



Develop PL/SQL Program Units

Student Guide

41024GC11
Production 1.1
April 1998
M06524

ORACLE®

Authors

Christian Bauwens
Ellen Gravina

Technical Contributors and Reviewers

Rhonda Bassett
Jacquelyn Bruce
Larry Cross
Gary Propeck
Bryan Roberts
Hazel Russell
Jason Schiedemeyer
Harry Siegersma
Vijay Venkatachalam

Publishers

Stephanie Jones
Kelly Lee
Lisa Patterson

Copyright © Oracle Corporation, 1998. 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

Introduction

- Course Objectives 1-2
- Overview 1-3
- Course Overview 1-4

1 Overview of PL/SQL

- Objectives 1-2
- PL/SQL Program Constructs 1-3
- Block Structure for Anonymous PL/SQL Blocks 1-4
- Block Structure for PL/SQL Subprograms 1-5
- Benefits of Subprograms 1-6
- Development Environments 1-7
- Developing Procedures and Functions Using SQL*Plus 1-8
- Developing Procedures and Functions Using Oracle Procedure Builder 1-9
- Invoking Stored Procedures and Functions 1-10
- Summary 1-11

2 Working with Procedure Builder

- Objectives 2-2
- Overview 2-3
- Procedure Builder Components: The Object Navigator 2-4
- Procedure Builder Components: The Program Unit Editor 2-5
- Procedure Builder Components: The Stored Program Unit Editor 2-6
- Creating a Program Unit on the Client Side 2-7
- Creating a Program Unit on the Server Side 2-8
- Transferring Program Units Between Client and Server 2-9
- Procedure Builder Components: The PL/SQL Interpreter 2-10
- Summary 2-11
- Practice Overview 2-12

3 Creating Procedures

- Objectives 3-2
- Overview of Procedures 3-3
- Syntax for Creating Procedures 3-4
- Developing Stored Procedures 3-5
- Creating a Stored Procedure Using SQL*Plus 3-6
- Creating a Procedure Using Procedure Builder 3-7
- Creating Server-Side Procedures Using Procedure Builder 3-8
- Creating Client-Side Procedures Using Procedure Builder 3-9
- Navigating Compilation Errors in Procedure Builder 3-10
- Procedural Parameter Modes 3-11
- Parameter Modes for Formal Parameters 3-12

- IN Arguments: Example 3-13
- OUT Arguments: Example 3-14
- OUT Arguments and SQL*Plus 3-16
- OUT Arguments and Procedure Builder 3-17
- IN OUT Arguments 3-18
- Invoking FORMAT_PHONE from SQL*Plus 3-19
- Invoking FORMAT_PHONE from Procedure Builder 3-20
- Methods for Passing Parameters 3-21
- Passing Parameters: Example Procedure 3-22
- Examples of Passing Parameters 3-23
- Invoking a Procedure from an Anonymous PL/SQL Block 3-24
- Invoking a Procedure from a Stored Procedure 3-25
- Removing Procedures 3-26
- Removing Server-Side Procedures 3-27
- Removing Client-Side Procedures 3-30
- Summary 3-31
- Practice Overview 3-32

4 Creating Functions

- Objectives 4-2
- Overview of Stored Functions 4-3
- Syntax for Creating Functions 4-4
- Creating a Function 4-5
- Creating a Stored Function Using SQL*Plus 4-6
- Creating a Stored Function Using SQL*Plus: Example 4-7
- Creating a Function Using Procedure Builder 4-8
- Creating Functions Using Procedure Builder: Example 4-9
- Executing Functions 4-10
- Executing Functions in SQL*Plus: Example 4-11
- Executing Functions in Procedure Builder: Example 4-12
- Advantages of User-Defined Functions in SQL Expressions 4-13
- Locations to Call User-Defined Functions 4-14
- Calling Functions from SQL Expressions: Restrictions 4-15
- Invoking Functions from a SQL Statement: Example 4-17
- Removing Functions 4-18
- Removing a Server-Side Function 4-19
- Removing Server-Side Functions 4-20
- Removing a Client-Side Function 4-22
- Procedure or Function? 4-23
- Comparing Procedures and Functions 4-24
- Benefits of Stored Procedures and Functions 4-25
- Summary 4-26
- Practice Overview 4-27

5 Creating Packages

- Objectives 5-2
- Overview of Packages 5-3
- Advantages of Packages 5-4
- Developing a Package 5-6
- Creating the Package Specification 5-10
- Declaring Public Constructs 5-11
- Creating a Package Specification: Example 5-12
- Declaring a Global Variable or a Public Procedure 5-13
- Creating the Package Body 5-14
- Public and Private Constructs 5-15
- Creating a Package Body: Example 5-16
- Developing Packages: Guidelines 5-18
- Invoking Package Constructs 5-19
- Referencing a Global Variable Within the Package 5-21
- Referencing a Global Variable from a Standalone Procedure 5-22
- Persistent State of Package Variables 5-23
- Persistent State of Package Cursor 5-24
- Persistent State of Package PL/SQL Tables and Records 5-27
- Removing Packages 5-28
- Managing Packages 5-29
- Summary 5-30
- Practice Overview 5-33

6 More Package Concepts

- Objectives 6-2
- Objectives 6-3
- Overloading 6-4
- Overloading: Example 6-5
- Forward Declarations 6-6
- Creating a One-Time-Only Procedure 6-8
- Restrictions on Package Functions Used in SQL 6-9
- Purity Level of a Packaged Functions 6-10
- Using PRAGMA RESTRICT_REFERENCES 6-11
- Invoke a User-Defined Package Function from an SQL Statement 6-12
- Oracle Supplied Packages 6-13
- Using DBMS_PIPE 6-15
- DBMS_PIPE Package 6-17
- DBMS_SQL Execution Flow 6-19
- Using DBMS_SQL 6-21
- DBMS_SQL Package 6-22
- Using DBMS_SQL: Example 6-26
- Invoking the General Purpose Procedure 6-29

DBMS_OUTPUT Package 6-30
DBMS_DDL Package 6-31
Summary 6-32
Practice Overview 6-35

7 Creating Database Triggers

Objectives 7-2
Overview of Triggers 7-3
Database Trigger 7-4
Cascading Triggers 7-5
Database Trigger 7-6
Creating Triggers 7-7
Trigger Components 7-8
Firing Sequence of Database Triggers on a Single Row 7-12
Statement and Row Triggers 7-13
Firing Sequence of Database Triggers on Multiple Rows 7-14
Syntax for Creating Statement Triggers 7-15
Creating Statement Triggers Using Procedure Builder 7-16
Before Statement Trigger: Example 7-17
Example 7-18
Using Conditional Predicates 7-19
After Statement Trigger: Example 7-20
User Audit Table 7-21
Creating a Row Trigger 7-22
Creating Row Triggers Using Procedure Builder 7-23
After Row Trigger: Example 7-24
Using Old and New Qualifiers 7-25
User Audit_Emp_Values Table 7-26
Restricting a Row Trigger 7-27
Differentiating Between Triggers and Stored Procedures 7-28
Managing Triggers 7-29
User_Triggers 7-31
Controlling Developer Security 7-33
Controlling User Security 7-34
Removing Triggers 7-35
Trigger Test Cases 7-36
Rules Governing Triggers 7-37
Changing Data in a Constraining Table 7-38
Constraining Table: Example 7-39
Reading Data from a Mutating Table 7-41
Mutating Table: Example 7-42
Implementation of Triggers 7-45
Controlling Security Within the Server 7-46

Controlling Security with a Database Trigger 7-47
Auditing Using the Server Facility 7-48
Auditing Using a Trigger 7-49
Enforce Data Integrity Within the Server 7-50
Protect Data Integrity with a Trigger 7-51
Enforce Referential Integrity Within the Server 7-52
Protect Referential Integrity with a Trigger 7-53
Replicate a Table Within the Server 7-54
Replicate a Table with a Trigger 7-55
Compute Derived Data Within the Server 7-56
Compute Derived Values with a Trigger 7-57
Log Events with a Trigger 7-58
Benefits of Database Triggers 7-59
Summary 7-60
Practice Overview 7-61

8 Managing Subprograms

Objectives 8-2
System Privileges Requirements 8-3
Example 8-4
Managing Stored PL/SQL Objects 8-5
USER_OBJECTS 8-6
List All Procedures and Functions 8-7
USER_SOURCE 8-8
List the Code of Procedures and Functions 8-9
List Code of Stored Procedures in Oracle Procedure Builder 8-10
Detecting Compile Errors Using the Stored Program Unit Editor 8-12
USER_ERRORS 8-13
Detecting Compilation Errors: Example 8-14
List Compilation Errors Using USER_ERRORS 8-15
List Compilation Errors Using SHOW ERRORS 8-16
USER_TRIGGERS 8-17
Describing PL/SQL Objects in SQL*Plus 8-18
Understanding Dependencies 8-20
Direct Dependency 8-21
Indirect Dependency 8-22
Local Dependencies 8-23
Remote Dependencies 8-24
A Scenario of Local Dependencies 8-25
Displaying Direct Dependencies using USER_DEPENDENCIES 8-26
Displaying Direct and Indirect Dependencies 8-27
DEPTREE View 8-28
IDEPTREE View 8-29

- Another Scenario of Local Dependencies 8-30
- A Scenario of Local Naming Dependencies 8-31
- Remote Dependencies 8-32
- Concepts of Remote Dependencies 8-33
- Setting the Remote Dependency Mode Parameter 8-34
- Remote Dependencies and Timestamp Mode: Scenario 8-35
- Remote Procedure B Compiled at 8:00 8-36
- Local Procedure A Compiles at 9:00 8-37
- Procedure A Is Invoked at 10:00; Procedure B Has Not Recompiled Since 8:00 8-38
- Procedure A Is Invoked at 12:00; Procedure B Has Recompiled at 11:00 (Local Time) 8-39
- Recompiling a PL/SQL Program Unit 8-40
- Recompilation of Procedures 8-41
- Debugging Using The DBMS_OUTPUT Package 8-45
- How to Use DBMS_OUTPUT 8-46
- Debugging a Stored Procedure Using DBMS_OUTPUT 8-47
- Debug Output 8-48
- Debugging Subprograms Using Procedure Builder 8-49
- Creating Breakpoints 8-50
- Using Debugging Levels 8-51
- Controlling Program Unit Execution 8-52
- Summary 8-53
- Practice Overview 8-56

9 Working with Object Types

- Objectives 9-2
- What Is an Object Type? 9-3
- The Structure of an Object Type 9-4
- Creating an Object Type Specification 9-5
- Creating an Object Type Body 9-6
- Example: Creating an Object Type Specification 9-7
- Example: Creating an Object Type Body 9-8
- Example: Creating a Complex Object Type Specification 9-9
- Example: Creating a Complex Object Type Body 9-10
- Obtaining Descriptions of Object Type Methods from the Data Dictionary 9-11
- Example: Calling Object Methods 9-12
- Using the Constructor Method 9-13
- Manipulating Objects 9-14
- Manipulating Objects: Selecting 9-15
- Manipulating Objects: Inserting 9-16
- Manipulating Objects: Updating 9-17
- Manipulating Objects: Deleting 9-18
- Summary 9-19
- Practice Overview 9-20

10 Manipulating Large Objects

Objectives 10-2

Overview 10-3

Contrasting LONG and LOB Datatypes 10-4

Anatomy of a LOB 10-5

Internal LOBs 10-6

Creating a Table with LOBs: Syntax 10-7

External LOBs 10-8

The Directory Alias 10-9

Managing LOBs 10-10

Managing BFILES 10-11

Example: Populating LOBs Using SQL 10-12

Populating LOBs with PL/SQL 10-13

Example: Removing LOBs 10-14

Working with the DBMS_LOB Package 10-15

The DBMS_LOB Package 10-16

DBMS_LOB WRITE and READ 10-18

Summary 10-19

Practice Overview 10-20

A Practices Solutions

B Table Descriptions and Data

Preface

Profile

Before You Begin This Course

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

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 introduced.

Related Publications

Oracle Publications

Title	Part Number
<i>Oracle8 SQL Reference</i>	<i>A-58225-01</i>
<i>PL/SQL User's Guide and Reference, Release 8.0</i>	<i>A-58236</i>
<i>Oracle8 Application Developer's Guide</i>	<i>A-58241</i>
<i>Oracle8 Concepts</i>	<i>A-58227-01</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

What follows are two lists of typographical conventions 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 SELECT command to view information stored in the LAST_NAME column of the EMP table.
Lowercase, italic	Filenames, syntax variables, usernames, passwords	where: <i>role</i> is the name of the role 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;

Typographic Conventions (continued)

Symbols Used in This Document



Indicates guidance relating to the subject matter, such as hints or advice.



Identifies a reference to other publications.



Identifies a reference to a Web site.



Indicates a warning; for example, “When deleting rows, word your WHERE clause carefully.”



Indicates that an expanded discussion on each numbered step is presented on the pages following the steps.



Indicates that a demonstration will be run at this point in the lesson.

