the Spatial Cartridge

Oracle Supplied Packages (continued)

Built-in Name	Description
STANDARD	Declares types, exceptions, and subprograms that are available automatically to every PL/SQL program
TIMESERIES	Provides functions that perform operations, such as extraction, retrieval, arithmetic, and aggregation, on time series data
TIMESCALE	Provides scale-up and scale-down functions
TSTOOLS	Provides administrative tools procedures
UTL_COLL	Enables PL/SQL programs to use collection locators to query and update
UTL_FILE	Enables your PL/SQL programs to read and write operating system (OS) text files and provides a restricted version of standard OS stream file I/O
UTL_HTTP	Enables HTTP callouts from PL/SQL and SQL to access data on the Internet or to call Oracle Web Server Cartridges
UTL_PG	Provides functions for converting COBOL numeric data into Oracle numbers and Oracle numbers into COBOL numeric data
UTL_RAW	Provides SQL functions for RAW data types that concatenate, obtain substring, and so on, to and from RAW data types
UTL_REF	Enables a PL/SQL program to access an object by providing a reference to the object
VIR_PKG	Provides analytical and conversion functions for visual information retrieval

Summary

In this lesson, you should have learned how to:

- Take advantage of the preconfigured packages that are provided by Oracle
- Create packages by using the `catproc.sql` script
- Create packages individually.

ORACLE

7-37

Copyright © Oracle Corporation, 2001. All rights reserved.

DBMS Packages and the Scripts to Execute Them

DBMS_ALERT	dbmsalrt.sql
DBMS_APPLICATION_INFO	dbmsutil.sql
DBMS_DDL	dbmsutil.sql
DBMS_LOCK	dbmslock.sql
DBMS_OUTPUT	dbmsotpt.sql
DBMS_PIPE	dbmspipe.sql
DBMS_SESSION	dbmsutil.sql
DBMS_SHARED_POOL	dbmsspool.sql
DBMS_SQL	dbmssql.sql
DBMS_TRANSACTION	dbmsutil.sql
DBMS_UTILITY	dbmsutil.sql

Note: For more information about these packages and scripts, refer to *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Practice 7 Overview

This practice covers using:

- **DBMS_SQL** for dynamic SQL
- **DBMS_DDL** to analyze a table
- **DBMS_JOB** to schedule a task
- **UTL_FILE** to generate text reports

ORACLE

7-38

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 7 Overview

In this practice, you use `DBMS_SQL` to implement a procedure to drop a table. You also use the `EXECUTE IMMEDIATE` command to drop a table. You use `DBMS_DDL` to analyze objects in your schema, and you can schedule the analyze procedure through `DBMS_JOB`.

In this practice, you also write a PL/SQL program that generates customer statuses into a text file.

Practice 7

1.
 - a. Create a DROP_TABLE procedure that drops the table specified in the input parameter. Use the procedures and functions from the supplied DBMS_SQL package.
 - b. To test the DROP_TABLE procedure, first create a new table called EMP_DUP as a copy of the EMPLOYEES table.
 - c. Execute the DROP_TABLE procedure to drop the EMP_DUP table.
2.
 - a. Create another procedure called DROP_TABLE2 that drops the table specified in the input parameter. Use the EXECUTE IMMEDIATE statement.
 - b. Repeat the test outlined in steps 1-b and 1-c.
3.
 - a. Create a procedure called ANALYZE_OBJECT that analyzes the given object that you specified in the input parameters. Use the DBMS_DDL package, and use the COMPUTE method.
 - b. Test the procedure using the EMPLOYEES table. Confirm that the ANALYZE_OBJECT procedure has run by querying the LAST_ANALYZED column in the USER_TABLES data dictionary view.

LAST_ANAL
01-MAY-01

If you have time:

4. Schedule ANALYZE_OBJECT by using DBMS_JOB. Analyze the DEPARTMENTS table, and schedule the job to run in five minutes time from now. Confirm that the job has been scheduled by using USER_JOBS.
5. Create a procedure called CROSS_AVGSAL that generates a text file report of employees who have exceeded the average salary of their department. The partial code is provided for you in the file lab07_5.sql.
 - a. Your program should accept two parameters. The first parameter identifies the output directory. The second parameter identifies the text file name to which your procedure writes.
 - b. Your instructor will inform you of the directory location. When you invoke the program, name the second parameter sal_rptxx.txt where xx stands for your user number, such as 01, 15, and so on.
 - c. Add an exception handling section to handle errors that may be encountered from using the UTL_FILE package.

Sample output from this file follows:

EMPLOYEES OVER THE AVERAGE SALARY OF THEIR DEPARTMENT:
REPORT GENERATED ON 26-FEB-01

Hartstein	20	\$13,000.00
Raphaely	30	\$11,000.00
Marvis	40	\$6,500.00

...

*** END OF REPORT ***

8

Manipulating Large Objects

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Compare and contrast LONG and large object (LOB) 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**

ORACLE

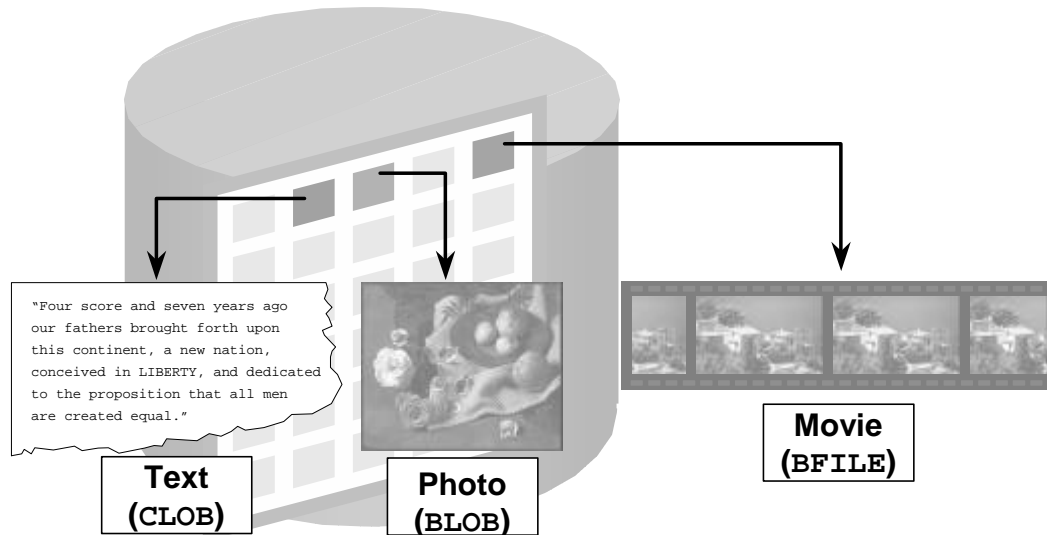
Lesson Aim

Databases have long been used to store large objects. However, the mechanisms built into databases have never been as useful as the new large object (LOB) data types provided in 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.



ORACLE

8-3

Copyright © Oracle Corporation, 2001. All rights reserved.

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 on 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 operating system 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 interpretation 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) are stored in the database.
- External files (BFILE) are stored outside the database.

The Oracle9i Server performs implicit conversion between CLOB and VARCHAR2 data types. The other implicit conversions between LOBs 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.

BFILES 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

ORACLE

8-4

Copyright © Oracle Corporation, 2001. All rights reserved.

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 9i provides a LONG-to-LOB API to migrate from LONG columns to LOB columns.

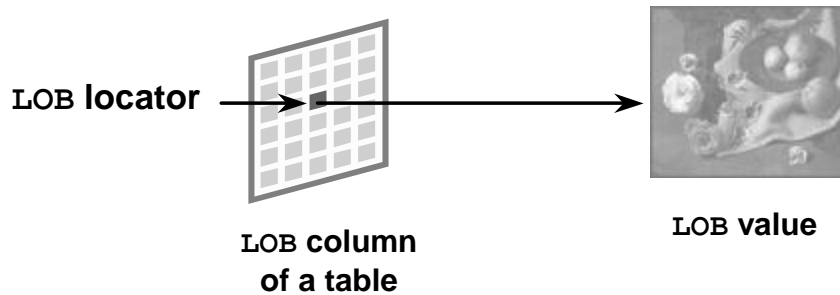
It is beneficial to discuss LOB functionality in comparison to the older types. In the bulleted list below, LONGs refers to LONG and LONG RAW, and LOBs refers to all LOB data types:

- 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 gigabytes; LOBs can be up to 4 gigabytes.
- 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.
- 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 covert 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.

Anatomy of a LOB

The LOB column stores a locator to the LOB's value.



ORACLE

8-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Components of a LOB

There are two distinct parts of 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 the 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.

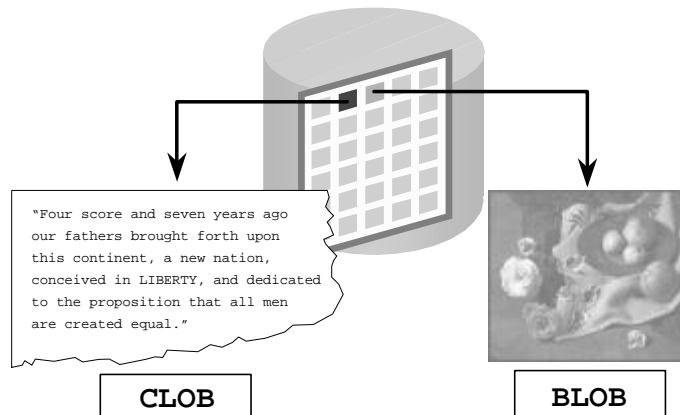
A LOB column does not contain the data; it contains the locator of the LOB value.

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 DML, you must create a LOB locator. Programmatic interfaces operate on the LOB values, using these locators in a manner similar to operating system file handles.

Internal LOBs

The LOB value is stored in the database.



ORACLE

8-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Features of Internal LOBs

The internal LOB is stored inside 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 rollbacks

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**
 - **JDBC**
- **The Oracle server provides some support for LOB management through SQL.**

ORACLE

8-7

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Manage LOBs

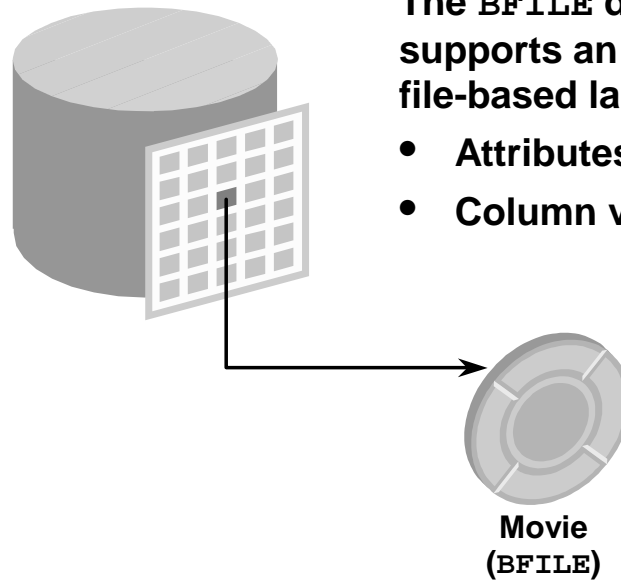
Use the following method to manage an internal LOB:

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.

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**

ORACLE

8-8

Copyright © Oracle Corporation, 2001. All rights reserved.

What Are BFILES?

BFILES are external large objects (LOBs) stored in operating system files outside of the database tablespaces. The Oracle SQL data type to support these large objects is BFILE. 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, CDROMs, photo CDs, or any such device, but a single LOB cannot extend from one device to another.

The BFILE data type is available so that database users can access the external file system. The Oracle9i server provides for:

- Definition of BFILE objects
- Association of BFILE objects to corresponding external files
- Security for BFILES

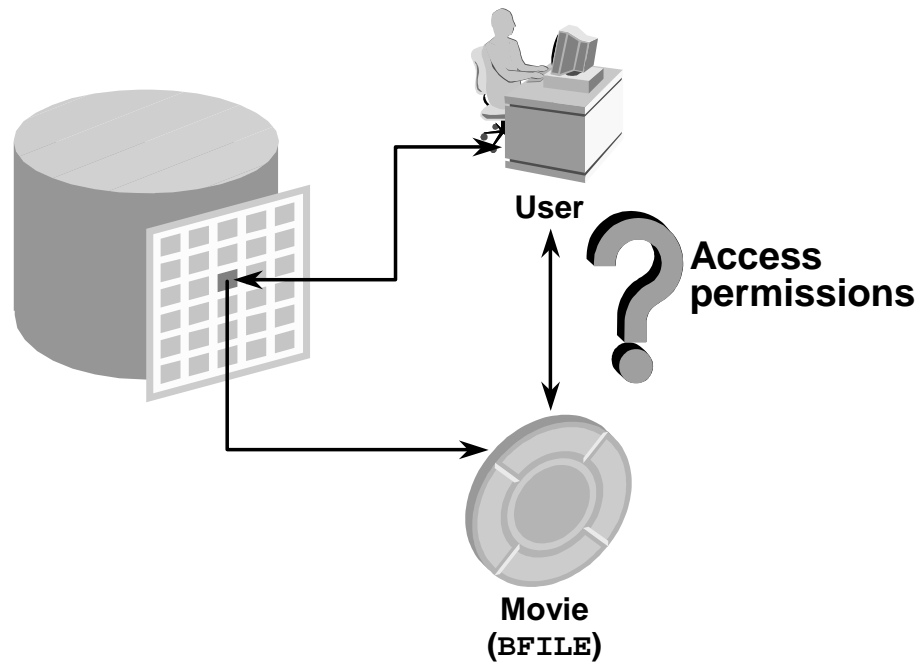
The rest of the operations required to use BFILES are possible through the DBMS_LOB package and the Oracle Call Interface.

BFILES are read-only, so they do not participate in transactions. Any support for integrity and durability must be provided by the operating system. The user must create the file and place it 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. The administration of the actual files and the OS directory structures to house the files is the responsibility of the database administrator (DBA), system administrator, or user. The maximum size of an external large object is operating system dependent but cannot exceed four gigabytes.

Note: BFILES are available in the Oracle8 database and in later releases.

Oracle9i: Develop PL/SQL Program Units 8-8

Securing BFILES



ORACLE

8-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Securing BFILES

Unauthenticated access to files on a server presents a security risk. The Oracle9i Server 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

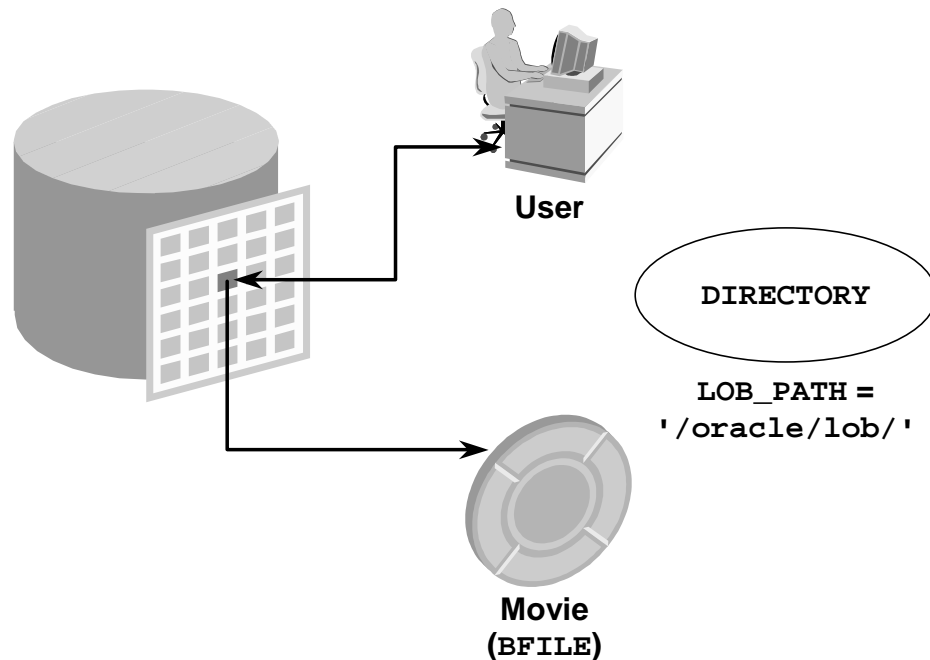
The file must reside on the machine where the database exists. A time-out to read a nonexistent BFILE is based on the operating system value.

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

The Oracle9i RDBMS 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



ORACLE

8-10

Copyright © Oracle Corporation, 2001. All rights reserved.

A New Database Object: DIRECTORY

A **DIRECTORY** is a nonschema database object that provides for administration of access and usage of **BFILE**s in an Oracle9i Server.

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).

Further, 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 **CREATE ANY DIRECTORY** privilege). 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 are operating system dependent. 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

8-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines for Creating Directory Objects

To associate an operating system file to a BFILE, you should first create a DIRECTORY object that is an alias for the full pathname to the operating system 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 system privileges CREATE ANY DIRECTORY and DROP ANY DIRECTORY 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 prior to creating the DIRECTORY object. Oracle does not create the OS path.

If you migrate the database to a different operating system, 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 data dictionary views DBA_DIRECTORIES and ALL_DIRECTORIES.

Managing BFILES

- **Create an OS directory and supply files.**
- **Create an Oracle table with a column that holds the BFILE data type.**
- **Create a DIRECTORY object.**
- **Grant privileges to read the DIRECTORY object to users.**
- **Insert rows into the table by using the BFILENAME function.**
- **Declare and initialize a LOB locator in a program.**
- **Read the BFILE.**

ORACLE

8-12

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Manage BFILES

Use the following method to manage the BFILE and DIRECTORY objects:

1. Create the 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 the OS directory.
2. Create a table containing the BFILE data type in the Oracle server.
3. Create the DIRECTORY object.
4. Grant the READ privilege to it.
5. Insert rows into the table using the BFILENAME function and associate the OS files with the corresponding row and column intersection.
6. Declare and initialize the LOB locator in a program.
7. Select the row and column containing the BFILE into the LOB locator.
8. Read the BFILE with an OCI or a DBMS_LOB function, using the locator as a reference to the file.

Preparing to Use BFILES

- **Create or modify an Oracle table with a column that holds the BFILE data type.**

```
ALTER TABLE employees
  ADD emp_video BFILE;
```

- **Create a DIRECTORY object by using the CREATE DIRECTORY command.**

```
CREATE DIRECTORY dir_name
  AS os_path;
```

- **Grant privileges to read the DIRECTORY object to users.**

```
GRANT READ ON DIRECTORY dir_name TO
user|role|PUBLIC;
```

ORACLE

8-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Preparing to Use BFILES

In order to use a BFILE within an Oracle table, you need to have a table with a column of BFILE type. For the Oracle server to access an external file, the server needs to know the location of the file on the operating system. The 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*;

Where:	<i>dir_name</i>	is the name of the directory database object
	<i>os_path</i>	is the location of the BFILES

The following commands set up a pointer to BFILES in the system directory /\$HOME/LOG_FILES and give users the privilege to read the BFILES from the directory.

```
DROP DIRECTORY log_files
CREATE OR REPLACE DIRECTORY log_files AS '/$HOME/LOG_FILES';
GRANT READ ON DIRECTORY log_files TO PUBLIC;
```

```
Directory dropped.
Directory created.
Grant succeeded.
```

In a session, the number of BFILES that can be opened in one session is limited by the parameter SESSION_MAX_OPEN_FILES. This parameter is set in the init.ora file. Its default value is 10.

The BFILENAME Function

Use the BFILENAME function to initialize a BFILE column.

```
FUNCTION BFILENAME (directory_alias IN VARCHAR2,  
                   filename IN VARCHAR2)  
RETURN BFILE;
```

ORACLE

8-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The BFILENAME Function

BFILENAME is a built-in function that initializes a BFILE column to point to an external file. Use the BFILENAME function as part of an INSERT statement to initialize a BFILE column by associating it with a physical file in the server file system. You can use the UPDATE statement to change the reference target of the BFILE. A BFILE can be initialized to NULL and updated later by using the BFILENAME function.

Syntax Definitions

Where: *directory_alias* is the name of the DIRECTORY database object

filename is the name of the BFILE to be read

Example

```
UPDATE employees  
SET emp_video = BFILENAME('LOG_FILES', 'King.avi')  
WHERE employee_id = 100;
```

Once physical files are associated with records using SQL DML, subsequent read operations on the BFILE can be performed using the PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILES, and so they cannot be updated or deleted through BFILES.

Loading BFILES

```
CREATE OR REPLACE PROCEDURE load_emp_bfile
  (p_file_loc IN VARCHAR2) IS
  v_file      BFILE;
  v_filename  VARCHAR2(16);
  CURSOR emp_cursor IS
    SELECT first_name FROM employees
    WHERE department_id = 60 FOR UPDATE;
BEGIN
  FOR emp_record IN emp_cursor LOOP
    v_filename := emp_record.first_name || '.bmp';
    v_file := BFILENAME(p_file_loc, v_filename);
    DBMS_LOB.FILEOPEN(v_file);
    UPDATE employees SET emp_video = v_file
      WHERE CURRENT OF emp_cursor;
    DBMS_OUTPUT.PUT_LINE('LOADED FILE: ' || v_filename
      || ' SIZE: ' || DBMS_LOB.GETLENGTH(v_file));
    DBMS_LOB.FILECLOSE(v_file);
  END LOOP;
END load_emp_bfile;
/
```

ORACLE

8-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Loading BFILES

Example

Load a BFILE pointer to an image of each employee into the EMPLOYEES table by using the DBMS_LOB package. The images are .bmp files stored in the /home/LOG_FILES directory.

Executing the procedure yields the following results:

```
EXECUTE load_emp_bfile('LOG_FILES')
```

```
LOADED FILE: Alexander.bmp SIZE: 22358
LOADED FILE: Bruce.bmp SIZE: 108082
LOADED FILE: David.bmp SIZE: 78736
LOADED FILE: Valli.bmp SIZE: 78736
LOADED FILE: Diana.bmp SIZE: 78736
PL/SQL procedure successfully completed.
```

Loading BFILES

Use the `DBMS_LOB.FILEEXISTS` function to verify that the file exists in the operating system. The function returns 0 if the file does not exist, and returns 1 if the file does exist.

```
CREATE OR REPLACE PROCEDURE load_emp_bfile
(p_file_loc IN VARCHAR2)
IS
  v_file          BFILE;  v_filename    VARCHAR2(16);
  v_file_exists   BOOLEAN;
  CURSOR emp_cursor IS ...
BEGIN
  FOR emp_record IN emp_cursor LOOP
    v_filename := emp_record.first_name || '.bmp';
    v_file := BFILENAME (p_file_loc, v_filename);
    v_file_exists := ((DBMS_LOB.FILEEXISTS(v_file) = 1));
    IF v_file_exists THEN
      DBMS_LOB.FILEOPEN (v_file); ...
```

ORACLE

8-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Using DBMS_LOB.FILEEXISTS

This function finds out whether a given BFILE locator points to a file that actually exists on the server's file system. This is the specification for the function:

Syntax Definitions

```
FUNCTION DBMS_LOB.FILEEXISTS
  (file_loc IN BFILE)
RETURN INTEGER;
```

Where: `file_loc` is name of the BFILE locator
 `RETURN INTEGER` returns 0 if the physical file does not exist
 returns 1 if the physical file exists

If the `FILE_LOC` parameter contains an invalid value, one of three exceptions may be raised.

In the example in the slide, the output of the `DBMS_LOB.FILEEXISTS` function is compared with value 1 and the result is returned to the `BOOLEAN` variable `V_FILE_EXISTS`.

Exception Name	Description
NOEXIST_DIRECTORY	The directory does not exist.
NOPRIV_DIRECTORY	You do not have privileges for the directory.
INVALID_DIRECTORY	The directory was invalidated after the file was opened.

Migrating from LONG to LOB

The Oracle9i server allows 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.

ORACLE

8-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Migrating from LONG to LOB

Oracle9i Server supports the LONG-to-LOB migration using API.

Data migration: Where existing tables that contain LONG columns need to be moved to use LOB columns. This can be done using the ALTER TABLE command. In Oracle8i, an operator named TO_LOB had to be used to copy a LONG to a LOB. In Oracle9i, 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, if you had a table with the following definition:

```
CREATE TABLE Long_tab (id NUMBER, long_col LONG);
```

you can change the LONG_COL column in table LONG_TAB to the CLOB data type as follows:

```
ALTER TABLE Long_tab MODIFY ( long_col CLOB );
```

For limitations on the LONG-to-LOB migration, refer to *Oracle9i Application Developer's Guide - Large Objects*.

Application Migration: Where the existing LONG applications change for using LOBs. You can use SQL and PL/SQL to access LONGs and LOBs. This API is provided for both OCI and PL/SQL.

Oracle9i: Develop PL/SQL Program Units 8-17

Migrating From LONG to LOB

- **Implicit conversion: 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:**
 - **CLOBs and BLOBs 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

8-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Migrating from LONG to LOB (continued)

With the new LONG-to-LOB API introduced in Oracle9i, 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 Oracle9i to convert other data types to CLOB and BLOB as part of LONG-to-LOB migration:

- **TO_CLOB () converts LONG, VARCHAR2, and CHAR to CLOB**
- **TO_BLOB () converts LONG RAW and RAW to BLOB**

TO_CHAR () is enabled to convert a CLOB to a CHAR type.

Function and procedure parameter passing: This allows all the user-defined procedures and functions to use CLOBs and BLOBs as actual parameters where VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa.

Accessing 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.

These details are discussed in detail later in this lesson.

For more information, refer to “Migrating from LONGs to LOBs” in *Oracle9i Application Developer’s Guide - Large Objects (LOBs)*.

Oracle9i: Develop PL/SQL Program Units 8-18

The DBMS_LOB Package

- **Working with LOB often requires the use of the Oracle-supplied package DBMS_LOB.**
- **DBMS_LOB provides routines to access and manipulate internal and external LOBs.**
- **Oracle9i enables retrieving LOB data directly using SQL, without using any special LOB API.**
- **In PL/SQL you can define a VARCHAR2 for a CLOB and a RAW for BLOB.**

ORACLE

8-19

Copyright © Oracle Corporation, 2001. All rights reserved.

The DBMS_LOB Package

In releases prior to Oracle9i, you need to 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.

Two constants are used in the specification of procedures in this package: `LOBMAXSIZE` and `FILE_READONLY`. These constants are used in the procedures and functions of DBMS_LOB; for example, you can use them to achieve the maximum possible level of purity so that they can be used in SQL expressions.

Using the DBMS_LOB Routines

Functions and procedures in this package can be broadly classified into two types: mutators or observers. Mutators can modify LOB values, whereas observers can only read LOB values.

- **Mutators:** APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL, and FILEOPEN
- **Observers:** COMPARE, FILEGETNAME, INSTR, GETLENGTH, READ, SUBSTR, FILEEXISTS, and FILEISOPEN

The 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

8-20

Copyright © Oracle Corporation, 2001. All rights reserved.

The DBMS_LOB Package (continued)

APPEND	Append the contents of the source LOB to the destination LOB
COPY	Copy all or part of the source LOB to the destination LOB
ERASE	Erase all or part of a LOB
LOADFROMFILE	Load BFILE data into an internal LOB
TRIM	Trim the LOB value to a specified shorter length
WRITE	Write data to the LOB from a specified offset
GETLENGTH	Get the length of the LOB value
INSTR	Return the matching position of the <i>n</i> th occurrence of the pattern in the LOB
READ	Read data from the LOB starting at the specified offset
SUBSTR	Return part of the LOB value starting at the specified offset
FILECLOSE	Close the file
FILECLOSEALL	Close all previously opened files
FILEEXISTS	Check if the file exists on the server
FILEGETNAME	Get the directory alias and file name
FILEISOPEN	Check if the file was opened using the input BFILE locators
FILEOPEN	Open a file

The 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.**

ORACLE

8-21

Copyright © Oracle Corporation, 2001. All rights reserved.

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

8-22

Copyright © Oracle Corporation, 2001. All rights reserved.

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 will be less than the one specified, if the end of the LOB is reached before the specified number of bytes or characters could 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 32767 for RAW and VARCHAR2 parameters. Make sure the allocated system resources are adequate to support these 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 will write AMOUNT bytes of the buffer contents into the LOB.

Adding LOB Columns to a Table

```
ALTER TABLE employees ADD  
  (resume      CLOB,  
   picture     BLOB);
```

Table altered.

ORACLE

8-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Adding LOB Columns to a Table

LOB columns are defined by way of SQL data definition language (DDL), as in the ALTER TABLE statement in the slide. The contents of a LOB column is 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 type LOB, which contains only the value of the LOB locator.

Populating LOB Columns

Insert a row into a table with LOB columns:

```
INSERT INTO employees (employee_id, first_name,
    last_name, email, hire_date, job_id,
    salary, resume, picture)
VALUES (405, 'Marvin', 'Ellis', 'MELLIS', SYSDATE,
    'AD_ASST', 4000, EMPTY_CLOB(), NULL);
```

1 row created.

Initialize a LOB column using the EMPTY_BLOB() function:

```
UPDATE employees
SET resume = 'Date of Birth: 8 February 1951',
    picture = EMPTY_BLOB()
WHERE employee_id = 405;
```

1 row updated.

ORACLE

8-24

Copyright © Oracle Corporation, 2001. All rights reserved.

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 functions `EMPTY_BLOB` and `EMPTY_CLOB` in `INSERT` or `UPDATE` statements of SQL DML to initialize a `NULL` or non-`NULL` internal LOB to empty. These are available as special functions in Oracle SQL DML, and are not part of the `DBMS_LOB` package.

Before you can start writing data to an internal LOB using OCI or the `DBMS_LOB` package, the LOB column must be made nonnull, that is, it must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column's value to empty by using the function `EMPTY_BLOB` in the `VALUES` clause of an `INSERT` statement. Similarly, a CLOB or NCLOB column's value can be initialized by using the function `EMPTY_CLOB`.

The result of using the function `EMPTY_CLOB()` or `EMPTY_BLOB()` means that the LOB is initialized, but not populated with data. To populate the LOB column, you can use an update statement.

You can use an `INSERT` statement to insert a new row and populate the LOB column at the same time.

When you create a LOB instance, the Oracle server creates and places a locator to the out-of-line LOB value in the LOB column of a particular row in the table. SQL, OCI, and other programmatic interfaces operate on LOBs through these locators.

Populating LOB Columns (continued)

The `EMPTY_B/CLOB()` function can be used as a `DEFAULT` column constraint, as in the example below. This initializes the LOB columns with locators.

```
CREATE TABLE emp_hiredata
  (employee_id    NUMBER(6),
   first_name     VARCHAR2(20),
   last_name      VARCHAR2(25),
   resume        CLOB   DEFAULT EMPTY_CLOB(),
   picture        BLOB   DEFAULT EMPTY_BLOB());
```

Table created.

Updating LOB by Using SQL

UPDATE CLOB column

```
UPDATE employees  
SET resume = 'Date of Birth: 1 June 1956'  
WHERE employee_id = 170;
```

1 row updated.

ORACLE

Updating LOB by Using SQL

You can update a LOB column by setting it to another LOB value, to NULL, or by using the empty function appropriate for the LOB data type (`EMPTY_CLOB()` or `EMPTY_BLOB()`). You can update the LOB using a bind variable in embedded SQL, the value of which may be NULL, empty, or populated. When you set one LOB equal to another, a new copy of the LOB value is created. These actions do not require a `SELECT FOR UPDATE` statement. You must lock the row prior to the update only when updating a piece of the LOB.

Updating LOB by Using DBMS_LOB in PL/SQL

```
DECLARE
  lobloc CLOB;          -- serves as the LOB locator
  text   VARCHAR2(32767):='Resigned: 5 August 2000';
  amount NUMBER;        -- amount to be written
  offset INTEGER;       -- where to start writing
BEGIN
  SELECT resume INTO lobloc
  FROM   employees
  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   employees
  WHERE  employee_id = 170 FOR UPDATE;
  amount := length(text);
  DBMS_LOB.WRITEAPPEND(lobloc, amount, text);
  COMMIT;
END;
```

ORACLE

8-27

Copyright © Oracle Corporation, 2001. All rights reserved.

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 package procedure WRITE 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 Oracle9i. 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: In versions prior to Oracle9i, Oracle did not allow LOBs in the WHERE clause of UPDATE and SELECT. 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, last_name , resume -- CLOB
FROM employees
WHERE employee_id IN (405, 170);
```

EMPLOYEE_ID	LAST_NAME	RESUME
170	Fox	Date of Birth: 1 June 1956 Resigned = 30 September 2000
405	Ellis	Date of Birth: 8 February 1951 Resigned = 5 August 2000

2 rows selected.

ORACLE[®]

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 *iSQL*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_column, no_of_chars, starting)**
- **DBMS_LOB.INSTR (lob_column, pattern)**

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),  
       DBMS_LOB.INSTR (resume, ' = ' )  
FROM   employees  
WHERE  employee_id IN (170, 405);
```

DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,' = ')
June	36
Febru	40

2 rows selected.

ORACLE

Selecting CLOB Values by Using DBMS_LOB

DBMS_LOB.SUBSTR

Use DBMS_LOB.SUBSTR to display part of a LOB. It is similar in functionality to the SQL function SUBSTR.

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 Oracle9i, you can also use SQL functions SUBSTR and INSTR to perform the operations shown in the slide.

Selecting CLOB Values in PL/SQL

```
DECLARE
    text VARCHAR2(4001);
BEGIN
    SELECT resume INTO text
    FROM employees
    WHERE employee_id = 170;
    DBMS_OUTPUT.PUT_LINE('text is: ' || text);
END;
/
```

text is: Date of Birth: 1 June 1956 Resigned: 30 September 2000 Resigned: 30 September 2000

ORACLE

8-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Selecting CLOB Values in PL/SQL

The slide shows the code for accessing CLOB values that can be implicitly converted to VARCHAR2 in Oracle9i. The value of the column RESUME, when selected into a VARCHAR2 variable TEXT, is implicitly converted.

In prior releases, to access a CLOB column, first you must retrieve the CLOB column value into a CLOB variable and specify the amount and offset size. Then you use the DBMS_LOB package to read the selected value. The code using DBMS_LOB is as follows:

```
DECLARE
    rlob clob;
    text VARCHAR2(4001);
    amt number := 4001;
    offset number := 1;
BEGIN
    SELECT resume INTO rlob
    FROM employees
    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 employees
WHERE employee_id = 405;
```

1 row deleted.

Disassociate a LOB value from a row:

```
UPDATE employees
SET resume = EMPTY_CLOB()
WHERE employee_id = 170;
```

1 row updated.

ORACLE

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 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 implicitly when all the LOB data is updated.

You must explicitly remove the file associated with a `BFILE` using operating system 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 `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.**

ORACLE

8-32

Copyright © Oracle Corporation, 2001. All rights reserved.

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.

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 IsTempLOBOpen
    (p_lob_loc IN OUT BLOB, p_retval OUT INTEGER)
IS
BEGIN
    -- create a temporary LOB
    DBMS_LOB.CREATETEMPORARY (p_lob_loc, TRUE);
    -- see if the LOB is open: returns 1 if open
    p_retval := DBMS_LOB.ISOPEN (p_lob_loc);
    DBMS_OUTPUT.PUT_LINE ('The file returned a value
        ....' || p_retval);

    -- free the temporary LOB
    DBMS_LOB.FREETEMPORARY (p_lob_loc);
END;
/
```

Procedure created

ORACLE

Creating a Temporary LOB

The example in the slide shows a user-defined PL/SQL procedure, `IsTempLOBOpen`, that 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 `IsTempLOBOpen` 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:**
 - **The 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

8-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

There are four LOB data types:

- A BLOB is a binary large object.
- A CLOB is a character large object.
- A 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 operating system file). You can manage LOBs by using the DBMS_LOB package and its procedures.

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables.

Practice 8 Overview

This practice covers the following topics:

- **Creating object types, using the new data types CLOB and BLOB**
- **Creating a table with LOB data types as columns**
- **Using the DBMS_LOB package to populate and interact with the LOB data**

ORACLE

8-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 8 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 8

1. Create a table called PERSONNEL by executing the script file lab08_1.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 employees 2034 and 2035. Use the empty function for the CLOB, and provide NULL as the value for the BLOB.
3. Examine and execute the script lab08_3.sql. 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 the following subquery in a SQL 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.

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2035;
```

Practice 8 (continued)

If you have time

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. The image files are named after the country IDs. You need to load an image file locator into all rows in Europe region (`REGION_ID = 1`) in the `COUNTRIES` table. The `DIRECTORY` object name that stores a pointer to the location of the binary files is called `COUNTRY_PIC`. This object is already created for you.
 - a. Use the command below to add the image column to the `COUNTRIES` table

```
ALTER TABLE countries ADD (picture BFILE);
```
 - b. Create a PL/SQL procedure called `load_country_image` that reads a locator into your picture column. Have the program test to see if the file exists, using the function `DBMS_LOB.FILEEXISTS`. If the file is not existing, your procedure should display a message that the file can not be opened. Have your program report information about the load to the screen.
 - c. Invoke the procedure by passing the name of the directory object `COUNTRY_PIC` as parameter. Note that you should pass the directory object in single quotation marks.

Sample output follows:

```
LOADING LOCATORS TO PICTURES...
LOADED LOCATOR TO FILE:  Be.tif SIZE:  24556
LOADED LOCATOR TO FILE:  Ch.tif SIZE:  44744
LOADED LOCATOR TO FILE:  De.tif SIZE:  9116
...
TOTAL FILES UPDATED: 8
```


9

Creating Database Triggers

ORACLE[®]

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe different types of triggers**
- **Describe database triggers and their use**
- **Create database triggers**
- **Describe database trigger firing rules**
- **Remove database triggers**

ORACLE®

9-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to create and use database triggers.

Types of Triggers

A trigger:

- **Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or the database**
- **Executes implicitly whenever a particular event takes place**
- **Can be either:**
 - **Application trigger: Fires whenever an event occurs with a particular application**
 - **Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database**

ORACLE

Types of Triggers

Application triggers execute implicitly whenever a particular data manipulation language (DML) event occurs within an application. An example of an application that uses triggers extensively is one developed with Oracle Forms Developer.