I

Introduction

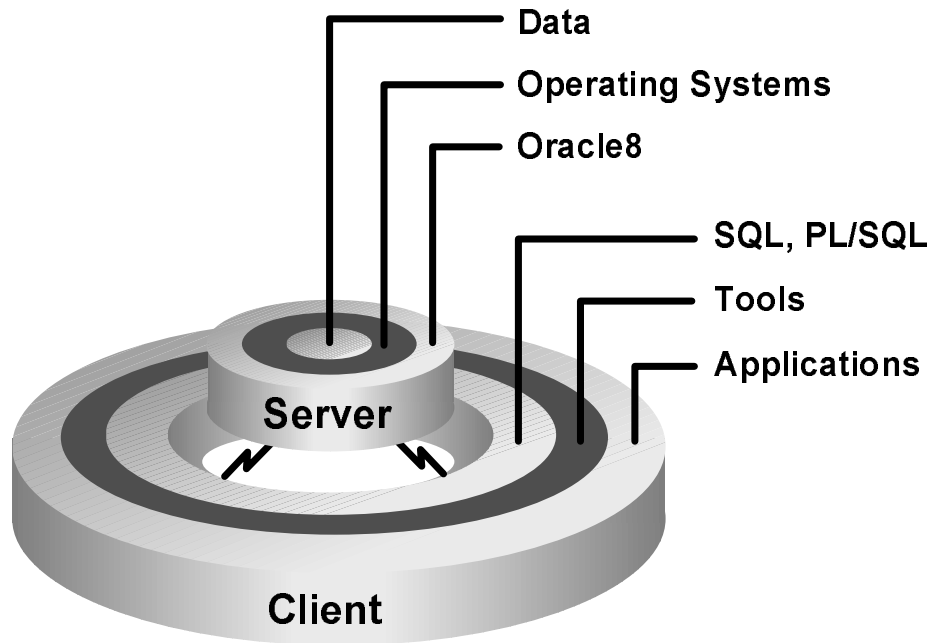
Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

Course Objectives

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

- **Describe the PL/SQL development environments**
- **Create, execute, and maintain procedures, functions, packages, database triggers, and object types**
- **Manage PL/SQL program constructs**
- **Describe Oracle supplied packages**
- **Manipulate Large Objects (LOB)**

Overview



I-3

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Course Overview

- **Database procedures allow for modularized application development using database objects such as the following:**
 - **Procedures and Functions**
 - **Packages**
 - **Database triggers**
 - **Object types**
- **Modular applications improve**
 - **Functionality**
 - **Security**
 - **Overall performance**

I-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1

Overview of PL/SQL

Objectives

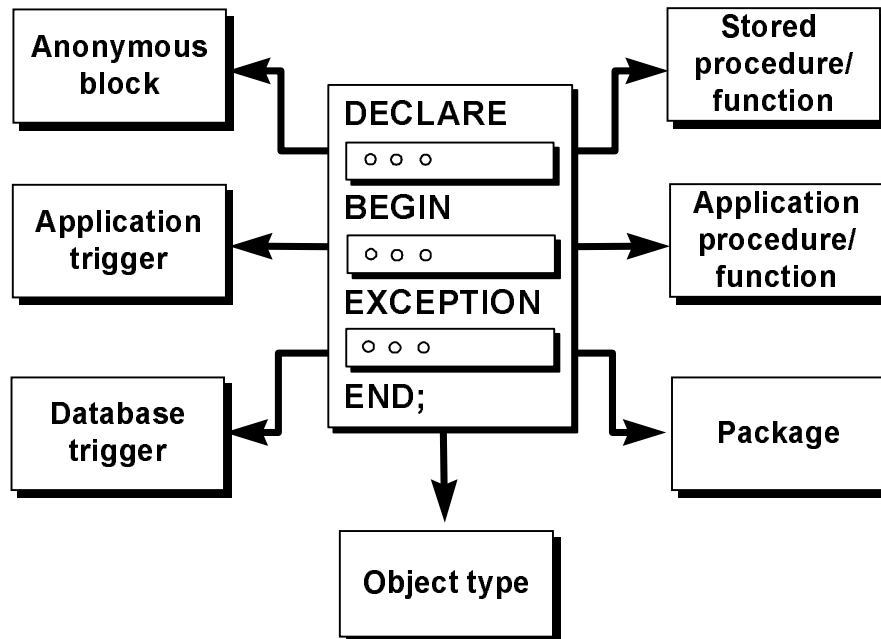
After completing this lesson, you should be able to do the following:

- **Distinguish between anonymous PL/SQL blocks and named PL/SQL blocks (subprograms)**
- **Describe the PL/SQL development environments**

Lesson Aim

PL/SQL supports many different program constructs. This lesson compares anonymous blocks with named PL/SQL blocks. It also introduces you to the different development environments.

PL/SQL Program Constructs



1-3

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

PL/SQL Program Constructs

The diagram above displays the variety of different PL/SQL program constructs using the basic PL/SQL block. They are available based on the environment where they are stored. In general, a block is either an anonymous block or a named block (subprogram).

PL/SQL Block Structure

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

Block Structure for Anonymous PL/SQL Blocks

- **DECLARE (optional)**
Define PL/SQL objects to be used within this block
- **BEGIN (mandatory)**
Executable statements
- **EXCEPTION (optional)**
What to do if the executable action causes an error condition
- **END; (mandatory)**

1-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Anonymous Blocks

These are blocks without names. You declare them at the point in an application where they are to be run, and passed to the PL/SQL engine for execution at runtime.

- The BEGIN and END elements are mandatory, and enclose the body of actions to be performed. This section is often referred to as the executable section of the block.
- The DECLARE section is optional. In it, you define PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block.
- The EXCEPTION section traps predefined error conditions. In it, you define actions to take if the specified condition arises. The EXCEPTION section is optional and must immediately precede the END keyword.



Section 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

Header

IS

Declaration section

BEGIN

Executable section

EXCEPTION

Exception section

END;

1-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Subprograms

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called procedures and functions.

- The header is relevant for named blocks only, and determines the way that the program unit will be called, or invoked. The header determines the PL/SQL block type (procedure, function), it contains the name of the program unit, the parameter list (if any), and a RETURN clause (for functions only).
- The BEGIN and END elements are mandatory, and enclose the body of actions to be performed. This section is often referred to as the executable section of the block.
- The DECLARE section is optional. In it, you define PL/SQL objects such as variables, constants, cursors, and user-defined exceptions that you want to reference within the block.
- The EXCEPTION section traps predefined error conditions. In it, you define actions to take if the specified condition arises. The EXCEPTION section is optional and must immediately precede the END keyword.

Benefits of Subprograms

- **Improved maintenance**
- **Improved data security and integrity**
- **Improved performance**

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

Improved 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 runtime by parsing at compile time.
- Reduce the number of calls to the database and decrease network traffic by bundling commands.

Development Environments

- **SQL*Plus**

Utilizes the PL/SQL engine in the Oracle Server

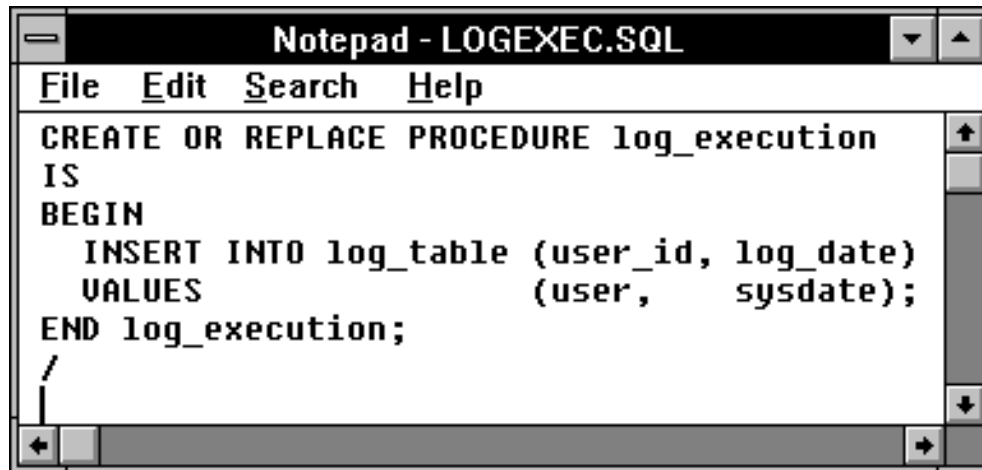
- **Procedure Builder**

Utilizes the PL/SQL engine in the client tool or in the Oracle Server

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 itself.

You will learn about the two main development environments for PL/SQL: SQL*Plus and Developer/2000 Procedure Builder.

Developing Procedures and Functions Using SQL*Plus

A screenshot of a Notepad window with the title bar 'Notepad - LOGEXEC.SQL'. The menu bar includes 'File', 'Edit', 'Search', and 'Help'. The text area contains the following SQL code:

```
CREATE OR REPLACE PROCEDURE log_execution
IS
BEGIN
    INSERT INTO log_table (user_id, log_date)
    VALUES                (user,      sysdate);
END log_execution;
/
```

The code is formatted with indentation for the procedure body. The window has standard scrollbars on the right and bottom.

1-8

Copyright © Oracle Corporation, 1998. All rights reserved.

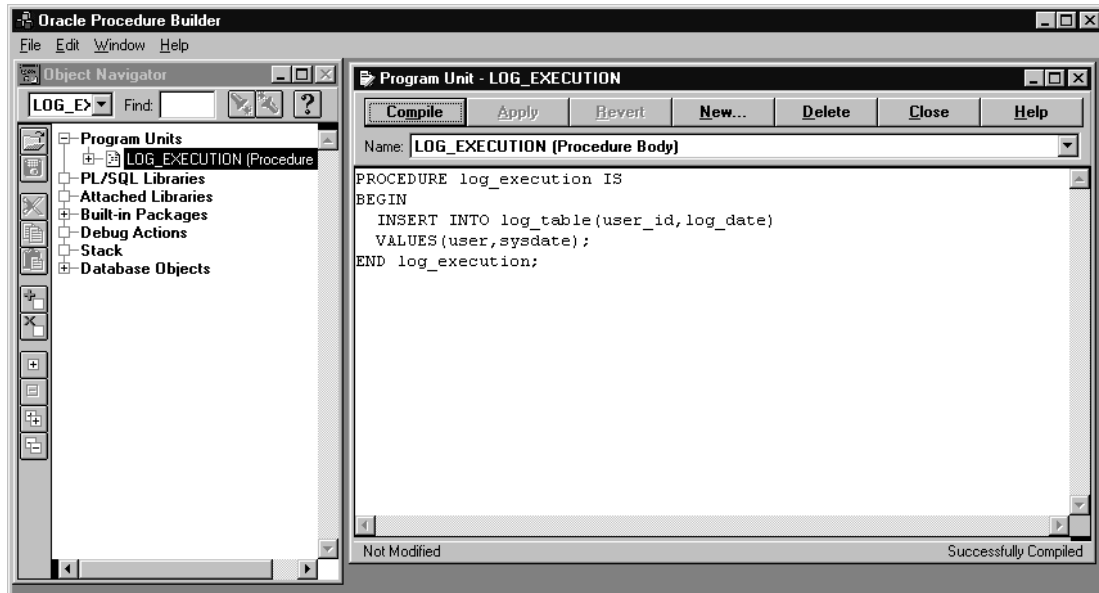
ORACLE®



Start SQL*Plus

The example above creates a stored procedure, without any arguments, to record the username and current date in a database table.

Developing Procedures and Functions Using Oracle Procedure Builder



1-9

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



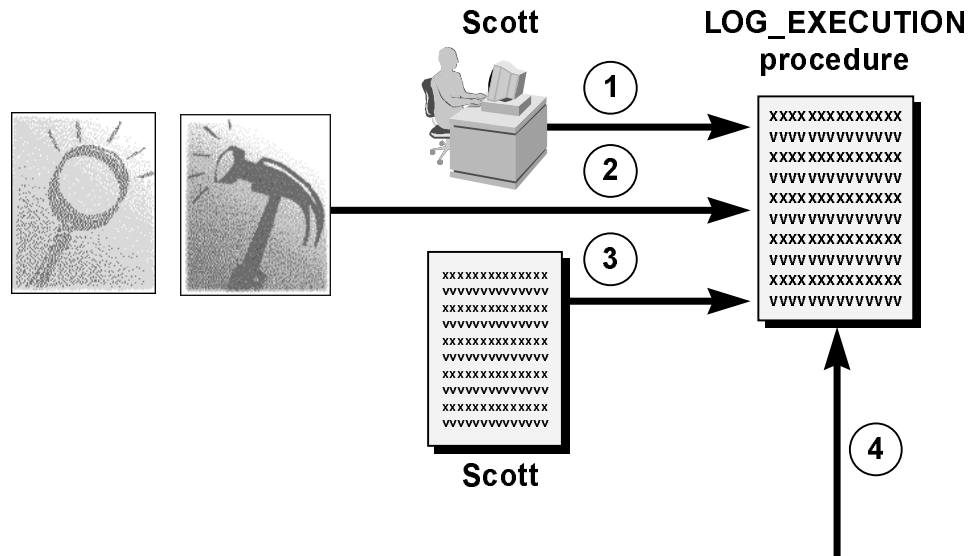
Start Developer/2000 Procedure Builder from Windows

Procedure Builder is a tool you can use to create, execute, and debug PL/SQL program units used in your application tools, such as a form or report, or on the Oracle Server through its graphical interface.

Procedure Builder's development environment contains a built-in editor for you to create or edit subprograms. You can compile, test, and debug your code.

Application partitioning is available to assist you with the distribution of logic between the client and the server. Developers can "drag and drop" a PL/SQL program unit between the client and the server.

Invoking Stored Procedures and Functions



1-10

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

You can invoke a previously created procedure or function from a variety of environments: SQL*Plus, Developer/2000, Oracle Discoverer, another stored procedure, and many other Oracle tools and precompiler applications.

1. SQL*Plus

```
SQL> EXECUTE log_execution
```

2. Developer/2000 or Oracle Discoverer; Example: Pre-Form trigger in a Form Builder application

Trigger code: log_execution;

3. Another procedure

```
CREATE OR REPLACE PROCEDURE fire_emp (v_id IN
emp.empno%TYPE) IS
BEGIN
    log_execution;
    DELETE FROM emp
    WHERE empno = v_id;
END fire_emp;
```

4. Other environments

Summary

- **Anonymous blocks are unnamed PL/SQL blocks.**
- **Subprograms are named PL/SQL blocks, declared as either procedures or functions.**
- **There are two main PL/SQL development environments:**
 - **SQL*Plus**
 - **Procedure Builder**

2

Working with Procedure Builder

Objectives

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

- **Describe the features of 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**

2-2

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Lesson Aim

In this lesson you learn about the features of the Procedure Builder tool, and how they can be used to create, compile, and invoke subprograms.

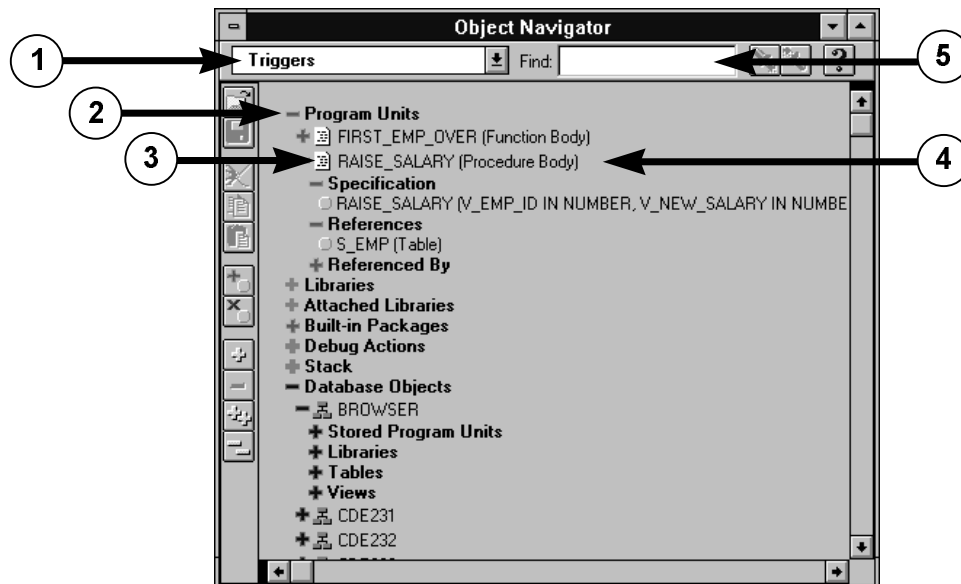
Overview

Component	Description
Object Navigator	Manage PL/SQL constructs; perform debug actions
PL/SQL Interpreter	Debug PL/SQL code; evaluate PL/SQL code in real time
Program Unit Editor	Create and edit PL/SQL source code
Stored Program Unit Editor	Create and edit server-side PL/SQL source code.
Database Trigger Editor	Create and edit database triggers

Procedure Builder Overview

Procedure Builder is an integrated development environment. It enables you to edit, compile, test, and debug client and server-side PL/SQL program units within a single tool.

Procedure Builder Components: The Object Navigator



2-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Navigator drop down list
2. Subobject indicator
3. Type icon
4. Object name
5. Find field

The Object Navigator

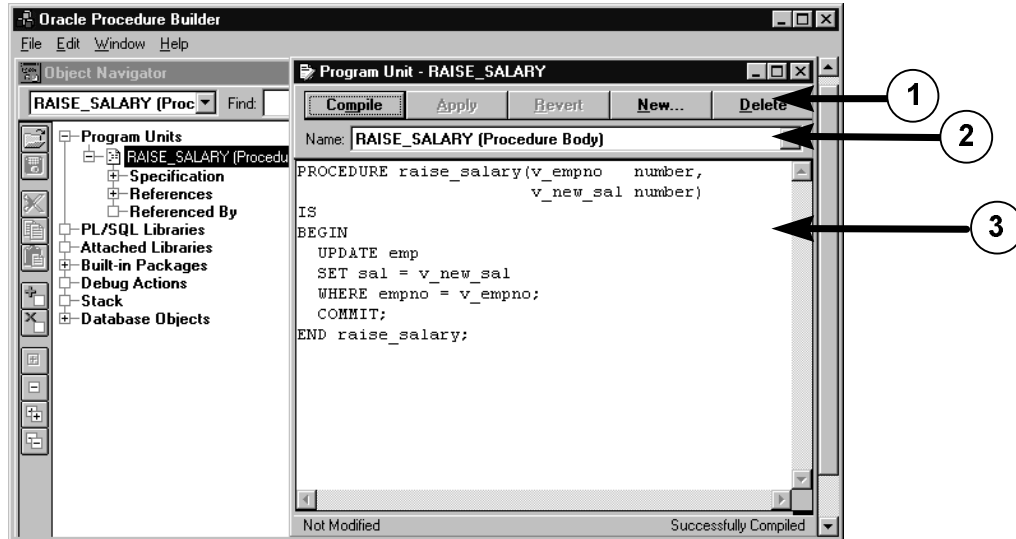
The Object Navigator provides an outline style interface with which you can browse objects, view the relationships between them, and edit properties.

You can create and manipulate all your PL/SQL program units, libraries, debug actions, and variables from the Object Navigator.

The Object Navigator also allows you to expand and collapse nodes; cut and paste; search for an object; and drag and drop PL/SQL program units between the client and the server side or vice versa.

Note: Make sure you connect to the database. Choose File—>Connect.

Procedure Builder Components: The Program Unit Editor



2-5

Copyright © Oracle Corporation, 1998. All rights reserved.

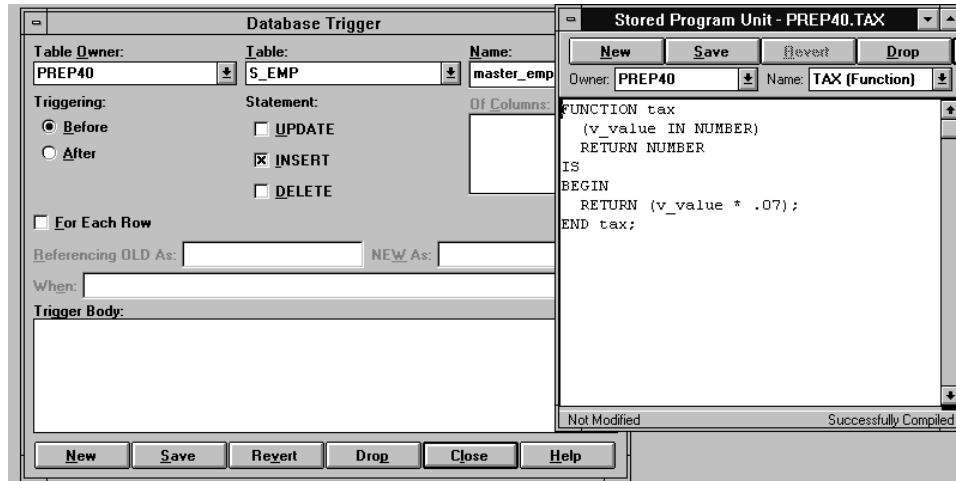
ORACLE®

1. Compile, Apply, Revert, New, Delete, Close, and Help buttons
2. Name drop down list
3. Source text pane

The Program Unit Editor

Use the Program Unit Editor to edit, compile, and browse warning and error messages during application development.

Procedure Builder Components: The Stored Program Unit Editor



2-6

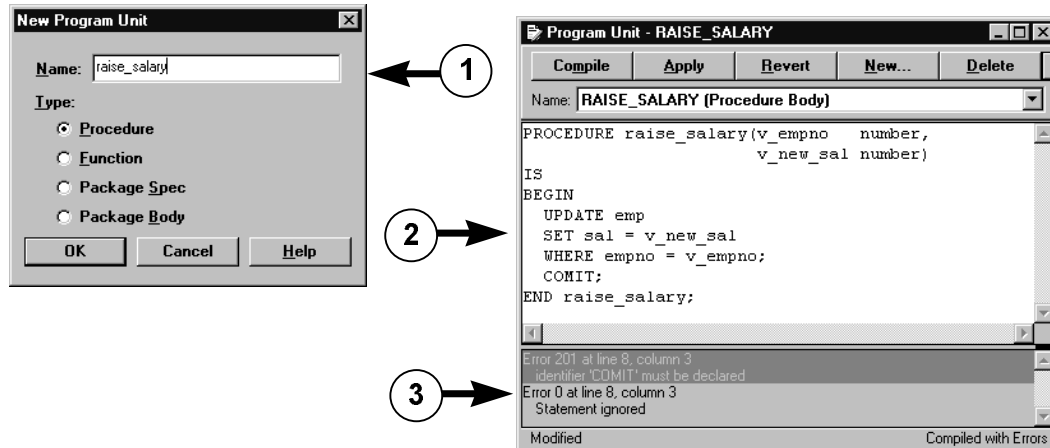
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

The Stored Program Unit Editor

The Stored Program Unit Editor is a GUI environment for editing server-side PL/SQL constructs. The save operation submits the source text to the server-side PL/SQL compiler.

Creating a Program Unit on the Client Side



2-7

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Create the program unit.
2. Write the code.
3. Compilation errors.

Creating a 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 and select the subprogram type. 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.

Navigating Compilation Errors

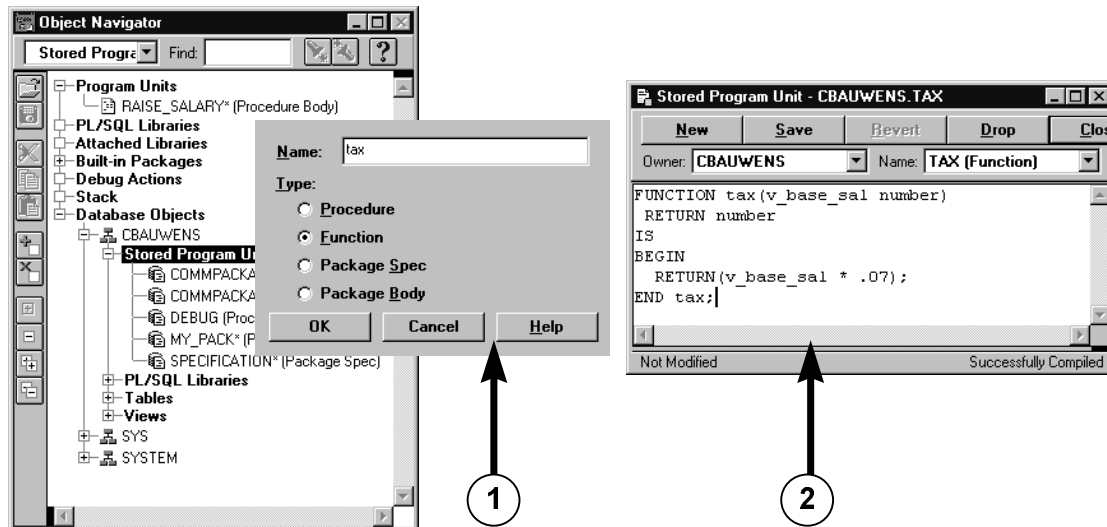
1. Click the Compile button in the Program Unit Editor.
Error messages generated during compilation are displayed in the Compilation Message pane at the bottom of the window.
2. Click an error message.

The cursor moves to the location of the error in the source pane.



Program Units that reside in the Program Units node are lost when you exit Procedure Builder. Either export them to a file, or store them in the database.

Creating a Program Unit on the Server Side



2-8

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Create the program unit.
2. Write the code.

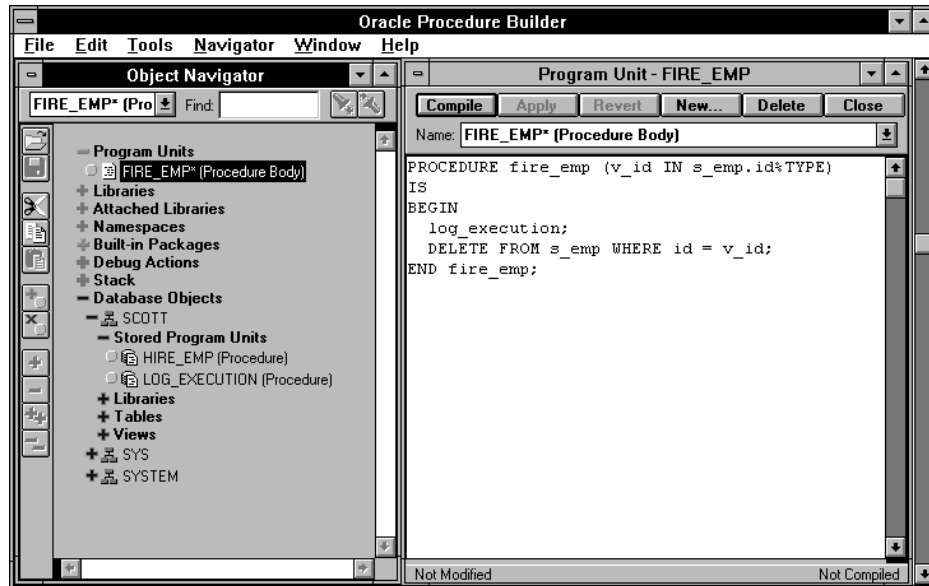
Creating a Program Unit

1. Select the Database Objects node in the Object Navigator.
2. Expand the schema and click on Stored Program Units.
3. Click on the create button. The Program Unit dialog box appears.
4. Enter the name and select the subprogram type. Click the OK button to accept the entries.
5. The Stored Program Unit Editor will be displayed. It contains the skeleton for your PL/SQL construct. The cursor is automatically positioned on the line beneath the BEGIN keyword.

Navigating Compilation Errors

1. Click the Compile button in the Stored Program Unit Editor.
Error messages generated during compilation are displayed in the Compilation Message pane.
2. Click an error message.
The cursor moves to the location of the error in the Source pane.

Transferring Program Units Between Client and Server



2-9

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

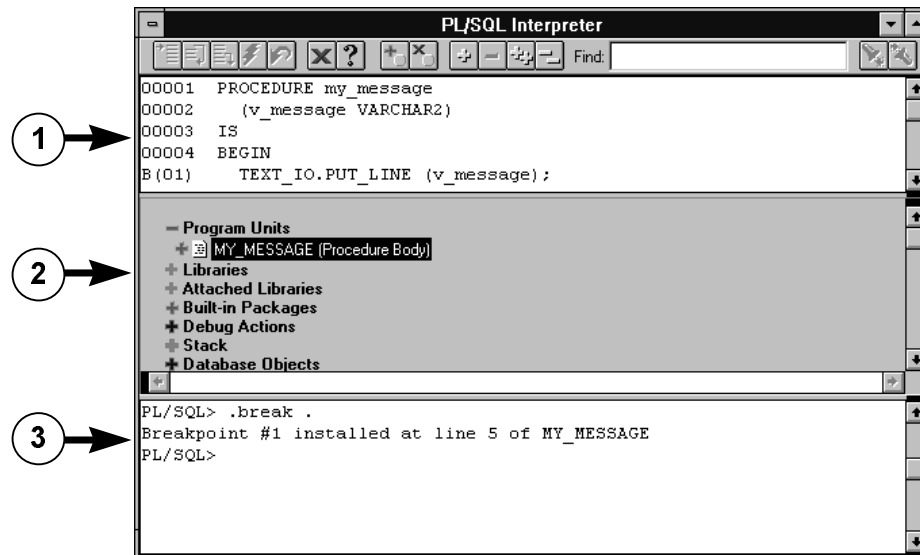
Application Partitioning

You have just learned that Procedure Builder can create PL/SQL program units on both the client and the server. Using Procedure Builder, you can also copy program units created on the client to stored program units on the server (or vice versa). 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



2-10

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Source pane
2. Navigator pane
3. Interpreter pane

The Interpreter

- Monitor debugging actions by using the program execution buttons.
- Execute PL/SQL program units and Procedure Builder commands by using the interpreter pane.
- Display and debug PL/SQL program units by using the Source pane.