Database triggers execute implicitly when a data event such as DML on a table (an INSERT, UPDATE, or DELETE triggering statement), an INSTEAD OF trigger on a view, or data definition language (DDL) statements such as CREATE and ALTER are issued, no matter which user is connected or which application is used. Database triggers also execute implicitly when some user actions or database system actions occur, for example, when a user logs on, or the DBA shut down the database.

Note: Database triggers can be defined on tables and on views. If a DML operation is issued on a view, the INSTEAD OF trigger defines what actions take place. If these actions include DML operations on tables, then any triggers on the base tables are fired.

Database triggers can be system triggers on a database or a schema. With a database, triggers fire for each event for all users; with a schema, triggers fire for each event for that specific user.

This course covers creating database triggers. Creating database triggers based on system events is discussed in the lesson “More Trigger Concepts.”

Guidelines for Designing Triggers

- **Design triggers to:**
 - **Perform related actions**
 - **Centralize global operations**
- **Do not design triggers:**
 - **Where functionality is already built into the Oracle server**
 - **That duplicate other triggers**
- **Create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.**
- **The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.**

ORACLE

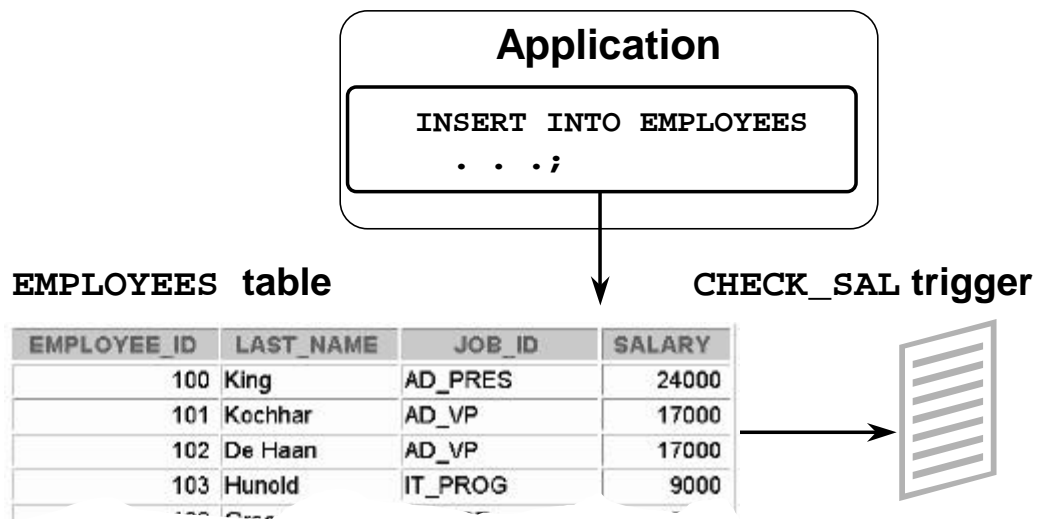
9-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines for Designing Triggers

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Use database triggers only for centralized, global operations that should be fired for the triggering statement, regardless of which user or application issues the statement.
- Do not define triggers to duplicate or replace the functionality already built into the Oracle database. For example do not define triggers to implement integrity rules that can be done by using declarative constraints. An easy way to remember the design order for a business rule is to:
 - Use built-in constraints in the Oracle server such as, primary key, foreign key and so on
 - Develop a database trigger or develop an application such as a servlet or Enterprise JavaBean (EJB) on your middle tier
 - Use a presentation interface such as Oracle Forms, dynamic HTML, Java ServerPages (JSP) and so on, if you cannot develop your business rule as mentioned above, which might be a presentation rule.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications. Only use triggers when necessary, and beware of recursive and cascading effects.
- If the logic for the trigger is very lengthy, create stored procedures with the logic and invoke them in the trigger body.
- Note that database triggers fire for every user each time the event occurs on which the trigger is created.

Database Trigger: Example



ORACLE

9-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a Database Trigger

In this example, the database trigger `CHECK_SAL` checks salary values whenever any application tries to insert a row into the `EMPLOYEES` table. Values that are out of range according to the job category can be rejected, or can be allowed and recorded in an audit table.

Creating DML Triggers

A triggering statement contains:

- **Trigger timing**
 - For table: **BEFORE, AFTER**
 - For view: **INSTEAD OF**
- **Triggering event: INSERT, UPDATE, or DELETE**
- **Table name: On table, view**
- **Trigger type: Row or statement**
- **WHEN clause: Restricting condition**
- **Trigger body: PL/SQL block**

ORACLE

9-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Trigger

Before coding the trigger body, decide on the values of the components of the trigger: the trigger timing, the triggering event, and the trigger type.

Part	Description	Possible Values
Trigger timing	When the trigger fires in relation to the triggering event	BEFORE AFTER INSTEAD OF
Triggering event	Which data manipulation operation on the table or view causes the trigger to fire	INSERT UPDATE DELETE
Trigger type	How many times the trigger body executes	Statement Row
Trigger body	What action the trigger performs	Complete PL/SQL block

If multiple triggers are defined for a table, be aware that the order in which multiple triggers of the same type fire is arbitrary. To ensure that triggers of the same type are fired in a particular order, consolidate the triggers into one trigger that calls separate procedures in the desired order.

DML Trigger Components

Trigger timing: When should the trigger fire?

- **BEFORE:** Execute the trigger body before the triggering DML event on a table.
- **AFTER:** Execute the trigger body after the triggering DML event on a table.
- **INSTEAD OF:** Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

ORACLE

9-7

Copyright © Oracle Corporation, 2001. All rights reserved.

BEFORE Triggers

This type of trigger is frequently used in the following situations:

- To determine whether that triggering statement should be allowed to complete. (This situation enables you to eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing a triggering INSERT or UPDATE statement.
- To initialize global variables or flags, and to validate complex business rules.

AFTER Triggers

This type of trigger is frequently used in the following situations:

- To complete the triggering statement before executing the triggering action.
- To perform different actions on the same triggering statement if a BEFORE trigger is already present.

INSTEAD OF Triggers

This type of trigger is used to provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because the view is not inherently modifiable.

You can write INSERT, UPDATE, and DELETE statements against the view. The INSTEAD OF trigger works invisibly in the background performing the action coded in the trigger body directly on the underlying tables.

DML Trigger Components

Triggering user event: Which DML statement causes the trigger to execute? You can use any of the following:

- **INSERT**
- **UPDATE**
- **DELETE**

ORACLE

9-8

Copyright © Oracle Corporation, 2001. All rights reserved.

The Triggering Event

The triggering event or statement can be an INSERT, UPDATE, or DELETE statement on a table.

- When the triggering event is an UPDATE statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement, because they always affect entire rows.

```
. . . UPDATE OF salary . . .
```

- The triggering event can contain one, two, or all three of these DML operations.

```
. . . INSERT or UPDATE or DELETE
```

```
. . . INSERT or UPDATE OF job_id . . .
```


DML Trigger Components

Trigger type: Should the trigger body execute for each row the statement affects or only once?

- **Statement:** The trigger body executes once for the triggering event. This is the default. A statement trigger fires once, even if no rows are affected at all.
- **Row:** The trigger body executes once for each row affected by the triggering event. A row trigger is not executed if the triggering event affects no rows.

ORACLE

Statement Triggers and Row Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple row UPDATE) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all.

Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself: for example, a trigger that performs a complex security check on the current user.

Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed.

Row triggers are useful if the trigger action depends on data of rows that are affected or on data provided by the triggering event itself.

DML Trigger Components

Trigger body: What action should the trigger perform?

The trigger body is a PL/SQL block or a call to a procedure.

ORACLE

9-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Trigger Body

The trigger action defines what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Additionally, row triggers use correlation names to access the old and new column values of the row being processed by the trigger.

Note: The size of a trigger cannot be more than 32 K.

Firing Sequence

Use the following firing sequence for a trigger on a table, when a single row is manipulated:

DML statement

```
INSERT INTO departments (department_id,  
                        department_name, location_id)  
VALUES (400, 'CONSULTING', 2400);
```

1 row created.

Triggering action

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
400	CONSULTING	2400

→ BEFORE statement trigger

→ BEFORE row trigger

→ AFTER row trigger

→ AFTER statement trigger

ORACLE

Creating Row or Statement Triggers

Create a statement trigger or a row trigger based on the requirement that the trigger must fire once for each row affected by the triggering statement, or just once for the triggering statement, regardless of the number of rows affected.

When the triggering data manipulation statement affects a single row, both the statement trigger and the row trigger fire exactly once.

Example

This SQL statement does not differentiate statement triggers from row triggers, because exactly one row is inserted into the table using this syntax.

Firing Sequence

Use the following firing sequence for a trigger on a table, when many rows are manipulated:

```
UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30;
```

6 rows updated.



ORACLE

9-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Row or Statement Triggers (continued)

When the triggering data manipulation statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Example

The SQL statement in the slide above causes a row-level trigger to fire a number of times equal to the number of rows that satisfy the WHERE clause, that is, the number of employees reporting to department 30.

Syntax for Creating DML Statement Triggers

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    event1 [OR event2 OR event3]
    ON table_name
    trigger_body
```

Note: Trigger names must be unique with respect to other triggers in the same schema.

ORACLE

9-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Syntax for Creating a Statement Trigger

<i>trigger name</i>	Is the name of the trigger
<i>timing</i>	Indicates the time when the trigger fires in relation to the triggering event: BEFORE AFTER
<i>event</i>	Identifies the data manipulation operation that causes the trigger to fire: INSERT UPDATE [OF <i>column</i>] DELETE
<i>table/view_name</i>	Indicates the table associated with the trigger
<i>trigger body</i>	Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, ending with END, or a call to a procedure

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique with respect to other schema objects, such as tables, views, and procedures.

Using column names along with the UPDATE clause in the trigger improves performance, because the trigger fires only when that particular column is updated and thus avoids unintended firing when any other column is updated.

Creating DML Statement Triggers

Example:

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
    (TO_CHAR(SYSDATE,'HH24:MI')
      NOT BETWEEN '08:00' AND '18:00')
    THEN RAISE_APPLICATION_ERROR (-20500,'You may
      insert into EMPLOYEES table only
      during business hours.');
```

```
END IF;
END;
/
```

Trigger created.

ORACLE

Creating DML Statement Triggers

You can create a **BEFORE** statement trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

For example, create a trigger to restrict inserts into the **EMPLOYEES** table to certain business hours, Monday through Friday.

If a user attempts to insert a row into the **EMPLOYEES** table on Saturday, the user sees the message, the trigger fails, and the triggering statement is rolled back. Remember that the **RAISE_APPLICATION_ERROR** is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server.

Testing SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,  
                        first_name, email, hire_date,  
                        job_id, salary, department_id)  
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,  
        'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,  
                        *  
ERROR at line 1:  
ORA-20500: You may only insert into EMPLOYEES during  
business hours.  
ORA-06512: at "NEWPL.SECURE_EMP", line 4  
ORA-04088: error during execution of trigger 'NEWPL.SECURE_EMP'
```

ORACLE

Example

Insert a row into the EMPLOYEES table during nonbusiness hours.

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
  IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN')) OR
     (TO_CHAR (SYSDATE, 'HH24') NOT BETWEEN '08' AND '18')
  THEN
    IF DELETING THEN
      RAISE_APPLICATION_ERROR (-20502, 'You may delete from
        EMPLOYEES table only during business hours.');
```

```
    ELSIF INSERTING THEN
      RAISE_APPLICATION_ERROR (-20500, 'You may insert into
        EMPLOYEES table only during business hours.');
```

```
    ELSIF UPDATING ('SALARY') THEN
      RAISE_APPLICATION_ERROR (-20503, 'You may update
        SALARY only during business hours.');
```

```
    ELSE
      RAISE_APPLICATION_ERROR (-20504, 'You may update
        EMPLOYEES table only during normal hours.');
```

```
    END IF;
  END IF;
END;
```

ORACLE

Combining Triggering Events

You can combine several triggering events into one by taking advantage of the special conditional predicates INSERTING, UPDATING, and DELETING within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the EMPLOYEES table to certain business hours, Monday through Friday.

Creating a DML Row Trigger

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
    event1 [OR event2 OR event3]
    ON table_name
    [REFERENCING OLD AS old / NEW AS new]
FOR EACH ROW
    [WHEN (condition)]
trigger_body
```

ORACLE

9-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Syntax for Creating a Row Trigger

<i>trigger_name</i>	Is the name of the trigger
<i>timing</i>	Indicates the time when the trigger fires in relation to the triggering event: BEFORE AFTER INSTEAD OF
<i>event</i>	Identifies the data manipulation operation that causes the trigger to fire: INSERT UPDATE [OF <i>column</i>] DELETE
<i>table_name</i>	Indicates the table associated with the trigger
REFERENCING	Specifies correlation names for the old and new values of the current row (The default values are OLD and NEW)
FOR EACH ROW	Designates that the trigger is a row trigger
WHEN	Specifies the trigger restriction; (This conditional predicate must be enclosed in parenthesis and is evaluated for each row to determine whether or not the trigger body is executed.)
<i>trigger body</i>	Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, ending with END, or a call to a procedure

Creating DML Row Triggers

```
CREATE OR REPLACE TRIGGER restrict_salary
  BEFORE INSERT OR UPDATE OF salary ON employees
  FOR EACH ROW
  BEGIN
    IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
      AND :NEW.salary > 15000
    THEN
      RAISE_APPLICATION_ERROR (-20202, 'Employee
                                   cannot earn this amount');
    END IF;
  END;
/
```

Trigger created.

ORACLE

Creating a Row Trigger

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

Create a trigger to allow only certain employees to be able to earn a salary of more than 15,000.

If a user attempts to do this, the trigger raises an error.

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

```
UPDATE EMPLOYEES
*
ERROR at line 1:
ORA-20202: EMPLOYEE CANNOT EARN THIS AMOUNT
ORA-06512: at "PLPU.RESTRICT_SALARY", line 5
ORA-04088: error during execution of trigger 'PLPU.RESTRICT_SALARY'
```

Using OLD and NEW Qualifiers

```
CREATE OR REPLACE TRIGGER audit_emp_values
  AFTER DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO audit_emp_table (user_name, timestamp,
    id, old_last_name, new_last_name, old_title,
    new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
    :OLD.last_name, :NEW.last_name, :OLD.job_id,
    :NEW.job_id, :OLD.salary, :NEW.salary );
END;
/
```

ORACLE

9-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Using OLD and NEW Qualifiers

Within a ROW trigger, reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifier.

Data Operation	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

- The OLD and NEW qualifiers are available only in ROW triggers.
- Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.
- There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.

Note: Row triggers can decrease the performance if you do a lot of updates on larger tables.

Using OLD and NEW Qualifiers: Example Using Audit_Emp_Table

USER_NAME	TIMESTAMP	ID	OLD_LAST_N	NEW_LAST_N	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
PLPU	09-MAR-01			An emp		SA_REP		1000
PLPU	09-MAR-01	999	An emp	New Emp	SA_REP	SA_MAN	1000	3000

ORACLE

9-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Using OLD and NEW Qualifiers: Example Using AUDIT_EMP_TABLE

Create a trigger on the EMPLOYEES table to add rows to a user table, AUDIT_EMP_TABLE, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name. There is additional column COMMENTS in the AUDIT_EMP_TABLE that is not shown in this slide.

Restricting a Row Trigger

```
CREATE OR REPLACE TRIGGER derive_commission_pct
  BEFORE INSERT OR UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF INSERTING
    THEN :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL
    THEN :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct + 0.05;
  END IF;
END;
/
```

ORACLE

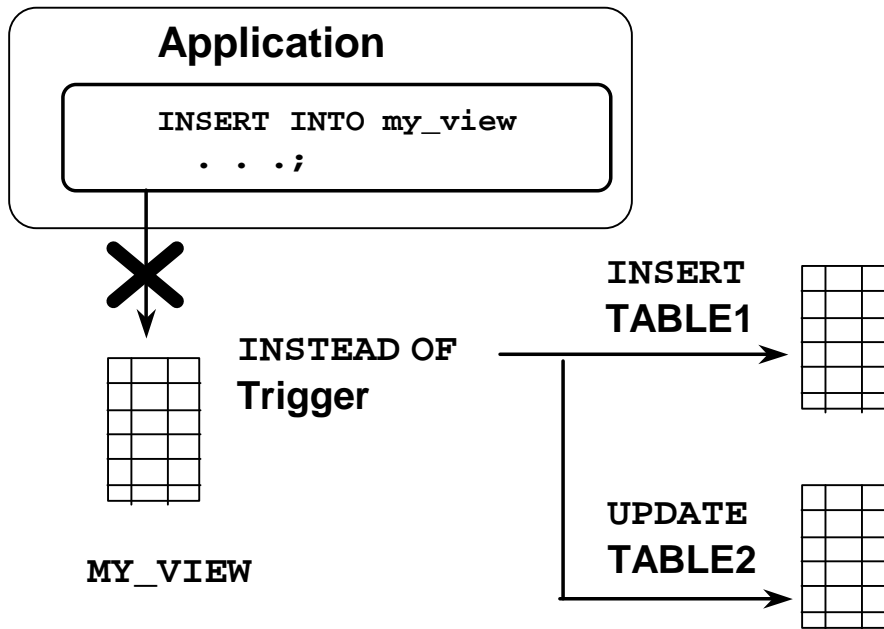
Example

To restrict the trigger action to those rows that satisfy a certain condition, provide a WHEN clause.

Create a trigger on the EMPLOYEES table to calculate an employee's commission when a row is added to the EMPLOYEES table, or when an employee's salary is modified.

The NEW qualifier cannot be prefixed with a colon in the WHEN clause because the WHEN clause is outside the PL/SQL blocks.

INSTEAD OF Triggers



ORACLE

9-22

Copyright © Oracle Corporation, 2001. All rights reserved.

INSTEAD OF Triggers

Use **INSTEAD OF** triggers to modify data in which the DML statement has been issued against an inherently nonupdatable view. These triggers are called **INSTEAD OF** triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. This trigger is used to perform an **INSERT**, **UPDATE**, or **DELETE** operation directly on the underlying tables.

You can write **INSERT**, **UPDATE**, or **DELETE** statements against a view, and the **INSTEAD OF** trigger works invisibly in the background to make the right actions take place.

Why Use **INSTEAD OF** Triggers?

A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as **GROUP BY**, **CONNECT BY**, **START**, the **DISTINCT** operator, or joins. For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So, you write an **INSTEAD OF** trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatable and has **INSTEAD OF** triggers, the triggers take precedence. **INSTEAD OF** triggers are row triggers.

The **CHECK** option for views is not enforced when insertions or updates to the view are performed by using **INSTEAD OF** triggers. The **INSTEAD OF** trigger body must enforce the check.

Creating an INSTEAD OF Trigger

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
  INSTEAD OF
    event1 [OR event2 OR event3]
    ON view_name
    [REFERENCING OLD AS old / NEW AS new]
  [FOR EACH ROW]
  trigger_body
```

ORACLE

9-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Syntax for Creating an INSTEAD OF Trigger

<i>trigger_name</i>	Is the name of the trigger.
INSTEAD OF	Indicates that the trigger belongs to a view
<i>event</i>	Identifies the data manipulation operation that causes the trigger to fire: INSERT UPDATE [OF <i>column</i>] DELETE
<i>view_name</i>	Indicates the view associated with trigger
REFERENCING	Specifies correlation names for the old and new values of the current row (The defaults are OLD and NEW)
FOR EACH ROW	Designates the trigger to be a row trigger; INSTEAD OF triggers can only be row triggers: if this is omitted, the trigger is still defined as a row trigger.
<i>trigger body</i>	Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, and ending with END or a call to a procedure

Note: INSTEAD OF triggers can be written only for views. BEFORE and AFTER options are not valid.

Creating an INSTEAD OF Trigger

Example:

The following example creates two new tables, NEW_EMPS and NEW_DEPTS, based on the EMPLOYEES and DEPARTMENTS tables respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables. The example also creates an INSTEAD OF trigger, NEW_EMP_DEPT. When a row is inserted into the EMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, based on the data in the INSERT statement. Similarly, when a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

```
CREATE TABLE new_emps AS
  SELECT employee_id, last_name, salary, department_id,
         email, job_id, hire_date
  FROM employees;

CREATE TABLE new_depts AS
  SELECT d.department_id, d.department_name, d.location_id,
         sum(e.salary) tot_dept_sal
  FROM employees e, departments d
  WHERE e.department_id = d.department_id
  GROUP BY d.department_id, d.department_name, d.location_id;

CREATE VIEW emp_details AS
  SELECT e.employee_id, e.last_name, e.salary, e.department_id,
         e.email, e.job_id, d.department_name, d.location_id
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;

CREATE OR REPLACE TRIGGER new_emp_dept
  INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
  FOR EACH ROW
  BEGIN
    IF INSERTING THEN
      INSERT INTO new_emps
      VALUES (:NEW.employee_id, :NEW.last_name, :NEW.salary,
              :NEW.department_id, :NEW.email, :NEW.job_id, SYSDATE);
      UPDATE new_depts
      SET tot_dept_sal = tot_dept_sal + :NEW.salary
      WHERE department_id = :NEW.department_id;
    ELSIF DELETING THEN
      DELETE FROM new_emps
      WHERE employee_id = :OLD.employee_id;
      UPDATE new_depts
      SET tot_dept_sal = tot_dept_sal - :OLD.salary
      WHERE department_id = :OLD.department_id;
```


Creating an INSTEAD OF Trigger (continued)

Example:

```
ELSIF UPDATING ('salary')
THEN
    UPDATE new_emps
    SET salary = :NEW.salary
    WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
    SET tot_dept_sal = tot_dept_sal + (:NEW.salary - :OLD.salary)
    WHERE department_id = :OLD.department_id;
ELSIF UPDATING ('department_id')
THEN
    UPDATE new_emps
    SET department_id = :NEW.department_id
    WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
    SET tot_dept_sal = tot_dept_sal - :OLD.salary
    WHERE department_id = :OLD.department_id;
    UPDATE new_depts
    SET tot_dept_sal = tot_dept_sal + :NEW.salary
    WHERE department_id = :NEW.department_id;
END IF;
END;
/
```

Note: This example is explained in the next page by using graphics.

Creating an INSTEAD OF Trigger

INSERT into EMP_DETAILS that is based on EMPLOYEES and DEPARTMENTS tables

```
INSERT INTO emp_details(employee_id, ... )
VALUES(9001,'ABBOTT',3000,10,'abbott.mail.com','HR_MAN');
```

INSTEAD OF
INSERT into
EMP_DETAILS →

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID	EMAIL	JOB_ID
100	King	24000	90	SKING	AD_PRES
206	Gietz	2000	110	WGIEZ	AC_ACCOUNT
210	Smith	1000	80	DASMITH	SA_REP

INSERT into
NEW_EMPS

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID	EMAIL
100	King	24000	90	SKING
101	Kochhar	17000	90	NKOCH
207	Harris	1000	80	DAHARRIS
210	Smith	1000	80	DASMITH
9001	ABBOTT	3000	10	abbott.m

UPDATE
NEW_DEPTS

DEPARTMENT_ID	DEPARTMENT_NAME	MT_DEPT_SAL
10	Administration	7400
20	Marketing	19000
30	Purchasing	27300

ORACLE

Creating an INSTEAD OF Trigger

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based.

Assume that an employee name will be inserted using the view. Create a trigger that results in the appropriate INSERT and UPDATE to the base tables.

Differentiating Between Database Triggers and Stored Procedures

Triggers	Procedures
Defined with <code>CREATE TRIGGER</code>	Defined with <code>CREATE PROCEDURE</code>
Data dictionary contains source code in <code>USER_TRIGGERS</code>	Data dictionary contains source code in <code>USER_SOURCE</code>
Implicitly invoked	Explicitly invoked
<code>COMMIT</code> , <code>SAVEPOINT</code> , and <code>ROLLBACK</code> are not allowed	<code>COMMIT</code> , <code>SAVEPOINT</code> , and <code>ROLLBACK</code> are allowed

ORACLE

9-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Triggers and Stored Procedures

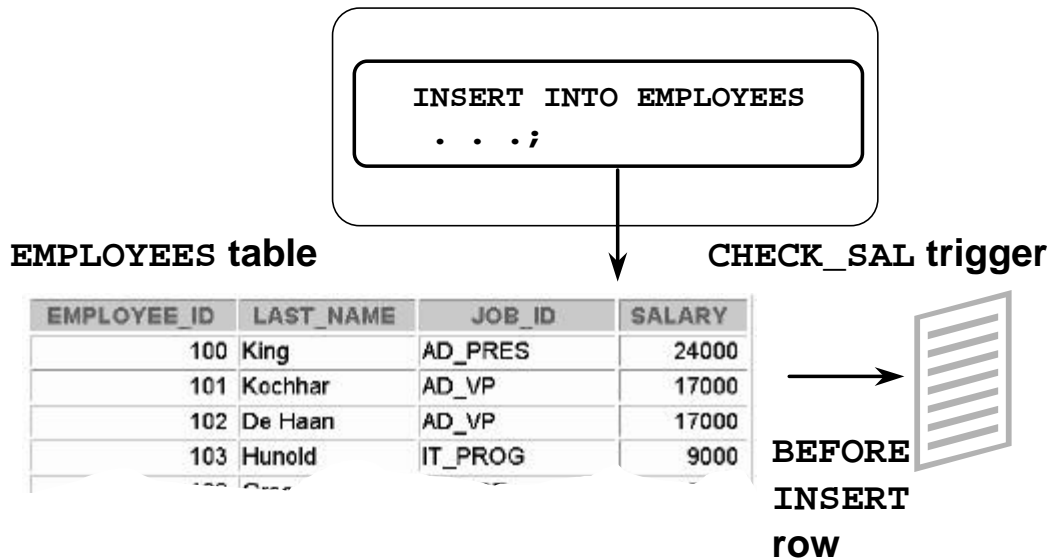
There are differences between database triggers and stored procedures:

Database Trigger	Stored Procedure
Invoked implicitly	Invoked explicitly
<code>COMMIT</code> , <code>ROLLBACK</code> , and <code>SAVEPOINT</code> statements are not allowed within the trigger body. It is possible to commit or rollback indirectly by calling a procedure, but it is not recommended because of side effects to transactions.	<code>COMMIT</code> , <code>ROLLBACK</code> , and <code>SAVEPOINT</code> statements are permitted within the procedure body.

Triggers are fully compiled when the `CREATE TRIGGER` command is issued and the P code is stored in the data dictionary.

If errors occur during the compilation of a trigger, the trigger is still created.

Differentiating Between Database Triggers and Form Builder Triggers



ORACLE

9-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Differences between a Database Trigger and a Form Builder Trigger

Database triggers are different from Form Builder triggers.

Database Trigger	Form Builder Trigger
Executed by actions from any database tool or application	Executed only within a particular Form Builder application
Always triggered by a SQL DML, DDL, or a certain database action	Can be triggered by navigating from field to field, by pressing a key, or by many other actions
Is distinguished as either a statement or row trigger	Is distinguished as a statement or row trigger
Upon failure, causes the triggering statement to roll back	Upon failure, causes the cursor to freeze and may cause the entire transaction to roll back
Fires independently of, and in addition to, Form Builder triggers	Fires independently of, and in addition to, database triggers
Executes under the security domain of the author of the trigger	Executes under the security domain of the Form Builder user

Managing Triggers

Disable or reenable a database trigger:

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

Disable or reenable all triggers for a table:

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS
```

Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE
```

ORACLE

Trigger Modes: Enabled or Disabled

- When a trigger is first created, it is enabled automatically.
- The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, manages the dependencies, and provides a two-phase commit process if a trigger updates remote tables in a distributed database.
- Disable a specific trigger by using the ALTER TRIGGER syntax, or disable *all* triggers on a table by using the ALTER TABLE syntax.
- Disable a trigger to improve performance or to avoid data integrity checks when loading massive amounts of data by using utilities such as SQL*Loader. You may also want to disable the trigger when it references a database object that is currently unavailable, owing to a failed network connection, disk crash, offline data file, or offline tablespace.

Compile a Trigger

- Use the ALTER TRIGGER command to explicitly recompile a trigger that is invalid.
- When you issue an ALTER TRIGGER statement with the COMPILE option, the trigger recompiles, regardless of whether it is valid or invalid.

DROP TRIGGER Syntax

To remove a trigger from the database, use the DROP TRIGGER syntax:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

Trigger dropped.

Note: All triggers on a table are dropped when the table is dropped.

ORACLE

Removing Triggers

When a trigger is no longer required, you can use a SQL statement in *iSQL*Plus* to drop it.

Trigger Test Cases

- **Test each triggering data operation, as well as nontriggering data operations.**
- **Test each case of the WHEN clause.**
- **Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.**
- **Test the effect of the trigger upon other triggers.**
- **Test the effect of other triggers upon the trigger.**

ORACLE

Testing Triggers

- Ensure that the trigger works properly by testing a number of cases separately.
- Take advantage of the DBMS_OUTPUT procedures to debug triggers. You can also use the Procedure Builder debugging tool to debug triggers. Using Procedure Builder is discussed in Appendix F, “Creating Program Units by Using Procedure Builder.”

Trigger Execution Model and Constraint Checking

1. Execute all **BEFORE STATEMENT** triggers.
2. Loop for each row affected:
 - a. Execute all **BEFORE ROW** triggers.
 - b. Execute all **AFTER ROW** triggers.
3. Execute the **DML** statement and perform integrity constraint checking.
4. Execute all **AFTER STATEMENT** triggers.

ORACLE

Trigger Execution Model

A single DML statement can potentially fire up to four types of triggers: **BEFORE** and **AFTER** statement and row triggers. A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. Triggers can also cause other triggers to fire (cascading triggers).

All actions and checks done as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, all actions performed because of the original SQL statement are rolled back. This includes actions performed by firing triggers. This guarantees that integrity constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may undergo changes by other users' transactions. In all cases, a read-consistent image is guaranteed for modified values the trigger needs to read (query) or write (update).

Trigger Execution Model and Constraint Checking: Example

```
UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Integrity constraint violation error
```

```
CREATE OR REPLACE TRIGGER constr_emp_trig
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO departments
        VALUES (999, 'dept999', 140, 2400);
END;
/
```

```
UPDATE employees SET department_id = 999
WHERE employee_id = 170;
-- Successful after trigger is fired
```

ORACLE

9-33

Copyright © Oracle Corporation, 2001. All rights reserved.

Trigger Execution Model and Constraint Checking: Example

The example in the slide explains a situation in which the integrity constraint can be taken care of by using a trigger. Table EMPLOYEES has a foreign key constraint on the DEPARTMENT_ID column of the DEPARTMENTS table.

In the first SQL statement, the DEPARTMENT_ID of the employee with EMPLOYEE_ID 170 is modified to 999.

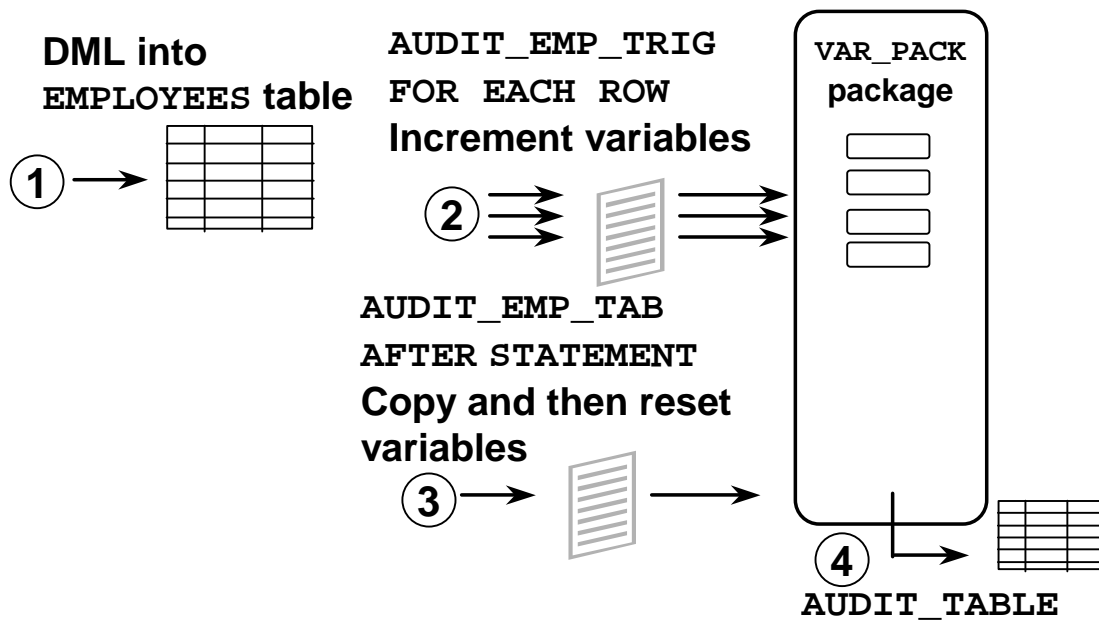
Because such a department does not exist in the DEPARTMENTS table, the statement raises the exception -2292 for the integrity constraint violation.

A trigger CONSTR_EMP_TRIG is created that inserts a new department 999 into the DEPARTMENTS table.

When the UPDATE statement that modifies the department of employee 170 to 999 is issued, the trigger fires. Then, the foreign key constraint is checked. Because the trigger inserted the department 999 into the DEPARTMENTS table, the foreign key constraint check is successful and there is no exception.

This process works with Oracle8i and later releases. The example described in the slide produces a run-time error in releases prior to Oracle8i.

A Sample Demonstration for Triggers Using Package Constructs



ORACLE

9-34

Copyright © Oracle Corporation, 2001. All rights reserved.

A Sample Demonstration

The following pages of PL/SQL subprograms are an example of the interaction of triggers, packaged procedures, functions, and global variables.

The sequence of events:

1. Issue an INSERT, UPDATE, or DELETE command that can manipulate one or many rows.
2. AUDIT_EMP_TRIG, the AFTER ROW trigger, calls the packaged procedure to increment the global variables in the package VAR_PACK. Because this is a row trigger, the trigger fires once for each row that you updated.
3. When the statement has finished, AUDIT_EMP_TAB, the AFTER STATEMENT trigger, calls the procedure AUDIT_EMP.
4. This procedure assigns the values of the global variables into local variables using the packaged functions, updates the AUDIT_TABLE, and then resets the global variables.

After Row and After Statement Triggers

```
CREATE OR REPLACE TRIGGER audit_emp_trig
AFTER      UPDATE or INSERT or DELETE on EMPLOYEES
FOR EACH ROW
BEGIN
    IF      DELETING      THEN  var_pack.set_g_del(1);
    ELSIF   INSERTING     THEN  var_pack.set_g_ins(1);
    ELSIF   UPDATING ('SALARY')
                                THEN  var_pack.set_g_up_sal(1);
    ELSE    var_pack.set_g_upd(1);
    END IF;
END audit_emp_trig;
```

```
CREATE OR REPLACE TRIGGER audit_emp_tab
AFTER      UPDATE or INSERT or DELETE on employees
BEGIN
    audit_emp;
END audit_emp_tab;
```

ORACLE

AFTER Row and AFTER Statement Triggers

The trigger `AUDIT_EMP_TRIG` is a row trigger that fires after every row manipulated. This trigger invokes the package procedures depending on the type of DML performed. For example, if the DML updates salary of an employee, then the trigger invokes the procedure `SET_G_UP_SAL`. This package procedure in turn invokes the function `G_UP_SAL`. This function increments the package variable `GV_UP_SAL` that keeps account of the number of rows being changed due to update of the salary.

The trigger `AUDIT_EMP_TAB` will fire after the statement has finished. This trigger invokes the procedure `AUDIT_EMP`, which is on the following pages. The `AUDIT_EMP` procedure updates the `AUDIT_TABLE` table. An entry is made into the `AUDIT_TABLE` table with the information such as the user who performed the DML, the table on which DML is performed, and the total number of such data manipulations performed so far on the table (indicated by the value of the corresponding column in the `AUDIT_TABLE` table). At the end, the `AUDIT_EMP` procedure resets the package variables to 0.

Demonstration: VAR_PACK Package Specification

var_pack.sql

```
CREATE OR REPLACE PACKAGE var_pack
IS
  -- these functions are used to return the
  -- values of package variables
  FUNCTION g_del RETURN NUMBER;
  FUNCTION g_ins RETURN NUMBER;
  FUNCTION g_upd RETURN NUMBER;
  FUNCTION g_up_sal RETURN NUMBER;
  -- these procedures are used to modify the
  -- values of the package variables
  PROCEDURE set_g_del (p_val IN NUMBER);
  PROCEDURE set_g_ins (p_val IN NUMBER);
  PROCEDURE set_g_upd (p_val IN NUMBER);
  PROCEDURE set_g_up_sal (p_val IN NUMBER);
END var_pack;
```

ORACLE

Demonstration: VAR_PACK Package Body

var_pack_body.sql

```
CREATE OR REPLACE PACKAGE BODY var_pack IS
  gv_del      NUMBER := 0;  gv_ins      NUMBER := 0;
  gv_upd      NUMBER := 0;  gv_up_sal  NUMBER := 0;
  FUNCTION g_del RETURN NUMBER IS
  BEGIN
    RETURN gv_del;
  END;
  FUNCTION g_ins RETURN NUMBER IS
  BEGIN
    RETURN gv_ins;
  END;
  FUNCTION g_upd RETURN NUMBER IS
  BEGIN
    RETURN gv_upd;
  END;
  FUNCTION g_up_sal RETURN NUMBER IS
  BEGIN
    RETURN gv_up_sal;
  END;
```

(continued on the next page)

VAR_PACK Package Body (continued)

```
PROCEDURE set_g_del (p_val  IN NUMBER) IS
BEGIN
    IF p_val = 0  THEN
        gv_del := p_val;
    ELSE gv_del := gv_del +1;
    END IF;
END set_g_del;
PROCEDURE set_g_ins (p_val  IN NUMBER) IS
BEGIN
    IF p_val = 0  THEN
        gv_ins := p_val;
    ELSE gv_ins := gv_ins +1;
    END IF;
END set_g_ins;
PROCEDURE set_g_upd (p_val  IN NUMBER) IS
BEGIN
    IF p_val = 0  THEN
        gv_upd := p_val;
    ELSE gv_upd := gv_upd +1;
    END IF;
END set_g_upd;
PROCEDURE set_g_up_sal (p_val  IN NUMBER) IS
BEGIN
    IF p_val = 0  THEN
        gv_up_sal := p_val;
    ELSE gv_up_sal := gv_up_sal +1;
    END IF;
END set_g_up_sal;
END var_pack;
/
```

Demonstration: Using the AUDIT_EMP Procedure

```
CREATE OR REPLACE PROCEDURE audit_emp IS
  v_del      NUMBER      := var_pack.g_del;
  v_ins      NUMBER      := var_pack.g_ins;
  v_upd      NUMBER      := var_pack.g_upd;
  v_up_sal   NUMBER      := var_pack.g_up_sal;
BEGIN
  IF v_del + v_ins + v_upd != 0 THEN
    UPDATE audit_table SET
      del = del + v_del, ins = ins + v_ins,
      upd = upd + v_upd
    WHERE user_name=USER AND tablename='EMPLOYEES'
    AND   column_name IS NULL;
  END IF;
  IF v_up_sal != 0 THEN
    UPDATE audit_table SET upd = upd + v_up_sal
    WHERE user_name=USER AND tablename='EMPLOYEES'
    AND   column_name = 'SALARY';
  END IF;
  -- resetting global variables in package VAR_PACK
  var_pack.set_g_del (0); var_pack.set_g_ins (0);
  var_pack.set_g_upd (0); var_pack.set_g_up_sal (0);
END audit_emp;
```

ORACLE

Updating the AUDIT_TABLE with the AUDIT_EMP Procedure

The AUDIT_EMP procedure updates the AUDIT_TABLE and calls the functions in the package VAR_PACK that reset the package variables, ready for the next DML statement.