Invoking PL/SQL Program Units

From the PL/SQL prompt in the Interpreter Source pane, enter the name of the PL/SQL unit you want to invoke. Terminate with a semicolon.

```
PL/SQL> construct_name [argument1|argument2,...];
```

Executing SQL Statements

From the PL/SQL prompt in the Interpreter Source pane, enter your SQL statement. Terminate with a semicolon.

```
PL/SQL> CREATE TABLE test(id NUMBER) ;
PL/SQL> SELECT * FROM dept;
PL/SQL> INSERT INTO DEPT
      +> VALUES (50, 'EDUCATION', 'DALLAS') ;
```

Develop PL/SQL Program Units 2-10

Summary

- **Advantages of using Procedure Builder**
 - Application partitioning
 - Editors
 - Execution environment
- **Procedure Builder components**
 - Object Navigator
 - Program Unit editors
 - PL/SQL Interpreter

Practice Overview

- **Exploring Procedure Builder menus and windows**
- **Creating a program unit**
- **Executing the program unit**

Practice 2

1. Load and execute a loop counter.
 - a. Launch Procedure Builder. Your instructor will give you the login information.
 - b. From the menu, load the *labs\p2loop.pls* file.
 - c. Examine the code using the Program Unit Editor.
 - d. Compile the procedure. Correct any errors.
 - e. Execute the procedure from the interpreter pane. Pass a numeric value into the procedure as demonstrated below.

```
PL/SQL> my_loop (4) :
```

2. Create a client-side procedure named MY_MESSAGE to print the message you provide at design time. The executable part in your procedure should contain the following line of code:

```
text_io.put_line('Hello world') ;
```

Compile and execute your procedure from the interpreter pane as shown below.

```
PL/SQL> my_message ;
```

3. Add a new department to your DEPT table using the interpreter pane. Verify the insert.
4. Insert a new department using a procedure.
 - a. From the menu, load the *labs\p2ins.pls* file.
 - b. Replace the comment inside the procedure with your INSERT statement.
 - c. Execute the procedure from the interpreter pane.
 - d. Verify the insert.

3

Creating Procedures

Objectives

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

- **Describe the uses of procedures**
- **Create client-side and server-side procedures**
- **Create procedures with arguments**
- **Invoke a procedure**
- **Remove a procedure**

Lesson Aim

In this lesson, you learn how to create and invoke procedures in client-side and server-side environments.

Overview of Procedures

- **A procedure is a named PL/SQL block that performs an action.**
- **A procedure can be stored in the database, as a database object, for repeated execution.**

A procedure is a named PL/SQL block that can take parameters and be invoked. Generally speaking you use a procedure to perform an action. As we have seen in Lesson 1, a procedure has a header, a declarative part, an executable part, and an optional exception-handling part.

Procedures promote reusability and maintainability. Once validated they can be used in any number of applications. If the definition changes, only the procedure is affected, this greatly simplifies maintenance.

Syntax for Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  (argument1 [mode1] datatype1,
   argument2 [mode2] datatype2,
   . . .
IS [AS]
PL/SQL Block;
```

3-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



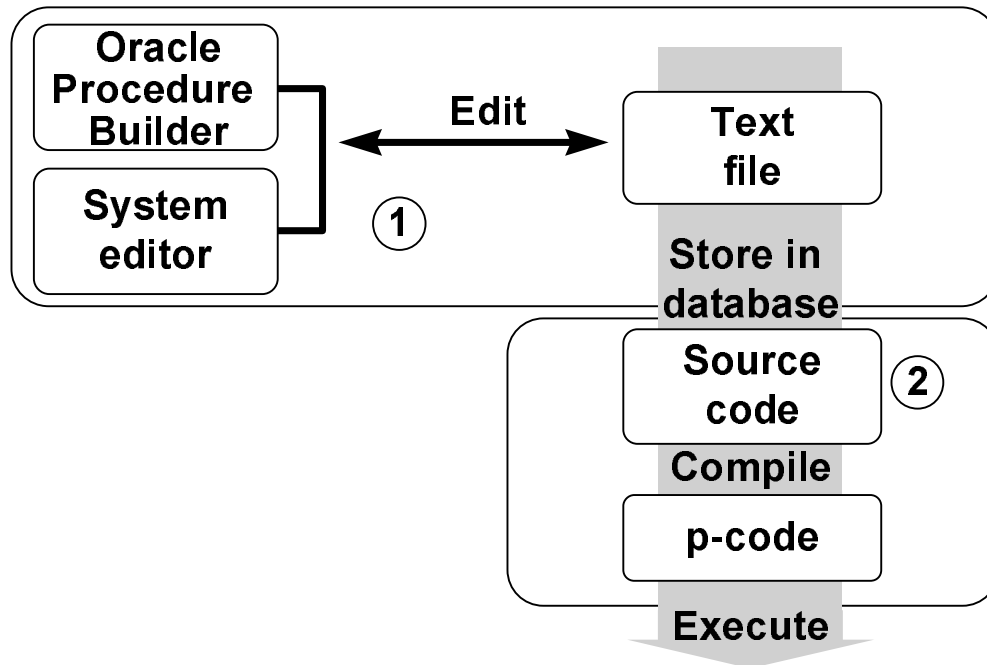
You create new procedures with the CREATE PROCEDURE statement, which may declare a list of arguments (sometimes referred to as parameters), and must define the actions to be performed by the standard PL/SQL block.

- 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.

Syntax Definitions

Parameter	Description
<i>procedure_name</i>	Name of the procedure
<i>argument</i>	Name of a PL/SQL variable whose value is passed to, 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>datatype</i>	Datatype of the argument
<i>PL/SQL block</i>	Procedural body that defines the action performed by the procedure

Developing Stored Procedures



3-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

How to Develop a Stored Procedure

The following are the main steps for developing a stored procedure. The next two pages provide more detail.

1. Select a development environment: Procedure Builder or SQL*Plus.

Write the syntax. If using Procedure Builder, enter the syntax in the Program Unit Editor. If using SQL*Plus, enter the code in a text editor and save it as a script file.

2. Compile the code.

The source code is compiled into *p-code*. If you are using Procedure Builder, click Compile. If you are using SQL*Plus, start the script file.

Creating a Stored Procedure Using SQL*Plus

- 1. Enter the text of the CREATE PROCEDURE statement in a system editor or word processor and save it as a script file (.sql extension).**
- 2. From SQL*Plus, run the script file to compile the source code into p-code and store both in the database.**
- 3. Invoke the procedure from an Oracle Server environment to determine whether it executes without error.**

A script file with the CREATE PROCEDURE (or CREATE FUNCTION) statement allows 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 (or function) that contains any compilation or run time errors. In SQL*Plus, use SHOW ERRORS to see any compilation errors.

Running the CREATE PROCEDURE (or CREATE FUNCTION) statement stores the source code in the data dictionary when the procedure (or function) contains compilation errors.

Creating a Procedure Using Procedure Builder

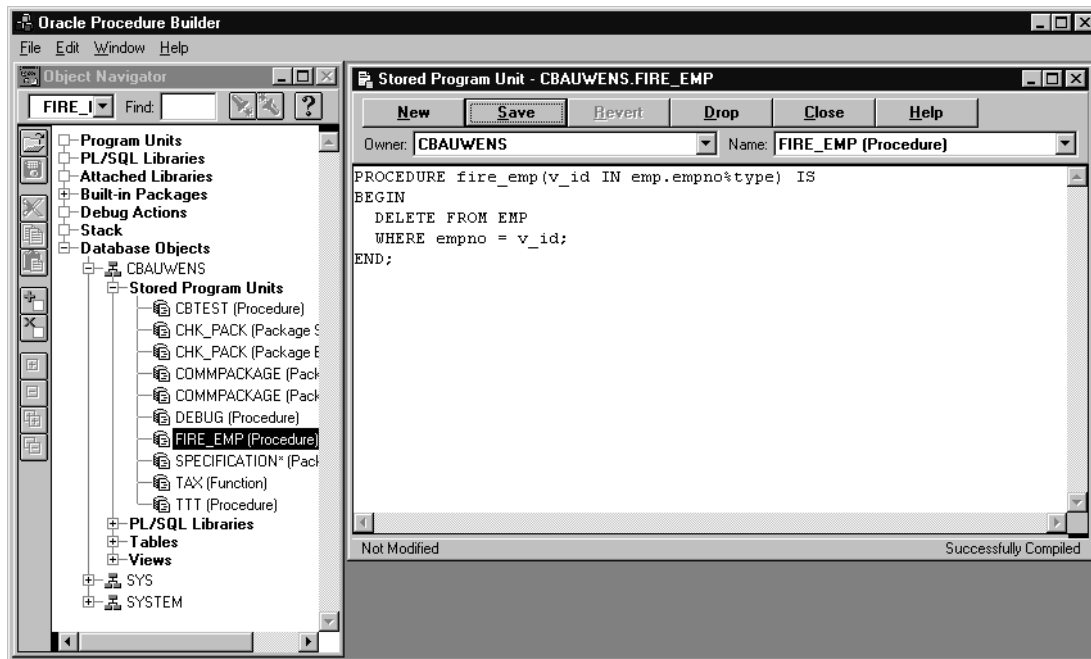
Procedure Builder allows you to:

- **Create a server-side procedure**
- **Create a client-side procedure**
- **Drag and drop procedures between
client and server**

Because there is a PL/SQL engine in the client tool Procedure Builder, you can develop client-side procedures. Using Procedure Builder, you can use the server's PL/SQL engine and develop server-side procedures.

Drag and drop functionality in Procedure Builder allows you to move your procedures easily between the client and the server.

Creating Server-Side Procedures Using Procedure Builder



3-8

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

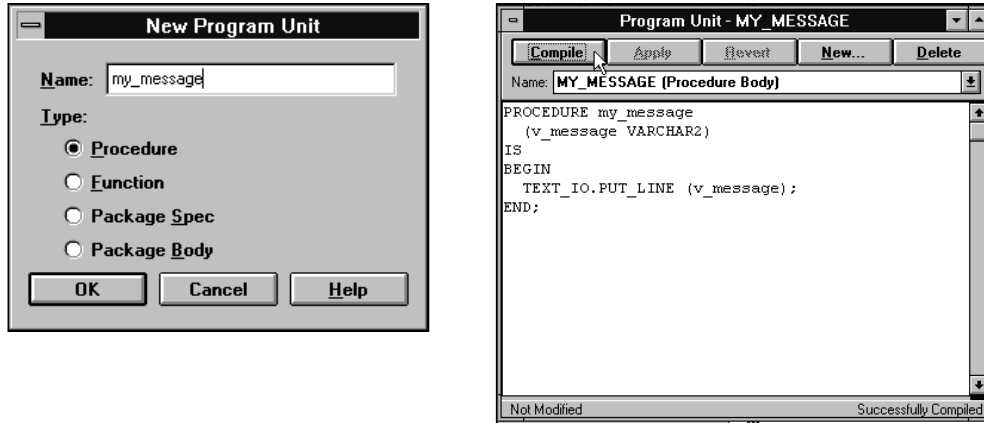
How to Create a Server-Side Procedure Using Procedure Builder

1. Choose File—>Connect, and 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.



The CREATE keyword is not necessary in Procedure Builder.

Creating Client-Side Procedures Using Procedure Builder



3-9

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

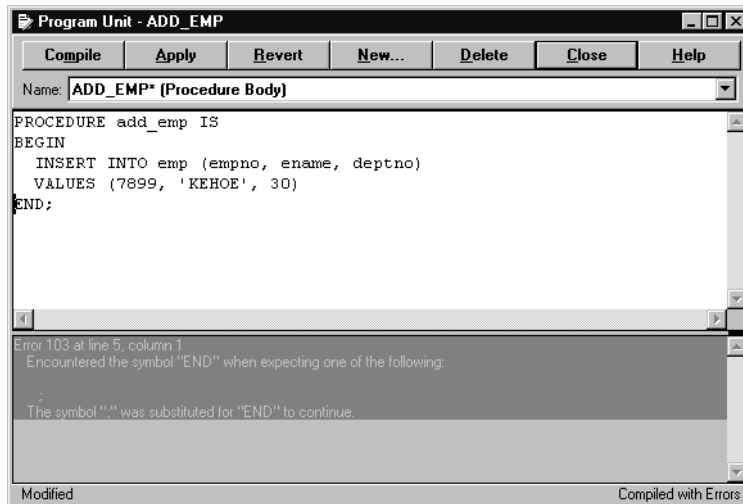
How to Create a Client-Side Procedure Using Procedure Builder

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. Click an error message to position the cursor at 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: Menu Edit—>Export.



The CREATE keyword is not necessary in Procedure Builder.

Navigating Compilation Errors in Procedure Builder



3-10

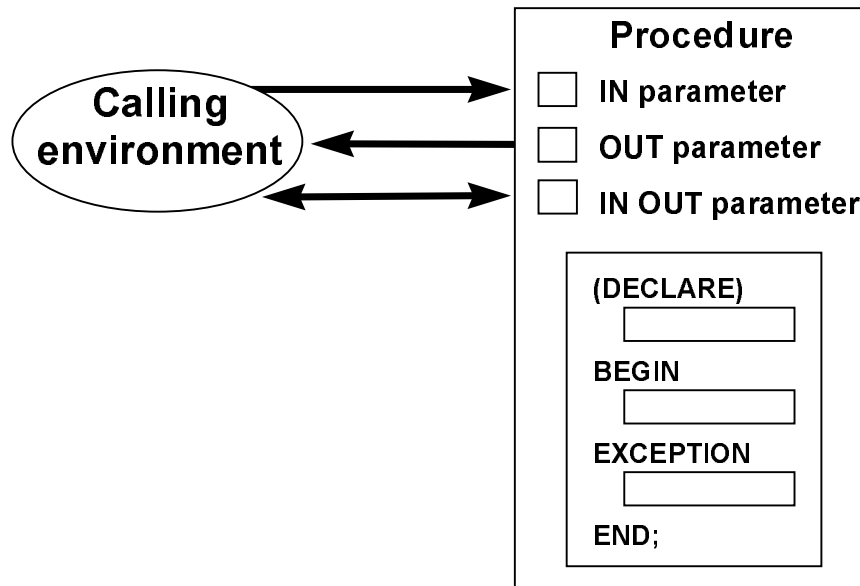
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

How to Resolve Compilation Errors

1. Click Compile.
2. Click an error message.
The cursor moves to the location of the error in the source pane.
3. Resolve the error and click Compile.

Procedural Parameter Modes



3-11

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

You can transfer values to and from the calling environment through parameters. Choose 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.

DATATYPE can only be the %TYPE definition, %ROWTYPE definition, or an explicit datatype 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

Parameter Modes for Formal Parameters

IN	OUT	IN OUT
Default	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

3-12

Copyright © Oracle Corporation, 1998. All rights reserved.

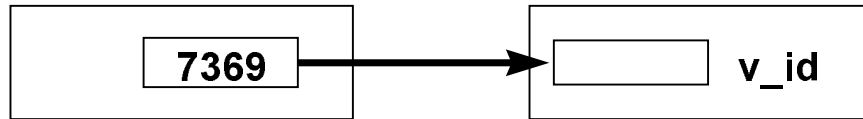
ORACLE®

Procedural 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.

IN Parameters: Example



```
SQL> CREATE OR REPLACE PROCEDURE raise_salary
  2  (v_id in emp.empno%TYPE)
  3  IS
  4  BEGIN
  5      UPDATE emp
  6      SET      sal = sal * 1.10
  7      WHERE   empno = v_id;
  8  END raise_salary;
  9  /
Procedure created.
```

```
SQL> EXECUTE raise_salary (7369)
PL/SQL procedure successfully completed.
```

3-13

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

The example above shows a procedure with one IN parameter. Running this statement in SQL*Plus creates the RAISE_SALARY procedure. When invoked, RAISE_SALARY takes the parameter for the employee number and updates the employee's record with a salary increase of 10 percent.

To invoke a procedure in SQL*Plus, use the EXECUTE command.

```
SQL> execute raise_salary (7369)
```

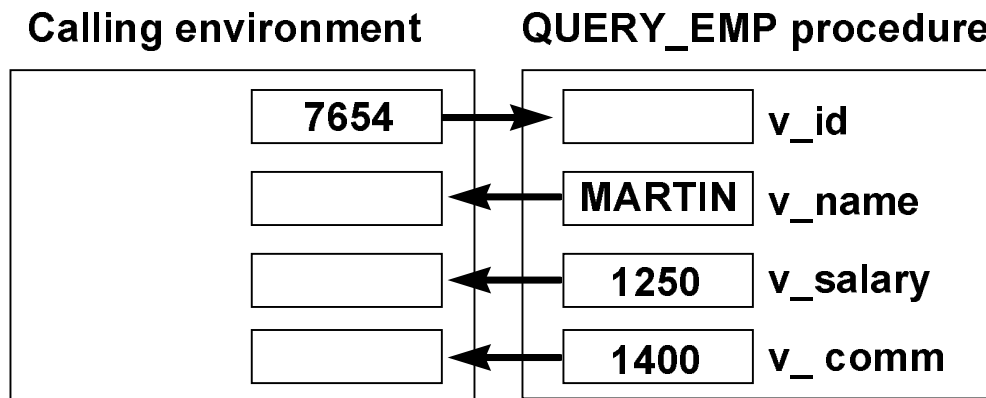
To invoke a procedure from the Procedure Builder environment, use a direct call. At the Procedure Builder Interpreter prompt, enter the procedure name and actual parameters.

```
PL/SQL> raise_salary (7369);
```



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



3-14

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Create a procedure with OUT parameters to retrieve information about an employee.

OUT Parameters: Example

```
SQL> CREATE OR REPLACE PROCEDURE query_emp
 1  (v_id      IN    emp.empno%TYPE,
 2   v_name    OUT   emp.ename%TYPE,
 3   v_salary  OUT   emp.sal%TYPE,
 4   v_comm    OUT   emp.comm%TYPE)
 5  IS
 6  BEGIN
 7      SELECT  ename, sal, comm
 8      INTO    v_name, v_salary, v_comm
 9      FROM    emp
10      WHERE   empno = v_id;
11  END query_emp;
12  /
```

Example

Run the above statement to create the QUERY_EMP procedure. This procedure has four formal parameters, three of them are OUT parameters that will return a value to the calling environment.

OUT Parameters and SQL*Plus

```
SQL> START emp_query.sql  
Procedure created.
```

```
SQL> VARIABLE g_name          varchar2(15)  
SQL> VARIABLE g_salary        number  
SQL> VARIABLE g_comm          number
```

```
SQL> EXECUTE query_emp (7654, :g_name, :g_salary,  
2  :g_comm)  
PL/SQL procedure successfully completed.
```

```
SQL> PRINT g_name  
G_NAME  
-----  
MARTIN
```

View the Value of OUT Parameters with SQL*Plus

1. Create host variables in SQL*Plus using the VARIABLE syntax.
2. 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 syntax.
3. To view the values passed from the procedure to the calling environment, use the PRINT syntax. Only one variable can be supplied to each PRINT command.

The above example shows the value of the variable *g_name* passed back to the the calling environment. Additional PRINT commands are needed to view the values for *g_salary* and *g_comm*.

OUT Parameters and Procedure Builder

```
PL/SQL> .CREATE CHAR g_name LENGTH 10
PL/SQL> .CREATE NUMBER g_salary PRECISION 4
PL/SQL> .CREATE NUMBER g_comm PRECISION 4
PL/SQL> RAISE_SALARY (7654, :g_name, :g_sal,
    +> :g_comm) ;
PL/SQL> TEXT_IO.PUT_LINE (:g_name || ' earns ' ||
    +> TO_CHAR(:g_sal) || ' and a commission of '
    +> || TO_CHAR(:g_comm)) ;
MARTIN      earns 1250 and a commission of 1400
```

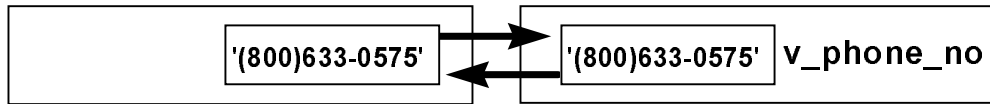
Viewing the Value of OUT Parameters with Procedure Builder

1. Create host variables using the .CREATE syntax.
2. Invoke the QUERY_EMP procedure, supplying these host variables as the OUT parameters. Note the use of the colon (:) to reference the host variables.
3. To view the values passed from the procedure to the calling environment, use the PUT_LINE procedure in the TEXT_IO package.

IN OUT Parameters

Calling environment

FORMAT_PHONE procedure



```
SQL> CREATE OR REPLACE PROCEDURE format_phone
 2  (v_phone_no IN OUT VARCHAR2)
 3  IS
 4  BEGIN
 5      v_phone_no := '(' || SUBSTR(v_phone_no,1,3) ||
 6                      ')' || SUBSTR(v_phone_no,4,3) ||
 7                      '-' || SUBSTR(v_phone_no,7);
 8  END format_phone;
 9  /
```

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

Invoking FORMAT_PHONE from SQL*Plus

```
SQL> VARIABLE g_phone_no varchar2(15)

SQL> BEGIN :g_phone_no := '8006330575'; END;
2 /
PL/SQL procedure successfully completed.

SQL> EXECUTE format_phone (:g_phone_no)
PL/SQL procedure successfully completed.

SQL> PRINT g_phone_no
```

```
G_PHONE_NO
-----
(800) 633-0575
```

Viewing IN OUT Parameters with SQL*Plus

1. Create a host variable using the VARIABLE syntax.
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 syntax.
4. To view the value passed back to the calling environment, use the PRINT syntax.

Invoking FORMAT_PHONE from Procedure Builder

```
PL/SQL> .CREATE CHAR g_phone_no LENGTH 15
PL/SQL> BEGIN
    +> :g_phone_no := '8006330575';
    +> END;
PL/SQL> FORMAT_PHONE (:g_phone_no);
PL/SQL> TEXT_IO.PUT_LINE (:g_phone_no);
(800) 633-0575
```

Viewing IN OUT Parameters with Procedure Builder

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

Methods for Passing Parameters

- **Positional**
- **Named**
- **Combination**

For a procedure containing multiple parameters, you can use a number of methods to specify the values of the parameters:

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

Passing Parameters: Example Procedure

```
SQL> CREATE OR REPLACE PROCEDURE add_dept
  1  (v_name  IN dept.dname%TYPE  DEFAULT 'unknown',
  2   v_loc   IN dept.loc%TYPE     DEFAULT 'unknown')
  3  IS
  4  BEGIN
  5      INSERT INTO dept
  6      VALUES (dept_deptno.NEXTVAL, v_name, v_loc);
  7  END add_dept;
  8  /
```

Example

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

Examples of Passing Parameters

```
SQL> begin
  2  add_dept;
  3  add_dept ( 'TRAINING', 'NEW YORK');
  4  add_dept ( v_loc => 'DALLAS', v_name =>
                'EDUCATION') ;
  5  add_dept ( v_loc => 'BOSTON') ;
  6  end;
  7  /
PL/SQL procedure successfully completed.
```

```
SQL>      SELECT * FROM dept;
```

DEPTNO	DNAME	LOC
-----	-----	-----
...
41	unknown	unknown
42	TRAINING	NEW YORK
43	EDUCATION	DALLAS
44	unknown	BOSTON

The anonymous block above shows the different ways the ADD_DEPT procedure can be invoked, and the results.

Invoking a Procedure from an Anonymous PL/SQL Block

```
DECLARE
  v_id NUMBER := 7900;
BEGIN
  raise_salary(v_id);    --invoke procedure
COMMIT;
...
END;
```

Example

You have already seen how to invoke an independent procedure from the two main PL/SQL development environments, SQL*Plus and Procedure Builder.

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

Invoke the RAISE_SALARY procedure from an anonymous PL/SQL block.

Invoking a Procedure from a Stored Procedure

```
SQL> CREATE OR REPLACE PROCEDURE process_emps
 2  IS
 3      CURSOR emp_cursor IS
 4          SELECT empno
 5              FROM emp;
 6  BEGIN
 7      FOR emp_rec IN emp_cursor LOOP
 8          raise_salary(emp_rec.empno);  --invoke procedure
 9      END LOOP;
10  COMMIT;
11  END process_emps;
12 /
```

Example

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

Removing Procedures

- **Using SQL*Plus:**
 - Drop a server-side procedure**
- **Using Procedure Builder:**
 - Drop a server-side procedure**
 - Drop a client-side procedure**

Removing Server-Side Procedures

Using SQL*Plus:

- **Syntax**

```
DROP PROCEDURE procedure_name
```

- **Example**

```
SQL> DROP PROCEDURE raise_salary;  
Procedure dropped.
```

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



Rollback is not possible after executing a data definition language (DDL) command such as DROP PROCEDURE.

Removing Server-Side Procedures

Using Procedure Builder:

1. **Select File—>Connect and enter your username, password, and database.**
2. **Expand the Database Objects node.**
3. **Expand the schema of the owner of the procedure.**

3-28

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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 RAISE_SALARY?



In the Stored Program Units Editor, you can also click **DROP** to remove the procedure from the server.

Removing Server-Side Procedures

- 4. Click the Stored Program Units node under that schema.**
- 5. Click the procedure you want to drop.**
- 6. Click delete in the Object Navigator.**
- 7. Click Yes to drop the procedure.**

Removing Client-Side Procedures

Using Procedure Builder:

1. Expand the Program Units node in the Object Navigator.
2. Click the procedure you want to remove.
3. Click delete in the Object Navigator.

3-30

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



Follow the above steps to remove the 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 go through the operating system.

Summary

- **A procedure is a named PL/SQL block that performs an action.**
- **Use parameters to pass data from the calling environment to the procedure.**
- **Procedures can be invoked from any tool or language that supports PL/SQL.**
- **Procedures can serve as building blocks for an application.**

Practice Overview

Creating stored procedures to implement a variety of data manipulation and query routines

3-32

Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

Use either SQL*Plus or Procedure Builder to do the practices.

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

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

Practice 3

1. Create and invoke the **ADD_PROD** procedure and consider the results.
 - a. Create a procedure called **ADD_PROD** to insert a new product into the **PRODUCT** table.
 - b. Compile the code, invoke the procedure, and then query the **PRODUCT** table to view the results.
 - c. Invoke your procedure again, passing a product Id of 100860. What happens and why?

2. Create a procedure called **UPD_PROD** to modify a product in the **PRODUCT** table.
 - a. Create a procedure called **UPD_PROD** to update the product description. Include the necessary exception handling.
 - b. Compile the code, invoke the procedure, and then query the **PRODUCT** table to view the results. Also check the exception handling by trying to update a product that does not exist.
3. Create a procedure called **DEL_PROD** to delete a product from the **PRODUCT** table.
 - a. Create a procedure called **DEL_PROD** to delete a product. Include the necessary exception handling.
 - b. Compile the code, invoke the procedure, and query the **PRODUCT** table to view the results. Also, check the exception handling by trying to delete a product that does not exist.
4. Create a procedure to query the **EMP** table, retrieving the salary and job title for employee 7839.
 - a. Create a procedure that returns a value from the **SAL** and **JOB** columns for a specified employee (use **EMPNO**).
Use host variables for the two **OUT** parameters.
Provide appropriate error handling if the user invokes the procedure with a non-existent value for **EMPNO**.
 - b. Compile the code, invoke the procedure, and display the salary and job title for employee 7839.
 - c. Invoke the procedure again, passing an **EMPNO** of 9898. What happens and why?

4

Creating Functions

Objectives

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

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

Lesson Aim

In this lesson, you will learn how to create and invoke functions in client-side and server-side environments.

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 database object, for repeated execution.**
- **A function can be called as part of an expression.**

A stored function is a named PL/SQL block that can take parameters and be invoked. Generally speaking you use a function to compute a value. Functions and procedures are structured alike, except that a function must return a value to the 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 the header, and at least one RETURN statement in the executable section.

Functions promote reusability and maintainability. Once validated they can be used in any number of applications. If the definition changes, only the function is affected, this greatly simplifies maintenance.

Functions can be called as part of a SQL expression or as part of a PL/SQL expression. In a SQL expression, a function must obey certain 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
(argument1 [mode1] datatype1,
 argument2 [mode2] datatype2,
 . . .
RETURN datatype
IS|AS
PL/SQL Block;
```

4-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



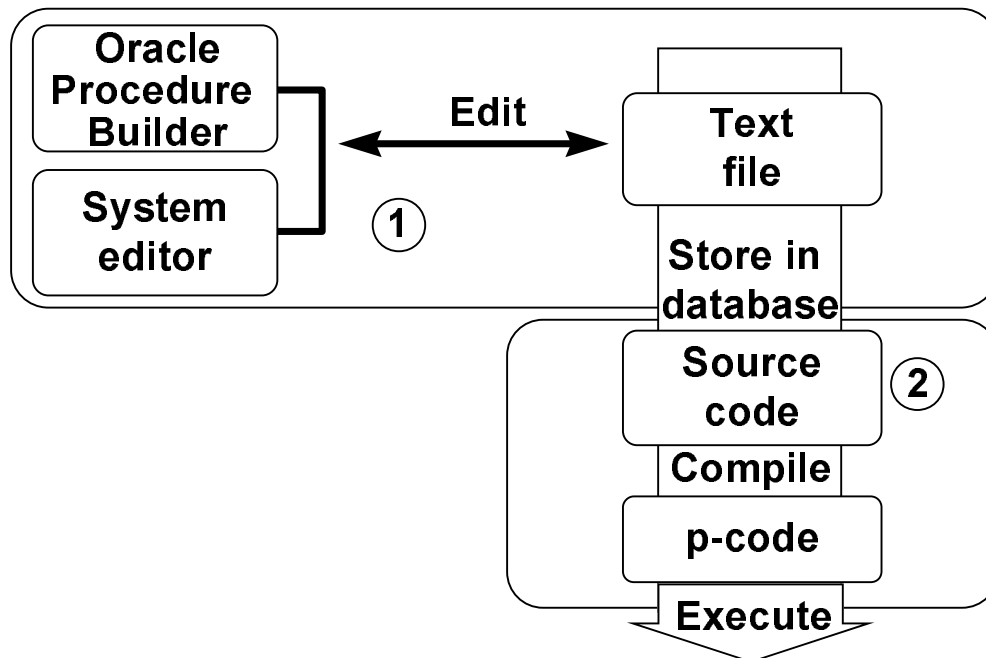
A function is a PL/SQL block that returns a value. You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

- 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 (variable) statement. You cannot reference host or bind variables in the PL/SQL block of a stored function.
- The REPLACE option indicates that if the function exists, it will be dropped and replaced with the new version created by the statement.
- RETURN datatype must not include size specification.