Summary

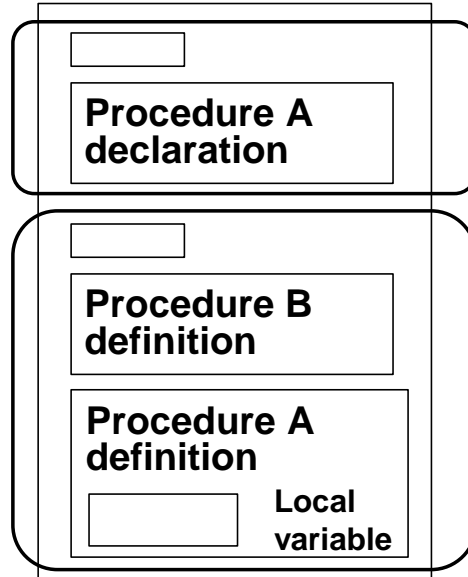
Procedure

```

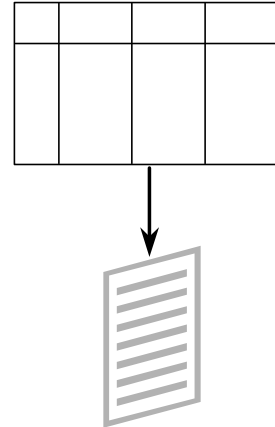
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxx

```

Package



Trigger



Develop different types of procedural database constructs depending on their usage.

Construct	Usage
Procedure	PL/SQL programming block that is stored in the database for repeated execution
Package	Group of related procedures, functions, variables, cursors, constants, and exceptions
Trigger	PL/SQL programming block that is executed implicitly by a data manipulation statement

Practice 9 Overview

This practice covers the following topics:

- **Creating statement and row triggers**
- **Creating advanced triggers to add to the capabilities of the Oracle database**

ORACLE

9-40

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 9 Overview

You create statement and row triggers in this practice. You create procedures that will be invoked from the triggers.

Practice 9

1. Changes to data are allowed on tables only during normal office hours of 8:45 a.m. until 5:30 p.m., Monday through Friday.

Create a stored procedure called `SECURE_DML` that prevents the DML statement from executing outside of normal office hours, returning the message, "You may only make changes during normal office hours."

- a. Create a statement trigger on the `JOBS` table that calls the above procedure.
- b. Test the procedure by temporarily modifying the hours in the procedure and attempting to insert a new record into the `JOBS` table. After testing, reset the procedure hours as specified in step 1.

If you have time:

3. Employees should receive an automatic increase in salary if the minimum salary for a job is increased. Implement this requirement through a trigger on the `JOBS` table.
 - a. Create a stored procedure named `UPD_EMP_SAL` to update the salary amount. This procedure accepts two parameters: the job ID for which salary has to be updated, and the new minimum salary for this job ID. This procedure is executed from the trigger on the `JOBS` table.
 - b. Create a row trigger named `UPDATE_EMP_SALARY` on the `JOBS` table that invokes the procedure `UPD_EMP_SAL`, when the minimum salary in the `JOBS` table is updated for a specified job ID.
 - c. Query the `EMPLOYEES` table to see the current salary for employees who are programmers.

LAST_NAME	FIRST_NAME	SALARY
Hunold	Alexander	9000
Ernst	Bruce	6000
Austin	David	4800
Pataballa	Valli	4800
Lorentz	Diana	4200

- d. Increase the minimum salary for the Programmer job from 4,000 to 5,000.
- e. Employee Lorentz (employee ID 107) had a salary of less than 4,500. Verify that her salary has been increased to the new minimum of 5,000.

LAST_NAME	FIRST_NAME	SALARY
Lorentz	Diana	5000

10

More Trigger Concepts

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Create additional database triggers**
- **Explain the rules governing triggers**
- **Implement triggers**

ORACLE

10-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to create more database triggers and learn the rules governing triggers. You also learn many applications of triggers.

Creating Database Triggers

- **Triggering user event:**
 - **CREATE, ALTER, or DROP**
 - **Logging on or off**
- **Triggering database or system event:**
 - **Shutting down or starting up the database**
 - **A specific error (or any error) being raised**

ORACLE

10-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Database Triggers

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level. Triggers on data definition language (DDL) statements, or a user logging on or off, can also be defined at either the database level or schema level. Triggers on DML statements are defined on a specific table or a view.

A trigger defined at the database level fires for all users, and a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shutdown or startup
- A specific or any error that occurs

Creating Triggers on DDL Statements

Syntax:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    [ddl_event1 [OR ddl_event2 OR ...]]
    ON {DATABASE|SCHEMA}
    trigger_body
```

ORACLE

10-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Create Trigger Syntax

DDL_Event	Possible Values
CREATE	Causes the Oracle server to fire the trigger whenever a CREATE statement adds a new database object to the dictionary
ALTER	Causes the Oracle server to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary
DROP	Causes the Oracle server to fire the trigger whenever a DROP statement removes a database object in the data dictionary

The trigger body represents a complete PL/SQL block.

You can create triggers for these events on DATABASE or SCHEMA. You also specify BEFORE or AFTER for the timing of the trigger.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, or user.

Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    [database_event1 [OR database_event2 OR ...]]
    ON {DATABASE|SCHEMA}
    trigger_body
```

ORACLE

10-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Create Trigger Syntax

Database_event	Possible Values
AFTER SERVERERROR	Causes the Oracle server to fire the trigger whenever a server error message is logged
AFTER LOGON	Causes the Oracle server to fire the trigger whenever a user logs on to the database
BEFORE LOGOFF	Causes the Oracle server to fire the trigger whenever a user logs off the database
AFTER STARTUP	Causes the Oracle server to fire the trigger whenever the database is opened
BEFORE SHUTDOWN	Causes the Oracle server to fire the trigger whenever the database is shut down

You can create triggers for these events on DATABASE or SCHEMA except SHUTDOWN and STARTUP, which apply only to the DATABASE.

LOGON and LOGOFF Trigger Example

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO log_trig_table(user_id, log_date, action)
  VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF ON SCHEMA
BEGIN
  INSERT INTO log_trig_table(user_id, log_date, action)
  VALUES (USER, SYSDATE, 'Logging off');
END;
/
```

ORACLE

LOGON and LOGOFF Trigger Example

You can create this trigger to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify ON SCHEMA, the trigger fires for the specific user. If you specify ON DATABASE, the trigger fires for all users.

CALL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    event1 [OR event2 OR event3]
    ON table_name
    [REFERENCING OLD AS old | NEW AS new]
    [FOR EACH ROW]
    [WHEN condition]
    CALL procedure_name;
```

```
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
    CALL log_execution
/
```

ORACLE

10-7

Copyright © Oracle Corporation, 2001. All rights reserved.

CALL Statements

A CALL statement enables you to call a stored procedure, rather than coding the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.

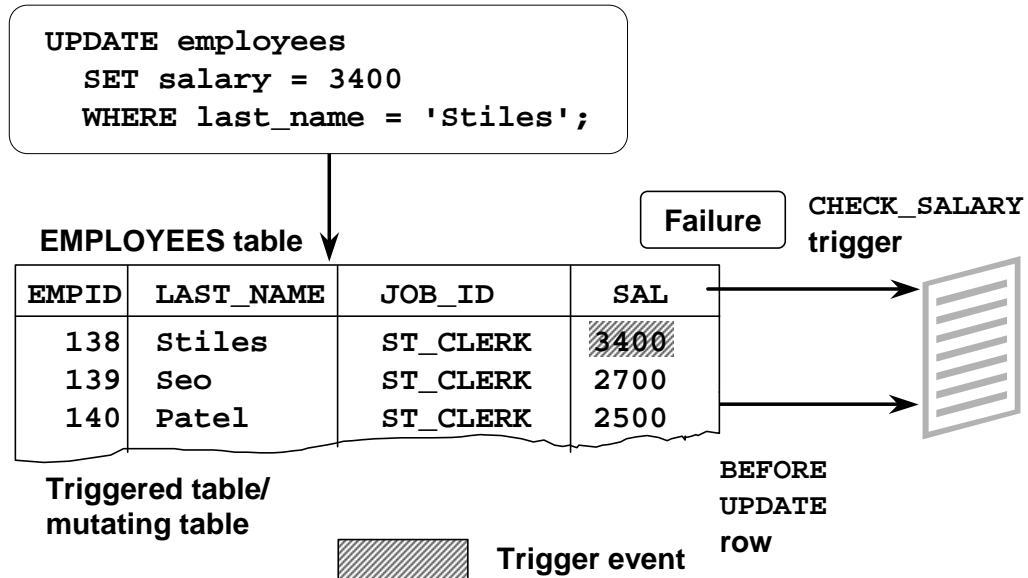
The call can reference the trigger attributes :NEW and :OLD as parameters as in the following example:

```
CREATE TRIGGER salary_check
    BEFORE UPDATE OF salary, job_id ON employees
    FOR EACH ROW
    WHEN (NEW.job_id <> 'AD_PRES')
    CALL check_sal(:NEW.job_id, :NEW.salary)
/
```

Note: There is no semicolon at the end of the CALL statement.

In the example above, the trigger calls a procedure `check_sal`. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

Reading Data from a Mutating Table



ORACLE

10-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of ON DELETE CASCADE.

Mutating Table

A mutating table is a table that is currently being modified by an UPDATE, DELETE, or INSERT statement, or a table that might need to be updated by the effects of a declarative DELETE CASCADE referential integrity action. A table is not considered mutating for STATEMENT triggers.

The triggered table itself is a mutating table, as well as any table referencing it with the FOREIGN KEY constraint. This restriction prevents a row trigger from seeing an inconsistent set of data.

Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE INSERT OR UPDATE OF salary, job_id
  ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
  v_minsalary employees.salary%TYPE;
  v_maxsalary employees.salary%TYPE;
BEGIN
  SELECT MIN(salary), MAX(salary)
  INTO   v_minsalary, v_maxsalary
  FROM   employees
  WHERE  job_id = :NEW.job_id;
  IF :NEW.salary < v_minsalary OR
     :NEW.salary > v_maxsalary THEN
    RAISE_APPLICATION_ERROR(-20505,'Out of range');
  END IF;
END;
/
```

ORACLE

Mutating Table: Example

The CHECK_SALARY trigger in the example, attempts to guarantee that whenever a new employee is added to the EMPLOYEES table or whenever an existing employee's salary or job ID is changed, the employee's salary falls within the established salary range for the employee's job.

When an employee record is updated, the CHECK_SALARY trigger is fired for each row that is updated. The trigger code queries the same table that is being updated. Hence, it is said that the EMPLOYEES table is mutating table.

Mutating Table: Example

```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```

```
UPDATE employees
*
ERROR at line 1:
ORA-04091: table PLPU.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "PLPU.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'PLPU.CHECK_SALARY'
```

ORACLE

Mutating Table: Example (continued)

Try to read from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value you get a run-time error. The EMPLOYEES table is mutating, or in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Implementing Triggers

You can use trigger for:

- **Security**
- **Auditing**
- **Data integrity**
- **Referential integrity**
- **Table replication**
- **Computing derived data automatically**
- **Event logging**

ORACLE

10-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Implementing Triggers

Develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server or as alternatives to those provided by the Oracle server.

Feature	Enhancement
Security	The Oracle server allows table access to users or roles. Triggers allow table access according to data values.
Auditing	The Oracle server tracks data operations on tables. Triggers track values for data operations on tables.
Data integrity	The Oracle server enforces integrity constraints. Triggers implement complex integrity rules.
Referential integrity	The Oracle server enforces standard referential integrity rules. Triggers implement nonstandard functionality.
Table replication	The Oracle server copies tables asynchronously into snapshots. Triggers copy tables synchronously into replicas.
Derived data	The Oracle server computes derived data values manually. Triggers compute derived data values automatically.
Event logging	The Oracle server logs events explicitly. Triggers log events transparently.

Controlling Security Within the Server

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON    employees
TO    clerk;                -- database role
GRANT clerk TO scott;
```

ORACLE

10-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling Security Within the Server

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data manipulation, and data definition privileges.

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
  v_dummy VARCHAR2(1);
BEGIN
  IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN'))
    THEN RAISE_APPLICATION_ERROR (-20506,'You may only
      change data during normal business hours.');
```

```
END IF;
SELECT COUNT(*) INTO v_dummy FROM holiday
WHERE holiday_date = TRUNC (SYSDATE);
IF v_dummy > 0 THEN RAISE_APPLICATION_ERROR(-20507,
  'You may not change data on a holiday.');
```

```
END IF;
END;
/
```

ORACLE

10-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling Security With a Database Trigger

Develop triggers to handle more complex security requirements.

- Base privileges on any database values, such as the time of day, the day of the week, and so on.
- Determine access to tables only.
- Determine data manipulation privileges only.

Using the Server Facility to Audit Data Operations

```
AUDIT INSERT, UPDATE, DELETE  
  ON departments  
  BY ACCESS  
WHENEVER SUCCESSFUL;
```

The Oracle server stores the audit information in a data dictionary table or operating system file.

ORACLE

10-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Auditing Data Operations

You can audit data operations within the Oracle server. Database auditing is used to monitor and gather data about specific database activities. The DBA can gather statistics about which tables are being updated, how many I/Os are performed, how many concurrent users connect at peak time, and so on.

- Audit users, statements, or objects.
- Audit data retrieval, data manipulation, and data definition statements.
- Write the audit trail to a centralized audit table.
- Generate audit records once per session or once per access attempt.
- Capture successful attempts, unsuccessful attempts, or both.
- Enable and disable dynamically.

Executing SQL through PL/SQL program units may generate several audit records because the program units may refer to other database objects.

Auditing by Using a Trigger

```
CREATE OR REPLACE TRIGGER audit_emp_values
  AFTER DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
BEGIN
  IF (audit_emp_package.g_reason IS NULL) THEN
    RAISE_APPLICATION_ERROR (-20059, 'Specify a reason
    for the data operation through the procedure SET_REASON
    of the AUDIT_EMP_PACKAGE before proceeding.');
```

```
  ELSE
    INSERT INTO audit_emp_table (user_name, timestamp, id,
      old_last_name, new_last_name, old_title, new_title,
      old_salary, new_salary, comments)
    VALUES (USER, SYSDATE, :OLD.employee_id, :OLD.last_name,
      :NEW.last_name, :OLD.job_id, :NEW.job_id, :OLD.salary,
      :NEW.salary, audit_emp_package.g_reason);
  END IF;
END;
```

```
CREATE OR REPLACE TRIGGER cleanup_audit_emp
  AFTER INSERT OR UPDATE OR DELETE ON employees
BEGIN
  audit_emp_package.g_reason := NULL;
END;
```

ORACLE

10-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Audit Data Values

Audit actual data values with triggers.

You can:

- Audit data manipulation statements only
- Write the audit trail to a user-defined audit table
- Generate audit records once for the statement or once for each row
- Capture successful attempts only
- Enable and disable dynamically

Using the Oracle server, you can perform database auditing. Database auditing cannot record changes to specific column values. If the changes to the table columns need to be tracked and column values need to be stored for each change, use application auditing. Application auditing can be done either through stored procedures or database triggers, as shown in the example in the slide.

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD  
CONSTRAINT ck_salary CHECK (salary >= 500);
```

ORACLE

10-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Enforcing Data Integrity within the Server

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key.

Use these rules to:

- Provide constant default values
- Enforce static constraints
- Enable and disable dynamically

Example

The code sample in the slide ensures that the salary is at least \$500.

Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
    'Do not decrease salary.');
```

```
END;
/
```

ORACLE

10-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Protecting Data Integrity with a Trigger

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

Example

The code sample in the slide ensures that the salary is never decreased.

Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
  FOREIGN KEY (department_id)
    REFERENCES departments(department_id)
  ON DELETE CASCADE;
```

ORACLE

10-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Enforcing Referential Integrity within the Server

Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

- Restrict updates and deletes.
- Cascade deletes.
- Enable and disable dynamically.

Example

When a department is removed from the DEPARTMENTS parent table, cascade the deletion to the corresponding rows in the EMPLOYEES child table.

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
  AFTER UPDATE OF department_id ON departments
  FOR EACH ROW
BEGIN
  UPDATE employees
    SET employees.department_id=:NEW.department_id
    WHERE employees.department_id=:OLD.department_id;
  UPDATE job_history
    SET department_id=:NEW.department_id
    WHERE department_id=:OLD.department_id;
END;
/
```

ORACLE

10-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Protecting Referential Integrity with a Trigger

Develop triggers to implement referential integrity rules that are not supported by declarative constraints.

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.

Example

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table.

For a complete referential integrity solution using triggers, a single trigger is not enough.

Replicating a Table Within the Server

```
CREATE SNAPSHOT emp_copy AS  
  SELECT * FROM employees@ny;
```

ORACLE

10-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Snapshot

A snapshot is a local copy of a table data that originates from one or more remote master tables. An application can query the data in a read-only table snapshot, but cannot insert, update, or delete rows in the snapshot. To keep a snapshot's data current with the data of its master, the Oracle server must periodically refresh the snapshot.

When this statement is used in SQL, replication is performed implicitly by the Oracle server by using internal triggers. This has better performance over using user-defined PL/SQL triggers for replication.

Copying Tables with Server Snapshots

Copy a table with a snapshot.

- Copy tables asynchronously, at user-defined intervals.
- Base snapshots on multiple master tables.
- Read from snapshots only.
- Improve the performance of data manipulation on the master table, particularly if the network fails.

Alternatively, you can replicate tables using triggers.

Example

In San Francisco, create a snapshot of the remote EMPLOYEES table in New York.

Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN /*Only proceed if user initiates a data operation,
      NOT through the cascading trigger.*/
  IF INSERTING THEN
    IF :NEW.flag IS NULL THEN
      INSERT INTO employees@sf
      VALUES(:new.employee_id, :new.last_name,..., 'B');
      :NEW.flag := 'A';
    END IF;
  ELSE /* Updating. */
    IF :NEW.flag = :OLD.flag THEN
      UPDATE employees@sf
      SET ename = :NEW.last_name, ...,
      flag = :NEW.flag
      WHERE employee_id = :NEW.employee_id;
    END IF;
    IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
    ELSE :NEW.flag := 'A';
    END IF;
  END IF;
END;
```

ORACLE

10-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Replicating a Table with a Trigger

Replicate a table with a trigger.

- Copy tables synchronously, in real time.
- Base replicas on a single master table.
- Read from replicas, as well as write to them.
- Impair the performance of data manipulation on the master table, particularly if the network fails.

Maintain copies of tables automatically with snapshots, particularly on remote nodes.

Example

In New York, replicate the local EMPLOYEES table to San Francisco.

Computing Derived Data Within the Server

```
UPDATE departments
SET total_sal=(SELECT SUM(salary)
                FROM employees
                WHERE employees.department_id =
                      departments.department_id);
```

ORACLE

10-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Computing Derived Data within the Server

Compute derived values in a batch job.

- Compute derived column values asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

Example

Keep the salary total for each department within a special TOTAL_SALARY column of the DEPARTMENTS table.

Computing Derived Values with a Trigger

```
CREATE OR REPLACE PROCEDURE increment_salary
(p_id      IN departments.department_id%TYPE,
p_salary IN departments.total_sal%TYPE)
IS
BEGIN
    UPDATE departments
    SET    total_sal = NVL (total_sal, 0)+ p_salary
    WHERE department_id = p_id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE ON employees
FOR EACH ROW
BEGIN
    IF DELETING THEN
        increment_salary(:OLD.department_id, (-1* :OLD.salary));
    ELSIF UPDATING THEN
        increment_salary(:NEW.department_id, (:NEW.salary-:OLD.salary));
    ELSE increment_salary(:NEW.department_id, :NEW.salary);--INSERT
    END IF;
END;
```

ORACLE

10-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Computing Derived Data Values with a Trigger

Compute derived values with a trigger.

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a single pass to the database.

Example

Keep a running total of the salary for each department within the special TOTAL_SALARY column of the DEPARTMENTS table.

Logging Events with a Trigger

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
ON inventories FOR EACH ROW
DECLARE
v_descrip product_descriptions.product_description%TYPE;
v_msg_text VARCHAR2(2000);
stat_send number(1);
BEGIN
IF :NEW.quantity_on_hand <= :NEW.reorder_point THEN
SELECT product_description INTO v_descrip
FROM product_descriptions
WHERE product_id = :NEW.product_id;
v_msg_text := 'ALERT: INVENTORY LOW ORDER:' || CHR(10) ||
... 'Yours,' || CHR(10) || user || '.' || CHR(10) || CHR(10);
ELSIF
:OLD.quantity_on_hand < :NEW.quantity_on_hand THEN NULL;
ELSE
v_msg_text := 'Product #' || ... CHR(10);
END IF;
DBMS_PIPE.PACK_MESSAGE(v_msg_text);
stat_send := DBMS_PIPE.SEND_MESSAGE('INV_PIPE');
END;
```

ORACLE

10-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Logging Events with a Trigger

Within the server, you can log events by querying data and performing operations manually. This sends a message using a pipe when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package DBMS_PIPE to send the message.

Logging Events within the Server

- Query data explicitly to determine whether an operation is necessary.
- In a second step, perform the operation, such as sending a message.

Using Triggers to Log Events

- Perform operations implicitly, such as firing off an automatic electronic memo.
- Modify data and perform its dependent operation in a single step.
- Log events automatically as data is changing.

Logging Events with a Trigger (continued)

Logging Events Transparently

In the trigger code:

- `CHR(10)` is a carriage return
- `Reorder_point` is not null
- Another transaction can receive and read the message in the pipe

Example

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF amount_in_stock, reorder_point
ON inventory FOR EACH ROW
DECLARE
    v_descrip product.descrip%TYPE;
    v_msg_text VARCHAR2(2000);
    stat_send  number(1);
BEGIN
    IF :NEW.amount_in_stock <= :NEW.reorder_point THEN
        SELECT descrip INTO v_descrip
        FROM PRODUCT WHERE prodid = :NEW.product_id;
        v_msg_text := 'ALERT: INVENTORY LOW ORDER:' || CHR(10) ||
        'It has come to my personal attention that, due to recent'
        || CHR(10) || 'transactions, our inventory for product # ' ||
        TO_CHAR(:NEW.product_id) || '-- ' || v_descrip ||
        ' -- has fallen below acceptable levels.' || CHR(10) ||
        'Yours,' || CHR(10) || user || '.' || CHR(10) || CHR(10);
    ELSIF
        :OLD.amount_in_stock < :NEW.amount_in_stock THEN NULL;
    ELSE
        v_msg_text := 'Product #' || TO_CHAR(:NEW.product_id)
        || ' ordered. ' || CHR(10) || CHR(10);    END IF;
    DBMS_PIPE.PACK_MESSAGE(v_msg_text);
    stat_send := DBMS_PIPE.SEND_MESSAGE('INV_PIPE');
END;
```

Benefits of Database Triggers

- **Improved data security:**
 - **Provide enhanced and complex security checks**
 - **Provide enhanced and complex auditing**
- **Improved data integrity:**
 - **Enforce dynamic data integrity constraints**
 - **Enforce complex referential integrity constraints**
 - **Ensure that related operations are performed together implicitly**

ORACLE

10-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Benefits of Database Triggers

You can use database triggers:

- As alternatives to features provided by the Oracle server
- If your requirements are more complex or more simple than those provided by the Oracle server
- If your requirements are not provided by the Oracle server at all

Managing Triggers

The following system privileges are required to manage triggers:

- **The `CREATE/ALTER/DROP (ANY) TRIGGER` privilege enables you to create a trigger in any schema**
- **The `ADMINISTER DATABASE TRIGGER` privilege enables you to create a trigger on `DATABASE`**
- **The `EXECUTE` privilege (if your trigger refers to any objects that are not in your schema)**

Note: Statements in the trigger body operate under the privilege of the trigger owner, not the trigger user.

ORACLE

10-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Managing Triggers

In order to create a trigger in your schema, you need the `CREATE TRIGGER` system privilege, and you must either own the table specified in the triggering statement, have the `ALTER` privilege for the table in the triggering statement, or have the `ALTER ANY TABLE` system privilege. You can alter or drop your triggers without any further privileges being required.

If the `ANY` keyword is used, you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.

You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the `EXECUTE` privilege on the referenced procedures, functions, or packages, and not through roles. As with stored procedures, the statement in the trigger body operates under the privilege domain of the trigger's owner, not that of the user issuing the triggering statement.

To create a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` privilege. If this privilege is later revoked, you can drop the trigger, but you cannot alter it.

Viewing Trigger Information

You can view the following trigger information:

- **USER_OBJECTS data dictionary view: object information**
- **USER_TRIGGERS data dictionary view: the text of the trigger**
- **USER_ERRORS data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger**

ORACLE

10-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Viewing Trigger Information

The slide shows the data dictionary views that you can access to get information regarding the triggers.

The USER_OBJECTS view contains the name and status of the trigger and the date and time when the trigger was created.

The USER_ERRORS view contains the details of the compilation errors that occurred while a trigger was compiling. The contents of these views are similar to those for subprograms.

The USER_TRIGGERS view contains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

The `SELECT Username FROM USER_USERS;` statement gives the name of the owner of the trigger, not the name of the user who is updating the table.

Using USER_TRIGGERS*

Column	Column Description
TRIGGER_NAME	Name of the trigger
TRIGGER_TYPE	The type is BEFORE, AFTER, INSTEAD OF
TRIGGERING_EVENT	The DML operation firing the trigger
TABLE_NAME	Name of the database table
REFERENCING_NAMES	Name used for :OLD and :NEW
WHEN_CLAUSE	The when_clause used
STATUS	The status of the trigger
TRIGGER_BODY	The action to take

* Abridged column list

ORACLE

10-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Using USER_TRIGGERS

If the source file is unavailable, you can use *iSQL*Plus* to regenerate it from USER_TRIGGERS. You can also examine the ALL_TRIGGERS and DBA_TRIGGERS views, each of which contains the additional column OWNER, for the owner of the object.

Listing the Code of Triggers

```
SELECT trigger_name, trigger_type, triggering_event,  
       table_name, referencing_names,  
       status, trigger_body  
FROM   user_triggers  
WHERE  trigger_name = 'RESTRICT_SALARY';
```

TRIGGER_NAME	TRIGGER_TYPE	TRIGGERING_EVENT	TABLE_NAME	REFERENCING_NAMES	STATUS	TRIGGER_BODY
RESTRICT_SALARY	BEFORE EACH ROW	INSERT OR UPDATE	EMPLOYEES	REFERENCING NEW AS NEW OLD AS OLD	ENABLED	BEGIN IF NOT (:NEW.JOB_ID IN ('AD_PRES', 'AD_VP')) AND :NEW.SAL

ORACLE®

Example

Use the USER_TRIGGERS data dictionary view to display information about the RESTRICT_SAL trigger.

Summary

In this lesson, you should have learned how to:

- **Use advanced database triggers**
- **List mutating and constraining rules for triggers**
- **Describe the real-world application of triggers**
- **Manage triggers**
- **View trigger information**

ORACLE

Practice 10 Overview

This practice covers creating advanced triggers to add to the capabilities of the Oracle database.

ORACLE

10-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 10 Overview

In this practice you decide how to implement a number of business rules. You will create triggers for those rules that should be implemented as triggers. The triggers will execute procedures that you have placed in a package.

Practice 10

1. A number of business rules that apply to the EMPLOYEES and DEPARTMENTS tables are listed below.

Decide how to implement each of these business rules, by means of declarative constraints or by using triggers.

Which constraints or triggers are needed and are there any problems to be expected?

Implement the business rules by defining the triggers or constraints that you decided to create.

A partial package is provided in file lab10_1.sql to which you should add any necessary procedures or functions that are to be called from triggers that you may create for the following rules.

(The triggers should execute procedures or functions that you have defined in the package.)

Business Rules

Rule 1. Sales managers and sales representatives should always receive commission. Employees who are not sales managers or sales representatives should never receive a commission. Ensure that this restriction does not validate the existing records of the EMPLOYEES table. It should be effective only for the subsequent inserts and updates on the table.

Rule 2. The EMPLOYEES table should contain exactly one president.

Test your answer by inserting an employee record with the following details: employee ID 400, last name Harris, first name Alice, e-mail ID AHARRIS, job ID AD_PRES, hire date SYSDATE, salary 10000, and department ID 20.

Note: You do not need to implement a rule for case sensitivity; instead you need to test for the number of people with the job title of President.

Rule 3. An employee should never be a manager of more than 15 employees.

Test your answer by inserting the following records into the EMPLOYEES table (perform a query to count the number of employees currently working for manager 100 before inserting these rows):

- i. Employee ID 401, last name Johnson, first name Brian, e-mail ID BJOHNSON, job ID SA_MAN, hire date SYSDATE, salary 11000, manager ID 100, and department ID 80. (This insertion should be successful, because there are only 14 employees working for manager 100 so far.)
- ii. Employee ID 402, last name Kellogg, first name Tony, e-mail ID TKELLOG, job ID ST_MAN, hire date SYSDATE, salary 7500, manager ID 100, and department ID 50. (This insertion should be unsuccessful, because there are already 15 employees working for manager 100.)

- . Rule 4. Salaries can only be increased, never decreased.

The present salary of employee 105 is 5000. Test your answer by decreasing the salary of employee 105 to 4500.

Practice 10 (continued)

Rule 5. If a department moves to another location, each employee of that department automatically receives a salary raise of 2 percent.

View the current salaries of employees in department 90.

LAST_NAME	SALARY	DEPARTMENT_ID
King	24000	90
Kochhar	17000	90
De Haan	17000	90

Test your answer by moving department 90 to location 1600. Query the new salaries of employees of department 90.

LAST_NAME	SALARY	DEPARTMENT_ID
King	24480	90
Kochhar	17340	90
De Haan	17340	90

11

Managing Dependencies

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Track procedural dependencies**
- **Predict the effect of changing a database object upon stored procedures and functions**
- **Manage procedural dependencies**

ORACLE

11-2

Copyright © Oracle Corporation, 2001. All rights reserved.

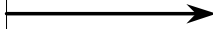
Lesson Aim

This lesson introduces you to object dependencies and implicit and explicit recompilation of invalid objects.

Understanding Dependencies

Dependent Objects

Table
View
Database Trigger
Procedure
Function
Package Body
Package Specification
User-Defined Object
and Collection Types



Referenced Objects

Function
Package Specification
Procedure
Sequence
Synonym
Table
View
User-Defined Object
and Collection Types

ORACLE

11-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Dependent and Referenced Objects

Some objects reference other objects as part of their definition. 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.

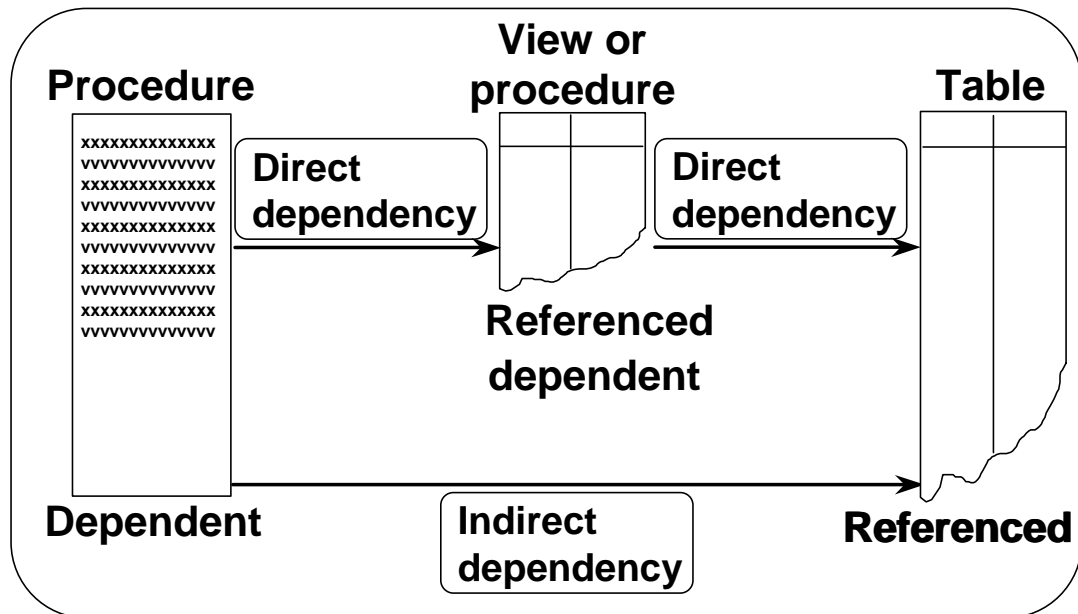
Dependency Issues

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.

Dependencies



ORACLE

11-4

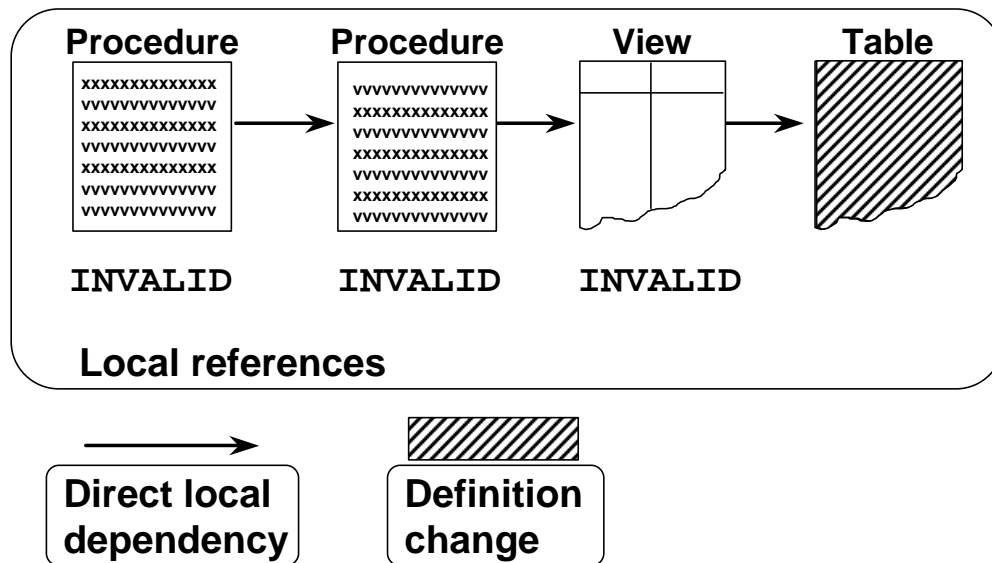
Copyright © Oracle Corporation, 2001. All rights reserved.

Dependent and Referenced Objects (continued)

A procedure or a 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



The Oracle server implicitly recompiles any **INVALID object when the object is next called.**

ORACLE

11-5

Copyright © Oracle Corporation, 2001. All rights reserved.

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.

Assume that the structure of the table on which a view is based is modified. When you describe the view by using *iSQL*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 and, at this stage, the view is invalid because the structure of its base table is changed. If you query the view now, 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
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv
```

EMP_VW view

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID	DEPARTMENT_NAME
100	King	24000	90	Executive
101	Kochhar	17000	90	Executive
102	De Haan	17000	90	Executive
103	Hunold	9000	60	IT
104	Ernst	6000	60	IT

QUERY_EMP procedure

```
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
xxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv
```

EMPLOYEES table

EMPLOYEE_ID	LAST_NAME	HIRE_DATE	JOB_ID
100	King	17-JUN-87	AD_PRES
101	Kochhar	21-SEP-89	AD_VP
102	De Haan	13-JAN-93	AD_VP
103	Hunold	03-JAN-90	IT_PROG

ORACLE

11-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

The QUERY_EMP procedure directly references the EMPLOYEES table. The ADD_EMP procedure updates the EMPLOYEES table indirectly, by way of the EMP_VW view.

In each of the following cases, will the ADD_EMP procedure be invalidated, and will 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
MGR_CONSTRAINTS_PKG	PACKAGE BODY	EMPLOYEES	TABLE
DETAIL_EMP	VIEW	EMPLOYEES	TABLE
QUERY_EMP	PROCEDURE	EMPLOYEES	TABLE
EMP_VW	VIEW	EMPLOYEES	TABLE
ADD_EMP	PROCEDURE	EMP_VW	VIEW

ORACLE

11-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Display 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, that reference 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 script `utldtree.sql` 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')
```

```
PL/SQL procedure successfully completed.
```

ORACLE

11-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle

Display direct and indirect dependencies from additional user views called `DEPTREE` and `IDETREE`; these view are provided by Oracle.

Example

1. Make sure 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:

<i>object_type</i>	Is the type of the referenced object
<i>object_owner</i>	Is the schema of the referenced object
<i>object_name</i>	Is the name of the referenced object

Displaying Dependencies

DEPTREE View

```
SELECT  nested_level, type, name
FROM    deptree
ORDER BY seq#;
```

NESTED_LEVEL	TYPE	NAME
0	TABLE	EMPLOYEES
1	PROCEDURE	NEW_EMP
1	PROCEDURE	NEW_EMP_TREE
1	VIEW	EMP_DETAILS_VIEW
1	PROCEDURE	QUERY_EMP
1	VIEW	EMP_VW
2	PROCEDURE	ADD_EMP

ORACLE

11-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

Display a tabular representation of all dependent objects by querying the DEPTREE view.

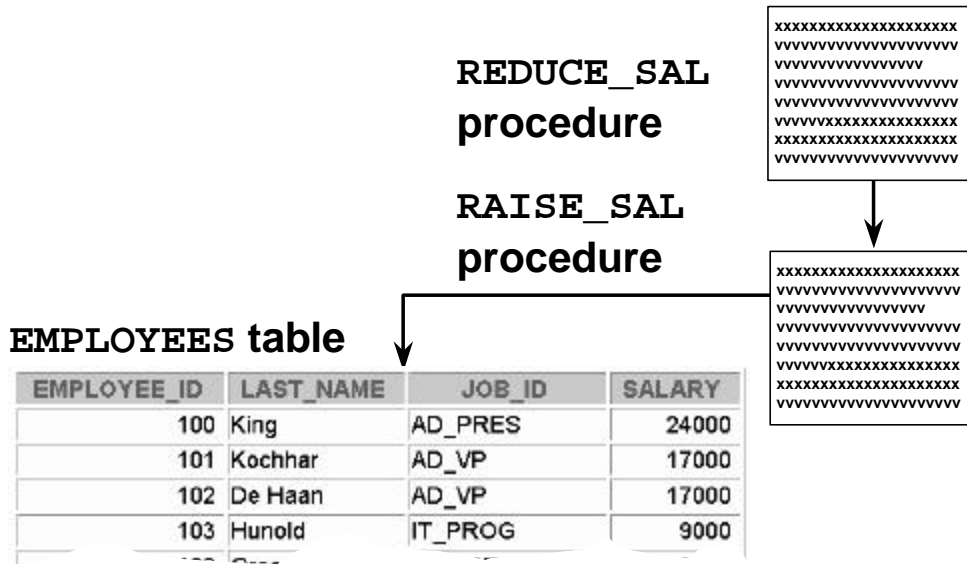
Display an indented representation of the same information by querying the IDEPTREE view, which consists of a single column named DEPENDENCIES.

For example,

```
SELECT *
FROM   ideptree;
```

provides a single column of indented output of the dependencies in a hierarchical structure.

Another Scenario of Local Dependencies



ORACLE

11-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Predicting the Effects of Changes on Dependent Objects

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, will 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**

```

xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv

```



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

**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

ORACLE

11-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Predicting Effects of Changes on Dependent Objects (continued)

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.

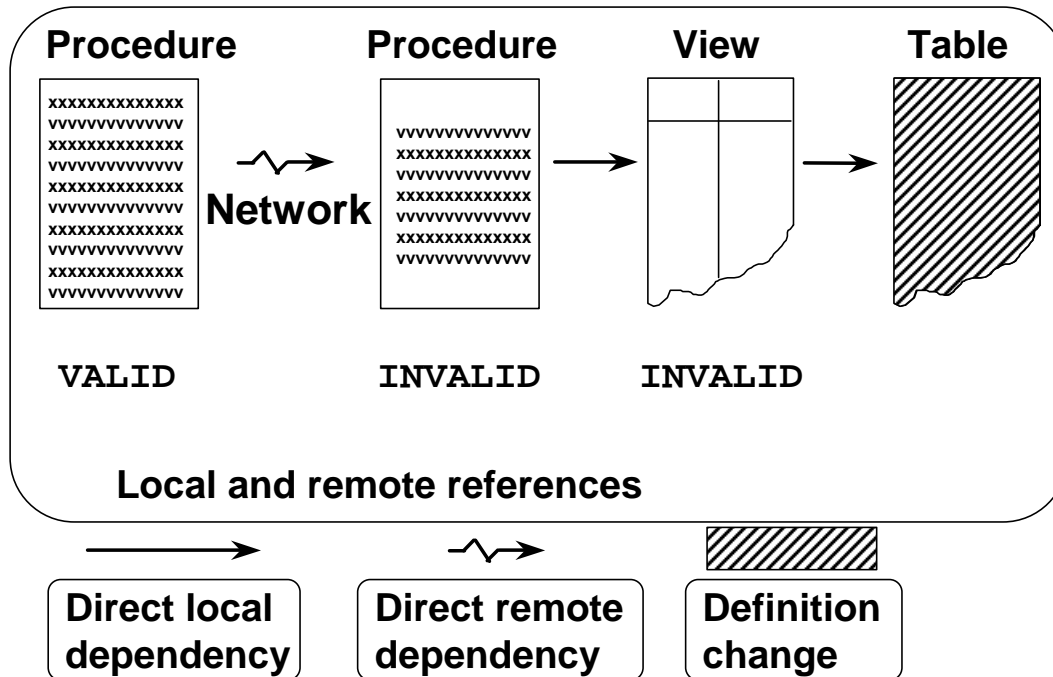
Predict the effect that the name of a new object has upon a dependent procedure.

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. Will this change invalidate the procedure? Which of the two `EMPLOYEES` objects will `QUERY_EMP` reference when the procedure recompiles?

Now suppose that you drop your private `EMPLOYEES` table. Will this invalidate the procedure? What will happen when the procedure recompiles?

You can track security dependencies within the `USER_TAB_PRIVS` data dictionary view.

Understanding Remote Dependencies



ORACLE

11-12

Copyright © Oracle Corporation, 2001. All rights reserved.

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 of its dependent objects will be invalidated but will not automatically recompile when called for the first time.

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 chosen by the user:

- **TIMESTAMP checking**
- **SIGNATURE checking**

ORACLE

11-13

Copyright © Oracle Corporation, 2001. All rights reserved.

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 of 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 normally. If they do not match, the Remote Procedure Calls (RPC) layer performs a simple test to compare 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 status is returned.

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

11-14

Copyright © Oracle Corporation, 2001. All rights reserved.

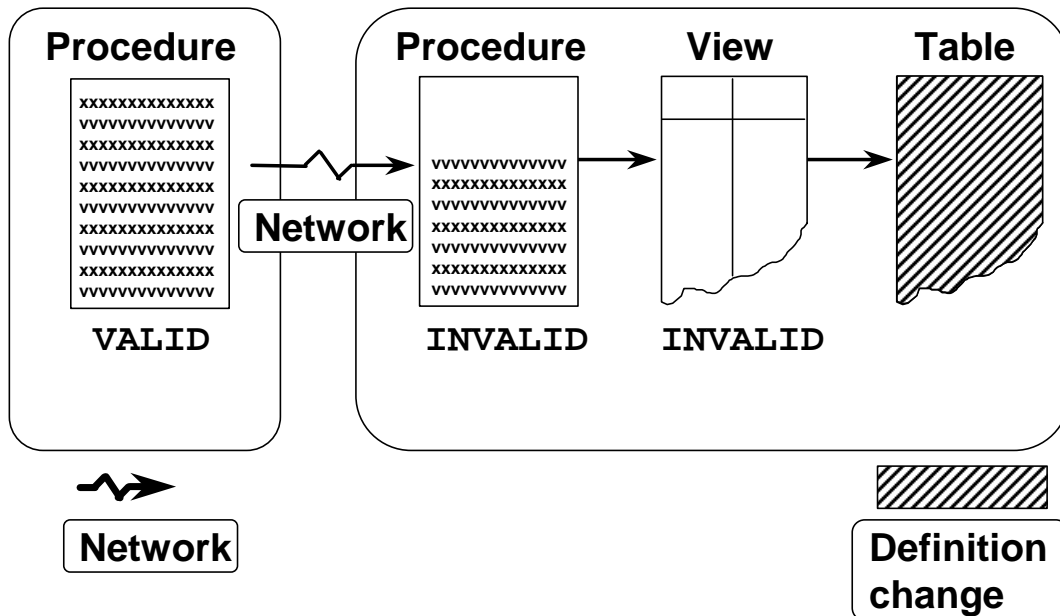
Setting the REMOTE_DEPENDENCIES_MODE

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



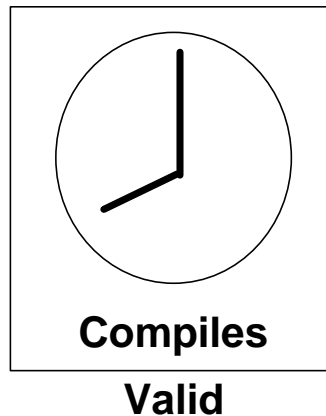
Using Time Stamp Mode for Automatic Recompilation of Local and Remote Objects

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 of 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



ORACLE

11-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Local Procedures Referencing Remote Procedures

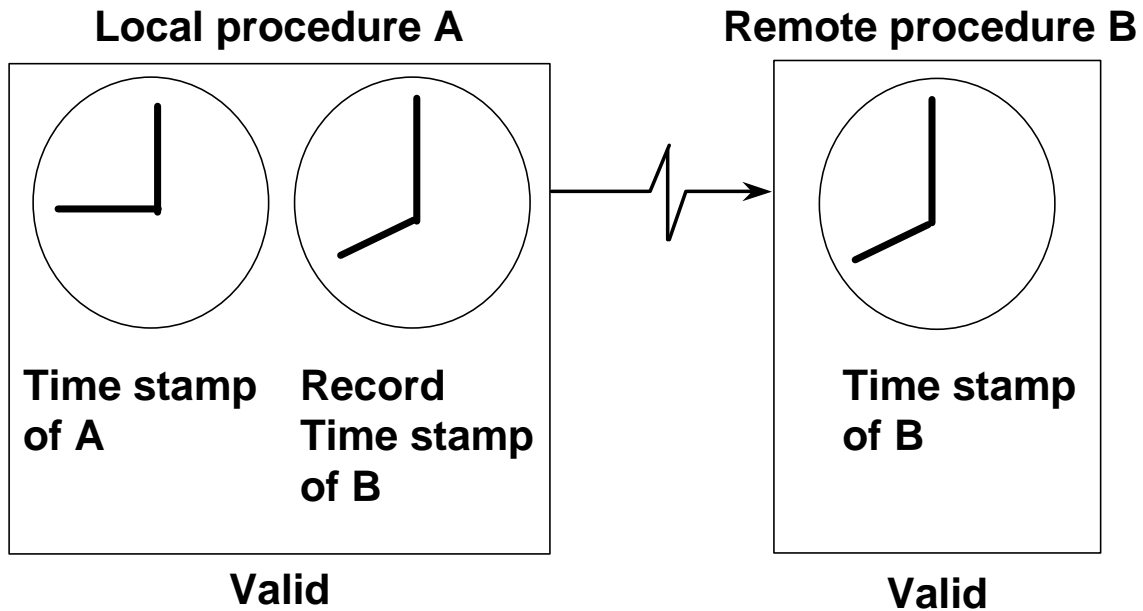
A local procedure that references a remote procedure is invalidated by the Oracle server if the remote procedure is recompiled after the local procedure is compiled.

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 was successfully compiled at 8 a.m., this time was recorded as its time stamp

Local Procedure A Compiles at 9:00 a.m.



ORACLE

11-17

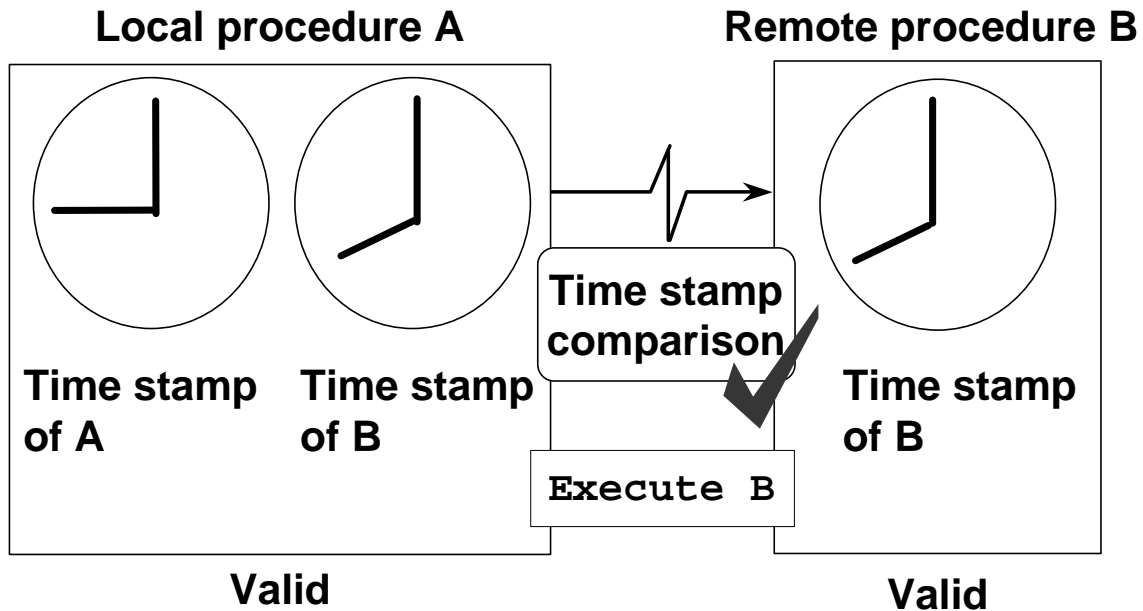
Copyright © Oracle Corporation, 2001. All rights reserved.

Automatic Remote Dependency Mechanism

When a local procedure referencing a remote procedure compiles, the Oracle server also records the time stamp of the remote procedure into 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



11-18

Copyright © Oracle Corporation, 2001. All rights reserved.

ORACLE

Automatic Remote Dependency

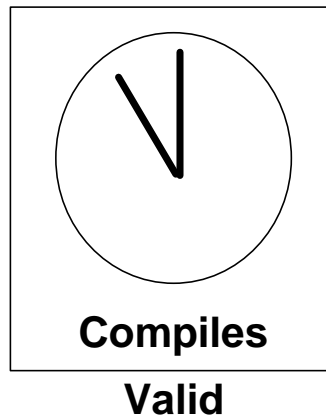
When the local procedure is invoked, at run time the Oracle server compares the two time stamps of the referenced remote procedure.

If the time stamps are equal (indicating that the remote procedure has not recompiled), the Oracle server executes the local procedure.

In the example in the slide, the time stamp recorded with P code of remote procedure B is the same as that recorded with local procedure A. Hence, local procedure A is valid.

Remote Procedure B Recompiled at 11:00 a.m.

Remote procedure B



ORACLE

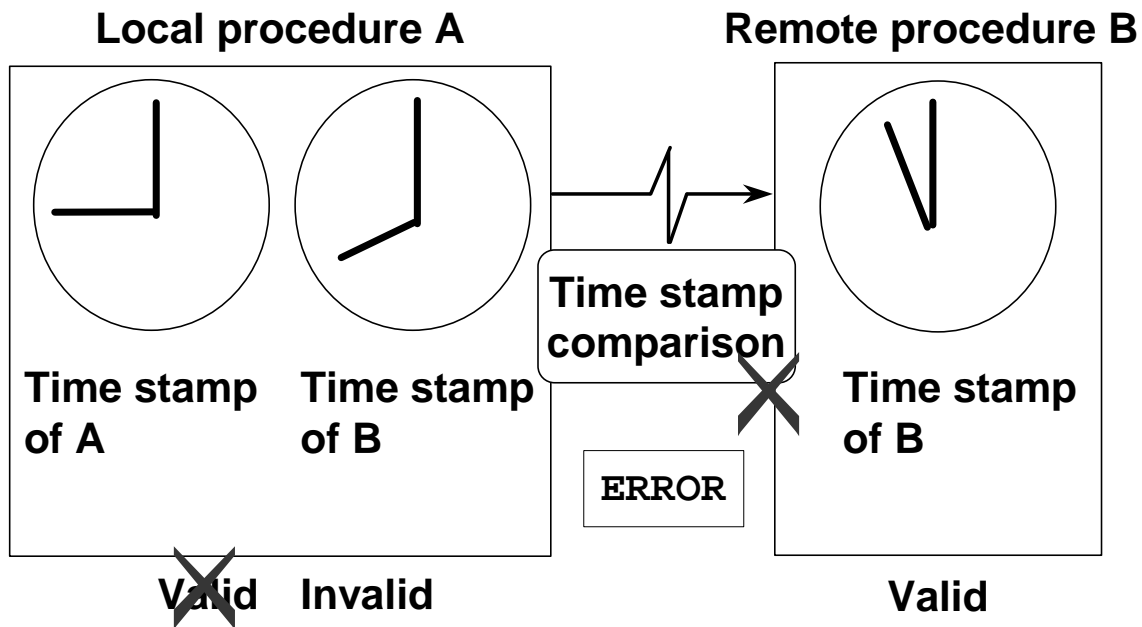
11-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Local Procedures Referencing Remote Procedures

Assume that the remote procedure B is successfully recompiled at 11 a.m. The new time stamp is recorded along with its P code.

Execute Procedure A



11-20

Copyright © Oracle Corporation, 2001. All rights reserved.

ORACLE

Automatic Remote Dependency

If the time stamps are not equal (indicating that the remote procedure has recompiled), the Oracle server invalidates the local procedure and returns a runtime error.

If the local procedure, which is now tagged as invalid, is invoked a second time, 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 that it is invoked, indicating that the remote procedure's time stamp has changed, you should develop a strategy to reinvoke the local procedure.

In the example in the slide, remote procedure is recompiled at 11 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 a.m. as time stamp for the remote procedure B.

Because the time stamp recorded with P code of local procedure A is different from that recorded with 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.

Disadvantage of time stamp mode

A disadvantage of the time stamp mode is that it is unnecessarily restrictive. Recompilation of dependent objects across the network are often performed when not strictly necessary, leading to performance degradation.

Signature Mode

- **The signature of a procedure is:**
 - The name of the procedure
 - The datatypes 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.**

ORACLE

11-21

Copyright © Oracle Corporation, 2001. All rights reserved.

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 datatypes of the parameters
- The modes of the parameters
- The number of parameters
- The datatype 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];  
ALTER PACKAGE [SCHEMA.]package_name COMPILE BODY;
```

```
ALTER TRIGGER trigger_name [COMPILE[DEBUG]];
```

ORACLE

11-22

Copyright © Oracle Corporation, 2001. All rights reserved.

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 objects 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 BODY option recompiles only the package body.

Recompiling a package specification invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. Note that the body of a package also depends on its specification.

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.

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

11-23

Copyright © Oracle Corporation, 2001. All rights reserved.

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 recreated, 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, 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, having the same name and structure, exists**
- **The PL/SQL body of a referenced procedure has been modified and recompiled successfully**

ORACLE

11-24

Copyright © Oracle Corporation, 2001. All rights reserved.

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 by using the `%ROWTYPE` attribute**
- **Declaring variables with the `%TYPE` attribute**
- **Querying with the `SELECT *` notation**
- **Including a column list with `INSERT` statements**

ORACLE

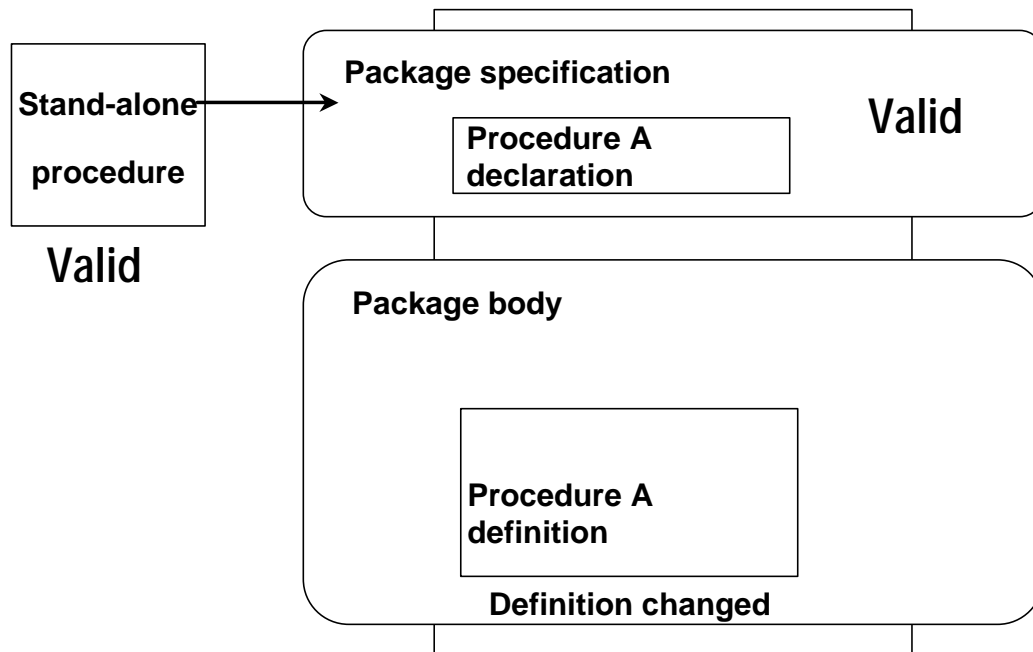
11-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Recompilation of Procedures

You can minimize recompilation failure by following the guidelines that are shown in the slide.

Packages and Dependencies



ORACLE

11-26

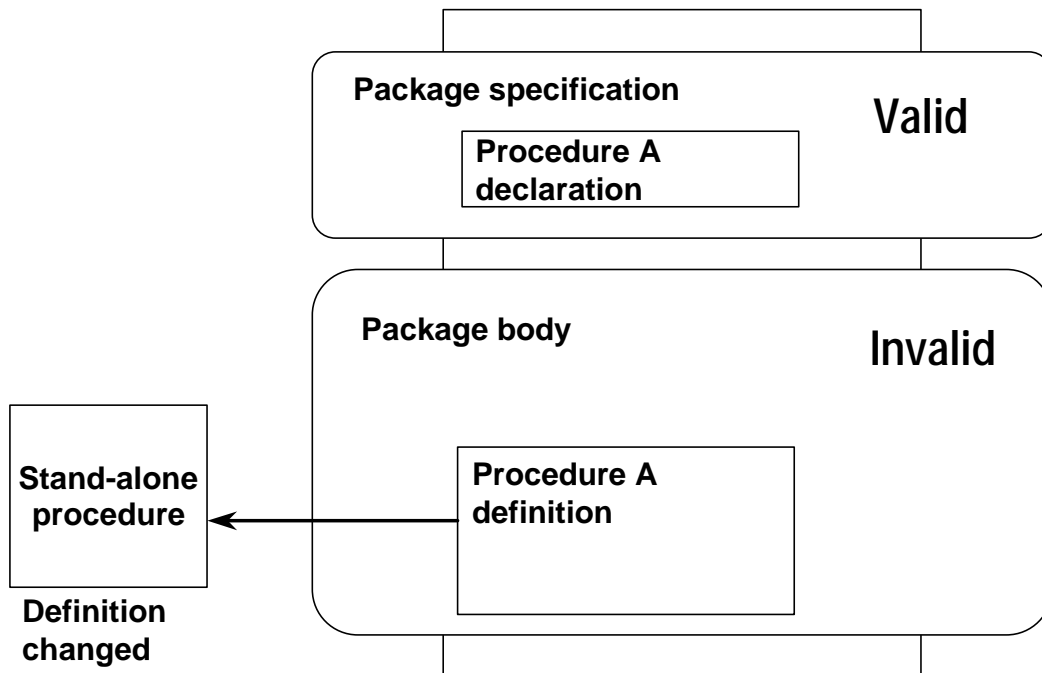
Copyright © Oracle Corporation, 2001. All rights reserved.

Managing Dependencies

You can greatly 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, the stand-alone procedure referencing a package construct remains valid.
- If the package specification changes, the outside procedure referencing a package construct is invalidated, as is the package body.

Packages and Dependencies



11-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Managing Dependencies (continued)

If a stand-alone procedure referenced within the package changes, 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**

ORACLE

11-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson 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 re-execution
Procedure depends on a remote procedure	Yes, but at second re-execution; use manual recompilation for first re-execution, or reinvoke it second time
Procedure depends on a remote object other than a procedure	No

Practice 11 Overview

This practice covers the following topics:

- **Using `DEPTREE_FILL` and `IDEPTREE` to view dependencies**
- **Recompiling procedures, functions, and packages**

ORACLE

11-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 11 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 11

1. Answer the following questions.
 - a. Can a table or a synonym be invalid?
 - b. Assuming the following scenario, is the stand-alone procedure MY_PROC invalidated?
The stand-alone procedure MY_PROC depends on the packaged procedure MY_PROC_PACK.
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.
2. Execute the utltdtree.sql script. This script is available in your lab folder. Print a tree structure showing all dependencies involving your NEW_EMP procedure and your VALID_DEPTID function.

DEPENDENCIES
PROCEDURE PLPU.NEW_EMP

Query the IDEPTREE view to see your results. (NEW_EMP and VALID_DEPTID were created in lesson 3, "Creating Functions." You can run the solution scripts for the practice if you need to create the procedure and function.)

DEPENDENCIES
FUNCTION PLPU.VALID_DEPTID
PROCEDURE PLPU.NEW_EMP

If you have time:

3. Dynamically validate invalid objects.
 - a. Make a copy of your EMPLOYEES table, called EMP_COP.
 - b. Alter your EMPLOYEES table and add the column TOTSAL with data type NUMBER (9 , 2).
 - c. Create a script file to print the name, type, and status of all objects that are invalid.
 - d. Create a procedure called COMPILE_OBJ that recompiles all invalid procedures, functions, packages and views in your schema.
Make use of the ALTER_COMPILE procedure in the DBMS_DDL package.
Execute the COMPILE_OBJ procedure.
 - e. Run the script file that you created in question 3c again and check the status column value.
Do you still have INVALID objects? If you do, why are they INVALID?

A

**PL/SQL
Fundamentals Quiz**

PL/SQL Fundamentals Quiz

The questions are designed as a review. Use the space provided for your answers. If you do not know the answer, go on to the next question.

1. What are the four keywords of the basic PL/SQL block? What happens in each area?
2. What must each executable statement end with?
3. What is the syntax to declare a variable?
4. What is the syntax to declare a constant?
5. What is the syntax to declare a cursor?
6. What is the syntax to declare an exception?
7. How do you link this exception with a known Oracle error?
8. What symbol do you use to assign values to variables?

PL/SQL Fundamentals Quiz (continued)

9. How does the syntax of the SQL `SELECT` statement differ from that of the PL/SQL `SELECT` statement?
10. How many rows can you `SELECT` in PL/SQL?
11. What happens if you select 0 rows?
12. What happens if you select more than 1 row?
13. Where do you handle these errors?
14. How do you handle these errors?
15. How do you give an alternative message to the user?
16. How many rows can you `INSERT`, `UPDATE`, and `DELETE`?
17. How do you test to see how many rows have been manipulated?
18. How do you find out whether some or no rows were changed?

PL/SQL Fundamentals Quiz (continued)

19. How do you cause an error of your own choice?

20. What is the syntax of the IF clause, including an ELSE and ELSE IF?

21. What is the syntax for a BASIC LOOP with an exit?

B

**PL/SQL
Fundamentals Quiz
Answers**

PL/SQL Fundamentals Quiz Answers

1. What are the four keywords of the basic PL/SQL block? What happens in each area?

```
[DECLARE
    Declare local variables
]BEGIN
    Executable statements
[EXCEPTION
    Error handling statements
]END;
```

2. What must each executable statement end with?

; (Semi-colon)

3. What is the syntax to declare a variable?

variable_name DATATYPE [:=] [NOT NULL];

4. What is the syntax to declare a constant?

constant_name CONSTANT DATATYPE := ;

5. What is the syntax to declare a cursor?

CURSOR cursor_name IS SELECT statement ;

6. What is the syntax to declare an exception?

exception_name EXCEPTION;

7. How do you link this exception with a known Oracle error?

PRAGMA EXCEPTION_INIT (exception_name, ERROR_NUMBER);

8. What symbol do you use to assign values to variables?

:=

PL/SQL Fundamentals Quiz Answers (continued)

9. How does the syntax of the SQL SELECT statement differ from that of the PL/SQL SELECT statement?

The PL/SQL SELECT has to have an INTO clause.

10. How many rows can you SELECT in PL/SQL?

1 (one)

11. What happens if you select 0 rows?

ERROR -1403: no data found

12. What happens if you select more than 1 row?

ERROR -1422: exact fetch returns more than the requested number of rows

13. Where do you handle these errors?

In the EXCEPTION section

14. How do you handle these errors?

**WHEN error_name
THEN action(s);**

15. How do you give an alternative message to the user?

**RAISE_APPLICATION_ERROR (error_number, error_text);
error_number must be between -20000 and -20999**

16. How many rows can you INSERT, UPDATE, and DELETE?

0, 1, or many. No restrictions.

17. How do you test to see how many rows have been manipulated?

Use the implicit cursor attribute, SQL%ROWCOUNT.

18. How do you find out whether some or no rows were changed?

Use the implicit cursor attribute, SQL%FOUND or SQL%NOTFOUND.

PL/SQL Fundamentals Quiz Answers (continued)

19. How do you cause an error of your own choice?

```
IF SQL%NOTFOUND
```

```
THEN
```

```
    RAISE_APPLICATION_ERROR (-20022, 'No rows changed.');
```

```
END IF;
```

or

```
RAISE no_data_found;
```

or

```
RAISE my_error; -- where my_error was previously declared
```

20. What is the syntax of the IF clause, including an ELSE and ELSE IF?

```
IF condition(s)
```

```
    THEN action(s);
```

```
ELSIF condition(s)
```

```
    THEN action(s);
```

```
ELSE
```

```
    action(s);
```

```
END IF;
```

21. What is the syntax for a BASIC LOOP with an exit?

```
LOOP
```

```
    action(s);
```

```
    EXIT WHEN condition;
```

```
END LOOP;
```

C

Practice Solutions

Practice 2 Solutions

Note: Save your subprograms as .sql files, using the Save Script button.

Remember to set the SERVEROUTPUT on if you set it off previously.

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.

```
CREATE OR REPLACE PROCEDURE add_job
  (p_jobid IN jobs.job_id%TYPE,
   p_jobtitle IN jobs.job_title%TYPE)
IS
BEGIN
  INSERT INTO jobs (job_id, job_title)
  VALUES (p_jobid, p_jobtitle);
  COMMIT;
END add_job;
```

- b. Compile the code, and invoke the procedure with IT_DBA as job ID and Database Administrator as job title. Query the JOBS table to view the results.

In iSQL*Plus, load and run the script file created in question 1a above.

Procedure created.

```
EXECUTE add_job ('IT_DBA', 'Database Administrator')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';
```

PL/SQL procedure successfully completed.

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?

```
EXECUTE add_job ('ST_MAN', 'Stock Manager')
BEGIN add_job ('ST_MAN', 'Stock Manager'); END;
*
ERROR at line 1:
ORA-00001: unique constraint (PLPU.JOB_ID_PK) violated
ORA-06512: at "PLPU.ADD_JOB", line 6
ORA-06512: at line 1
```

There is a primary key integrity constraint on the JOB_ID column.

Practice 2 Solutions (continued)

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.

```
CREATE OR REPLACE PROCEDURE upd_job
(p_jobid IN jobs.job_id%TYPE,
p_jobtitle IN jobs.job_title%TYPE)
IS
BEGIN
UPDATE jobs
SET    job_title = p_jobtitle
WHERE  job_id = p_jobid;
IF SQL%NOTFOUND THEN
RAISE_APPLICATION_ERROR(-20202,'No job updated.');
```

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

In iSQL*Plus, load and run the script file created in the above question.

Procedure created.

```
EXECUTE upd_job ('IT_DBA', 'Data Administrator')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';
```

PL/SQL procedure successfully completed.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		

```
EXECUTE upd_job ('IT_WEB', 'Web Master')
```

```
BEGIN upd_job ('IT_WEB', 'Web Master'); END;
```

*

ERROR at line 1:

ORA-20202: No job updated.

ORA-06512: at "PLPU.UPD_JOB", line 10

ORA-06512: at line 1

Practice 2 Solutions (continued)

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 from the JOBS table. Include the necessary exception handling if no job is deleted.

```
CREATE OR REPLACE PROCEDURE del_job
(p_jobid IN jobs.job_id%TYPE)
IS
BEGIN
    DELETE FROM jobs
    WHERE job_id = p_jobid;
    IF SQL%NOTFOUND THEN
        RAISE_APPLICATION_ERROR(-20203,'No jobs deleted.');
```

```
    END IF;
```

```
END DEL_JOB;
```

- b. Compile the code; invoke the procedure using job ID IT_DBA. Query the JOBS table to view the results.

In iSQL*Plus, load and run the script file created in the above question.

Procedure created.

```
EXECUTE del_job ('IT_DBA')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';
```

PL/SQL procedure successfully completed.

no rows selected

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

```
EXECUTE del_job ('IT_WEB')

BEGIN del_job ('IT_WEB'); END;
*
ERROR at line 1:
ORA-20203: No jobs deleted.
ORA-06512: at "PLPU.DEL_JOB", line 8
ORA-06512: at line 1
```

Practice 2 Solutions (continued)

4. Create a procedure called QUERY_EMP 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.

Use host variables for the two OUT parameters salary and job ID.

```
CREATE OR REPLACE PROCEDURE query_emp
(p_empid IN  employees.employee_id%TYPE,
 p_sal    OUT employees.salary%TYPE,
 p_job    OUT employees.job_id%TYPE)
IS
BEGIN
    SELECT  salary, job_id
    INTO    p_sal, p_job
    FROM    employees
    WHERE   employee_id = p_empid;
END query_emp;
```

- b. Compile the code, invoke the procedure to display the salary and job ID for employee ID 120.

In iSQL*Plus, load and run the script file created in the above question.

Procedure created.

```
VARIABLE g_sal  NUMBER
VARIABLE g_job  VARCHAR2(15)
EXECUTE query_emp (120, :g_sal, :g_job)
PRINT g_sal
PRINT g_job
```

PL/SQL procedure successfully completed.

G_SAL	
	8000

G_JOB	
ST_MAN	

Practice 2 Solutions (continued)

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

```
EXECUTE query_emp (300, :g_sal, :g_job)
```

```
BEGIN query_emp (300, :g_sal, :g_job); END;  
*  
ERROR at line 1:  
ORA-01403: no data found  
ORA-06512: at "PLPU.QUERY_EMP", line 7  
ORA-06512: at line 1
```

There is no employee in the `EMPLOYEES` table with an `EMPLOYEE_ID` of 300. The `SELECT` statement retrieved no data from the database, resulting in a fatal PL/SQL error, `NO_DATA_FOUND`.

Practice 3 Solutions

1. Create and invoke the Q_JOB function to return a job title.
 - a. Create a function called Q_JOB to return a job title to a host variable.

```
CREATE OR REPLACE FUNCTION q_job
(p_jobid IN jobs.job_id%TYPE)
RETURN VARCHAR2
IS
    v_jobtitle jobs.job_title%TYPE;
BEGIN
    SELECT job_title
    INTO v_jobtitle
    FROM jobs
    WHERE job_id = p_jobid;
    RETURN (v_jobtitle);
END q_job;
```

- b. Compile the code; create a host variable G_TITLE and invoke the function with job ID SA_REP. Query the host variable to view the result.

In iSQL*Plus, load and run the script file created in the above question.

Function created.

```
VARIABLE g_title VARCHAR2(30)
EXECUTE :g_title := q_job ('SA_REP')
PRINT g_title
```

PL/SQL procedure successfully completed.

G_TITLE
Sales Representative

Practice 3 Solutions (continued)

2. Create a function called ANNUAL_COMP to return the annual salary by accepting two parameters: an employee's monthly salary and commission. The function should address NULL values.

- a. Create and invoke the function ANNUAL_COMP, passing in values for monthly salary and commission. Either or both values passed can be NULL, but the function should still return an annual salary, which is not NULL. The annual salary is defined by the basic formula:

$(\text{sal} * 12) + (\text{commission_pct} * \text{salary} * 12)$

```
CREATE OR REPLACE FUNCTION annual_comp
(p_sal IN employees.salary%TYPE,
 p_comm IN employees.commission_pct%TYPE)
RETURN NUMBER
IS
BEGIN
    RETURN (NVL(p_sal,0) * 12 + (NVL(p_comm,0)* p_sal * 12));
END annual_comp;
/
```

- b. Use the function in a SELECT statement against the EMPLOYEES table for department 80.

```
SELECT employee_id, last_name, annual_comp(salary,commission_pct)
       "Annual Compensation"
FROM   employees
WHERE  department_id=80;
```

EMPLOYEE_ID	LAST_NAME	Annual Compensation
145	Russell	235200
146	Partners	210600
147	Errazuriz	187200
148
177	Livingston	120960
179	Johnson	81840

34 rows selected.

Practice 3 Solutions (continued)

3. Create a procedure, NEW_EMP, to insert a new employee into the EMPLOYEES table. The procedure should contain a call to the 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. The function should return a BOOLEAN value.

```
CREATE OR REPLACE FUNCTION valid_deptid
  (p_deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN
IS
  v_dummy  VARCHAR2(1);
BEGIN
  SELECT  'x'
  INTO    v_dummy
  FROM    departments
  WHERE   department_id = p_deptid;
  RETURN (TRUE);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN (FALSE);
END valid_deptid;
/
```

Practice 3 Solutions (continued)

- b. Create the procedure NEW_EMP to add an employee to the EMPLOYEES table. A new row should be added to EMPLOYEES if the function returns TRUE. If the function returns FALSE, the procedure should alert the user with an appropriate message.