Syntax Definitions

Parameter	Description
<i>function_name</i>	Name of the function
<i>argument</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>	Datatype of the parameter
RETURN <i>datatype</i>	Datatype 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



4-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

How to Develop Stored Functions

1. Select your development environment: Procedure Builder or SQL*Plus.

Write the syntax. If using Procedure Builder, enter the code in the Program Unit Editor. If using SQL*Plus, enter the code in a text editor and save it as a script file.

2. Compile the code.

The source code is compiled into *p-code*. If developing in Procedure Builder, click Compile. If developing in SQL*Plus, start the script file to compile your code.

Returning a Value

- Add a RETURN clause with the datatype in the header of the function.
- Include at least one RETURN statement in the executable section.



Multiple RETURN statements are allowed, usually within an IF statement. Only one RETURN statement will be executed.

Creating a Stored Function Using SQL*Plus

- 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 SQL*Plus, run the script file to compile the source code into p-code and store both in the database.**
- 3. Invoke the function from an Oracle Server environment to determine whether it executes without error.**

Creating a Stored Function Using SQL*Plus: Example

```
SQL> CREATE OR REPLACE FUNCTION get_sal
 2   (v_id IN emp.empno%TYPE)
 3   RETURN NUMBER
 4   IS
 5     v_salary emp.sal%TYPE :=0;
 6   BEGIN
 7     SELECT sal
 8     INTO   v_salary
 9     FROM   emp
10     WHERE  empno = v_id;
11     RETURN (v_salary);
12 END get_sal;
13 /
```

4-7

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

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

Run the statement to create the GET_SAL function. Invoke a function as part of a PL/SQL expression. This is because of the way the RETURN statement works.



The CREATE keyword is not required in Procedure Builder.

Creating a Function Using Procedure Builder

Procedure Builder allows you to:

- **Create a client-side function**
- **Create a server-side function**
- **Drag and drop functions between client and server**

Creating Functions Using Procedure Builder: Example

Return the tax based on a specified value.

```
FUNCTION tax
  (v_value IN NUMBER)
  RETURN NUMBER
IS
BEGIN
  RETURN (v_value * .08);
END tax;
```

Steps to Create a Client-Side Function Using Procedure Builder

1. Click the Program Units node in the Object Navigator.
2. Click Create.
3. Enter the function name in the New Program Units dialog box: tax.
Select the radio button for the program unit type: function.
4. Click OK to accept these entries.
5. In the Program Unit Editor, enter the code shown above.
6. Click Compile. Note “Successfully Compiled.”
7. Click Close.



Avoid using OUT and IN OUT mode parameters with functions. Functions are designed to return a single value.

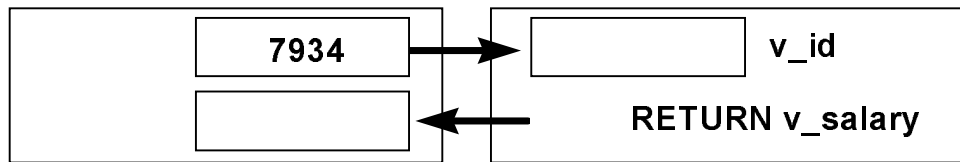
Executing Functions

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

Executing Functions in SQL*Plus: Example

Calling environment

GET_SAL function



```
SQL> START get_salary.sql
Procedure created.
```

```
SQL> VARIABLE g_salary number
```

```
SQL> EXECUTE :g_salary := get_sal(7934)
PL/SQL procedure successfully completed.
```

```
SQL> PRINT g_salary
          G_SALARY
-----
              1300
```

4-11

Copyright © Oracle Corporation, 1998. All rights reserved.

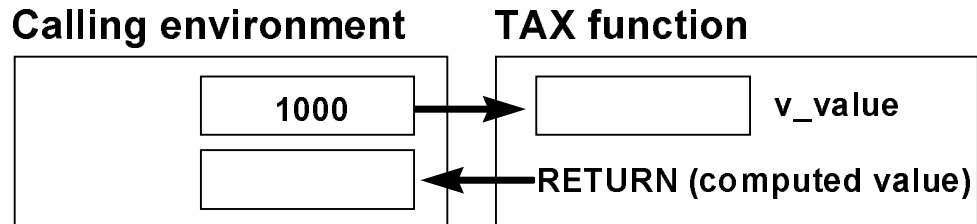
ORACLE®

Example

Execute the GET_SAL function from SQL*Plus:

1. Run the statement to create the stored function GET_SAL.
2. Create a host variable that will be populated by the RETURN (variable) syntax within the function.
3. Using the EXECUTE syntax in SQL*Plus, invoke GET_SAL by creating a PL/SQL expression. Supply a value for the parameter (employee ID number 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 syntax. Employee Miller, empno 7934, earns a monthly salary of 1300.

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
```

Example

Execute the TAX function from Procedure Builder:

1. Create a 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.

Advantages of User-Defined Functions in SQL Expressions

- **Extend SQL where activities are too complex, too awkward, or unavailable with SQL**
- **Query efficiency: functions used in the WHERE clause can filter data**
- **Manipulate character strings**
- **Provide parallel query execution**

4-13

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Calling Stored 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
- Provides parallel query execution; if the query is paralleled, it can be executed in parallel (using the Parallel Query option)

Example

In SQL*Plus, invoke the TAX function inside a query displaying employee id, name, and salary.

```
SQL> SELECT empno, ename, sal, tax(sal)
2 FROM emp;
```

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 clauses of the INSERT command**
- **SET clause of the UPDATE command**

PL/SQL user-defined functions can be called from any SQL expression wherever a built-in function can be called.

Calling Functions from SQL Expressions: Restrictions

- **A user-defined function must be a stored function.**
- **A user-defined function must be a ROW function, not a GROUP function.**
- **A user-defined function only takes IN parameters, not OUT, or IN OUT.**
- **Datatypes must be CHAR, DATE, or NUMBER, not PL/SQL types such as BOOLEAN, RECORD, or TABLE.**
- **Return type must be an Oracle Server internal type.**

4-15

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



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

- Only stored functions are callable from SQL statements. Stored procedures cannot be called.
- This feature is only available with PL/SQL 2.1 and later. Tools using earlier versions of PL/SQL do not support this functionality.
- Parameters to a PL/SQL function called from a SQL statement must use the positional notation. The 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 EXECUTE privilege of the function in order to call it from a SQL statement.

Calling Functions from SQL Expressions: Restrictions

- **INSERT, UPDATE, or DELETE commands are not allowed.**
- **Calls to subprograms that break the above restriction are not allowed.**

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 changes to database tables. Side effects could delay the executions of a query or yield order-dependent (therefore indeterminate) results. Therefore, restrictions apply to stored functions called from SQL expressions.

Restrictions

- The function cannot modify database tables; it cannot execute an INSERT, UPDATE, or DELETE statement.
- The function cannot call another subprogram that breaks one of the above restrictions.

Removing Functions

- **Using SQL*Plus:**
Drop a server-side function.
- **Using Procedure Builder:**
 - **Drop a server-side function.**
 - **Drop a client-side function.**

Removing a Server-Side Function

Using SQL*Plus

- **Syntax**

```
DROP FUNCTION function_name
```

- **Example**

```
SQL> DROP FUNCTION get_salary;  
Function dropped.
```

To remove a server-side function using SQL*Plus, execute the SQL command **DROP FUNCTION**.



Rollback is not possible after executing a data definition language (DDL) command such as **DROP FUNCTION**.

Removing Server-Side Functions

Using Procedure Builder

1. **Choose File—>Connect and enter your username, password, and database.**
2. **Expand the Database Objects node.**
3. **Expand the schema of the owner of the function.**

4-19

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

When you decide to delete a stored function, an alert box displays with the following message:

Do you really want to drop stored program unit TAX?



In the Stored Program Units Editor, click **DROP** to remove the function from the server.

Removing Server-Side Functions

- 4. Click the Stored Program Units node under that schema.**
- 5. Click the function you want to drop.**
- 6. Click Delete in the Object Navigator.**
- 7. Click Yes to drop the function.**

Removing a Client-Side Function

Using Procedure Builder

1. **Expand the Program Units node in the Object Navigator.**
2. **Click the function you want to remove.**
3. **Click Delete in the Object Navigator.**

4-21

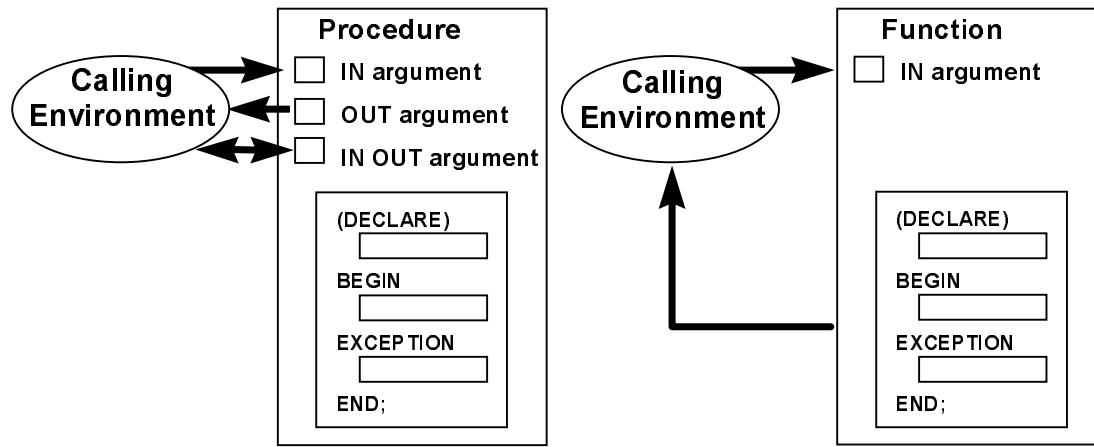
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



Follow the above steps to remove the function from Procedure Builder. If you have *exported* the code that built your function to a text file and you want to delete that file from the client, you must do so in the operating system.

Procedure or Function?



4-22

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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 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.

Comparing Procedures and Functions

Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No RETURN datatype	Must contain a RETURN datatype
Can return one or more values	Must return a value

4-23

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



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

Benefits of Stored Procedures and Functions

- Improved performance
- Improved maintenance
- Improved data security and integrity

Benefits of Stored Procedures and Functions

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 runtime by parsing at compile time.
 - Reduce the number of calls to the database and decrease network traffic by bundling commands.
- Improved 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.

Summary

- **A function is a named PL/SQL block that must return a value.**
- **A function is invoked as part of an expression.**
- **A stored function can be called in SQL statements.**

Practice Overview

- **Creating stored functions**
- **Invoking a stored function from a SQL statement**
- **Invoking a stored function from a stored procedure**

Use either SQL*Plus or Procedure Builder to do the practices.

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

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

Practice 4

1. Create and invoke the Q_PROD function to return a product description.
 - a. Create a function called Q_PROD to return a product description to a host variable.
 - b. Compile the code, invoke the function, and then query the host variable to view the result.
2. Create a stored function ANNUAL_COMP to return the annual salary when passed an employee's monthly salary and commission. Be sure the function addresses NULL values.
 - a. Create and invoke the function ANNUAL_COMP, passing in values for monthly salary and commission. The function should return the annual salary, as defined by:

`(sal*12) + comm`

- b. Use the stored function in a SELECT statement against the EMP table.
3. Create a procedure, NEW_EMP, to insert a new employee into the EMP table. The procedure should contain a call to the function VALID_DEPTNO to check whether the department number specified for the new employee exists in the DEPT table.

- a. Create a function VALID_DEPTNO to validate a specified department number. The function should return a BOOLEAN.
 - b. Then create the procedure NEW_EMP to add an employee to the EMP table. A new record should be added to EMP 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 arguments. The default commission is 0, the default salary is 1000, the default department number is 30, the default job is SALESMAN and the default manager number is 7839. For the employee's ID number, use the sequence SEQ_EMPNO. Run the `\labs\cre_seq.sql` file to create the sequence.

- c. Test your NEW_EMP procedure by adding a new employee named HARRIS to department 99. Let all other parameters default. What was the result?
 - d. Test your NEW_EMP procedure by adding a new employee named HARRIS to department 30. Let all other parameters default. What was the result?

5

Creating Packages

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**
- **Make a package construct either public or private**
- **Invoke a package construct**

Lesson Aim

In this lesson you learn what a package is and what it can comprise. You also learn how to create and use packages.

Overview of Packages

- **Group logically related PL/SQL types, items, and subprograms**
- **Consist of two parts:**
 - **Specification**
 - **Body**
- **Cannot be called, parameterized, or nested**
- **Allow Oracle8 to read multiple objects into memory at once**

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.

Usually a package 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 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 calling a packaged PL/SQL construct for the first time, the whole package is loaded into memory. Thus, later calls to related constructs require no disk I/O.

Advantages of Packages

- **Modularity**
- **Easier application design**
- **Information hiding**
- **Added functionality**
- **Better performance**
- **Overloading**

Using packages is an alternative to creating procedures and functions as standalone schema objects and offers 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.

Information Hiding

You can decide which constructs are public (visible and accessible) or private (hidden and inaccessible). The package hides the definition of the private constructs so that only the package is affected (not your application) if the definition changes. Also, by hiding implementation details from users, you protect the integrity of the package.

Advantages of Packages

- **Modularity**
- **Easier application design**
- **Information hiding**
- **Added functionality**
- **Better performance**
- **Overloading**

Added Functionality

Packaged public variables and cursors persist for the duration of a session. So, they can be shared by all subprograms that execute in the environment. Also, they allow you to maintain data across transactions without having to store it in the database.

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.

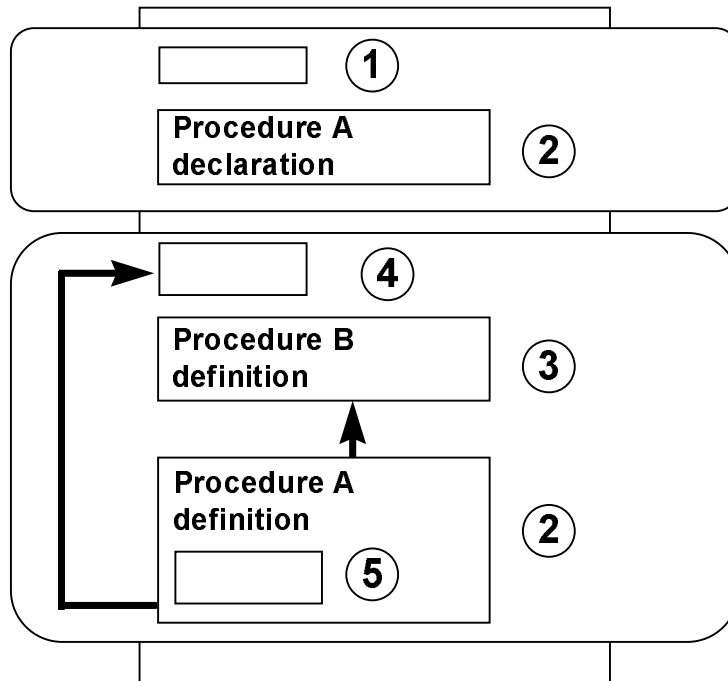
Overloading

Packages allow you to 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.

Developing a Package

**Package
specification**

**Package
body**



5-6

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Public variable
2. Public procedure
3. Private procedure

You create a package in two parts: create the package specification and then create 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.

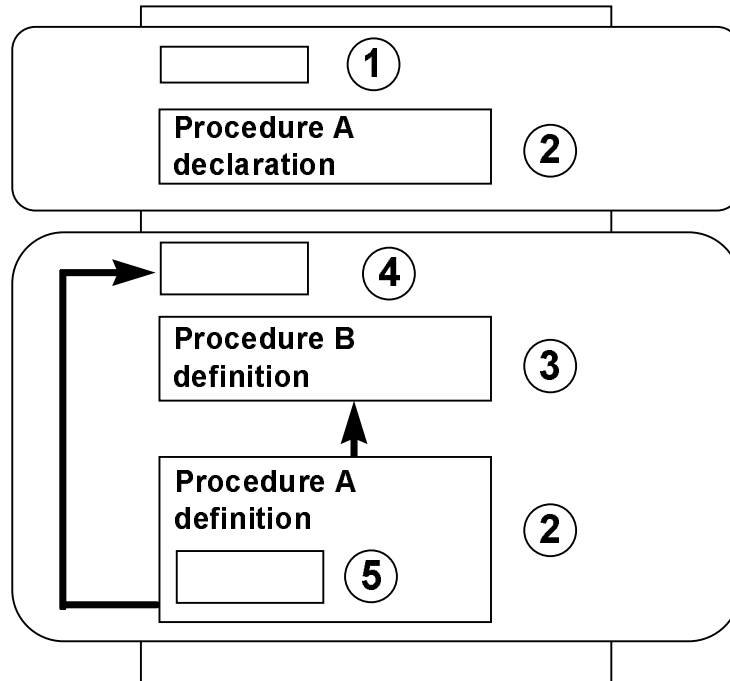
Scope of Construct	Description	Placement Within Package
Public	Can be referenced from any Oracle Server environment	Declared within the package specification and defined within the package body
Private	Can only be referenced by other constructs which are part of the same package	Declared and defined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

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

Developing a Package

**Package
specification**

**Package
body**



5-7

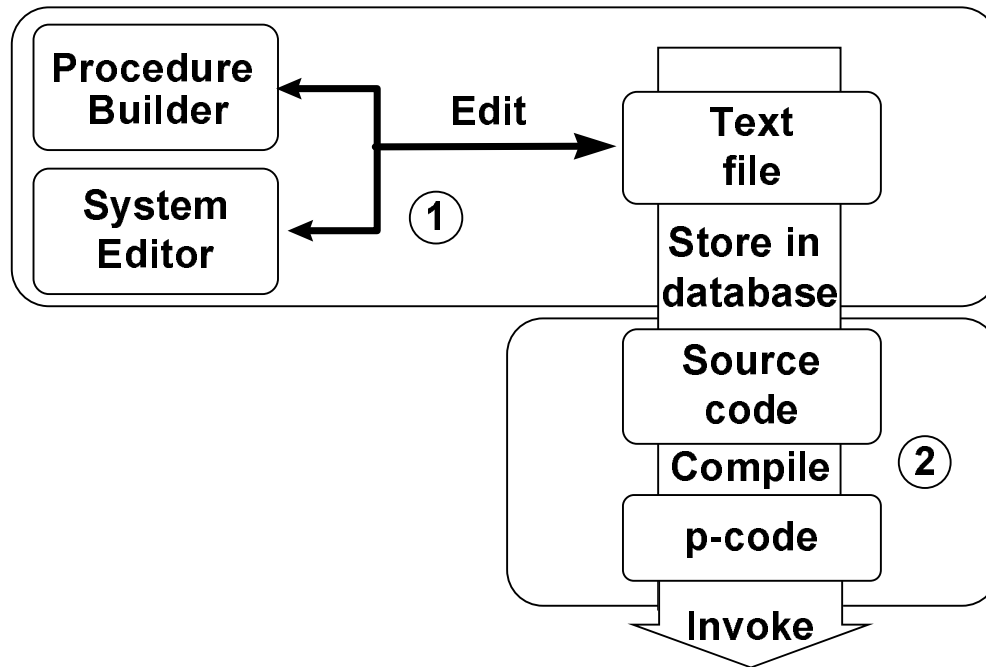
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

- 4. Global variable
- 5. Local variable

Visibility of Construct	Description
Local	A variable or subprogram defined within another subprogram. (Cannot be referenced by other applications and is only visible to the enclosing block.)
Global	A variable or subprogram that can be referenced (and changed) outside the package.

Developing a Package



5-8

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

How to Develop a Package

1. Select a development environment: Procedure Builder or SQL*Plus.

Write the syntax. If you are using Procedure Builder, enter the syntax in the Program Unit Editor. If you are using SQL*Plus, enter the code in a text editor and save it as a script file.

2. Compile the code.

The source code is compiled into p-code. If you are using Procedure Builder, click Compile. If you are using SQL*Plus script files, start the script file.

Developing a Package

- **Saving the text of the CREATE PACKAGE statement in two different text 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.**
- **If you have incorporated a standalone procedure into a package, you should drop your standalone procedure.**

5-9

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Overview

There are three basic steps to follow to develop a package, similar to those used to develop a standalone procedure, as shown below.

Develop a Package

1. Write the text of the CREATE PACKAGE statement within a script file using either SQL*Plus or Oracle Procedure Builder to create the package specification and process the statement. 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 script file using either SQL*Plus or Procedure Builder to create the package body and process the statement. 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;
```

5-10

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Create Packages

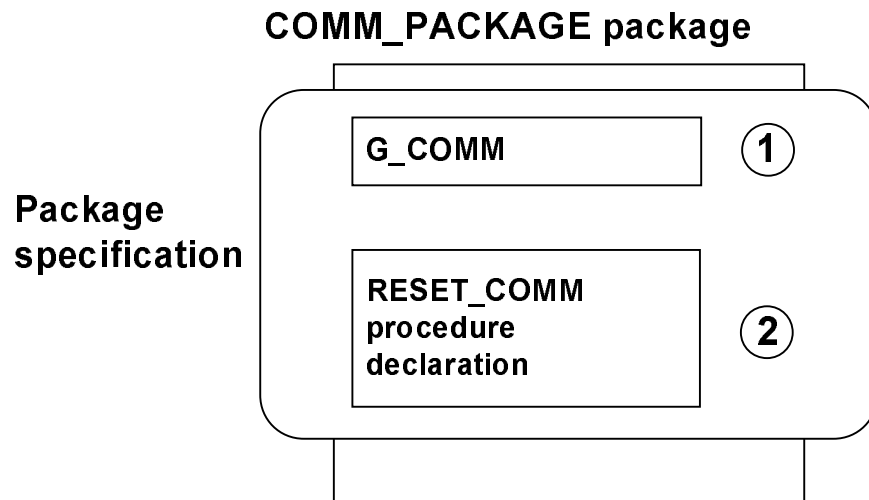
To create packages, you declare all public constructs within the package specification.

- Specify the REPLACE option when the package specification already exists.
- Use the keyword IS instead of AS if you are using Procedure Builder. If you are not using Procedure Builder, IS and AS are equivalent.
- Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.

Syntax Description

where:	<i>package_name</i>	is the name of the package.
	<i>type and item declarations</i>	declare variables, constants, cursors, exceptions, or types.
	<i>subprogram specifications</i>	declare the PL/SQL subprograms.

Declaring Public Constructs



5-11

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Global variable
2. Public procedure

To declare a public variable to store a value that can be referenced and changed outside the package, declare a public procedure or function to implement a routine that can be invoked repeatedly outside the package.

Creating a Package Specification: Example

```
SQL>CREATE OR REPLACE PACKAGE comm_package IS
  2   g_comm  NUMBER := 10;    --initialized to 10
  3   PROCEDURE reset_comm
  4     (v_comm      IN      NUMBER) ;
  5 END comm_package;
  6 /
```

In the example above both the (global) variable G_COMM and the procedure RESET_COMM are public constructs.

Declaring a Global Variable or a Public Procedure

```
SQL>EXECUTE comm_package.g_comm := 5
```

```
SQL>EXECUTE comm_package.reset_comm(8)
```

You can declare a global variable to keep track of a prevailing commission for the user session, which can be changed by the user directly. Or you can declare a public procedure to allow the user to reset the commission at any time during the session.

Creating the Package Body

Syntax

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

Syntax Description

You define all public and private procedures and functions in the package body.

where:	<i>package_name</i>	is the name of the package.
	<i>type and item declarations</i>	declare variables, constants, cursors, exceptions, or types.
	<i>subprogram bodies</i>	define the PL/SQL subprograms.

- Specify the REPLACE option when the package body already exists.
- Use the keyword IS instead of AS if you are using Procedure Builder. If you are not using Procedure Builder, IS and AS are equivalent.
- The order in which constructs are defined within the package body makes a difference; you must define a construct before referencing it from another.

Public and Private Constructs

COMM_PACKAGE package

**Package
specification**

G_COMM

①

RESET_COMM
procedure declaration

②

**Package
body**

VALIDATE_COMM
function definition

③

RESET_COMM
procedure definition

②



5-15

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

1. Public (Global) variable
2. Public procedure
3. Private function

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

Creating a Package Body: Example

```
SQL>CREATE OR REPLACE PACKAGE BODY comm_package IS
  2 FUNCTION validate_comm
  3   (v_comm    IN NUMBER) RETURN  BOOLEAN
  4 IS
  5   v_max_comm NUMBER;
  6 BEGIN
  7   SELECT MAX(comm)
  8   INTO    v_max_comm
  9   FROM    emp;
 10   IF v_comm > v_max_comm THEN RETURN(FALSE) ;
 11   ELSE RETURN(TRUE) ;
 12   END IF;
 13 END validate_comm;
 14 END comm_package;
 15 /
```

Define a function to validate the commission. The commission may not be greater than the highest commission among all existing employees. We also define a procedure that allows you to reset and validate the prevailing commission.

Creating a Package Body: Example

```
SQL>PROCEDURE reset_comm
 2 (v_comm IN NUMBER)
 3 IS
 4   v_valid  BOOLEAN;
 5 BEGIN
 6   v_valid := validate_comm(v_comm);
 7   IF v_valid = TRUE THEN
 8     g_comm := v_comm;
 9   ELSE
10     RAISE_APPLICATION_ERROR
11       (-20210,'Invalid commission');
12   END IF;
13 END reset_comm;
14 END comm_package;
15 /
```

Developing Packages: Guidelines

- **Keep packages as general as possible.**
- **Define the package specification before the body.**
- **The package specification should only contain public constructs.**
- **The package specification should contain as few constructs as possible.**

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 Oracle.

Package specifications reflect the design of your application, so define them before defining the package bodies.

The package specification should only contain 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.

To reduce the need for recompiling when code is changed, place as few constructs as possible in a package specification. 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.