Define DEFAULT values for most parameters. The default commission is 0, the default salary is 1000, the default department ID is 30, the default job is SA_REP and the default manager number is 145. For the employee's ID, use the sequence EMPLOYEES_SEQ. Provide the last name, first name and e-mail for the employee.

```
CREATE OR REPLACE PROCEDURE new_emp
(p_lname    employees.last_name%TYPE,
 p_fname    employees.first_name%TYPE,
 p_email     employees.email%TYPE,
 p_job       employees.job_id%TYPE          DEFAULT 'SA_REP',
 p_mgr       employees.manager_id%TYPE      DEFAULT 145,
 p_sal       employees.salary%TYPE          DEFAULT 1000,
 p_comm      employees.commission_pct%TYPE  DEFAULT 0,
 p_deptid    employees.department_id%TYPE   DEFAULT 30)
IS
BEGIN
    IF valid_deptid(p_deptid) THEN
        INSERT INTO employees(employee_id, last_name, first_name,
                               email, job_id, manager_id, hire_date,
                               salary, commission_pct, department_id)
        VALUES (employees_seq.NEXTVAL, p_lname, p_fname, p_email,
                p_job, p_mgr, TRUNC (SYSDATE, 'DD'), p_sal,
                p_comm, p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204,
                                'Invalid department ID. Try again.');
```

END IF;

END new_emp;

/

Practice 3 Solutions (continued)

- c. Test your NEW_EMP procedure by adding a new employee named Jane Harris to department 15. Allow all other parameters to default. What was the result?

```
EXECUTE new_emp(p_lname=>'Harris', p_fname=>'Jane',  
               p_email=>'JAHARRIS', p_deptid => 15)  
  
BEGIN new_emp(p_lname=>'Harris', p_fname=>'Jane', p_email=>'JAHARRIS',  
p_deptid=>15); END;  
*  
ERROR at line 1:  
ORA-20204: Invalid department ID. Try again.  
ORA-06512: at "PLPU.NEW_EMP", line 18  
ORA-06512: at line 1
```

- d. Test your NEW_EMP procedure by adding a new employee named Joe Harris to department 80. Allow all other parameters to default. What was the result?

```
EXECUTE new_emp(p_lname=>'Harris',p_fname=>'Joe',  
               p_email=>'JOHARRIS',p_deptno => 80)  
  
PL/SQL procedure successfully completed.
```

Practice 4 Solutions

Suppose you have lost the code for the `NEW_EMP` procedure and the `VALID_DEPTID` function that you created in lesson 3. (If you did not complete the practices in lesson 3, you can run the solution scripts to create the procedure and function.)

Create a *iSQL*Plus* spool file to query the appropriate data dictionary view to regenerate the code.

Hint:

```
SET                -- options ON|OFF
SELECT            -- statement(s) to extract the code
SET              -- reset options ON|OFF
```

To spool the output of the file to a `.sql` file from *iSQL*Plus*, select the Save option for the Output and execute the code.

```
SET ECHO OFF HEADING OFF FEEDBACK OFF VERIFY OFF
COLUMN  LINE NOPRINT
SET PAGESIZE 0
```

```
SELECT 'CREATE OR REPLACE ', 0 line
FROM   DUAL
UNION
SELECT text, line
FROM   USER_SOURCE
WHERE  name IN ('NEW_EMP', 'VALID_DEPTNO')
ORDER BY line;
```

```
SELECT '/'
FROM DUAL;
```

```
SET PAGESIZE 24
COLUMN  LINE  CLEAR
SET FEEDBACK ON VERIFY ON HEADING ON ECHO ON
```

Practice 5 Solutions

1. Create a package specification and body called JOB_PACK. (You can save the package body and specification in two separate files.) This package contains your ADD_JOB, UPD_JOB, and DEL_JOB procedures, as well as your Q_JOB function.

Note: Use the code in your previously saved script files when creating the package.

- a. Make all the constructs public.

Note: Consider whether you still need the stand-alone procedures and functions you just packaged.

```
CREATE OR REPLACE PACKAGE job_pack IS
  PROCEDURE add_job
    (p_jobid IN jobs.job_id%TYPE,
     p_jobtitle IN jobs.job_title%TYPE);
  PROCEDURE upd_job
    (p_jobid IN jobs.job_id%TYPE,
     p_jobtitle IN jobs.job_title%TYPE);
  PROCEDURE del_job
    (p_jobid IN jobs.job_id%TYPE);
  FUNCTION q_job
    (p_jobid IN jobs.job_id%TYPE)
    RETURN VARCHAR2;
END job_pack;
/
```

Package Created.

Practice 5 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY job_pack IS
  PROCEDURE add_job
    (p_jobid      IN jobs.job_id%TYPE,
     p_jobtitle IN jobs.job_title%TYPE)
  IS
  BEGIN
    INSERT INTO jobs (job_id, job_title)
      VALUES          (p_jobid, p_jobtitle);
  END add_job;
  PROCEDURE upd_job
    (p_jobid      IN jobs.job_id%TYPE,
     p_jobtitle IN jobs.job_title%TYPE)
  IS
  BEGIN
    UPDATE jobs
      SET    job_title = p_jobtitle
    WHERE   job_id = p_jobid;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR(-20202,'No job updated.');
```

```
    END IF;
  END upd_job;
  PROCEDURE del_job
    (p_jobid IN jobs.job_id%TYPE)
  IS
  BEGIN
    DELETE FROM jobs
      WHERE job_id = p_jobid;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR (-20203,'No job deleted.');
```

```
    END IF;
  END del_job;
  FUNCTION q_job
    (p_jobid IN jobs.job_id%TYPE)
    RETURN VARCHAR2
  IS
    v_jobtitle jobs.job_title%TYPE;
  BEGIN
    SELECT  job_title
    INTO    v_jobtitle
    FROM    jobs
    WHERE   job_id = p_jobid;
    RETURN (v_jobtitle);
  END q_job;
END job_pack;
/
```

Package Body Created.

Practice 5 Solutions (continued)

- b. Invoke your ADD_JOB procedure by passing values IT_SYSAN and SYSTEMS ANALYST as parameters.

```
EXECUTE job_pack.add_job('IT_SYSAN', 'Systems Analyst')
```

PL/SQL procedure successfully completed.

- c. Query the JOBS table to see the result.

```
SELECT * FROM jobs
WHERE job_id = 'IT_SYSAN';
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst		

1 row selected.

2. Create and invoke a package that contains private and public constructs.
- a. Create a package specification and package body called EMP_PACK that contains your NEW_EMP procedure as a public construct, and your VALID_DEPTID function as a private construct. (You can save the specification and body into separate files.)

```
CREATE OR REPLACE PACKAGE emp_pack IS
  PROCEDURE new_emp
    (p_lname  employees.last_name%TYPE,
     p_fname  employees.first_name%TYPE,
     p_email  employees.email%TYPE,
     p_job    employees.job_id%TYPE      DEFAULT 'SA_REP',
     p_mgr    employees.manager_id%TYPE  DEFAULT 145,
     p_sal    employees.salary%TYPE      DEFAULT 1000,
     p_comm   employees.commission_pct%TYPE DEFAULT 0,
     p_deptid employees.department_id%TYPE DEFAULT 80);
END emp_pack;
/
```

Package Created.

Practice 5 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY emp_pack IS
  FUNCTION valid_deptid
    (p_deptid IN departments.department_id%TYPE)
    RETURN BOOLEAN
  IS
    v_dummy  VARCHAR2(1);
  BEGIN
    SELECT 'x'
    INTO    v_dummy
    FROM    departments
    WHERE   department_id = p_deptid;
    RETURN (TRUE);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN(FALSE);
  END valid_deptid;
  PROCEDURE new_emp
    (p_lname   employees.last_name%TYPE,
     p_fname   employees.first_name%TYPE,
     p_email    employees.email%TYPE,
     p_job      employees.job_id%TYPE          DEFAULT 'SA_REP',
     p_mgr      employees.manager_id%TYPE      DEFAULT 145,
     p_sal      employees.salary%TYPE          DEFAULT 1000,
     p_comm     employees.commission_pct%TYPE  DEFAULT 0,
     p_deptid   employees.department_id%TYPE   DEFAULT 80)
  IS
  BEGIN
    IF valid_deptid(p_deptid) THEN
      INSERT INTO employees (employee_id, last_name, first_name,
        email, job_id, manager_id, hire_date, salary, commission_pct,
        department_id)
      VALUES (employees_seq.NEXTVAL, p_lname, p_fname, p_email,
        p_job, p_mgr, TRUNC (SYSDATE, 'DD'), p_sal, p_comm,
        p_deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20205,
        'Invalid department number. Try again.');
```

```
    END IF;
  END new_emp;
END emp_pack;
/
```

Package Body Created.

Practice 5 Solutions (continued)

- b. Invoke the NEW_EMP procedure, using 15 as a department number. As the department ID 15 does not exist in the DEPARTMENTS table, you should get an error message as specified in the exception handler of your procedure.

```
EXECUTE emp_pack.new_emp(p_lname=>'Harris',p_fname=>'Jane',  
  p_email=>'JAHARRIS', p_deptid => 15)
```

```
BEGIN emp_pack.new_emp(p_lname=>'Harris',p_fname=>'Jane',  
  p_email=>'JAHARRIS', p_deptid=> 15); END;  
*
```

ERROR at line 1:

ORA-20205: Invalid department number. Try again.

ORA-06512: at "PLPU.EMP_PACK", line 36

ORA-06512: at line 1

- c. Invoke the NEW_EMP procedure, using an existing department ID 80.

```
EXECUTE emp_pack.new_emp(p_lname =>'Smith', p_fname=>'David',  
  p_email=>'DASMITH', p_deptid=>80)
```

PL/SQL procedure successfully completed.

If you have time:

3. a. Create a package called CHK_PACK that contains the procedures CHK_HIREDATE and CHK_DEPT_MGR. Make both constructs public. (You can save the specification and body into separate files.)

The procedure CHK_HIREDATE checks whether an employee's hire date is within the following range: [SYSDATE - 50 years, SYSDATE + 3 months].

Note:

- If the date is invalid, you should raise an application error with an appropriate message indicating why the date value is not acceptable.
- Make sure the time component in the date value is ignored.
- Use a constant to refer to the 50 years boundary.
- A null value for the hire date should be treated as an invalid hire date.

The procedure CHK_DEPT_MGR checks the department and manager combination for a given employee. The CHK_DEPT_MGR procedure accepts an employee ID and a manager

ID. The procedure checks that the manager and employee work in the same department.

The procedure also checks that the job title of the manager ID provided is MANAGER.

Note: If the department ID and manager combination is invalid, you should raise an application error with an appropriate message.

```
CREATE OR REPLACE PACKAGE chk_pack IS  
  PROCEDURE chk_hiredate  
    (p_date in employees.hire_date%type);  
  PROCEDURE chk_dept_mgr  
    (p_empid in employees.employee_id%type,  
     p_mgr in employees.manager_id%type);  
END chk_pack;
```

/

Package Created.

Practice 5 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY chk_pack IS

    PROCEDURE chk_hiredate(p_date in employees.hire_date%TYPE)
    IS
        v_low date := ADD_MONTHS (SYSDATE, - (50 * 12));
        v_high date := ADD_MONTHS (SYSDATE, 3);
    BEGIN
        IF TRUNC(p_date) NOT BETWEEN v_low AND v_high
            OR p_date IS NULL THEN
            RAISE_APPLICATION_ERROR(-20200,'Not a valid hiredate');
        END IF;
    END chk_hiredate;

    PROCEDURE chk_dept_mgr(p_empid in employees.employee_id%TYPE,
                           p_mgr in employees.manager_id%TYPE)
    IS
        v_empnr employees.employee_id%TYPE;
        v_deptid employees.department_id%TYPE;
    BEGIN
        BEGIN
            SELECT department_id
            INTO v_deptid
            FROM employees
            WHERE employee_id = p_empid;
        EXCEPTION
            WHEN NO_DATA_FOUND
            THEN RAISE_APPLICATION_ERROR(-20000, 'Not a valid emp id');
        END;
        BEGIN
            SELECT employee_id
            INTO v_empnr
            FROM employees
            WHERE department_id = v_deptid
            AND employee_id = p_mgr
            AND job_id like '%MAN';
        EXCEPTION
            WHEN NO_DATA_FOUND
            THEN RAISE_APPLICATION_ERROR (-20000,
                'Not a valid manager for this department');
        END;
    END chk_dept_mgr;
END chk_pack;
/
Package Body Created.
```

Practice 5 Solutions (continued)

- b. Test the CHK_HIREDATE procedure with the following command:

```
EXECUTE chk_pack.chk_hiredate('01-JAN-47')
```

What happens, and why?

```
BEGIN chk_pack.chk_hiredate('01-JAN-47'); END;  
*  
ERROR at line 1:  
ORA-20200: Not a valid hiredate  
ORA-06512: at "PLPU.CHK_PACK", line 10  
ORA-06512: at line 1
```

- c. Test the CHK_HIREDATE procedure with the following command:

```
EXECUTE chk_pack.chk_hiredate(NULL)
```

What happens, and why?

```
BEGIN chk_pack.chk_hiredate(NULL); END;  
*  
ERROR at line 1:  
ORA-20200: Not a valid hiredate  
ORA-06512: at "PLPU.CHK_PACK", line 10  
ORA-06512: at line 1
```

- d. Test the CHK_DEPT_MGR procedure with the following command:

```
EXECUTE chk_pack.chk_dept_mgr(117, 100)
```

What happens, and why?

```
BEGIN chk_pack.chk_dept_mgr(117,100); END;  
*  
ERROR at line 1:  
ORA-20000: Not a valid manager for this department  
ORA-06512: at "PLPU.CHK_PACK", line 38  
ORA-06512: at line 1
```

Practice 6 Solutions

1. Create a package called OVER_LOAD. Create two functions in this package, name each function PRINT_IT. The function accepts a date or a character string and prints a date or a number, depending on how the function is invoked.

Note:

- To print the date value, use DD-MON-YY as the input format, and FmMonth,dd yyyy as the output format. Make sure you handle invalid input.
- To print out the number, use 999,999.00 as the input format.

The package specification:

```
CREATE OR REPLACE PACKAGE over_load IS
    FUNCTION print_it(p_arg    IN    DATE)
        RETURN VARCHAR2;
    FUNCTION print_it(p_arg    IN    VARCHAR2)
        RETURN NUMBER;
END over_load;
/
```

Package Created.

The package body:

```
CREATE OR REPLACE PACKAGE BODY over_load
IS
    FUNCTION print_it(p_arg    IN    DATE)
        RETURN VARCHAR2
    IS
    BEGIN
        RETURN to_char(p_arg, 'FmMonth,dd yyyy');
    END print_it;

    FUNCTION print_it(p_arg    IN    VARCHAR2)
        RETURN NUMBER
    IS
    BEGIN
        RETURN TO_NUMBER(p_arg, '999,999.00');
        -- or use the NLS characters for grands and decimals
        -- RETURN TO_NUMBER(p_arg, '999G999D00');
    END print_it;
END over_load;
/
```

Package Body Created.

Practice 6 Solutions (continued)

- a. Test the first version of PRINT_IT with the following set of commands:

```
VARIABLE display_date VARCHAR2(20)
EXECUTE :display_date := over_load.print_it('08-MAR-01')
PRINT display_date
```

PL/SQL procedure successfully completed.

TODAYS_DATE
March,8 2001

- b. Test the second version of PRINT_IT with the following set of commands:

```
VARIABLE g_emp_sal number
EXECUTE :g_emp_sal := over_load.print_it('33,600')
PRINT g_emp_sal
```

PL/SQL procedure successfully completed.

G_EMP_SAL
33600

2. Create a new package, called CHECK_PACK, to implement a new business rule.
- a. Create a procedure called CHK_DEPT_JOB to verify whether a given combination of department ID and job is a valid one. In this case *valid* means that it must be a combination that currently exists in the EMPLOYEES table.

Note:

- Use a PL/SQL table to store the valid department and job combination.
- The PL/SQL table needs to be populated only once.
- Raise an application error with an appropriate message if the combination is not valid.

```
CREATE OR REPLACE PACKAGE check_pack IS
  PROCEDURE chk_dept_job
    (p_deptid IN employees.department_id%TYPE,
     p_job     IN employees.job_id%TYPE);
END check_pack;
/
Package Created.
```

Practice 6 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY check_pack
IS
    i NUMBER := 0;
    CURSOR emp_cur IS
        SELECT department_id, job_id
        FROM employees;
    TYPE emp_table_type IS TABLE OF emp_cur%ROWTYPE
        INDEX BY BINARY_INTEGER;
    deptid_job emp_table_type;

    PROCEDURE chk_dept_job
        (p_deptid IN employees.department_id%TYPE,
         p_job     IN employees.job_id%TYPE)
    IS
    BEGIN
        FOR k IN deptid_job.FIRST .. deptid_job.LAST LOOP
            IF p_deptid = deptid_job(k).department_id
               AND p_job = deptid_job(k).job_id THEN
                RETURN;
            END IF;
        END LOOP;
        RAISE_APPLICATION_ERROR
            (-20500, 'Not a valid job for this dept');
    END chk_dept_job;

    BEGIN -- one-time-only-procedure
        FOR emp_rec IN emp_cur LOOP
            deptid_job(i) := emp_rec;
            i := i + 1;
        END LOOP;
    END check_pack;
/
Package Body Created.
```


Practice 6 Solutions (continued)

- b. Test your CHK_DEPT_JOB package procedure by executing the following command:

```
EXECUTE check_pack.chk_dept_job(50,'ST_CLERK')
```

What happens, and why?

PL/SQL procedure successfully completed.

- c. Test your CHK_DEPT_JOB package procedure by executing the following command:

```
EXECUTE check_pack.chk_dept_job(20,'ST_CLERK')
```

What happens, and why?

```
BEGIN check_pack.chk_dept_job(20,'ST_CLERK'); END;  
*  
ERROR at line 1:  
ORA-20500: Not a valid job for this dept  
ORA-06512: at "PLPU.CHECK_PACK", line 21  
ORA-06512: at line 1
```

Practice 7 Solutions

- 1a. Create a procedure DROP_TABLE that drops the table specified in the input parameter. Use the procedures and functions from the supplied DBMS_SQL package.

```
CREATE OR REPLACE PROCEDURE drop_table
  (p_table_name IN VARCHAR2)
IS
  dyn_cur NUMBER;
  dyn_err VARCHAR2(255);
BEGIN
  dyn_cur := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(dyn_cur, 'DROP TABLE ' ||
                  p_table_name, DBMS_SQL.NATIVE);
  DBMS_SQL.CLOSE_CURSOR(dyn_cur);
EXCEPTION
  WHEN OTHERS THEN dyn_err := SQLERRM;
  DBMS_SQL.CLOSE_CURSOR(dyn_cur);
  RAISE_APPLICATION_ERROR(-20600, dyn_err);
END drop_table;
/
Procedure created.
```

- b. To test the DROP_TABLE procedure, first create a new table called EMP_DUP as a copy of the EMPLOYEES table.

```
CREATE TABLE emp_dup AS
SELECT * FROM employees;
Table created.
```

- c. Execute the DROP_TABLE procedure to drop the EMP_DUP table.

```
EXECUTE drop_table('emp_dup')
SELECT * FROM emp_dup;
```

PL/SQL procedure successfully completed.

```
SELECT * FROM emp_dup
      *
```

ERROR at line 1:

ORA-00942: table or view does not exist

Practice 7 Solutions (continued)

- 2a. Create a procedure called DROP_TABLE2 that drops the table specified in the input parameter. Use the EXECUTE IMMEDIATE statement.

```
CREATE PROCEDURE DROP_TABLE2
  (p_table_name IN VARCHAR2)
IS
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE ' || p_table_name;
END;
/
```

Procedure created.

- b. Repeat the test outlined in steps 1b and 1c.

```
CREATE TABLE emp_dup AS
  SELECT * FROM employees;
```

Table created.

```
EXECUTE drop_table2('emp_dup')
SELECT * FROM emp_dup;
```

PL/SQL procedure successfully completed.

```
SELECT * FROM emp_dup
      *
```

ERROR at line 1:

ORA-00942: table or view does not exist

Practice 7 Solutions (continued)

- 3a. Create a procedure called `ANALYZE_OBJECT` that analyzes the given object that you specified in the input parameters. Use the `DBMS_DDL` package, and use the `COMPUTE` method.

```
CREATE OR REPLACE procedure analyze_object
  (p_obj_type IN  VARCHAR2,
   p_obj_name IN  VARCHAR2)
IS
BEGIN
  DBMS_DDL.ANALYZE_OBJECT(
    p_obj_type,
    USER,
    UPPER(p_obj_name),
    'COMPUTE' );
END;
/
Procedure created.
```

- b. Test the procedure using the table `EMPLOYEES`.

Confirm that the `ANALYZE_OBJECT` procedure has run by querying the `LAST_ANALYZED` column in the `USER_TABLES` data dictionary view.

```
EXECUTE ANALYZE_OBJECT ('TABLE', 'EMPLOYEES')
SELECT LAST_ANALYZED FROM USER_TABLES
WHERE TABLE_NAME = 'EMPLOYEES';
```

PL/SQL procedure successfully completed.

LAST_ANAL
01-MAY-01

1 row selected.

Practice 7 Solutions (continued)

If you have time:

- 4a. Schedule ANALYZE_OBJECT by using DBMS_JOB. Analyze the DEPARTMENTS table, and schedule the job to run in five minutes time from now. (To start the job in five minutes from now, set the parameter NEXT_DATE = 5/(24*60) = 1/288.)

```
VARIABLE jobno NUMBER
```

```
EXECUTE DBMS_JOB.SUBMIT(:jobno,  
    'ANALYZE_OBJECT (''TABLE'', ''DEPARTMENTS'');',  
    SYSDATE + 1/288)  
PRINT jobno
```

PL/SQL procedure successfully completed.

JOBNO
122

- b. Confirm that the job has been scheduled by using USER_JOBS.

```
SELECT JOB, NEXT_DATE, NEXT_SEC, WHAT FROM USER_JOBS;
```

JOB	NEXT_DATE	NEXT_SEC	WHAT
121	09-MAR-01	06:00:00	OVER_PACK.ADD_DEPT('EDUCATION',2710);
122	08-MAR-01	13:31:54	ANALYZE_OBJECT ('TABLE','DEPARTMENTS');

2 rows selected.

Practice 7 Solutions (continued)

5. Create a procedure called CROSS_AVGSAL that generates a text file report of employees who have exceeded the average salary of their department. The partial code is provided for you in the file lab07_5.sql.
 - a. Your program should accept two parameters. The first parameter identifies the output directory. The second parameter identifies the text file name to which your procedure writes.

```
CREATE OR REPLACE PROCEDURE cross_avgsal
(p_filedir IN VARCHAR2, p_filename1 IN VARCHAR2)
IS
v_fh_1 UTL_FILE.FILE_TYPE;
CURSOR cross_avg IS
SELECT last_name, department_id, salary
  FROM employees outer
 WHERE salary > (SELECT AVG(salary)
                  FROM   employees inner
                  GROUP BY outer.department_id)
 ORDER BY department_id;
BEGIN
  v_fh_1 := UTL_FILE.FOPEN(p_filedir, p_filename1, 'w');
  UTL_FILE.PUTF(v_fh_1, 'Employees with more than average salary:\n');
  UTL_FILE.PUTF(v_fh_1, 'REPORT GENERATED ON  %s\n\n', SYSDATE);
  FOR v_emp_info IN cross_avg
  LOOP
    UTL_FILE.PUTF(v_fh_1, '%s    %s \n',
      RPAD(v_emp_info.last_name, 30, ' '),
      LPAD(TO_CHAR(v_emp_info.salary, '$99,999.00'), 12, ' '));
  END LOOP;
  UTL_FILE.NEW_LINE(v_fh_1);
  UTL_FILE.PUT_LINE(v_fh_1, '*** END OF REPORT ***');
  UTL_FILE.FCLOSE(v_fh_1);
END cross_avgsal;
/
```

Practice 7 Solutions (continued)

- b. Your instructor will inform you of the directory location. When you invoke the program, name the second parameter `sal_rptxx.txt` where `xx` stands for your user number, such as 01, 15, and so on.

EXECUTE `cross_avgsal('$HOME/Utl_File', 'sal_rptxx.txt')`

(Replace `$HOME` with the path to the directory `Utl_File` and `xx` with your user number)

- c. Add an exception handling section to handle errors that may be encountered from using the `UTL_FILE` package.

Sample output from this file follows:

```
EMPLOYEES OVER THE AVERAGE SALARY OF THEIR DEPARTMENT:
REPORT GENERATED ON 26-FEB-01
```

Hartstein	20	\$13,000.00
Raphaely	30	\$11,000.00
Marvis	40	\$6,500.00
Weiss	50	\$8,000.00

```
...
*** END OF REPORT ***
```

Note: The solution appears on the next page.

Practice 7 Solutions (continued)

```
CREATE OR REPLACE PROCEDURE cross_avgsal
(p_filedir IN VARCHAR2, p_filename1 IN VARCHAR2)
IS
  v_fh_1 UTL_FILE.FILE_TYPE;
  CURSOR cross_avg IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                     FROM employees inner
                     GROUP BY outer.department_id)
    ORDER BY department_id;
BEGIN
  v_fh_1 := UTL_FILE.FOPEN(p_filedir, p_filename1, 'w');
  UTL_FILE.PUTF(v_fh_1, 'Employees with more than average salary:\n');
  UTL_FILE.PUTF(v_fh_1, 'REPORT GENERATED ON  %s\n\n', SYSDATE);
  FOR v_emp_info IN cross_avg
  LOOP
    UTL_FILE.PUTF(v_fh_1, '%s  %s \n',
      RPAD(v_emp_info.last_name, 30, ' '),
      LPAD(TO_CHAR(v_emp_info.salary, '$99,999.00'), 12, ' '));
  END LOOP;
  UTL_FILE.NEW_LINE(v_fh_1);
  UTL_FILE.PUT_LINE(v_fh_1, '*** END OF REPORT ***');
  UTL_FILE.FCLOSE(v_fh_1);

EXCEPTION
  WHEN UTL_FILE.INVALID_FILEHANDLE THEN
    RAISE_APPLICATION_ERROR (-20001, 'Invalid File. ');
    UTL_FILE.FCLOSE_ALL;
  WHEN UTL_FILE.WRITE_ERROR THEN
    RAISE_APPLICATION_ERROR (-20002,
      'Unable to write to file');
    UTL_FILE.FCLOSE_ALL;
END cross_avgsal;
/
```


Practice 8 Solutions

1. Create a table called PERSONNEL by executing the script file lab08_1.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

```
CREATE TABLE personnel
(id NUMBER(6) constraint personnel_id_pk PRIMARY KEY,
last_name VARCHAR2(35),
review CLOB,
picture BLOB);
```

2. Insert two rows into the PERSONNEL table, one each for employees 2034 and 2035. Use the empty function for the CLOB, and provide NULL as the value for the BLOB.

```
INSERT INTO personnel
VALUES(2034, 'Allen', EMPTY_CLOB(), NULL);
```

```
INSERT INTO personnel
VALUES(2035, 'Bond', EMPTY_CLOB(), NULL);
```

3. Execute the script lab08_3.sql. 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.

```
CREATE TABLE review_table
(employee_id number,
ann_review VARCHAR2(2000));
```

```
INSERT INTO review_table
VALUES(2034,'Very good performance this year. Recommended to
increase salary by $500');
```

```
INSERT INTO review_table
VALUES(2035,'Excellent performance this year. Recommended to
increase salary by $1000');
COMMIT;
```

Practice 8 Solutions (continued)

4. Update the PERSONNEL table.

a. Populate the CLOB for the first row, using the following query in a SQL UPDATE statement:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2034;

UPDATE personnel
SET review = (SELECT ann_review
              FROM   review_table
              WHERE  employee_id = 2034)
WHERE last_name = 'Allen';
```

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:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2035;

DECLARE
  lobloc CLOB;
  text VARCHAR2(2000);
  amount NUMBER ;
  offset INTEGER;
BEGIN
  SELECT ann_review INTO text
  FROM review_table
  WHERE employee_id =2035;
  SELECT review INTO lobloc
  FROM personnel
  WHERE last_name = 'Bond' FOR UPDATE;
  offset := 1;
  amount := length(text);
  DBMS_LOB.WRITE ( lobloc, amount, offset, text );
END;
```

Practice 8 Solutions (continued)

If you have time...

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. The image files are named after the country IDs. You need to load an image file locator into all rows in Europe region (REGION_ID = 1) in the COUNTRIES table. The DIRECTORY object name that stores a pointer to the location of the binary files is called COUNTRY_PIC. This object is already created for you.

- a. Use the command below to add the image column to the COUNTRIES table.

```
ALTER TABLE countries ADD (picture BFILE);
```

- b. Create a PL/SQL procedure called load_country_image that reads a locator into your picture column. Have the program test to see if the file exists, using the function DBMS_LOB.FILEEXISTS. If the file is not existing, your procedure should display a message that the file can not be opened. Have your program report information about the load to the screen.

Note: The solution appears on the next page.

- c. Invoke the procedure by passing the name of the directory object COUNTRY_PIC as the parameter. Note that you should pass the directory object in single quotation marks.

```
EXECUTE load_country_image('COUNTRY_PIC')
```

Practice 8 Solutions (continued)

```
CREATE OR REPLACE PROCEDURE load_country_image
    (p_file_loc IN VARCHAR2)
IS
    v_file          BFILE;
    v_filename       VARCHAR2(40);
    v_record_number  NUMBER;
    v_file_exists    BOOLEAN;
    CURSOR country_pic_cursor IS
        SELECT country_id
        FROM countries
        WHERE region_id = 1
        FOR UPDATE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('LOADING LOCATORS TO IMAGES...');
    FOR country_record IN country_pic_cursor
    LOOP
        v_filename := country_record.country_id || '.tif';
        v_file := bfilename(p_file_loc, v_filename);
        v_file_exists := (DBMS_LOB.FILEEXISTS(v_file) = 1);
        IF v_file_exists THEN
            DBMS_LOB.FILEOPEN(v_file);
            UPDATE countries
            SET picture = bfilename(p_file_loc, v_filename)
            WHERE CURRENT OF country_pic_cursor;
            DBMS_OUTPUT.PUT_LINE('LOADED LOCATOR TO FILE: ' || v_filename
                                || ' SIZE: ' || DBMS_LOB.GETLENGTH(v_file));
            DBMS_LOB.FILECLOSE(v_file);
            v_record_number := country_pic_cursor%ROWCOUNT;
        ELSE
            DBMS_OUTPUT.PUT_LINE('Can not open the file ' || v_filename);
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('TOTAL FILES UPDATED: ' || v_record_number);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_LOB.FILECLOSE(v_file);
        DBMS_OUTPUT.PUT_LINE('Program Error Occurred: '
                            || to_char(SQLCODE) || SQLERRM);
END load_country_image;
/
```

Practice 9 Solutions

1. Changes to data are allowed on tables only during normal office hours of 8:45 a.m. until 5:30 p.m., Monday through Friday.

Create a stored procedure called SECURE_DML that prevents the DML statement from executing outside of normal office hours, returning the message, "You may make changes only during normal office hours."

```
CREATE OR REPLACE PROCEDURE secure_dml
IS
BEGIN
    IF TO_CHAR (SYSDATE, 'HH24:MI') NOT BETWEEN '08:45' AND '17:30'
        OR TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN')
    THEN RAISE_APPLICATION_ERROR (-20205,
        'You may make changes only during normal office hours');
    END IF;
END secure_dml;
```

2. a. Create a statement trigger on the JOBS table that calls the above procedure.

```
CREATE OR REPLACE TRIGGER secure_prod
BEFORE INSERT OR UPDATE OR DELETE ON jobs
BEGIN
    secure_dml;
END secure_prod;
```

b. Test the procedure by temporarily modifying the hours in the procedure and attempting to insert a new record into the JOBS table. After testing, reset the procedure hours as specified in step 1.

```
INSERT INTO jobs (job_id, job_title)
VALUES ('HR_MAN', 'Human Resources Manager');
```

```
INSERT INTO jobs (job_id, job_title)
*
ERROR at line 1:
ORA-20205: You may only make changes during normal office hours
ORA-06512: at "PLPU.SECURE_DML", line 6
ORA-06512: at "PLPU.SECURE_PROD", line 2
ORA-04088: error during execution of trigger 'PLPU.SECURE_PROD'
```

Practice 9 Solutions (continued)

3. Employees should receive an automatic increase in salary if the minimum salary for a job is increased. Implement this requirement through a trigger on the JOBS table.

- a. Create a stored procedure named UPD_EMP_SAL to update the salary amount. This procedure accepts two parameters: the job ID for which salary has to be updated, and the new minimum salary for this job ID. This procedure is executed from the trigger on the JOBS table.

```
CREATE OR REPLACE PROCEDURE upd_emp_sal
(p_jobid  IN employees.job_id%TYPE,
 p_minsal IN employees.salary%TYPE)
IS
BEGIN
    UPDATE employees
    SET salary = p_minsal
    WHERE job_id = p_jobid
    AND SALARY < p_minsal;
END upd_emp_sal;
/
```

- b. Create a row trigger named UPDATE_EMP_SALARY on the JOBS table that invokes the procedure UPD_EMP_SAL when the minimum salary in the JOBS table is updated for a specified job ID.

```
CREATE OR REPLACE TRIGGER update_emp_salary
AFTER UPDATE OF min_salary ON jobs
FOR EACH ROW
BEGIN
    upd_emp_sal(:NEW.job_id, :NEW.min_salary);
END;
/
```

- c. Query the EMPLOYEES table to see the current salary for employees who are programmers.

```
SELECT last_name, first_name, salary
FROM employees
WHERE job_id = 'IT_PROG';
```

LAST_NAME	FIRST_NAME	SALARY
Hunold	Alexander	9000
Ernst	Bruce	6000
Austin	David	4800
Pataballa	Valli	4800
Lorentz	Diana	4200

Practice 9 Solutions (continued)

- d. Increase the minimum salary for the programmer job from 4,000 to 5,000.

```
UPDATE jobs
SET min_salary = 5000
WHERE job_id = 'IT_PROG';
```

- e. Employee Lorentz (employee ID 107) had a salary of less than 4,500. Verify that her salary has been increased to the new minimum of 5,000.

```
SELECT last_name, first_name, salary
FROM employees
WHERE employee_id = 107;
```

LAST_NAME	FIRST_NAME	SALARY
Lorentz	Diana	5000

Practice 10 Solutions

1. A number of business rules that apply to the EMPLOYEES and DEPARTMENTS tables are listed below.

Decide how to implement each of these business rules, by means of declarative constraints or by using triggers.

Which constraints or triggers are needed and are there any problems to be expected?

Implement the business rules by defining the triggers or constraints that you decided to create.

A partial package is provided in file lab10_1.sql to which you should add any necessary procedures or functions that are to be called from triggers that you may create for the following rules.

(The triggers should execute procedures or functions that you have defined in the package.)

The following code is from the lab10_1.sql file:

```
REM    Package specification with sample procedure specifications
CREATE OR REPLACE PACKAGE mgr_constraints_pkg
IS
    PROCEDURE check_president;
    PROCEDURE check_mgr;
    PROCEDURE new_location(p_deptid IN departments.department_id%TYPE);
    new_mgr employees.manager_id%TYPE := NULL;
END mgr_constraints_pkg;
/

REM    Package Body - fill in the code for the procedures
CREATE OR REPLACE PACKAGE BODY mgr_constraints_pkg
IS
    PROCEDURE check_president IS

    END check_president;
    PROCEDURE check_mgr IS

    END check_mgr;
    PROCEDURE new_location(p_deptid IN departments.department_id%TYPE)
    IS

    END new_location;

END mgr_constraints_pkg;
/
```


Practice 10 Solutions (continued)

The following code is the complete solution for the package specification.

```
CREATE OR REPLACE PACKAGE mgr_constraints_pkg
IS
    PROCEDURE check_president;
    PROCEDURE check_mgr;
    PROCEDURE new_location
        (p_deptid IN departments.department_id%TYPE);
    new_mgr employees.manager_id%TYPE := NULL;
END mgr_constraints_pkg;
```

Practice 10 Solutions (continued)

The following code is the solution for the package body.

```
CREATE OR REPLACE PACKAGE BODY mgr_constraints_pkg
IS
PROCEDURE check_president
IS
    v_dummy CHAR(1);
BEGIN
    SELECT 'x'
    INTO   v_dummy
    FROM   employees
    WHERE  job_id = 'AD_PRES';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        NULL;
    WHEN TOO_MANY_ROWS THEN
        RAISE_APPLICATION_ERROR(-20001,'President title
        already exists');
END check_president;
PROCEDURE check_mgr
IS
    count_emps NUMBER := 0;
BEGIN
    IF new_mgr IS NOT NULL
    THEN
        -- count the number of people
        -- working for the mgr
        SELECT count(*)
        INTO   count_emps
        FROM   employees
        WHERE  manager_id = new_mgr;
    END IF;
    -- if there are now more than 15,
    -- raise an error
    IF count_emps > 15
    THEN RAISE_APPLICATION_ERROR (-20202,
        'Max number of emps exceeded for ' || TO_CHAR(new_mgr));
    END IF;
END check_mgr;
```

Practice 10 Solutions (continued)

```
PROCEDURE new_location
    (p_deptid  IN  departments.department_id%TYPE)
IS
    v_sal  employees.salary%TYPE;
BEGIN
    UPDATE employees
        SET salary = salary*1.02
        WHERE department_id = p_deptid;
END new_location;
END mgr_constraints_pkg;
/
```

Practice 10 Solutions (continued)

Business Rules

Rule 1. Sales managers and sales representatives should always receive commission. Employees who are not sales managers or sales representatives should never receive a commission. Ensure that this restriction does not validate the existing records of the EMPLOYEES table. It should be effective only for the subsequent inserts and updates on the table.

Implement rule 1 with a constraint.

```
ALTER TABLE employees
ADD CONSTRAINT emp_comm_chk
CHECK ((job_id = 'SA_REP' and commission_pct>0) OR
      (job_id = 'SA_MAN' and commission_pct>0) OR
      (job_id != 'SA_REP' and commission_pct=0))
NOVALIDATE;
```

Rule 2. The EMPLOYEES table should contain exactly one president.

Test your answer by inserting an employee record with the following details: employee ID 400, last name Harris, first name Alice, e-mail ID AHARRIS, job ID AD_PRES, hire date SYSDATE, salary 20000, and department ID 20.

Note: You do not need to implement a rule for case sensitivity; instead, you need to test for the number of people with the job title of President.

Implement rule 2 with a trigger.

```
CREATE OR REPLACE TRIGGER check_pres_title
AFTER INSERT OR UPDATE OF job_id ON employees
BEGIN
    mgr_constraints_pkg.check_president;
END check_pres_title;

Trigger created.

INSERT INTO employees
      (employee_id, last_name, first_name, email, job_id,
       hire_date, salary, department_id)
VALUES (400,'Harris','Alice', 'AHARRIS', 'AD_PRES',
        SYSDATE, 20000, 20);
```

```
INSERT INTO employees
```

```
*
```

```
ERROR at line 1:
```

```
ORA-20001: President title already exists
```

```
ORA-06512: at "PLPU.MGR_CONSTRAINTS_PKG", line 15
```

```
ORA-06512: at "PLPU.CHECK_PRE_TITLE", line 2
```

```
ORA-04088: error during execution of trigger 'PLPU.CHECK_PRE_TITLE'
```

Practice 10 Solutions (continued)

Rule 3. An employee should never be a manager of more than 15 employees.

Test your answer by inserting the following records into the EMPLOYEES table (perform a query to count the number of employees currently working for manager 100 before inserting these rows):

- i. Employee ID 401, last name Johnson, first name Brian, e-mail ID BJOHNSON, job ID SA_MAN, hire date SYSDATE , salary 11000, manager ID 100, and department ID 80.
(This insertion should be successful, because there are only 14 employees working for manager 100 so far.)
- ii. Employee ID 402, last name Kellogg, first name Tony, e-mail ID TKELLOG, job ID ST_MAN, hire date SYSDATE , salary 7500, manager ID 100, and department ID 50.
(This insertion should be unsuccessful, because there are already 15 employees working for manager 100.)

Implement rule 3 with a trigger.

```
CREATE OR REPLACE TRIGGER set_mgr
AFTER INSERT or UPDATE of manager_id on employees
FOR EACH ROW
BEGIN
    -- To get round MUTATING TABLE ERROR
    mgr_constraints_pkg.new_mgr := :NEW.manager_id;
END set_mgr;
```

```
CREATE OR REPLACE TRIGGER chk_emps
AFTER INSERT or UPDATE of manager_id on employees
BEGIN
    mgr_constraints_pkg.check_mgr;
    -- if for some reason, SET_MGR is disabled,
    -- the global variable is set to null
    -- to stop the SELECT COUNT running
    mgr_constraints_pkg.new_mgr := NULL;
END chk_emps;
```

```
INSERT INTO employees
(employee_id, last_name, first_name, email, job_id,
hire_date, salary, manager_id, department_id)
VALUES (401,'Johnson','Brian', 'BJOHNSON', 'SA_MAN',
        SYSDATE, 11000, 100, 80);
1 row created.
```

Practice 10 Solutions (continued)

```
SELECT count(*)
FROM employees
WHERE manager_id = 100;
```

COUNT(*)
15

```
INSERT INTO employees
(employee_id, last_name, first_name, email, job_id,
hire_date, salary, manager_id, department_id)
VALUES (402, 'Kellogg', 'Tony', 'TKELLOGG', 'ST_MAN',
        SYSDATE, 7500, 100, 50);
```

```
INSERT INTO employees
```

*

ERROR at line 1:

ORA-20202: Max number of emps exceeded for 100

ORA-06512: at "HR.MGR_CONSTRAINTS_PKG", line 34

ORA-06512: at "HR.CHK_EMPS", line 2

ORA-04088: error during execution of trigger 'HR.CHK_EMPS'

Practice 10 Solutions (continued)

Rule 4. Salaries can only be increased, never decreased.

The present salary of employee 105 is 5000. Test your answer by decreasing the salary of employee 105 to 4500.

Implement rule 4 with a trigger.

```
CREATE OR REPLACE TRIGGER check_sal
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR(-20002,'Salary may not be reduced');
END check_sal;
```

Trigger Created.

```
UPDATE employees
SET    salary = 4500
WHERE  employee_id = 105;
```

```
UPDATE employees
*
```

ERROR at line 1:

ORA-20002: Salary may not be reduced

ORA-06512: at "HR.CHECK_SAL", line 2

ORA-04088: error during execution of trigger 'HR.CHECK_SAL'

Practice 10 Solutions (continued)

Rule 5. If a department moves to another location, each employee of that department automatically receives a salary raise of 2 percent.

View the current salaries of employees in department 90.

Test your answer by moving department 90 to location 1600. Query the new salaries of employees of department 90.

Implement rule 5 with a trigger.

```
CREATE OR REPLACE TRIGGER change_location
BEFORE UPDATE OF location_id ON departments
FOR EACH ROW
BEGIN
    mgr_constraints_pkg.new_location(:OLD.department_id);
END change_location;
Trigger created.
```

```
SELECT last_name, salary, department_id
FROM    employees
WHERE   department_id = 90;
```

LAST_NAME	SALARY	DEPARTMENT_ID
King	24000	90
Kochhar	17000	90
De Haan	17000	90

```
UPDATE departments
SET    location_id = 1600
WHERE  department_id = 90;
1 row updated.
SELECT last_name, salary, department_id
FROM    employees
WHERE   department_id = 90;
```

LAST_NAME	SALARY	DEPARTMENT_ID
King	24480	90
Kochhar	17340	90
De Haan	17340	90

Practice 11 Solutions

1. Answer the following questions.

a. Can a table or a synonym be invalid?

A table or a synonym can never be invalidated; however, dependent objects can be invalidated.

b. Assuming the following scenario, is the stand-alone procedure MY_PROC invalidated?

- The stand-alone procedure MY_PROC depends on the packaged procedure MY_PROC_PACK.
- 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.

Although the package body is recompiled, the stand-alone procedure MY_PROC that depends on the packaged procedure MY_PROC_PACK is not invalidated because the package specification is not altered

2. Execute the utldtree.sql script. Print a tree structure showing all dependencies involving your NEW_EMP procedure and your VALID_DEPTID function. Query the ideptree view to see your results. (NEW_EMP and VALID_DEPTID were created in lesson 3, "Creating Functions." You can run the solution scripts for the practice if you need to create the procedure and function.)

Replace 'your USERNAME' with your username in the following statements.

```
EXECUTE deptree_fill('PROCEDURE', 'your USERNAME', 'NEW_EMP')
```

PL/SQL procedure successfully completed.

```
SELECT * FROM ideptree;
```

DEPENDENCIES
PROCEDURE PLPU.NEW_EMP

```
EXECUTE deptree_fill('FUNCTION', 'your USERNAME',  
                    'VALID_DEPTID')
```

PL/SQL procedure successfully completed.

```
SELECT * FROM ideptree;
```

DEPENDENCIES
FUNCTION PLPU.VALID_DEPTID
PROCEDURE PLPU.NEW_EMP

Practice 11 Solutions (continued)

If you have time:

3. Dynamically validate invalid objects.

- a. Make a copy of your EMPLOYEES table, called EMP_COP.

```
CREATE TABLE emp_cop AS  
SELECT * FROM employees;
```

- b. Alter your EMPLOYEES table and add the column TOTSAL with data type NUMBER (9 , 2).

```
ALTER TABLE employees  
ADD (totsal NUMBER(9,2));
```

- c. Create a script file to print the name, type, and status of all objects that are invalid.

This is the code that your script file should contain:

```
SELECT object_name, object_type, status  
FROM user_objects  
WHERE status = 'INVALID';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
ANNUAL_COMP	FUNCTION	INVALID
AUDIT_EMP_TAB	TRIGGER	INVALID
AUDIT_EMP_TRIG	TRIGGER	INVALID
AUDIT_EMP_VALUES	TRIGGER	INVALID
CHECK_PACK	PACKAGE	INVALID
CHECK_PACK	PACKAGE BODY	INVALID
CHECK PRES_TITLE	TRIGGER	INVALID
CHECK_SALARY	TRIGGER	INVALID
CHK_EMPS	TRIGGER	INVALID
CHK_PACK	PACKAGE	INVALID
SAL_STATUS	PROCEDURE	INVALID
SECURE_EMP	TRIGGER	INVALID
SECURE_EMPLOYEES	TRIGGER	INVALID
SET_MGR	TRIGGER	INVALID
UPDATE_EMP_SALARY	TRIGGER	INVALID
UPDATE_JOB_HISTORY	TRIGGER	INVALID
OBJECT_NAME	OBJECT_TYPE	STATUS
UPD_EMP_SAL	PROCEDURE	INVALID

Practice 11 Solutions (continued)

- d. Create a procedure called `COMPILE_OBJ` that recompiles all invalid procedures, functions, packages and views in your schema.

Make use of the `ALTER_COMPILE` procedure in the `DBMS_DDL` package.

```
CREATE OR REPLACE PROCEDURE compile_obj
IS
CURSOR obj_cur IS
    SELECT object_type, object_name
    FROM user_objects
    WHERE status = 'INVALID'
    AND object_type IN ('PROCEDURE', 'FUNCTION', 'PACKAGE',
                        'PACKAGE BODY', 'VIEW')
    ORDER BY object_type;
BEGIN
    FOR obj_rec IN obj_cur LOOP
        DBMS_DDL.ALTER_COMPILE(obj_rec.object_type, user,
                               obj_rec.object_name);
    END LOOP;
END compile_obj;
/
```

Execute the `COMPILE_OBJ` procedure.

```
EXECUTE compile_obj
```

- e. Run the script file that you created in question 3c again and check the status column value.

Do you still have `INVALID` objects? If you do, why are they `INVALID`?

```
SELECT object_name, object_type, status
FROM user_objects
WHERE status = 'INVALID';
```

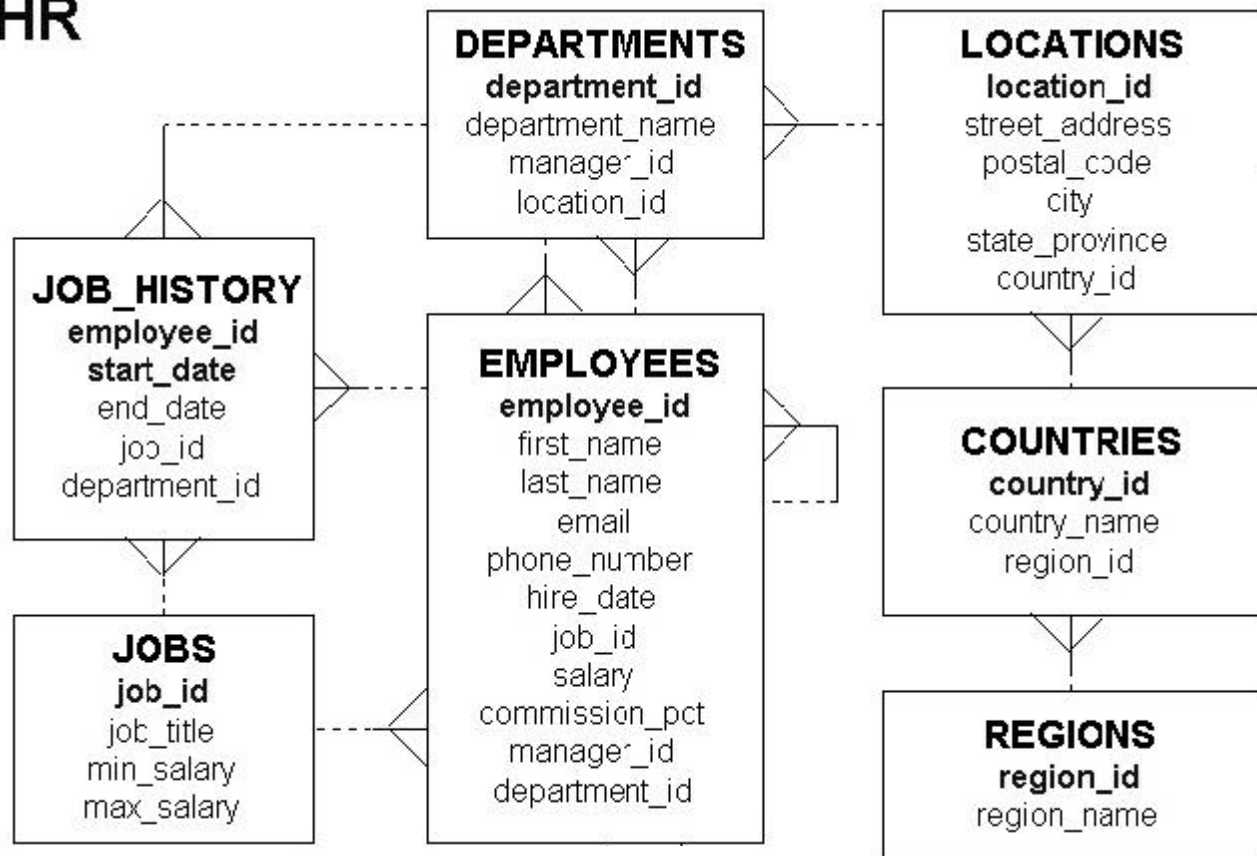
You may still have invalid objects because the procedure does not take into account object dependencies.

D

Table Descriptions and Data

ENTITY RELATIONSHIP DIAGRAM

HR



Tables in the Schema

```
SELECT * FROM tab;
```

TNAME	TABTYPE	CLUSTERID
COUNTRIES	TABLE	
DEPARTMENTS	TABLE	
EMPLOYEES	TABLE	
EMP_DETAILS_VIEW	VIEW	
JOBS	TABLE	
JOB_HISTORY	TABLE	
LOCATIONS	TABLE	
REGIONS	TABLE	

8 rows selected.

REGIONS Table

```
DESCRIBE regions
```

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

```
SELECT * FROM regions;
```

REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

COUNTRIES Table

DESCRIBE countries

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

SELECT * FROM countries;

CO	COUNTRY_NAME	REGION_ID
AR	Argentina	2
AU	Australia	3
BE	Belgium	1
BR	Brazil	2
CA	Canada	2
CH	Switzerland	1
CN	China	3
DE	Germany	1
DK	Denmark	1
EG	Egypt	4
FR	France	1
HK	HongKong	3
IL	Israel	4
IN	India	3
CO	COUNTRY_NAME	REGION_ID
IT	Italy	1
JP	Japan	3
KW	Kuwait	4
MX	Mexico	2
NG	Nigeria	4
NL	Netherlands	1
SG	Singapore	3
UK	United Kingdom	1
US	United States of America	2
ZM	Zambia	4
ZW	Zimbabwe	4

25 rows selected.

LOCATIONS Table

```
DESCRIBE locations;
```

Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

```
SELECT * FROM locations;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1000	1297 Via Cola di Rie	00989	Roma		IT
1100	93091 Calle della Testa	10934	Venice		IT
1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
1300	9450 Kamiya-cho	6823	Hiroshima		JP
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
2000	40-5-12 Laogianggen	190518	Beijing		CN
2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
2200	12-98 Victoria Street	2901	Sydney	New South Wales	AU
2300	198 Clementi North	540198	Singapore		SG
LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
2400	8204 Arthur St		London		UK
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
3000	Murtenstrasse 921	3095	Postfach	Berne	CH
3100	Pieter Breughelstraat 837	3029	Utrecht	SK	NL
3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

23 rows selected.

DEPARTMENTS Table

DESCRIBE departments

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

SELECT * FROM departments;

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700
200	Operations		1700
210	IT Support		1700
220	NOC		1700
230	IT Helpdesk		1700
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700

27 rows selected.

JOBS Table

```
DESCRIBE jobs
```

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

```
SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

19 rows selected.

EMPLOYEES Table

DESCRIBE employees

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

EMPLOYEES Table

The headings for columns COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID are set to COMM, MGRID, and DEPTID in the following screenshot, to fit the table values across the page.

```
SELECT * FROM employees;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000			90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000		100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000		100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000		102	60
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000		103	60
105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800		103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800		103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200		103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000		101	100
109	Daniel	Faviet	DFAVET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000		108	100
110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200		108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700		108	100
112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800		108	100
113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900		108	100
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000		100	30
115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100		114	30
116	Shelli	Baida	SBaida	515.127.4563	24-DEC-97	PU_CLERK	2900		114	30
117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800		114	30
118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600		114	30
119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500		114	30
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000		100	50
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200		100	50
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900		100	50
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500		100	50
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800		100	50
125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200		120	50
126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700		120	50
127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400		120	50

EMPLOYEES Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200		120	50
129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300		121	50
130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800		121	50
131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500		121	50
132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100		121	50
133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300		122	50
134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900		122	50
135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400		122	50
136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200		122	50
137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600		123	50
138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200		123	50
139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700		123	50
140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500		123	50
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
142	Curtis	Davies	CDAVES	650.121.2994	29-JAN-97	ST_CLERK	3100		124	50
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600		124	50
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500		124	50
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	SA_MAN	14000	.40	100	80
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	SA_MAN	13500	.30	100	80
147	Alberto	Erazuriz	AERAZUR	011.44.1344.429278	10-MAR-97	SA_MAN	12000	.30	100	80
148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-99	SA_MAN	11000	.30	100	80
149	Beni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	SA_MAN	10500	.20	100	80
150	Peter	Tucker	PTUCKER	011.44.1344.129268	30-JAN-97	SA_REP	10000	.30	145	80
151	David	Bernstein	DBERNSTE	011.44.1344.345268	24-MAR-97	SA_REP	9500	.25	145	80
152	Peter	Hall	PHALL	011.44.1344.478968	20-AUG-97	SA_REP	9000	.25	145	80
153	Christopher	Olsen	COLSEN	011.44.1344.498718	30-MAR-98	SA_REP	8000	.20	145	80
154	Nanette	Cambrault	NCAMBRAU	011.44.1344.987668	09-DEC-98	SA_REP	7500	.20	145	80
155	Oliver	Tuvault	OTUVAULT	011.44.1344.486508	23-NOV-99	SA_REP	7000	.15	145	80
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
156	Janette	King	JKING	011.44.1345.429268	30-JAN-96	SA_REP	10000	.35	146	80
157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-96	SA_REP	9500	.35	146	80
158	Allan	McEwen	AMCEWEN	011.44.1345.829268	01-AUG-96	SA_REP	9000	.35	146	80
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	.30	146	80
160	Louise	Doran	LDORAN	011.44.1345.629268	15-DEC-97	SA_REP	7500	.30	146	80
161	Sarath	Sewall	SSEWALL	011.44.1345.529268	03-NOV-98	SA_REP	7000	.25	146	80
162	Clara	Vishney	CVISHNEY	011.44.1346.129268	11-NOV-97	SA_REP	10500	.25	147	80
163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-99	SA_REP	9500	.15	147	80
164	Mattea	Marvins	MMARVINS	011.44.1346.329268	24-JAN-00	SA_REP	7200	.10	147	80
165	David	Lee	DLEE	011.44.1346.529268	23-FEB-00	SA_REP	6800	.10	147	80
166	Sundar	Ande	SANDE	011.44.1346.629268	24-MAR-00	SA_REP	6400	.10	147	80
167	Amit	Banda	ABANDA	011.44.1346.729268	21-APR-00	SA_REP	6200	.10	147	80
168	Lisa	Ozer	LOZER	011.44.1343.929268	11-MAR-97	SA_REP	11500	.25	148	80
169	Harrison	Bloom	HBLOOM	011.44.1343.829268	23-MAR-98	SA_REP	10000	.20	148	80

EMPLOYEES Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
170	Taylor	Fox	TFOX	011.44.1343.729268	24-JAN-98	SA_REP	9600	.20	148	80
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	.15	148	80
172	Elizabeth	Bates	EBATES	011.44.1343.529268	24-MAR-99	SA_REP	7300	.15	148	80
173	Sundita	Kumar	SKUMAR	011.44.1343.329268	21-APR-00	SA_REP	6100	.10	148	80
174	Elen	Abel	EABEL	011.44.1644.429267	11-MAY-96	SA_REP	11000	.30	149	80
175	Alyssa	Hutton	AHUTTON	011.44.1644.429266	19-MAR-97	SA_REP	8800	.25	149	80
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98	SA_REP	8600	.20	149	80
177	Jack	Livingston	JLIVINGS	011.44.1644.429264	23-APR-98	SA_REP	8400	.20	149	80
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	.15	149	
179	Charles	Johnson	CJOHNSON	011.44.1644.429262	04-JAN-00	SA_REP	6200	.10	149	80
180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200		120	50
181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100		120	50
182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500		120	50
183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800		120	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200		121	50
185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100		121	50
186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400		121	50
187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000		121	50
188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800		122	50
189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600		122	50
190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900		122	50
191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500		122	50
192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000		123	50
193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900		123	50
194	Samuel	Mc Cain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200		123	50
195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800		123	50
196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100		124	50
197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMM	MGRID	DEPTID
198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-99	SH_CLERK	2600		124	50
199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	SH_CLERK	2600		124	50
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400		101	10
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000		100	20
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000		201	20
203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-94	HR_REP	6500		101	40
204	Hermann	Baer	HBAER	515.123.8888	07-JUN-94	PR_REP	10000		101	70
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000		101	110
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300		205	110

107 rows selected.

JOB_HISTORY Table

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	DEPTID
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

Review of PL/SQL

ORACLE[®]

Copyright © Oracle Corporation, 2001. All rights reserved.

Block Structure for Anonymous PL/SQL Blocks

DECLARE (optional)
 Declare PL/SQL objects to be used
 within this block

BEGIN (mandatory)
 Define the executable statements

EXCEPTION (optional)
 Define the actions that take place if
 an error or exception arises

END ; (mandatory)

ORACLE



E-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Anonymous Blocks

Anonymous blocks do not have names. You declare them at the point in an application where they are to be run, and they are passed to the PL/SQL engine for execution at run time.

- The section between the keywords **DECLARE** and **BEGIN** is referred to as the declaration section. In the declaration section, you define the PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block. The **DECLARE** keyword is optional if you do not declare any PL/SQL objects.
- The **BEGIN** and **END** keywords are mandatory and enclose the body of actions to be performed. This section is referred to as the executable section of the block.
- The section between **EXCEPTION** and **END** is referred to as the exception section. The exception section traps error conditions. In it, you define actions to take if the specified condition arises. The exception section is optional.

The keywords **DECLARE**, **BEGIN**, and **EXCEPTION** are not followed by semicolons, but **END** and all other PL/SQL statements do require semicolons.

Declaring PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
    [:= | DEFAULT expr];
```

Examples:

```
Declare
    v_hiredate      DATE;
    v_deptno        NUMBER(2) NOT NULL := 10;
    v_location      VARCHAR2(13) := 'Atlanta';
    c_comm          CONSTANT NUMBER := 1400;
    v_count         BINARY_INTEGER := 0;
    v_valid         BOOLEAN NOT NULL := TRUE;
```

ORACLE

E-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Declaring PL/SQL Variables

You need to declare all PL/SQL identifiers within the declaration section before referencing them within the PL/SQL block. You have the option to assign an initial value. You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must be sure to declare them separately in a previous statement.

In the syntax,

<i>identifier</i>	is the name of the variable.
CONSTANT	constrains the variable so that its value cannot change; constants must be initialized.
<i>datatype</i>	is a scalar, composite, reference, or LOB data type (this course covers only scalar and composite data types).
NOT NULL	constrains the variable so that it must contain a value; NOT NULL variables must be initialized.
<i>expr</i>	is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions.

Declaring Variables with the %TYPE Attribute

Examples:

```
...  
  v_ename          employees.last_name%TYPE;  
  v_balance        NUMBER(7,2);  
  v_min_balance    v_balance%TYPE := 10;  
...
```

ORACLE

E-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

```
...  
v_ename          employees.last_name%TYPE;  
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10.

```
...  
v_balance        NUMBER(7,2);  
v_min_balance    v_balance%TYPE := 10;  
...
```

A NOT NULL column constraint does not apply to variables declared using %TYPE. Therefore, if you declare a variable using the %TYPE attribute and a database column defined as NOT NULL, you can assign the NULL value to the variable.

Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

Example:

```
...  
  TYPE emp_record_type IS RECORD  
    (ename   VARCHAR2(25),  
     job     VARCHAR2(10),  
     sal     NUMBER(8,2));  
  emp_record emp_record_type;  
...
```

ORACLE

E-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a PL/SQL Record

Field declarations are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

The following example shows that you can use the %TYPE attribute to specify a field data type:

```
DECLARE  
  TYPE emp_record_type IS RECORD  
    (empid  NUMBER(6) NOT NULL := 100,  
     ename  employees.last_name%TYPE,  
     job    employees.job_id%TYPE);  
  emp_record emp_record_type;  
...
```

Note: You can add the NOT NULL constraint to any field declaration and so prevent the assigning of nulls to that field. Remember, fields declared as NOT NULL must be initialized.

The %ROWTYPE Attribute

Examples:

Declare a variable to store the same information about a department as it is stored in the DEPARTMENTS table.

```
dept_record    departments%ROWTYPE;
```

Declare a variable to store the same information about an employee as it is stored in the EMPLOYEES table.

```
emp_record     employees%ROWTYPE;
```

ORACLE

E-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Examples

The first declaration in the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID.

The second declaration above creates a record with the same field names and field data types as a row in the EMPLOYEES table. The fields are EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID.

In the following example, you select column values into a record named item_record.

```
DECLARE
    job_record  jobs%ROWTYPE;
    ...
BEGIN
    SELECT * INTO job_record
    FROM    jobs
    WHERE   ...
```


Creating a PL/SQL Table

```
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  ename_table      ename_table_type;
  hiredate_table   hiredate_table_type;
BEGIN
  ename_table(1) := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
  IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
    ...
END;
```

ORACLE

E-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a PL/SQL Table

There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore you must create the data type first and then declare an identifier using that data type.

Referencing a PL/SQL table

Syntax

```
pl/sql_table_name(primary_key_value)
```

where: primary_key_value belongs to type BINARY_INTEGER.

Reference the third row in a PL/SQL table ENAME_TABLE.

```
ename_table(3) ...
```

The magnitude range of a BINARY_INTEGER is -2147483647 ... 2147483647, so the primary key value can be negative. Indexing need not start with 1.

Note: The `table.EXISTS(i)` statement returns TRUE if at least one row with index `i` is returned. Use the EXISTS statement to prevent an error that is raised in reference to a nonexisting table element.

SELECT Statements in PL/SQL

INTO clause is required.

Example:

```
DECLARE
  v_deptid  NUMBER(4);
  v_loc     NUMBER(4);
BEGIN
  SELECT  department_id, location_id
  INTO    v_deptno, v_loc
  FROM    departments
  WHERE   department_name = 'Sales';
  ...
END;
```

ORACLE

E-8

Copyright © Oracle Corporation, 2001. All rights reserved.

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables to hold the values that SQL returns from the SELECT clause. You must give one variable for each item selected, and their order must correspond to the items selected.

You use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return One and Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of Embedded SQL, for which the following rule applies: queries must return one and only one row. More than one row or no row generates an error.

PL/SQL deals with these errors by raising standard exceptions, which you can trap in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions (exception handling is covered in a subsequent lesson). You should code SELECT statements to return a single row.

Inserting Data

Add new employee information to the EMPLOYEES table.

Example:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
BEGIN
  SELECT  employees_seq.NEXTVAL
  INTO    v_empno
  FROM    dual;
  INSERT INTO  employees(employee_id, last_name,
                        job_id, department_id)
  VALUES(v_empno, 'HARDING', 'PU_CLERK', 30);
END;
```

ORACLE

E-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Inserting Data

- Use SQL functions, such as USER and SYSDATE.
- Generate primary key values by using database sequences.
- Derive values in the PL/SQL block.
- Add column default values.

Note: There is no possibility for ambiguity with identifiers and column names in the INSERT statement. Any identifier in the INSERT clause must be a database column name.

Updating Data

Increase the salary of all employees in the `EMPLOYEES` table who are purchasing clerks.

Example:

```
DECLARE
    v_sal_increase    employees.salary%TYPE := 2000;
BEGIN
    UPDATE employees
    SET      salary = salary + v_sal_increase
    WHERE    job_id = 'PU_CLERK';
END;
```

ORACLE

Updating and Deleting Data

There may be ambiguity in the `SET` clause of the `UPDATE` statement because although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable.

Remember that the `WHERE` clause is used to determine which rows are affected. If no rows are modified, no error occurs, unlike the `SELECT` statement in PL/SQL.

Note: PL/SQL variable assignments always use `:=` and SQL column assignments always use `=`. Recall that if column names and identifier names are identical in the `WHERE` clause, the Oracle server looks to the database first for the name.

Deleting Data

Delete rows that belong to department 190 from the EMPLOYEES table.

Example:

```
DECLARE
  v_deptid    employees.department_id%TYPE := 190;
BEGIN
  DELETE FROM employees
  WHERE department_id = v_deptid;
END;
```

ORACLE

E-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Deleting Data

Delete a specific job.

```
DECLARE
  v_jobid      jobs.job_id%TYPE := 'PR_REP';
BEGIN
  DELETE FROM jobs
  WHERE job_id = v_jobid;
END;
```

COMMIT and ROLLBACK Statements

- **Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK.**
- **Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.**

ORACLE

E-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling Transactions

You control the logic of transactions with COMMIT and ROLLBACK SQL statements, rendering some groups of database changes permanent while discarding others. As with the Oracle server, DML transactions start at the first command to follow a COMMIT or ROLLBACK and end on the next successful COMMIT or ROLLBACK. These actions may occur within a PL/SQL block or as a result of events in the host environment (for example, ending a SQL*Plus session automatically commits the pending transaction).

COMMIT Statement

COMMIT ends the current transaction by making all pending changes to the database permanent.

Syntax

```
COMMIT [WORK];
```

```
ROLLBACK [WORK];
```

where: WORK is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as LOCK TABLE and SELECT . . . FOR UPDATE) in a block (a subsequent lesson will cover more information on the FOR_UPDATE command). They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.

SQL Cursor Attributes

Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value)
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed

ORACLE

E-13

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL Cursor Attributes

SQL cursor attributes enable you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the attributes `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN` in the exception section of a block to gather information about the execution of a data manipulation statement. PL/SQL does not consider a DML statement that affects no rows to have failed, unlike the `SELECT` statement, which returns an exception.

IF-THEN-ELSIF Statements

For a given value entered, return a calculated value.

Example:

```
. . .  
IF v_start > 100 THEN  
    v_start := 2 * v_start;  
ELSIF v_start >= 50 THEN  
    v_start := 0.5 * v_start;  
ELSE  
    v_start := 0.1 * v_start;  
END IF;  
. . .
```

ORACLE

E-14

Copyright © Oracle Corporation, 2001. All rights reserved.

IF-THEN-ELSIF Statements

When possible, use the ELSIF clause instead of nesting IF statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the ELSE clause consists purely of another IF statement, it is more convenient to use the ELSIF clause. This makes the code clearer by removing the need for nested END IFs at the end of each further set of conditions and actions.

Example

```
IF condition1 THEN  
    statement1;  
ELSIF condition2 THEN  
    statement2;  
ELSIF condition3 THEN  
    statement3;  
END IF;
```

The example IF-THEN-ELSIF statement above is further defined as follows:

For a given value entered, return a calculated value. If the entered value is over 100, then the calculated value is two times the entered value. If the entered value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.

Basic Loop

Example:

```
DECLARE
  v_ordid    order_items.order_id%TYPE := 101;
  v_counter  NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

ORACLE

E-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Basic Loop

The basic loop example shown in the slide is defined as follows: Insert the first 10 new line items for order number 101.

Note: A basic loop allows execution of its statements at least once, even if the condition has been met upon entering the loop.

FOR Loop

Insert the first 10 new line items for order number 101.

Example:

```
DECLARE
  v_ordid      order_items.order_id%TYPE := 101;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO order_items(order_id,line_item_id)
      VALUES(v_ordid, i);
  END LOOP;
END;
```

ORACLE

WHILE Loop

Example:

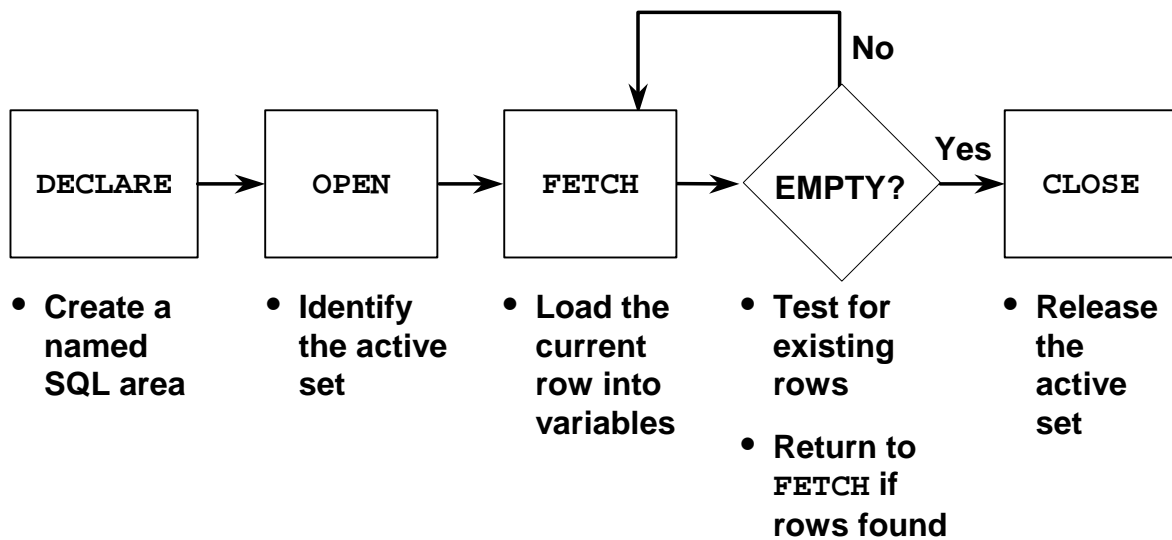
```
ACCEPT p_price PROMPT 'Enter the price of the item: '  
ACCEPT p_itemtot -  
  PROMPT 'Enter the maximum total for purchase of item: '  
DECLARE  
...  
v_qty          NUMBER(8) := 1;  
v_running_total NUMBER(7,2) := 0;  
BEGIN  
  ...  
  WHILE v_running_total < &p_itemtot LOOP  
    ...  
    v_qty := v_qty + 1;  
    v_running_total := v_qty * &p_price;  
  END LOOP;  
  ...
```

ORACLE

WHILE Loop

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

Controlling Explicit Cursors



ORACLE

E-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Cursors

Controlling Explicit Cursors, Using Four Commands

1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The **OPEN** statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor. The **FETCH** statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore each fetch accesses a different row returned by the query. In the flow diagram shown in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise, it closes the cursor.
4. Close the cursor. The **CLOSE** statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Declaring the Cursor

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR c2 IS
    SELECT *
    FROM   departments
    WHERE  department_id = 10;
BEGIN
  ...
```

ORACLE

E-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Cursor Declaration

Retrieve the employees one by one.

```
DECLARE
  v_empid  employees.employee_id%TYPE;
  v_ename  employees.last_name%TYPE;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  ...
```

Note: You can reference variables in the query, but you must declare them before the CURSOR statement.

Opening the Cursor

Syntax:

```
OPEN cursor_name;
```

- **Open the cursor to execute the query and identify the active set.**
- **If the query returns no rows, no exception is raised.**
- **Use cursor attributes to test the outcome after a fetch.**

ORACLE

E-20

Copyright © Oracle Corporation, 2001. All rights reserved.

OPEN Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax,

`cursor_name` is the name of the previously declared cursor.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information.
2. Parses the SELECT statement.
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses.
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
5. Positions the pointer just before the first row in the active set.

Note: If the query returns no rows when the cursor is opened, PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared using the FOR UPDATE clause, the OPEN statement also locks those rows.

Fetching Data from the Cursor

Examples:

```
FETCH c1 INTO v_empid, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
    FETCH defined_cursor INTO defined_variables
    EXIT WHEN ...;
    ...
    -- Process the retrieved data
    ...
END;
```

ORACLE

E-21

Copyright © Oracle Corporation, 2001. All rights reserved.

FETCH Statement (continued)

You use the `FETCH` statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the `INTO` list. Also, their data types must be compatible.

Retrieve the first 10 employees one by one.

```
DECLARE
    v_empid employees.employee_id%TYPE;
    v_ename employees.last_name%TYPE;
    i        NUMBER := 1;
    CURSOR c1 IS
        SELECT employee_id, last_name
        FROM   employees;
BEGIN
    OPEN c1;
    FOR i IN 1..10 LOOP
        FETCH c1 INTO v_empid, v_ename;
        ...
    END LOOP;
END ;
```

Closing the Cursor

Syntax:

```
CLOSE      cursor_name;
```

- Close the cursor after completing the processing of the rows.
- Reopen the cursor, if required.
- Do not attempt to fetch data from a cursor once it has been closed.

ORACLE

E-22

Copyright © Oracle Corporation, 2001. All rights reserved.

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore you can establish an active set several times.

In the syntax,

cursor_name is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor once it has been closed, or the INVALID_CURSOR exception will be raised.

Note: The CLOSE statement releases the context area. Although it is possible to terminate the PL/SQL block without closing cursors, you should get into the habit of closing any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the OPEN_CURSORS parameter in the database parameter field. OPEN_CURSORS = 50 by default.

```
...
FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename;
    ...
END LOOP;
CLOSE c1;
END;
```


Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	BOOLEAN	Evaluates to TRUE if the cursor is open
%NOTFOUND	BOOLEAN	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	BOOLEAN	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	NUMBER	Evaluates to the total number of rows returned so far

ORACLE

E-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a data manipulation statement.

Note: Do not reference cursor attributes directly in a SQL statement.

Cursor FOR Loops

Retrieve employees one by one until there are no more left.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
    -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
    END LOOP; -- implicit close occurs
  END;
```

ORACLE

Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. The cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched.

The FOR UPDATE Clause

Retrieve the orders for amounts over \$1000 that were processed today.

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
          AND order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```

ORACLE

E-25

Copyright © Oracle Corporation, 2001. All rights reserved.

The FOR UPDATE Clause

If the Oracle8 Server cannot acquire the locks on the rows it needs in a `SELECT FOR UPDATE`, it waits indefinitely. You can use the `NOWAIT` clause in the `SELECT FOR UPDATE` statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore you can retry opening the cursor *n* times before terminating the PL/SQL block.

If you intend to update or delete rows using the `WHERE CURRENT OF` clause, you must specify a column name in the `FOR UPDATE OF` clause.

If you have a large table, you can achieve better performance by using the `LOCK TABLE` statement to lock all rows in the table. However, when using `LOCK TABLE`, you cannot use the `WHERE CURRENT OF` clause and must use the notation `WHERE column = identifier`.

The WHERE CURRENT OF Clause

Example:

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
    ...
  END LOOP;
  COMMIT;
END;
```

ORACLE

The WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your DELETE or UPDATE statement to contain the WHERE CURRENT OF cursor_name clause to refer to the latest row processed by the FETCH statement. When you use this clause, the cursor you reference must exist and must contain the FOR UPDATE clause in the cursor query; otherwise you will obtain an error. This clause enables you to apply updates and deletes to the currently addressed row without the need to explicitly reference the ROWID pseudocolumn.

Trapping Predefined Oracle Server Errors

- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX

ORACLE

E-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Trapping Predefined Oracle server Errors

Trap a predefined Oracle server error by referencing its standard name within the corresponding exception-handling routine.

Note: PL/SQL declares predefined exceptions in the `STANDARD` package.

It is a good idea to always consider the `NO_DATA_FOUND` and `TOO_MANY_ROWS` exceptions, which are the most common.

Predefined Exception

Syntax:

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

ORACLE

E-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Trapping Predefined Oracle server Exceptions

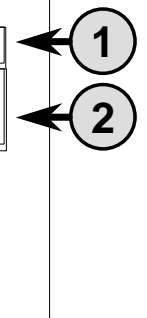
In the slide example for each exception, a message is printed out to the user.

Only one exception is raised and handled at any time.

Nonpredefined Error

Trap for Oracle server error number -2292, an integrity constraint violation

```
DECLARE
  e_products_invalid EXCEPTION;
  PRAGMA EXCEPTION_INIT (
    e_products_invalid, -2292);
  v_message VARCHAR2(50);
BEGIN
  . . .
EXCEPTION
  WHEN e_products_invalid THEN
    :g_message := 'Product ID
                  specified is not valid.';
  . . .
END;
```



ORACLE

E-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Trapping a Nonpredefined Oracle server Exception

1. Declare the name for the exception within the declarative section.

Syntax

```
exception          EXCEPTION;
```

where: *exception* is the name of the exception.

2. Associate the declared exception with the standard Oracle server error number, using the PRAGMA EXCEPTION_INIT statement.

Syntax

```
PRAGMA EXCEPTION_INIT(exception, error_number);
```

where: *exception* is the previously declared exception.

error_number is a standard Oracle server error number.

3. Reference the declared exception within the corresponding exception-handling routine.
In the slide example: If there is product in stock, halt processing and print a message to the user.

User-Defined Exception

Example:

```
[DECLARE]
  e_amount_remaining EXCEPTION; ← 1
. . .
BEGIN
. . .
  RAISE e_amount_remaining; ← 2
. . .
EXCEPTION
  WHEN e_amount_remaining THEN ← 3
    :g_message := 'There is still an amount
                  in stock.';
. . .
END;
```

ORACLE

E-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Trapping User-Defined Exceptions

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.

Syntax

exception EXCEPTION;

where: *exception* is the name of the exception.

2. Use the RAISE statement to raise the exception explicitly within the executable section.

Syntax

RAISE *exception*;

where: *exception* is the previously declared exception.

3. Reference the declared exception within the corresponding exception handling routine.

In the slide example: This customer has a business rule that states that a product can not be removed from its database if there is any inventory left in stock for this product. Because there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a DELETE on the PRODUCT_INFORMATION table, the block queries the INVENTORIES table to see if there is any stock for the product in question. If so, raise an exception.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception back to the calling environment.

RAISE_APPLICATION_ERROR

Syntax:

```
raise_application_error (error_number,  
                        message[, {TRUE | FALSE}]);
```

- A procedure that lets you issue user-defined error messages from stored subprograms.
- Called only from an executing stored subprogram.

ORACLE

E-31

Copyright © Oracle Corporation, 2001. All rights reserved.

RAISE_APPLICATION_ERROR

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR you can report errors to your application and avoid returning unhandled exceptions.

In the syntax,

error_number is a user specified number for the exception between -20000 and -20999.

message is the user-specified message for the exception. It is a character string up to 2048 bytes long.

TRUE | FALSE is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example

```
...  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    RAISE_APPLICATION_ERROR (-20201,  
      'Manager is not a valid employee.');
```

END;

RAISE_APPLICATION_ERROR

- **Used in two different places:**
 - Executable section
 - Exception section
- **Returns error conditions to the user in a manner consistent with other Oracle server errors**

ORACLE

E-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

```
...  
DELETE FROM employees  
WHERE manager_id = v_mgr;  
IF SQL%NOTFOUND THEN  
    RAISE_APPLICATION_ERROR(-20202, 'This is not a valid manager');  
END IF;  
...
```



Creating Program Units by Using Procedure Builder

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Objectives

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

- **Describe the features of Oracle Procedure Builder**
- **Manage program units using the Object Navigator**
- **Create and compile program units using the Program Unit Editor**
- **Invoke program units using the PL/SQL Interpreter**
- **Debug subprograms using the debugger**
- **Control execution of an interrupted PL/SQL program unit**
- **Test possible solutions at run time**

ORACLE

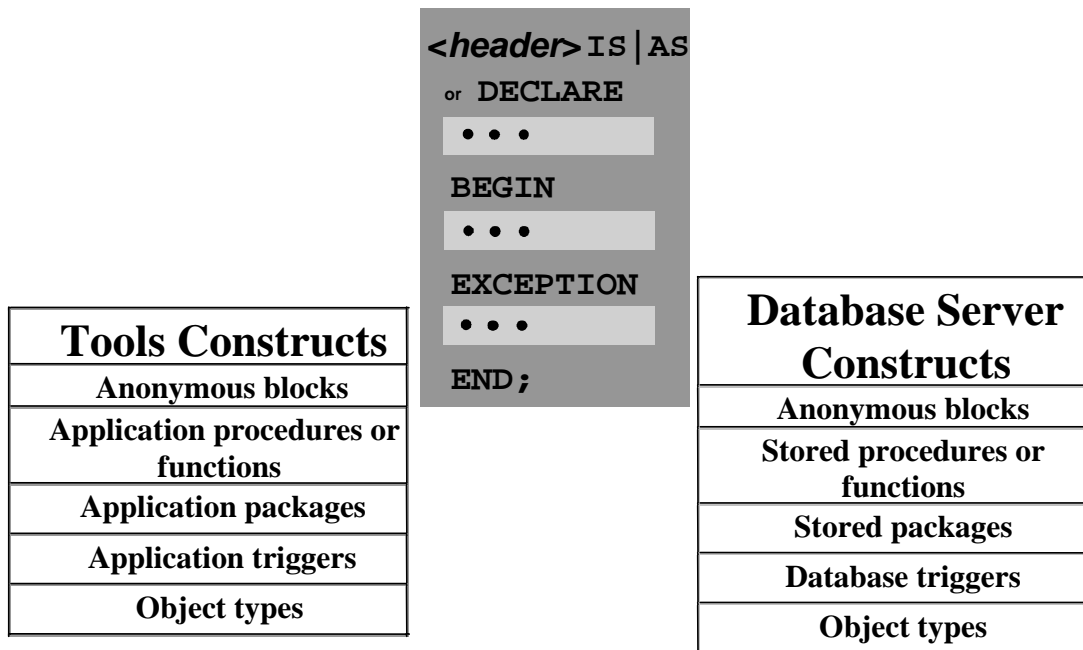
F-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

You can use different development environments to create PL/SQL program units. In this appendix you learn to use Oracle Procedure Builder as one of the development environments to create and debug different types of program units. You also learn about the features of the Procedure Builder tool and how they can be used to create, compile, and invoke subprograms.

PL/SQL Program Constructs



PL/SQL Program Constructs

The diagram above displays a variety of different PL/SQL program constructs using the basic PL/SQL block. In general, a block is either an anonymous block or a named block (subprogram or program unit).

PL/SQL Block Structure

Every PL/SQL construct is composed of one or more blocks. These blocks can be entirely separate or nested within one another. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Note: In the slide, the word “or” prior to the keyword DECLARE is not part of the syntax. It is used in the diagram to differentiate between starting subprograms and anonymous blocks.

The PL/SQL blocks can be constructed on and use the Oracle server (stored PL/SQL program units). They can also be constructed using the Oracle Developer tools such as Oracle Forms Developer, Oracle Report Developer, and so on (application or client-side PL/SQL program units).

Object types are user-defined composite data types that encapsulate a data structure along with the functions and procedures needed to manipulate the data. You can create object types either on the Oracle server or using the Oracle Developer tools.

You can create both application program units and stored program units using Oracle Procedure Builder. Application program units are used in graphical user environment tools such as Oracle Forms. Stored program units are stored on the database server and can be shared by multiple applications.

Development Environments

- **iSQL*Plus uses the PL/SQL engine in the Oracle Server**
- **Oracle Procedure Builder uses the PL/SQL engine in the client tool or in the Oracle Server. It includes:**
 - **A GUI development environment for PL/SQL code**
 - **Built-in editors**
 - **The ability to compile, test, and debug code**
 - **Application partitioning that allows drag-and-drop of program units between client and server**

ORACLE

F-4

Copyright © Oracle Corporation, 2001. All rights reserved.

iSQL*Plus and Oracle Procedure Builder

PL/SQL is not an Oracle product in its own right. It is a technology employed by the Oracle Server and by certain Oracle development tools. Blocks of PL/SQL are passed to, and processed by, a PL/SQL engine. That engine may reside within the tool or within the Oracle Server.

There are two main development environments for PL/SQL: *iSQL*Plus* and Oracle Procedure Builder. This course covers creating program units using *iSQL*Plus*.

About Procedure Builder

Oracle Procedure Builder is a tool you can use to create, execute, and debug PL/SQL programs used in your application tools, such as a form or report, or on the Oracle server through its graphical interface.

Integrated PL/SQL Development Environment

Procedure Builder's development environment contains a build-in editor for you to create or edit subprograms. You can compile, test, and debug your code.

Unified Client-Server PL/SQL Development

Application partitioning through Procedure Builder is available to assist you with distribution of logic between client and server. Users can drag and drop a PL/SQL program unit between the client and the server.

Developing Procedures and Functions Using *iSQL*Plus*

The screenshot shows the iSQL*Plus web interface. At the top, there is a section labeled 'Enter statements: Script location:' with a text input field and two buttons: 'Browse...' and 'Load Script'. Below this is a large text area containing the following SQL script:

```
CREATE OR REPLACE PROCEDURE log_execution
IS
BEGIN
  INSERT INTO log_table (user_id, log_date)
  VALUES (USER, SYSDATE);
END log_execution;
```

At the bottom of the text area is a small '1' icon. Below the text area is a row of buttons: 'Execute', 'Output', 'Display' (with a dropdown arrow), 'Clear Screen', and 'Save Script'.

ORACLE

F-5

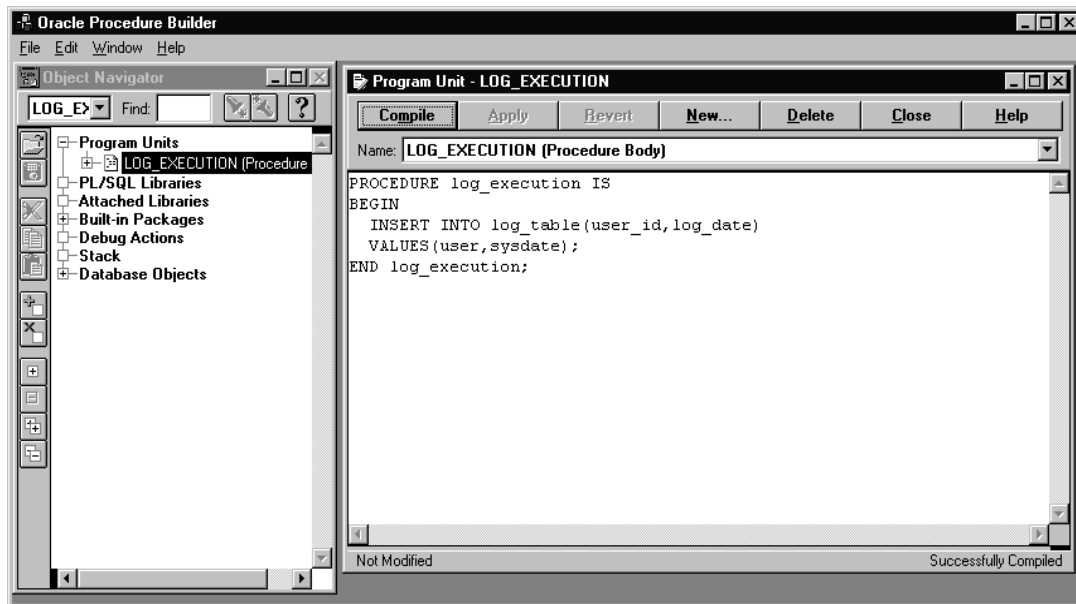
Copyright © Oracle Corporation, 2001. All rights reserved.

Using *iSQL*Plus*

Use a text editor to create a script to define your procedure or function. Browse and upload the script into the *iSQL*Plus* input window. Execute the script by clicking the EXECUTE button.

The example in the slide creates a stored procedure without any parameters. The procedure records the username and current date in a database table.

Developing Procedures and Functions Using Oracle Procedure Builder



F-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Start Procedure Builder from Windows

Procedure Builder contains object navigator where you can see all the program units that you created. You can open, edit, compile, debug, and save the program units by using a graphical editor.

Components of Procedure Builder

Component	Function
Object Navigator	Manages PL/SQL constructs; performs debug actions
PL/SQL Interpreter	Debugs PL/SQL code; evaluates PL/SQL code in real time
Program Unit Editor	Creates and edits PL/SQL source code
Stored Program Unit Editor	Creates and edits server-side PL/SQL source code
Database Trigger Editor	Creates and edits database triggers

ORACLE

F-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Components of Procedure Builder

Procedure Builder is an integrated development environment. It enables you to edit, compile, test, and debug client-side and server-side PL/SQL program units within a single tool.

The Object Navigator

The Object Navigator provides an outline-style interface to browse objects, view the relationships between them, and edit their properties.

The Interpreter Pane

The Interpreter pane is the central debugging workspace of the Oracle Procedure Builder. It is a window with two regions where you display, debug, and run PL/SQL program units. It also interactively supports the evaluation of PL/SQL constructs, SQL commands, and Procedure Builder commands.

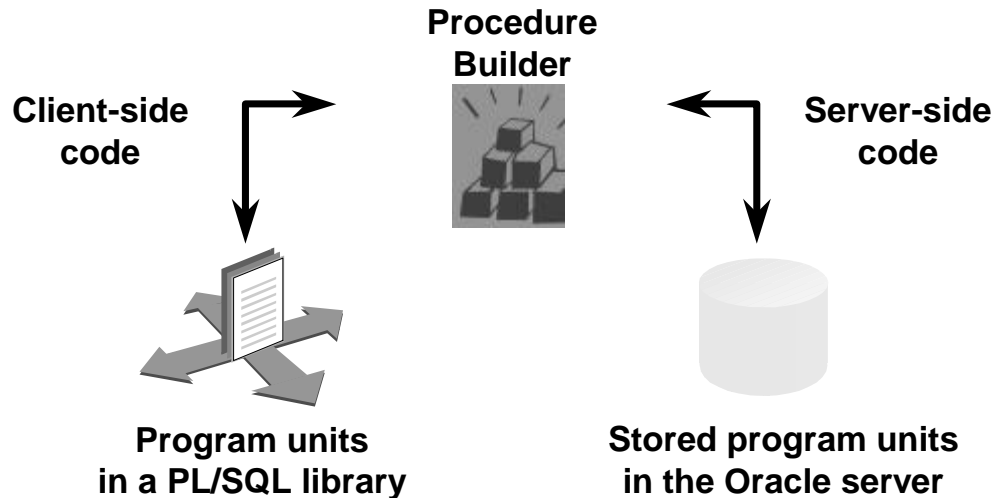
The Program Unit Editor

The easiest and most common place to enter PL/SQL source code is in the Program Unit Editor. You can use it to edit, compile, and browse warning and error messages during application development. The Stored Program Unit Editor is a GUI environment for editing server-side packages and subprograms. The compile operation submits the source text to the server-side PL/SQL compiler.

The Database Trigger Editor

The Database Trigger Editor is a GUI environment for editing database triggers. The compile operation submits the source text to the server-side PL/SQL compiler.

Developing Program Units and Stored Programs Units



ORACLE

F-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Program Units and Stored Program Units

Use Procedure Builder to develop PL/SQL subprograms that can be used by client and server applications.

Program units are client-side PL/SQL subprograms that you use with client applications, such as Oracle Developer. Stored program units are server-side PL/SQL subprograms that you use with all applications, client or server.

Developing PL/SQL Code

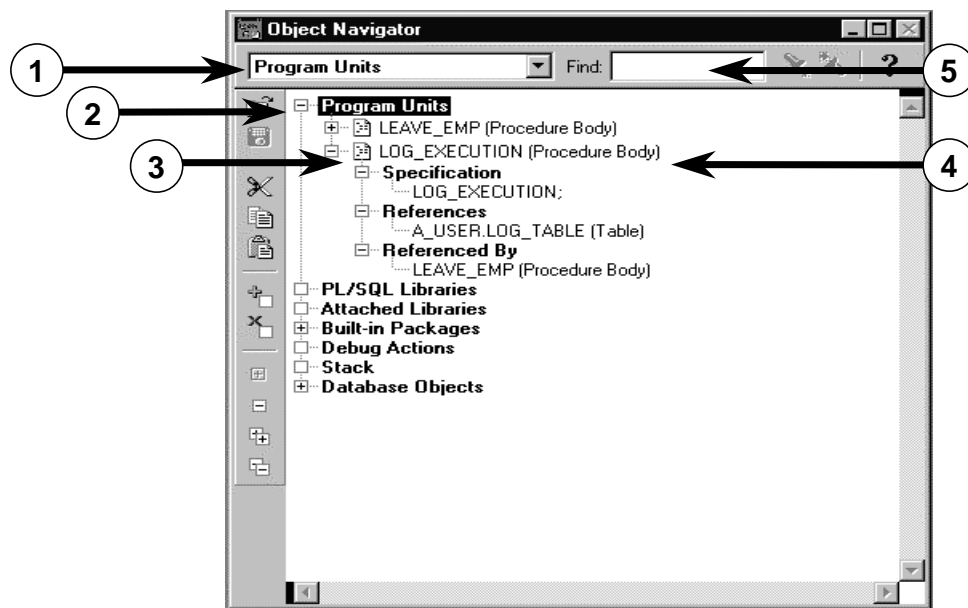
Client-side code:

- Create program units by using the Program Unit Editor
- Drag a server-side subprogram to the client by using the Object Navigator

Server-side code:

- Create stored programs by using the Stored Program Unit Editor
- Drag a client-side program unit to the server by using the Object Navigator

Procedure Builder Components: The Object Navigator



ORACLE

F-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Components of the Object Navigator

The following descriptions correspond to the numbered components on the slide:

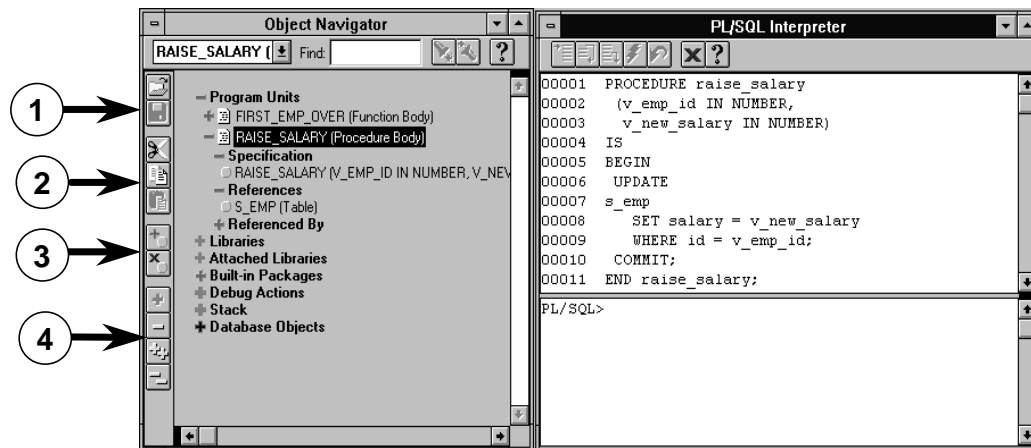
1. Location indicator: Shows your current location in the hierarchy.
2. Subobject indicator: Allows you to expand and collapse nodes to view or hide object information. Different icons represent different classes of objects.
3. Type icon: Indicates the type of object, followed by the name of the object. In the example, the icon indicates that LOG_EXECUTION is a PL/SQL block. If you double-click the icon, Procedure Builder opens the Program Unit Editor and displays the code of that object.
4. Object name: Shows you the names of the objects.
5. Find field: Allows you to search for objects.

Object Navigator

The Object Navigator is Procedure Builder's browser for locating and working with both client and server program units, libraries, and triggers.

The Object Navigator allows you to expand and collapse nodes, cut and paste, search for an object, and drag PL/SQL program units between the client and the server side.

Procedure Builder Components: The Object Navigator



Components of the Object Navigator: Vertical Button Bar

The vertical button bar on the Object Navigator provides convenient access for many of the actions frequently performed from the File, Edit, and Navigator menus.

1. **Open:** Opens a library from the file system or from the Oracle server.
Save: Saves a library in the file system or on the Oracle server.
2. **Cut:** Cuts the selected object and stores it in the clipboard. Cutting an object also cuts any objects owned by that object.
Copy: Makes a copy of the selected object and stored it in the clipboard. Copying an object also copies any objects owned by that object.
Paste: Pastes the cut or copied module into the selected location. Note that objects must be copied to a valid location in the object hierarchy.
3. **Create:** Creates a new instance of the currently selected object.
Delete: Deletes the selected object with confirmation.
4. **Expand, Collapse, Expand All, and Collapse All:** Expands or collapses one or all levels of subobjects of the currently selected object.

Procedure Builder Components: Objects of the Navigator

- **Program Units**
 - **Specification**
 - **References**
 - **Referenced By**
- **Libraries**
- **Attached Libraries**
- **Built-in Packages**
- **Debug Actions**
- **Stack**
- **Database Objects**

ORACLE

F-11

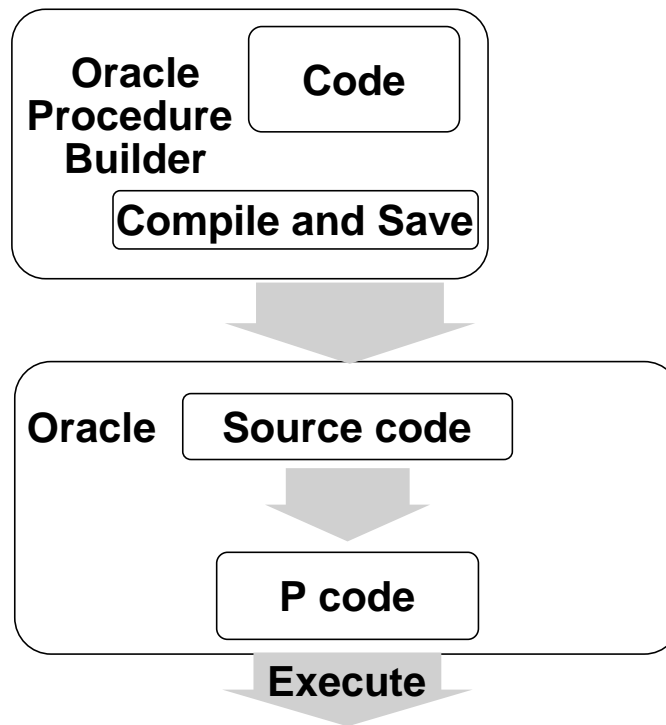
Copyright © Oracle Corporation, 2001. All rights reserved.

Objects of the Object Navigator

By using the Object Navigator, you can display a hierarchical listing of all objects you have access to during your session.

Object Nodes	Description
Program Units	PL/SQL constructs that can be independently recognized and processed by the PL/SQL compiler.
Program Units: Specification	Name, parameter, and return type (functions only) of the program unit.
Program Units: References	Procedures, functions, anonymous blocks, and tables that the program unit references.
Program Units: Referenced By	Procedures, functions, anonymous blocks, and tables that reference the program unit.
Libraries	Collection of PL/SQL packages, procedures, and functions stored in the database or the file system.
Attached Libraries	Referenced libraries stored in the database or the file system.
Built-in Packages	PL/SQL constructs that can be referenced while debugging program units.
Debug Actions	Actions that enable you to monitor or interrupt the execution of PL/SQL program units.
Stack	Chain of subprogram calls, from the initial entry point down to the currently executing subprogram.
Database Objects	Collection of server-side stored program units, libraries, tables, and views.

Developing Stored Procedures



ORACLE

F-12

Copyright © Oracle Corporation, 2001. All rights reserved.

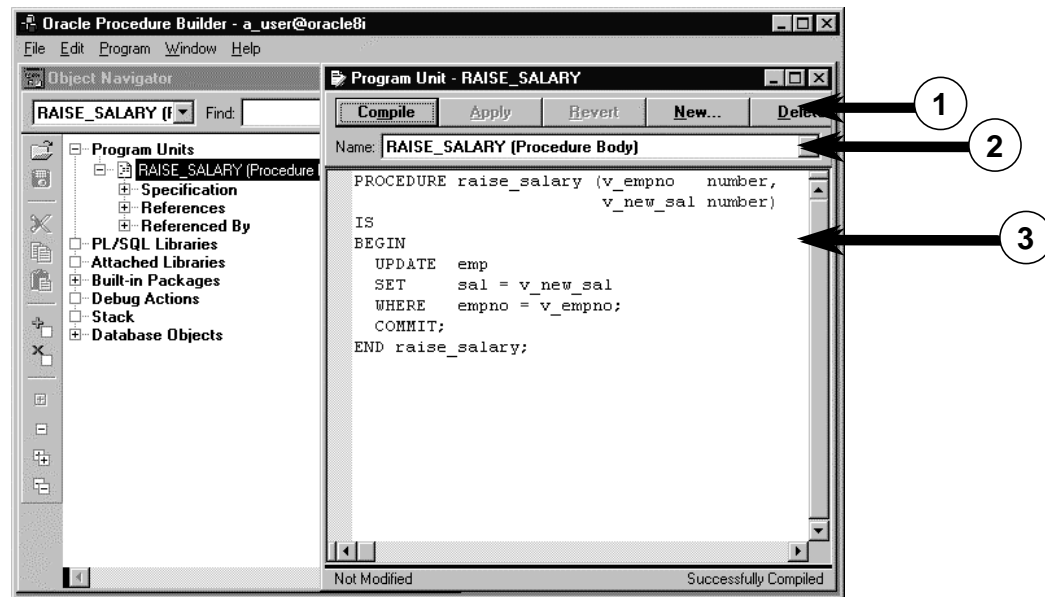
How to Develop Stored Program Units

Use the following steps to develop a stored program unit:

1. Enter the syntax in the Program Unit editor.
2. Click the Save button to compile and save the code.

The source code is compiled into P code.

Procedure Builder Components: The Program Unit Editor



F-13

Copyright © Oracle Corporation, 2001. All rights reserved.

ORACLE

Program Unit Editor

The following descriptions correspond to the numbered components on the slide:

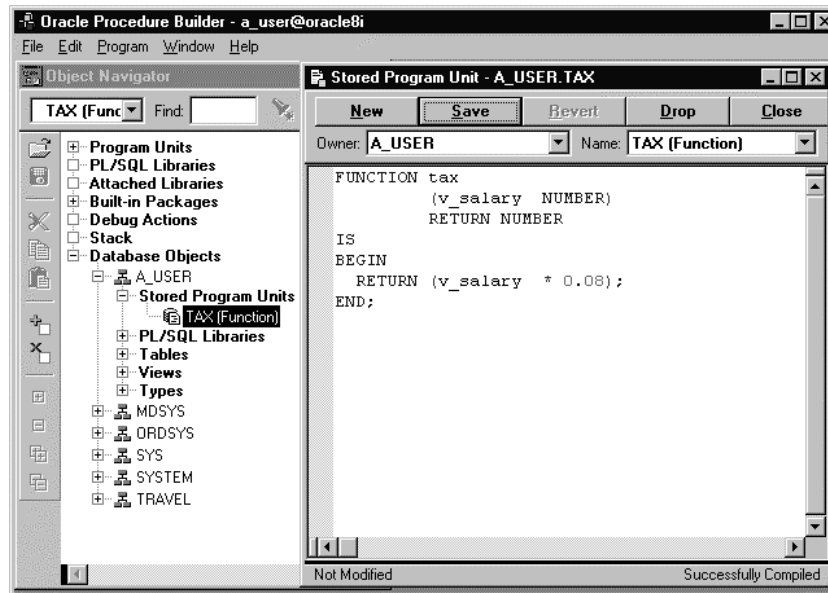
1. Compile, Apply, Revert, New, Delete, Close, and Help buttons
2. Name drop-down list
3. Source text pane

Program Unit Editor

Use the Program Unit Editor to edit, compile, and browse warning and error messages during development of client-side PL/SQL subprograms.

To bring a subprogram into the source text pane, select an option from the Name drop-down list. Use the buttons to decide which action to take once you are in the Program Unit Editor.

Procedure Builder Components: The Stored Program Unit Editor



ORACLE

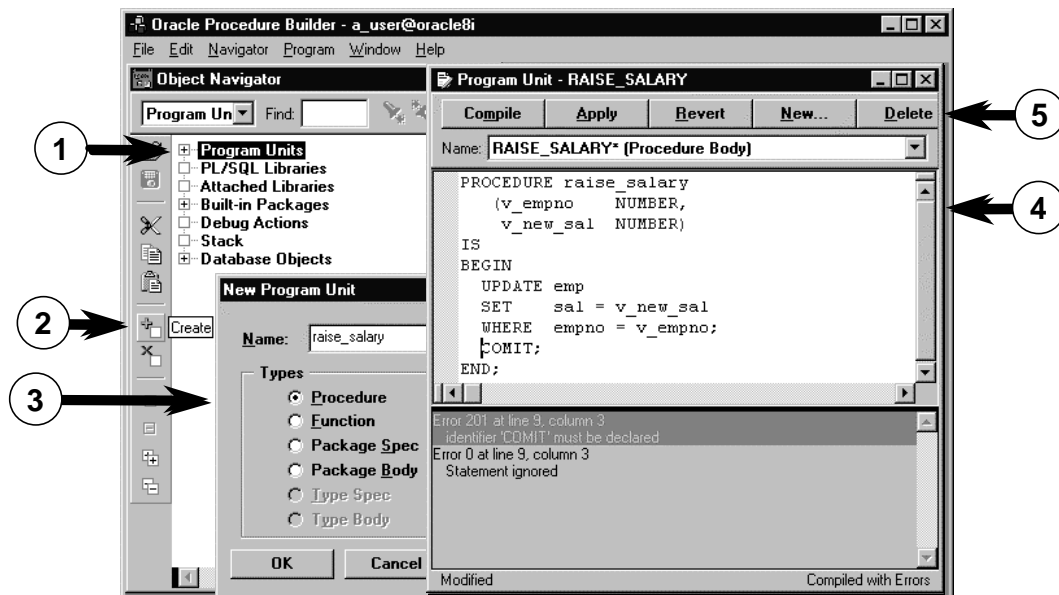
F-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The Stored Program Unit Editor

Use the Stored Program Unit Editor to edit server-side PL/SQL constructs. The Save operation submits the source text to the server-side PL/SQL compiler.

Creating a Client-Side Program Unit



F-15

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Create a Client-Side Program Unit

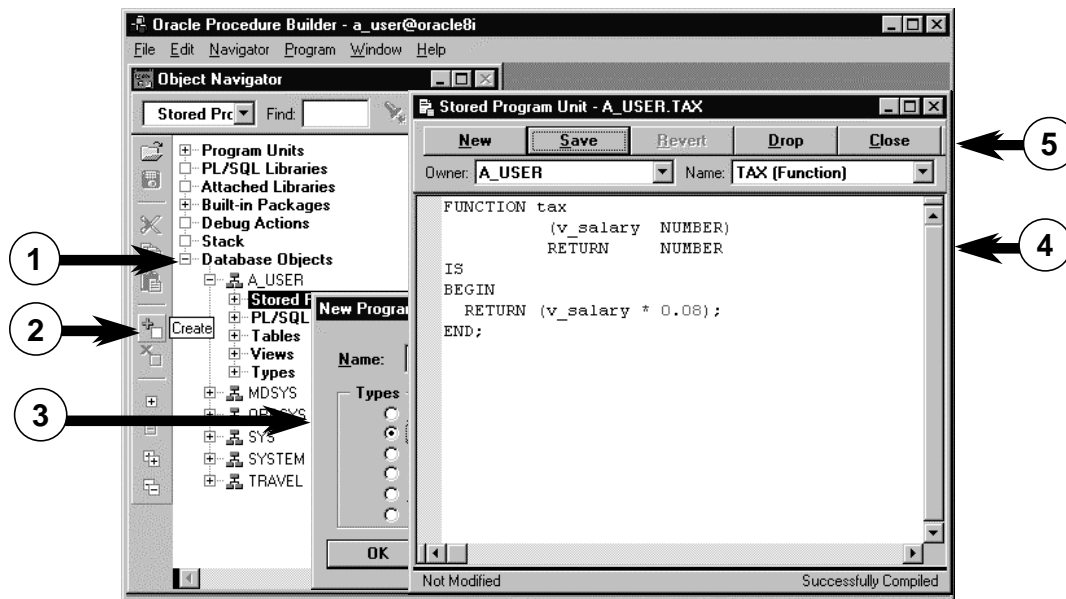
1. Select the Program Units object or subobject.
2. Click the Create button. The New Program Unit dialog box appears.
3. Enter the name of your subprogram, select the subprogram type, and click the OK button to accept the entries.
4. The Program Unit editor is displayed. It contains the skeleton for your PL/SQL construct. The cursor is automatically positioned on the line beneath the BEGIN keyword. You can now write the code.
5. When you finish writing the code, click Compile in the Program Unit Editor.

Error messages generated during compilation are displayed in the compilation message pane in the Program Unit window. When you select an error message, the cursor moves to the location of the error in the program screen.

When your PL/SQL code is error free, the compilation message disappears, and the Successfully Compiled message appears in the status line of the Program Unit Editor.

Note: Program units that reside in the Program Units node are lost when you exit Procedure Builder. You must export them to a file, save them in a PL/SQL library, or store them in the database.

Creating a Server-Side Program Unit



F-16

Copyright © Oracle Corporation, 2001. All rights reserved.

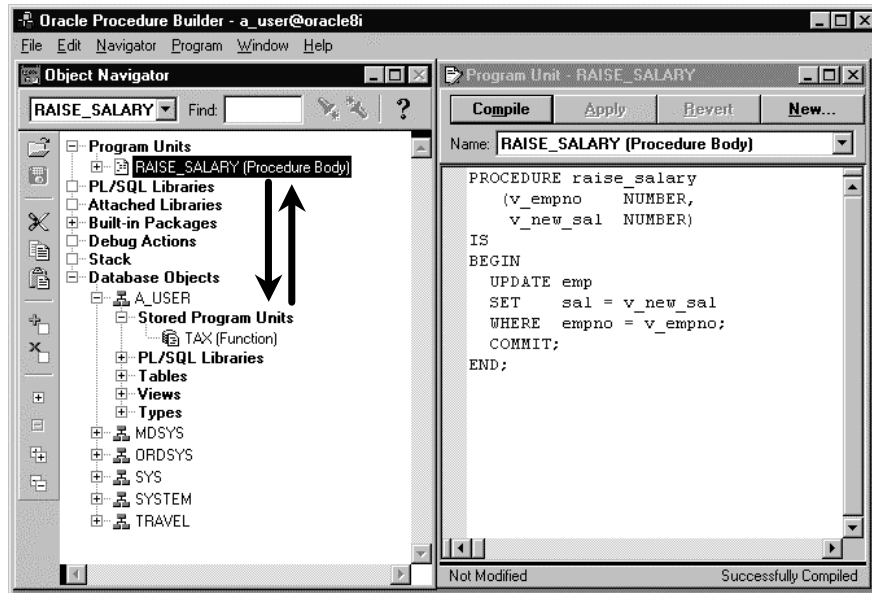
How to Create a Server-Side Program Unit

1. Select the Database Objects node in the Object Navigator, expand the schema name, and click Stored Program Units.
2. Click Create.
3. In the New Program Unit window, enter the name of the subprogram, select the subprogram type, and click OK to accept the entries.
4. The Stored Program Unit editor is displayed. It contains the skeleton for your PL/SQL construct. The cursor is automatically positioned on the line beneath the BEGIN keyword. You can now write the code.
5. When you finish writing the code, click Save in the Stored Program Unit Editor.

Error messages generated during compilation are displayed in a compilation message at the bottom of the window. Click an error message to move to the location of the error.

When the PL/SQL code is error-free, the compilation message does not appear. The Successfully Compiled message appears in the status line at the bottom of the Stored Program Unit Editor window.

Transferring Program Units Between Client and Server



F-17

Copyright © Oracle Corporation, 2001. All rights reserved.

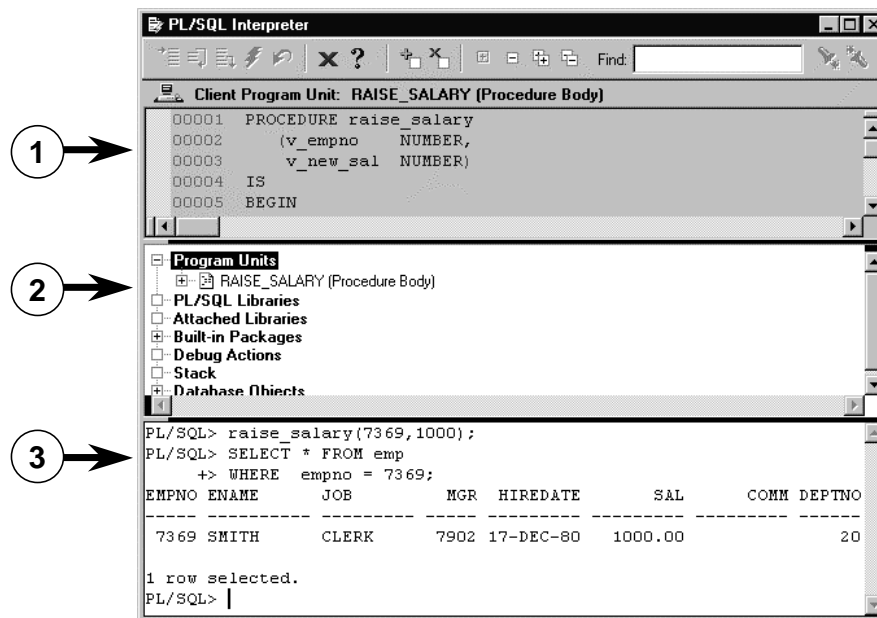
Application Partitioning

Using Procedure Builder you can create PL/SQL program units on both the client and the server. You can also use Procedure Builder to copy program units created on the client into stored program units on the server (or vice versa). You can do this by dragging the program unit to the destination Stored Program Units node in the appropriate schema.

PL/SQL code that is stored in the server is processed by the server-side PL/SQL engine; therefore, any SQL statements contained within the program unit do not have to be transferred between a client application and the server.

Program units on the server are potentially accessible to all applications (subject to user security privileges).

Procedure Builder Components: The PL/SQL Interpreter



Components of the PL/SQL Interpreter

1. Source pane: Displays the PL/SQL code of your program.
2. Navigator pane: Displays the same information as the Object Navigator, but within the PL/SQL Interpreter.
3. Interpreter pane: Allows you to execute subprograms, Procedure Builder commands, and SQL statements.

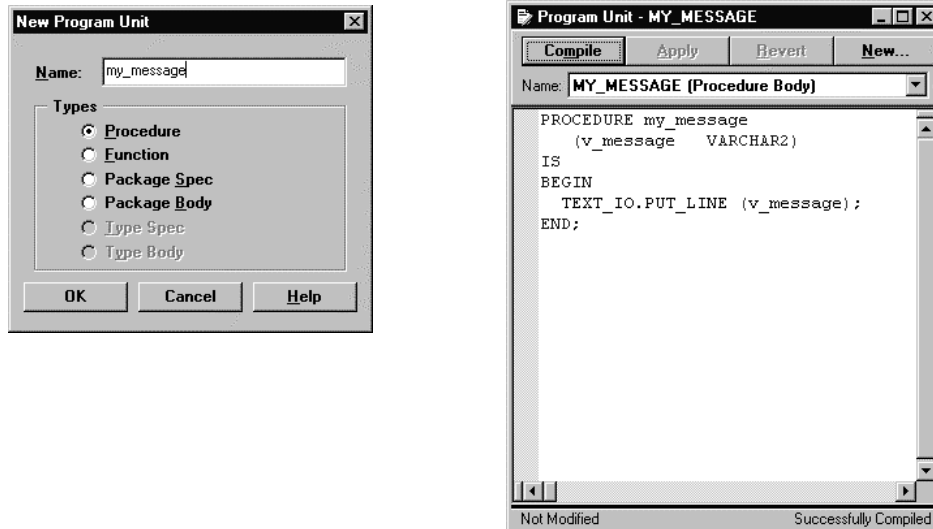
To execute subprograms, enter the name of your PL/SQL program at the PL/SQL prompt, provide any parameters, and terminate with a semicolon.

```
PL/SQL> construct_name [parameter1|parameter2,...];
```

To execute SQL statements, enter your SQL statement and terminate with a semicolon.

```
PL/SQL> SELECT      *  
      +> FROM        departments;
```

Creating Client-Side Program Units



ORACLE

F-19

Copyright © Oracle Corporation, 2001. All rights reserved.

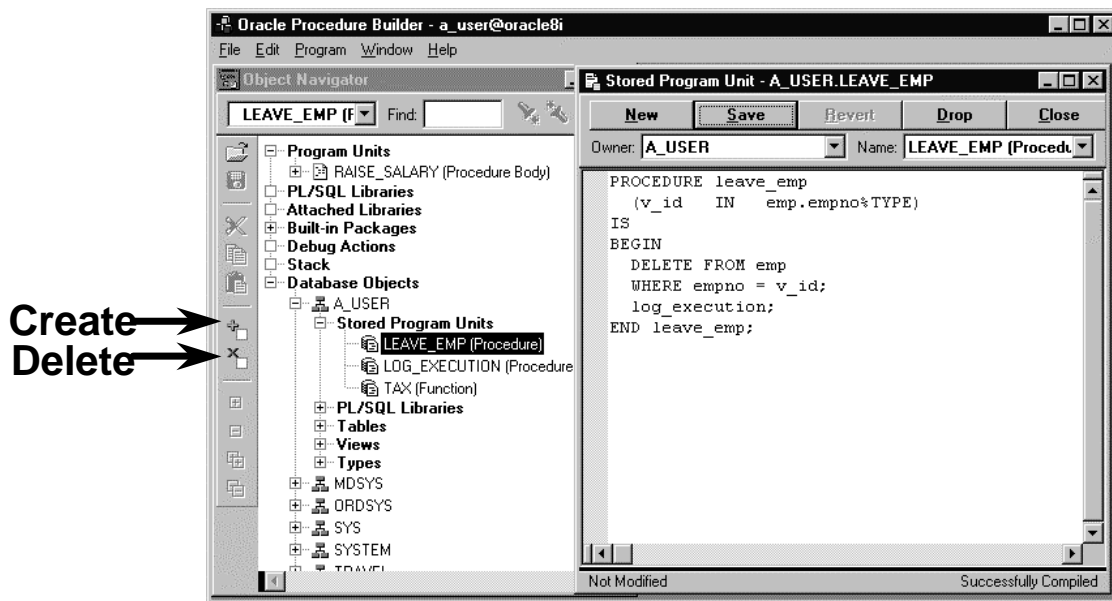
How to Create a Client-Side Program Units

1. Select the Program Units node in the Object Navigator.
2. Click Create. The New Program Unit dialog box appears.
3. Enter a name for the procedure. Note that the default program unit type is Procedure. Click OK to accept these entries. The program unit name appears in the Object Navigator.
 - The Program Unit editor appears, containing the procedure name and IS, BEGIN, and END statements.
 - The cursor is automatically positioned on the line beneath the BEGIN keyword.
4. Enter the source code.
5. Click Compile. Error messages generated during compilation are displayed in the compilation message pane (the lower half of the window).
6. Select an error message to go to the location of the error in the source text pane.

When successfully compiled, a message is displayed in the lower right hand corner of the Program Unit Editor window.
7. Save the source code in a file (M) File > Export.

Note: The keywords CREATE, and CREATE OR REPLACE and the forward slash are invalid in Procedure Builder.

Creating Server-Side Program Units



F-20

Copyright © Oracle Corporation, 2001. All rights reserved.

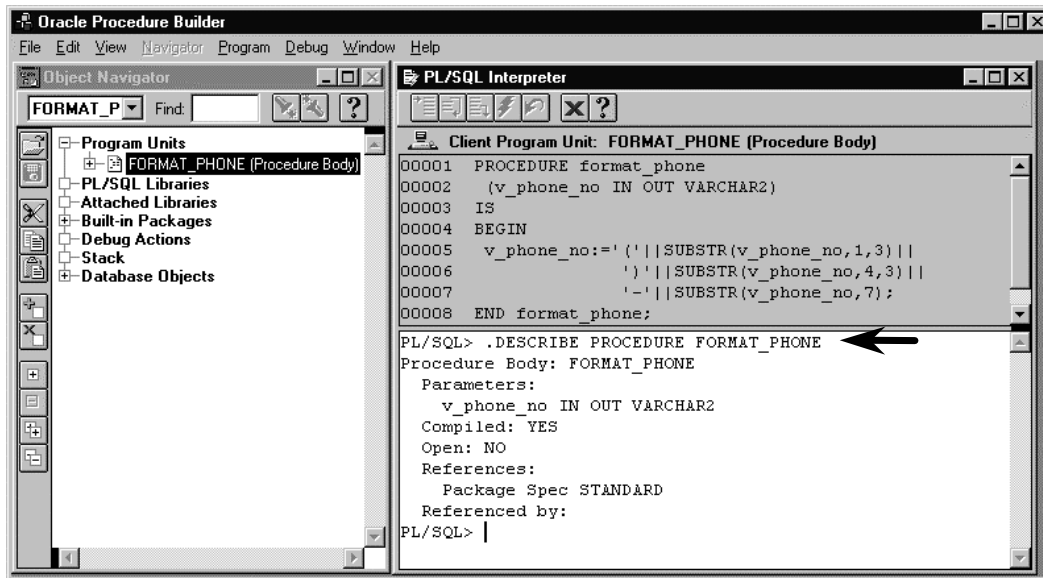
ORACLE

How to Create a Server-Side Program Units

1. Select File > Connect. Then enter your username, password, and database connect string.
2. Expand the Database Objects node in the Object Navigator.
3. Expand your schema name.
4. Click the Stored Program Units node under that schema.
5. Click Create in the Object Navigator.
6. Enter the name for the procedure in the New Program Unit dialog box.
7. Click OK to accept.
8. Enter the source code and click save.

Note: The keywords CREATE, and CREATE OR REPLACE and the forward slash are invalid in Procedure Builder.

The DESCRIBE Command in Procedure Builder



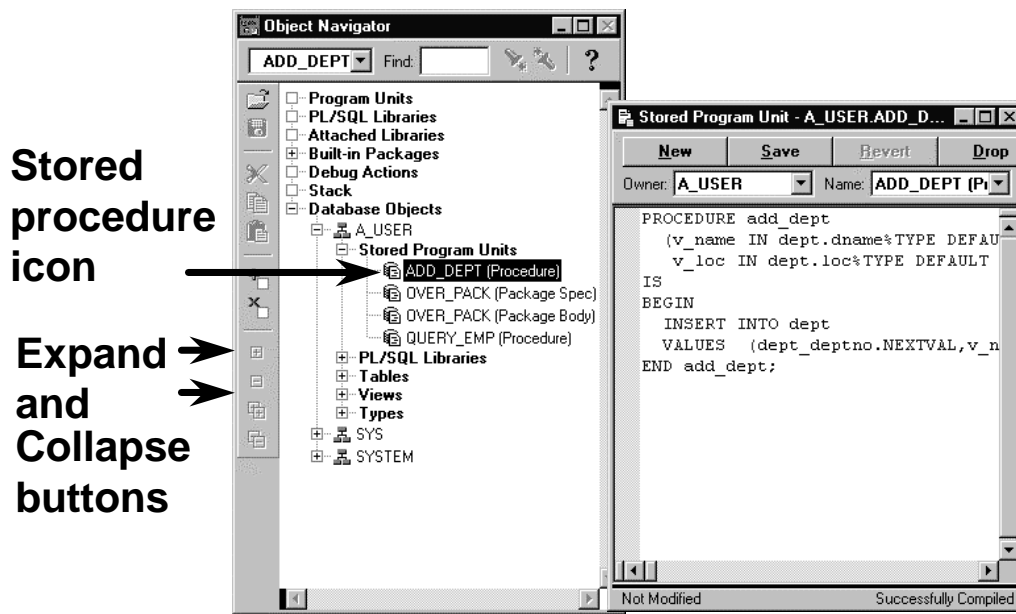
Describing Procedures and Functions

To display a procedure or function, its parameter list, and other information, use the `.DESCRIBE` command in Procedure Builder.

Example

Display information about the `FORMAT_PHONE` procedure.

Listing Code of Stored Program Units



F-22

Copyright © Oracle Corporation, 2001. All rights reserved.

ORACLE

Listing Code of a Stored Procedure

1. Select File > Connect and enter your username, password, and database.
2. Select Database Objects and click the Expand button.
3. Select the schema of the procedure owner and click the Expand button.
4. Select Stored Program Units and click the Expand button.
5. Double-click the icon of the stored procedure. The Stored Program Unit editor appears in the window and contains the code of the procedure.

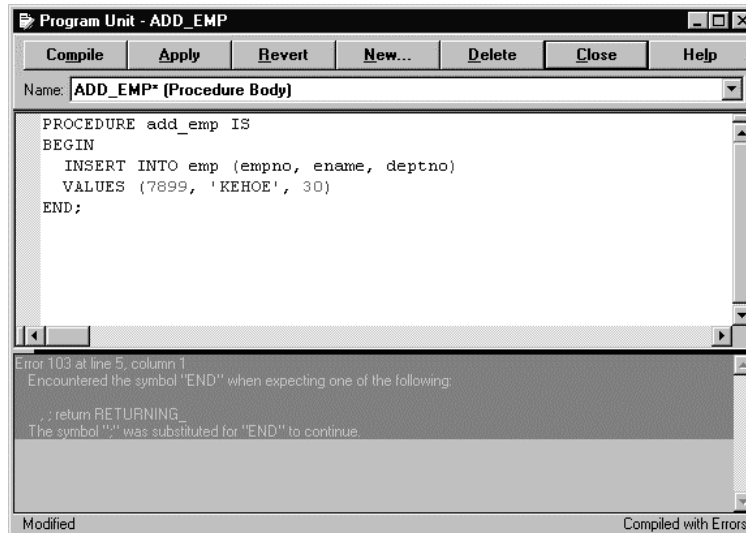
The ADD_DEPT Procedure Code

The example above shows the PL/SQL Program Unit editor with the code for the ADD_DEPT procedure.

The code can now be saved to a file.

1. Select File > Export and enter the name of your file in the Open dialog box.
2. Click OK. A file containing your stored procedure text (.pls extension) is created.

Navigating Compilation Errors in Procedure Builder



ORACLE

F-23

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Resolve Compilation Errors

1. Click Compile.
2. Select an error message.
The cursor moves to the location of the error in the source pane.
3. Resolve the syntax error and click Compile.

Procedure Builder Built-in Package: TEXT_IO

- The **TEXT_IO** package:
 - Contains a procedure **PUT_LINE**, which writes information to the PL/SQL Interpreter window
 - Is used for client-side program units
- The **TEXT_IO.PUT_LINE** accepts one parameter

```
PL/SQL> TEXT_IO.PUT_LINE(1);  
1
```

ORACLE

F-24

Copyright © Oracle Corporation, 2001. All rights reserved.

TEXT_IO Built-in Package

You can use **TEXT_IO** packaged procedures to output values and messages from a client-side procedure or function to the PL/SQL Interpreter window.

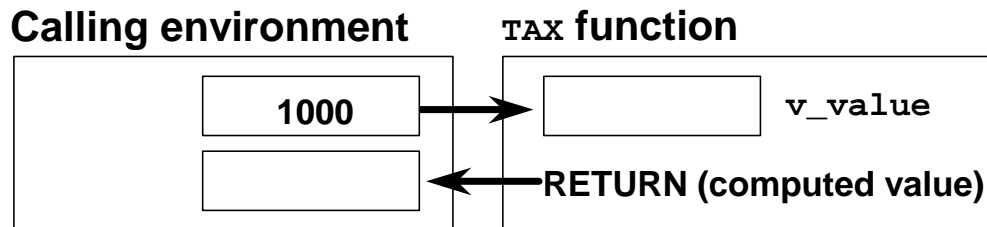
TEXT_IO is a built-in package that is part of Procedure Builder.

Use the Oracle supplied package **DBMS_OUTPUT** to debug server-side procedures, and the Procedure Builder built-in, **TEXT_IO**, to debug client-side procedures.

Note:

- You cannot use **TEXT_IO** to debug server-side procedures. The program will fail to compile successfully because **TEXT_IO** is not stored in the database.
- **DBMS_OUTPUT** does not display messages in the PL/SQL Interpreter window if you execute a procedure from Procedure Builder.

Executing Functions in Procedure Builder: Example



Display the tax based on a specified value.

```
PL/SQL> .CREATE NUMBER x PRECISION 4
PL/SQL> :x := tax(1000);
PL/SQL> TEXT_IO.PUT_LINE (TO_CHAR(:x));
80
```

ORACLE

F-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

Execute the TAX function from Procedure Builder:

1. Create a host variable to hold the value returned from the function. Use the `.CREATE` syntax at the Interpreter prompt.
2. Create a PL/SQL expression to invoke the function `TAX`, passing a numeric value to the function. Note the use of the colon (`:`) to reference a host variable.
3. View the result of the function call by using the `PUT_LINE` procedure in the `TEXT_IO` package.

Creating Statement Triggers

The screenshot shows the 'Database Trigger' dialog box in Oracle. The 'Table Owner' is 'A_USER', the 'Table' is 'EMP', and the 'Name' is 'SECURE_EMP'. The 'Triggering' section has 'Before' selected. The 'Statement' section has 'INSERT' checked. The 'For Each' section has 'Statement' selected. The 'Trigger Body' contains the following PL/SQL code:

```
BEGIN
  IF TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN')
  OR TO_CHAR(SYSDATE, 'HH24') NOT BETWEEN '08' AND '18'
  THEN
    RAISE_APPLICATION_ERROR (-20500,
      'You may only insert into the EMP table during business hours.');
```

At the bottom of the dialog are buttons for 'New', 'Save', 'Revert', 'Drop', 'Close', and 'Help'.

ORACLE

F-26

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Create a Statement Trigger When Using Procedure Builder

You can also create the same BEFORE statement trigger in Procedure Builder.

1. Connect to the database.
2. Click the Database Objects node in the Object Navigator.
3. Select the Database Trigger editor from the Program menu.
4. Select a table owner and a table from the Table owner and Table drop-down lists.
5. Click New to start creating the trigger.
6. Select one of the Triggering option buttons to choose the timing component.
7. Select Statement to choose the event component.
8. In the Trigger Body region, enter the trigger code.
9. Click Save. Your trigger code will now be compiled by the PL/SQL engine in the server. Once successfully compiled, your trigger is stored in the database and automatically enabled.

Note: If the trigger has compilation errors, the error message appears in a separate window.

Creating Row Triggers

Database Trigger

Table Owner: A_USER Table: EMP Name: DERIVE_COMMISSION_PCT

Triggering: ☒ Before ☐ After ☐ Instead Of

Statement: ☒ UPDATE ☒ INSERT ☐ DELETE

Of Columns: EMPNO, ENAME, JOB, MGR, HIREDATE, SAL

For Each: ☐ Statement ☒ Row

Referencing OLD As: OLD NEW As: NEW

When:

Trigger Body:

```
BEGIN
  IF NOT (:NEW.JOB IN ('MANAGER' , 'PRESIDENT'))
    AND :NEW.SAL > 5000
  THEN
    RAISE_APPLICATION_ERROR
      (-20202, 'EMPLOYEE CANNOT EARN THIS AMOUNT');
  END IF;
END;
```

New Save Revert Drop Close Help

ORACLE

F-27

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Create a Row Trigger When Using Procedure Builder

You can also create the same BEFORE row trigger in Procedure Builder.

1. Connect to the database.
2. Click the Database Objects node in the Object Navigator.
3. Select the Database Trigger Editor from the Program menu.
4. Select a table owner and a table from the corresponding drop-down lists.
5. Click New to start creating the trigger.
6. Select the Triggering option button to choose the timing component.
7. Select the appropriate Statement check boxes to choose the events component.
8. In the For Each region, select the Row option button to designate the trigger as a row trigger.
9. Complete the Referencing OLD As and NEW As fields if you want to modify the correlation names. In the When field, enter a WHEN condition to restrict the execution of the trigger. These fields are optional and are available only with row triggers.
10. Enter the trigger code.
11. Click Save. The trigger code is now compiled by the PL/SQL engine in the server. When successfully compiled, the trigger is stored in the database and automatically enabled.

Removing Server-Side Program Units

Using Procedure Builder:

1. Connect to the database.
2. Expand the Database Objects node.
3. Expand the schema of the owner of the program unit.
4. Expand the Stored Program Units node.
5. Click the program unit that you want to drop.
6. Click Delete in the Object Navigator.
7. Click Yes to confirm.

ORACLE

F-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Removing a Server-Side Program Unit

When you decide to delete a stored program unit, an alert box displays with the following message:

"Do you really want to drop stored program unit *<program unit name>*?". Click Yes to drop the unit.

In the Stored Program Units Editor, you can also click DROP to remove the procedure from the server.

Removing Client-Side Program Units

Using Procedure Builder:

1. Expand the Program Units node.
2. Click the program unit that you want to remove.
3. Click Delete in the Object Navigator.
4. Click Yes to confirm.

ORACLE

F-29

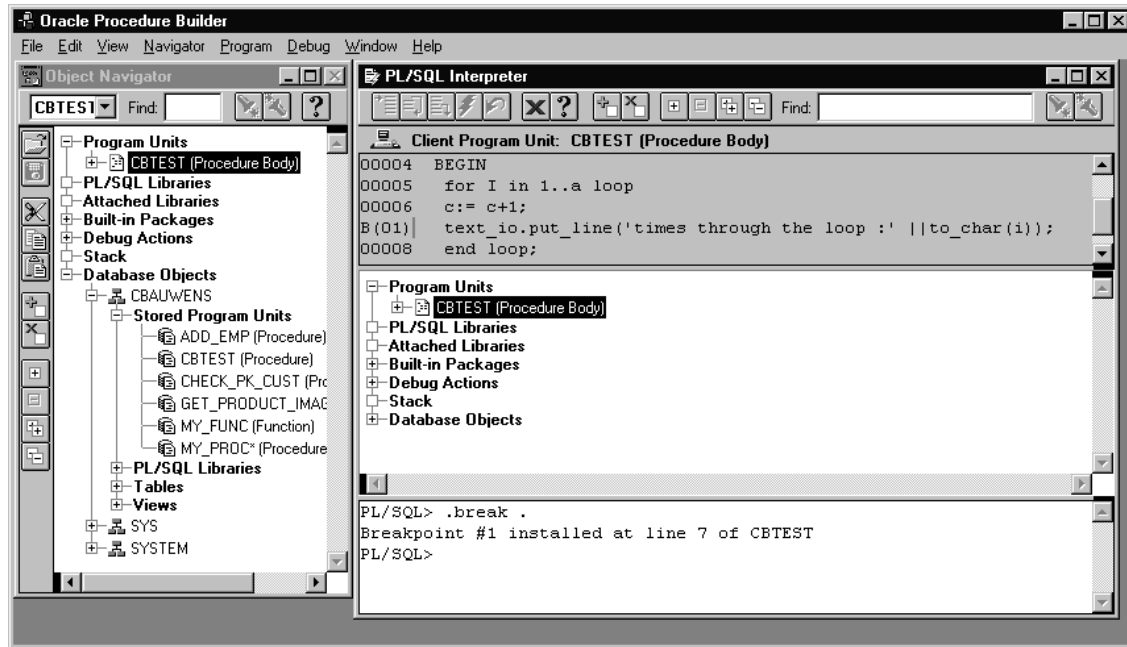
Copyright © Oracle Corporation, 2001. All rights reserved.

Removing a Client-Side Program Unit

Follow the steps in the slide to remove a procedure from Procedure Builder.

If you have exported the code that built your procedure to a text file and you want to delete that file from the client, you must use the appropriate operating system command.

Debugging Subprograms by Using Procedure Builder

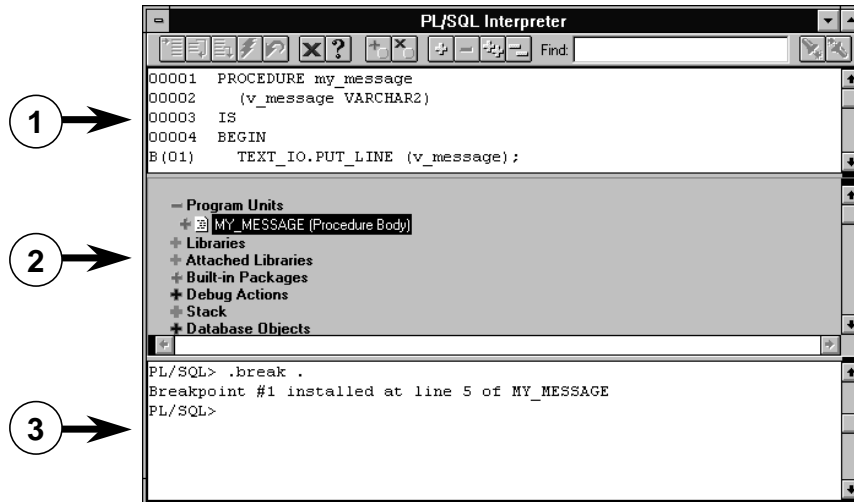


Debugging Subprograms by Using Procedure Builder

You can perform debug actions on a server-side or client-side subprogram using Procedure Builder. Use the following steps to load the subprogram:

1. From the Object Navigator, select Program > PL/SQL Interpreter.
2. In the menu, select View > Navigator Pane.
3. From the Navigator pane, expand either the Program Units or the Database objects node.
4. Locate the program unit that you want to debug and click it.

Listing Code in the Source Pane



Listing Code in the Source Pane

Performing Debug Actions in the Interpreter

You can use the Object Navigator to examine and modify parameters in an interrupted program. By invoking the Object Navigator within the Interpreter, you can perform debugging actions entirely within the Interpreter window. Alternatively, you can interact with the Object Navigator and Interpreter windows separately.

1. Invoking the Object Navigator Pane

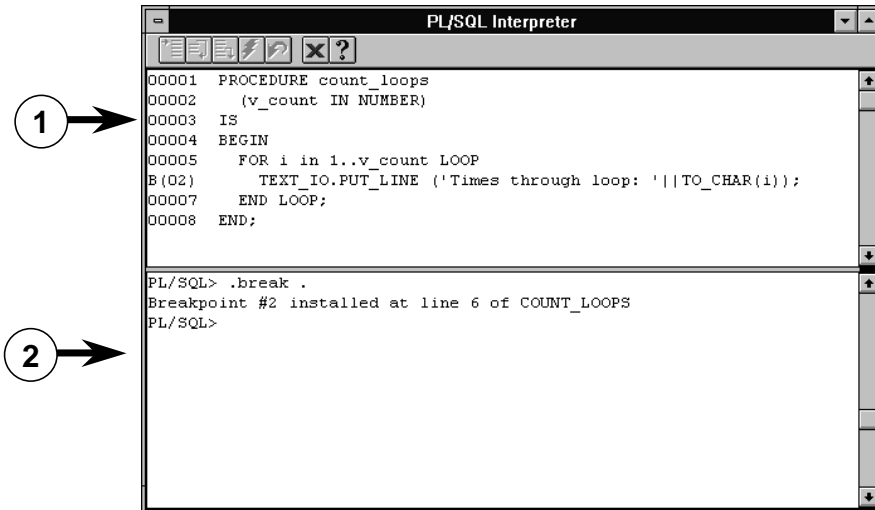
- Select PL/SQL Interpreter from the Tools menu to open the Interpreter if it is not already open.
- Select Navigator Pane from the View menu.
- The Navigator pane is inserted between the Source and the Interpreter panes.
- Drag the split bars to adjust the size of each pane.

2. Listing Source Text in the Source Pane

- Click the Program Units node in the Navigator pane to expand the list.
The list of program units is displayed.
- Click the object icon of the program unit to be listed.

3. The source code is listed in the Source pane of the Interpreter.

Setting a Breakpoint



ORACLE

F-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Setting a Breakpoint

If you encounter errors while compiling or running your application, you should test the code and determine the cause for the error. To determine the cause of the error effectively, review the code, line by line. Eventually, you should identify the exact line of code causing the error. You can use a breakpoint to halt execution at any given point and to permit you to examine the status of the code on a line-by-line basis.

Setting a Breakpoint

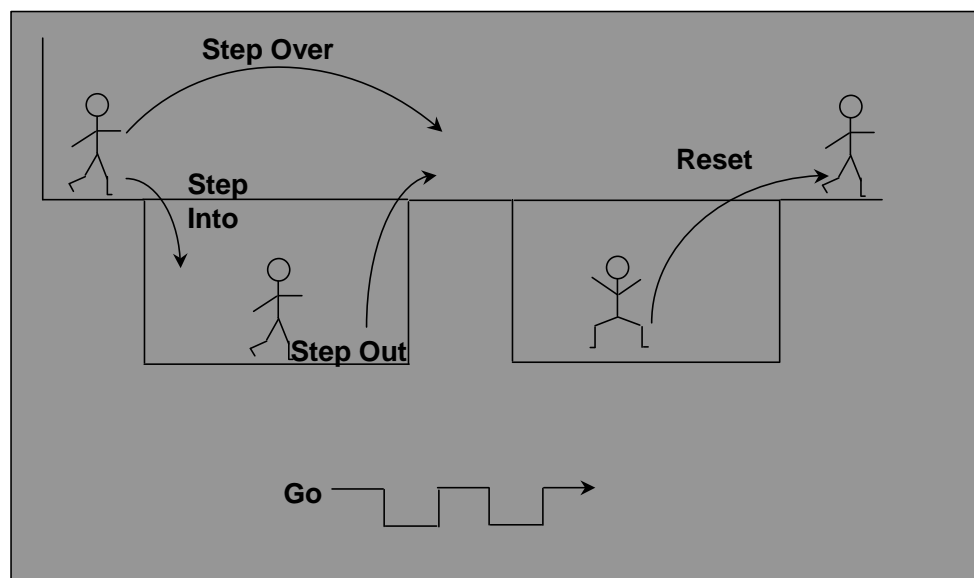
1. Double click the executable line of code on which to break. A "B(n)" is placed in the line where the break is set.
2. The message Breakpoint #n installed at line i of name is shown in the Interpreter pane.

Note: Breakpoints also can be set using debugger commands in the Interpreter pane. Test breakpoints by entering the program unit name at the Interpreter PL/SQL prompt.

Monitoring Debug Actions

Debug actions, like breakpoints, can be viewed in the Object Navigator under the heading Debug Actions. Double-click the Debug Actions icon to view a description of the breakpoint. Remove breakpoints by double-clicking the breakpoint line number

Debug Commands



ORACLE

F-33

Copyright © Oracle Corporation, 2001. All rights reserved.

Debug Commands

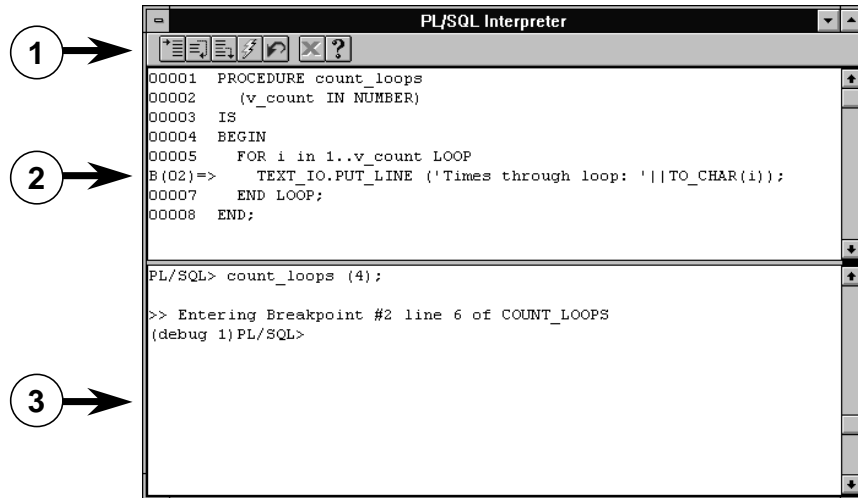
Reviewing Code

When a breakpoint is reached, you can use a set of commands to step through the code. You can execute these commands by clicking the command buttons on the Interpreter toolbar or by entering the command at the Interpreter prompt.

Commands for Stepping through Code

Command	Description
Step Into	Advances execution into the next executable line of code
Step Over	Bypasses the next executable line of code and advances to the subsequent line
Step Out	Resumes to the end of the current level of code, such as the subprogram
Go	Resumes execution until either the program unit ends or is interrupted again by a debug action
Reset	Aborts the execution at the current levels of debugging

Stepping through Code



ORACLE

F-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Stepping Through Code

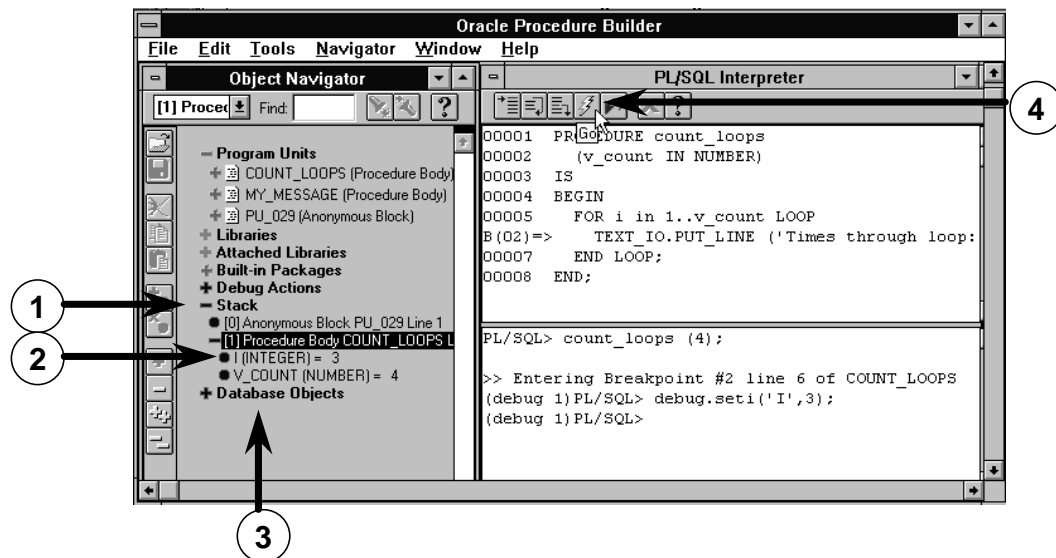
Determining the Cause of Error

After the breakpoint is found at run time, you can begin stepping through the code. An arrow (`=>`) indicates the next line of code to execute.

1. Click the Step Into button.
2. A single line of code is executed. The arrow moves to the next line of code.
3. Repeat step 1 as necessary until the line causing the error is found.

The arrow continues to move forward until the erroneous line of code is found. At that time, PL/SQL displays an error message.

Changing a Value



F-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Changing a Value

Examining Local Variables

Using Procedure Builder, you can examine and modify local variables and parameters in an interrupted program. Use the Stack node in the Navigator pane to view and change the values of local variables and parameters associated with the current program unit located in the call stack. When debugging code, check for the absence of values as well as incorrect values.

Examining Values and Testing the Possible Solution

1. Click the Stack node in the Object Navigator or Navigator pane to expand it.
2. Click the value of the variable to edit. For example, select variable 1.
The value 1 becomes an editable field.
3. Enter the new value and click anywhere in the Navigator pane to end the variable editing, for example, enter 3.

The following statement is displayed in the Interpreter pane:

```
(debug1) PL/SQL> debug.seti('I',3);
```

4. Click the Go button to resume execution through the end of the program unit.

Note: Variables and parameters can also be changed by using commands at the Interpreter PL/SQL prompt.

Summary

In this appendix, you should have learned how to:

- **Use Procedure Builder:**
 - Application partitioning
 - Built-in editors
 - GUI execution environment
- **Describe the components of Procedure Builder**
 - Object Navigator
 - Program Unit Editor
 - PL/SQL Interpreter
 - Debugger

ORACLE

Index

A

actual parameter 2-6

anonymous blocks 1-7

application trigger 9-3

aUTHID CURRENT_USE 4-5

B

BEFORE statement trigger 9-14

BEGIN 1-7

benefits 2-26

BFILE 8-12

BFILENAME 8-12

binding 7-5

C

CALL statement 10-7

CLOSE_CONNECTION 7-31

CREATE PROCEDURE 2-5

CREATE ANY DIRECTORY 8-13

D

database trigger 9-3, 10-11

DBA_JOB 7-19

DBA_JOBS_RUNNING 7-19

DBMS_DDL 7-12

DBMS_JOB 7-13

DBMS_JOB.BROKEN 7-18

DBMS_JOB.REMOVE 7-18

DBMS_JOB.RUN 7-18

DBMS_LOB 7-21, 8-12

data dictionary view 4-9, 4-11

data type 3-4

DBMS_OUTPUT 4-16

DBMS_SQL 7-6

DECLARE 1-7

definer's-rights 4-4

DEPTREE 11-8

DIRECTORY 8-10

DROP PROCEDURE 2-25

dynamic SQL 7-4

E

EMPTY_BLOB 8-24

EMPTY_CLOB 8-24

END 1-7

environments 1-13

EXCEPTION 1-7, 2-21

EXECUTE 4-3, 7-11

external large object 8-8

F

fetch 7-5

FILE_LOCATOR 8-16

file_type 7-27

formal parameter 2-6

forward declaration 6-8

function 3-3, 8-12

H

host variable 2-14

I

IDEPTREE 11-8

IMMEDIATE 7-11

INSTEAD OF 9-22

INSTEAD OF 9-7

internal 8-6

invoke a procedure 2-9

IS_OPEN 7-26

L

LOB 8-3, 8-5, 8-32

LOB locator 8-5

local dependencies 11-5

locator 8-12

LONG 8-4

LONG-to-LOB 8-17

M

migration 8-17

modularization 1-6

modules 1-6

mutating table 10-8

N

NEW 9-19

O

object privilege 4-3
OCI 8-10
OLD 9-19
one-time-only procedure 6-10
OPEN_CONNECTION 7-31
Oracle Internet Platform 1-4
overload 6-3
OLD and NEW qualifiers 9-19

P

package 4-16, 5-3, 7-22, 8-9, 8-19
package body 5-11
package specification 5-8
parameter mode 2-8, 3-8
Parsing 7-5
persistent state 6-14
PL/SQL block 2-4
PL/SQL construct 1-5
PROCEDURE 2-5, 2-25
procedures and functions within the 7-23
purity level 6-11

R

recompile a PL/SQL object 11-22
remote dependencies 11-12
row trigger 9-18
READ 8-12
REPLACE 2-4
REQUEST 7-29
REQUEST_PIECES 7-29
RETURN 3-4
row trigger 9-9

S

schedule batch job 7-13
security mechanism 8-9
SESSION_MAX_OPEN_FILE 8-13
SHOW ERROR 4-11

signature 11-13
SQL*Plus 1-4
statement trigger 9-9, 9-14
SUBMIT 7-15
submit PL/SQL program 7-13
subprogram 1-6
system event 10-3
system privileges 4-3

T

temporary 8-32
time stamp 11-13, 11-20
TO_BLOB 8-18
TO_CLOB 8-18
trigger action 9-10
trigger name 9-13
trigger timing 9-6
trigger type 9-6
triggering event 9-8

U

user-defined PL/SQL function 3-13
USER_DEPENDENCIES 11-7
USER_ERRORS 4-11
USER_JOBS 7-19
USER_OBJECTS 4-7
USER_SOURCE 4-9
USER_TRIGGER 10-28
UTL_FILE 7-21
UTL_FILE_DIR 7-22
UTL_HTTP 7-29
UTL_TCP 7-31