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(v_comm IN NUMBER)
    IS
        v_valid      BOOLEAN;
    BEGIN
        v_valid := validate_comm(v_comm);
        IF v_valid = TRUE THEN
            g_comm := v_comm;
        ELSE
            RAISE_APPLICATION_ERROR (-20210, 'Invalid comm');
        END IF;
    END reset_comm;
END comm_package;
```

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 within the package, you do not need to qualify its name.

Example 1

Call the `VALIDATE_COMM` function from the `RESET_COMM` procedure.

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

Invoking Package Constructs

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

```
SQL> EXECUTE comm_package.reset_comm(1500);
```

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

```
SQL> EXECUTE scott.comm_package.reset_comm(1500);
```

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

```
SQL> EXECUTE comm_package.reset_comm@ny (1500);
```

5-20

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example 2

Call the RESET_COMM procedure from SQL*Plus, making the prevailing commission \$1500 for the user session.

Example 3

Call the RESET_COMM procedure that is located in the scott schema from SQL*Plus, making the prevailing commission \$1500 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 SQL*Plus, making the prevailing commission \$1500 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.

Referencing a Global Variable Within the Package

Example 1

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

5-21

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Similarly, when you reference a variable, cursor, constant or exception within the package, it is not necessary to qualify its name.

Example 1

Set the prevailing commission percentage from the RESET_COMM procedure.

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

Referencing a Global Variable from a Standalone Procedure

Example 2

```
CREATE OR REPLACE PROCEDURE hire_emp
(v_ename IN      emp.ename%TYPE,
 v_mgr   IN      emp.mgr%TYPE,
 v_job   IN      emp.job%TYPE,
 v_sal   IN      emp.sal%TYPE)
IS
    v_comm emp.comm%TYPE;
    . . .
BEGIN
    . . .

    v_comm := comm_package.g_comm;
    . . .
END hire_emp;
```

Example 2

Revise the HIRE_EMP procedure to compute the commission for a new employee based on the prevailing commission.

Persistent State of Package Variables

```
SCOTT-09:00> EXECUTE comm_package.reset_comm(120);  
Initially g_comm equals 10  
After executing the procedure, g_comm equals 120  
  
JONES-09:30> INSERT INTO emp (ename,...,comm)  
VALUES ('Madona',...,2000);  
JONES-09:35> EXECUTE comm_package.reset_comm(170);  
Initially g_comm equals 10  
After executing the procedure, g_comm equals 170  
  
SCOTT-10:00> EXECUTE comm_package.reset_comm(4000);  
Results in: 'Invalid commission'  
  
JONES-11:00> ROLLBACK;  
JONES-11:01> EXIT;  
  
JONES-12:00> EXECUTE comm_package.reset_comm(150);  
Initially g_comm equals 10  
After executing the procedure, g_comm equals 150
```

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, first-time-only procedure.
2. Change the value of the variable by means of package procedures.
3. The value of the variable is released from the SGA when the user disconnects.

Persistent State of a Package Cursor

Example

```
SQL> CREATE OR REPLACE PACKAGE pack_cur
  2  IS
  3      CURSOR c1 IS SELECT empno FROM emp
  4                      ORDER BY empno desc;
  5      PROCEDURE proc1_3rows;
  6      PROCEDURE proc4_6rows;
  7  END pack_cur;
  8  /
```

Controlling the Persistent State of a Package Cursor

1. Open the cursor within an automatic, first-time-only procedure.
2. Fetch successive rows from the cursor by means of package procedures.
3. Close the cursor for clean termination.

Persistent State of a Package Cursor

```
SQL> CREATE OR REPLACE PACKAGE BODY pack_cur
2  IS
3      v_empno NUMBER;
4      PROCEDURE proc1_3rows IS
5      BEGIN OPEN c1;
6          LOOP FETCH c1 INTO v_empno;
7              DBMS_OUTPUT.PUT_LINE('Id : ' || (v_empno));
8              EXIT WHEN c1%ROWCOUNT >= 3;
9          END LOOP;
10     END proc1_3rows;
11     PROCEDURE proc4_6rows IS
12     BEGIN
13         LOOP FETCH c1 INTO v_empno;
14             DBMS_OUTPUT.PUT_LINE('Id : ' || (v_empno));
15             EXIT WHEN c1%ROWCOUNT >= 6;
16         END LOOP;
17     CLOSE c1;
18     END proc4_6rows;
19 END pack_cur;
20 /
```

Persistent State of a Package Cursor

```
SQL> SET SERVEROUTPUT ON
SQL> EXECUTE pack1.proc1_3rows;
      Id : 7934
      Id : 7902
      Id : 7900
SQL> EXECUTE pack1.proc4_6rows;
      Id : 7876
      Id : 7844
      Id : 7839
```

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.

The state does not persist from user to user.

Persistent State of Package PL/SQL Tables and Records

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

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

5-27

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Passing Tables of Records to Procedures or Functions Inside a Package

Invoke the READ_EMP_TABLE procedure from an anonymous PL/SQL Block using SQL*Plus.

```
SQL>DECLARE
2  emp_table emp_package.emp_table_type;
3 BEGIN
4  emp_package.read_emp_table(emp_table);
5  dbms_output.put_line('An example: ' || emp_table(4).ename);
6 END;
7 /
```

Removing Packages

To remove the package specification and the body:

```
DROP PACKAGE package_name
```

To remove the package body:

```
DROP PACKAGE BODY package_name
```

Summary

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

Summary

- **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.**

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

Practice Overview

- **Creating Packages**
- **Invoking Package Program Units**

Use either SQL*Plus or Procedure Builder to do the practices.

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

If you correct any compilation errors in SQL*Plus, do so in the original script file, not in the buffer, and then re-run the new version of the file. This will save a new version of the package to the data dictionary.

Practice 5

1. Create a package specification and body called **PROD_PACK** that contains your **ADD_PROD**, **UPD_PROD**, **DEL_PROD** procedures, and your **Q_PROD** function.
 - a. Make all the constructs public.

Note: Consider whether you still need the standalone procedures and functions you just packaged.
 - b. Invoke your **DEL_PROD** procedure.
 - c. Query the **PRODUCT** table to see the result.
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_DEPTNO** function as a private construct.
 - b. Invoke the **NEW_EMP** procedure using 99 as a department number.
 - c. Invoke the **NEW_EMP** procedure using 30 as a department number.
3. Create a package called **CHK_PACK** that contains the procedures **CHK_HIREDATE** and **CHK_DEPT_MGR**. Make both constructs public.
 - a. The procedure **CHK_HIREDATE** checks whether an employee's hiredate is within the following range: [sysdate - 50 years, sysdate + 3 months]

Notes:

- 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 hiredate should be treated as an invalid hiredate.

- b. The procedure **CHK_DEPT_MGR** checks the department and manager combination for a given employee. This means that the manager number provided must be equal to the manager number supervising the employee's department.

Notes:

- If the department number/manager combination is invalid, you should raise an application error with an appropriate message.
- Make sure you handle the case where there is no manager for the department.

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

```
SQL> execute chk_pack.chk_hiredate('01-JAN-47')
```

- d. Test the **CHK_HIREDATE** procedure with the following command.

```
SQL> execute chk_pack.chk_hiredate(NULL)
```

- e. Test the **CHK_HIREDATE** procedure with the following command.

```
SQL> execute chk_pack.chk_hiredate('01-JAN-98')
```


6

More Package Concepts

Objectives

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

- **Write packages that use the overloading feature**
- **Avoid errors with mutually referential subprograms**
- **Initialize variables with a one-time-only procedure**

Lesson Aim

This lesson introduces some more advanced features of PL/SQL. It also gives an overview of the Oracle Server supplied packages.

Objectives

- **Control side effects of functions**
- **Declare and use cursor variables in a package**
- **Describe the use and application of the Oracle Server supplied packages**

Overloading

- **Allows you to use the same name for different subprograms inside a package**
- **The formal parameters of the subprograms must differ in number, order, or datatype family**
- **Overloaded subprograms can be placed in local or packaged subprograms**

Overloading

Sometimes the processing in two subprograms is the same. In that case it is logical to give them the same name. PL/SQL determines which subprogram is being called by checking its formal parameters.

Restrictions

- Only local or packaged subprograms can be overloaded.
- You cannot overload two subprograms if their formal parameters differ only in name or parameter mode.
- You cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family.
- You cannot overload two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family.
- You cannot overload two functions that differ only in return type, even if the types are in different families.

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 datatype between the actual and formal parameters.

Overloading: Example

Initialize the first 50 rows in two PL/SQL tables.

```
PROCEDURE initialize(tab OUT datetabtyp, n INTEGER) IS
BEGIN
  FOR i in 1..n LOOP
    tab(i) := SYSDATE;
  END LOOP;
END initialize;

PROCEDURE initialize(tab OUT realtabtyp, n INTEGER) IS
BEGIN
  FOR i in 1..n LOOP
    tab(i) := 1000;
  END LOOP;
END initialize;
```

6-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

If you call INITIALIZE with a datetabtyp parameter, PL/SQL uses the first version of the procedure. If you call INITIALIZE with a realtabtyp parameter, PL/SQL uses the second version.

```
DECLARE
  TYPE datetabtyp IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  TYPE realtabtyp IS TABLE OF REAL
    INDEX BY BINARY_INTEGER;
  hiredate_tab    datetabtyp;
  sal_tab         realtabtyp;
  indx            BINARY_INTEGER;
BEGIN
  indx := 50;
  initialize(hiredate_tab,indx); -- will call first version
  initialize(sal_tab,indx);      -- will call second version
  . . .
END;
```

Forward Declarations

Identifiers must be declared before referencing them.

```
. . .  
PROCEDURE award_bonus(. . .) IS  
BEGIN  
    calc_rating(. . .);          --illegal reference  
. . .  
END;  
PROCEDURE calc_rating(. . .) IS  
BEGIN  
    . . .  
END;  
. . .
```

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 above the `CALC_RATING` procedure cannot be referenced at this point as 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.

Forward Declarations

PL/SQL allows for a special subprogram declaration called 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

Forward Declarations

```
. . .  
PROCEDURE calc_rating(. . .);      -- forward declaration  
. . .  
PROCEDURE award_bonus(. . .) IS    -- subprograms defined  
BEGIN                              -- in alphabetical order  
    calc_rating(. . .);  
. . .  
END;  
PROCEDURE calc_rating(. . .) IS  
BEGIN  
    . . .  
END;  
. . .
```

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 they 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 allow you to hide implementation details.

Creating a One-Time-Only Procedure

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
  FUNCTION validate_comm
    . . .
  END validate_comm;

  PROCEDURE reset_comm
    . . .
  END reset_comm;

  BEGIN
    SELECT AVG(comm)
    INTO g_comm
    FROM emp;
  END comm_package;
```

Define an Automatic, One-Time-Only Procedure

A one-time-only procedure is executed *one time only*, when the package is first invoked within the user session. In the example above the prevailing commission value is set at the beginning of the session to the average commission among all employees.

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, since the value will be reset by the one-time-only procedure.

Restrictions on Package Functions Used in SQL

- **INSERT, UPDATE, or DELETE are not allowed.**
- **Only local functions can update package variables.**
- **Remote functions cannot read or write remote package variables.**
- **Functions that read or write package variables cannot use the parallel query option.**
- **Calls to subprograms that break the above restrictions are not allowed.**

6-9

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Controlling Side Effects

For the Oracle Server to execute a SQL statement that calls a stored function, it must know 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 (which is not allowed). Therefore, the following restrictions apply to stored functions called from SQL expressions:

- The function cannot modify database tables; therefore it cannot execute an INSERT, UPDATE, or DELETE.
- Functions that read or write the values of packaged variables cannot be executed remotely or in parallel.
- Only functions called from a SELECT, a VALUES, or a SET clause can write the values of packaged variables.
- The function cannot call another subprogram that breaks one of the foregoing rules. Also the function cannot reference a view that breaks one of the foregoing rules.

Purity Level of a Package Function

```
PRAGMA RESTRICT_REFERENCES (function_name,  
                             WNDS  
                             [,WNPS]  
                             [,RNDS]  
                             [,RNPS] );
```

6-10

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Package Function State

where:	<i>WNDS</i>	Means Writes No Database State (mandatory). The function cannot modify the database tables.
	<i>WNPS</i>	Means Writes No Package State. The function cannot change the values of package variables.
	<i>RNDS</i>	Means Reads No Database State. The function cannot query database tables.
	<i>RNPS</i>	Means Reads No Package State. The function cannot reference the value of public packaged variables.

Specify the purity level of a packaged function when creating the package. The purity level asserts the extent to which the function permits database operations. You specify the purity level with the `PRAGMA RESTRICT_REFERENCES` compiler directive. The pragma tells the PL/SQL compiler to deny the packaged function read/write access to database tables, packaged variables or both. If you try to compile a function body that violates the pragma, you get a compilation error.



For standalone functions, the Oracle Server can enforce these rules by checking the function body. However, the body of a packaged function is hidden; only the specification is visible. Hence for packaged functions, you must use the `PRAGMA RESTRICT_REFERENCES` compiler directive.

Using PRAGMA RESTRICT_REFERENCES

```
CREATE OR REPLACE PACKAGE finance as -- package specification
    interest REAL;                    -- public variable
    . . .
    FUNCTION compound (years IN NUMBER,
                      amount IN NUMBER,
                      rate  IN NUMBER)
    RETURN NUMBER;
    . . .
    PRAGMA RESTRICT_REFERENCES (compound, WNDS, RNPS, WNPS);
END finance;
CREATE OR REPLACE PACKAGE BODY finance AS -- package body

    FUNCTION compound (years IN NUMBER,
                      amount IN NUMBER,
                      rate  IN NUMBER) RETURN NUMBER
    IS
    BEGIN
        RETURN (amount * POWER ((rate/100) + 1, years));
    END compound;
END finance;
```

6-11

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example



Create a function, COMPOUND, in a package, FINANCE. The function will be called from SQL statements on remote databases. Therefore, you need to assert the RNPS, WNPS, and WNDS purity levels, because remote functions can neither read nor write the package variables, nor modify database tables when called from a SQL statement.

- Assert the purity level of a function in the package specification.
- If a package contains multiple functions with the same name, the PRAGMA applies to the last one.

Invoke a User-Defined Package Function from an SQL Statement

```
DECLARE
    interest NUMBER
BEGIN
    . . .
    SELECT finance.compound@ny(yrs, amt, rte) -- function call
    INTO   interest
    FROM   accounts
    WHERE  acctno = acct_id;
    . . .
END;
```

Calling Package Functions

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

Example

Call the COMPOUND function (in the FINANCE package) from a SELECT statement within a PL/SQL block. The COMPOUND function resides on a database in New York.

Cursor Variables and Packages

```
CREATE OR REPLACE PACKAGE emp_data AS      -- package specification
    . . .
    TYPE emp_cur_typ IS REF CURSOR RETURN emp%ROWTYPE; --define type
    PROCEDURE use_emp_cv(emp_cv IN OUT emp_cur_typ);
END emp_data;

CREATE OR REPLACE PACKAGE BODY emp_data AS -- package body
    . . .
    PROCEDURE use_emp_cv(emp_cv IN OUT emp_cur_typ)
    IS
    BEGIN
        OPEN emp_cv FOR SELECT * FROM emp;
    END use_emp_cv;
END emp_data;
```

Package Cursor Variables

To create a cursor variable, you normally need two steps. First, you define a REF CURSOR type, then you declare a cursor variable of that type. You can define a REF CURSOR TYPE in any PL/SQL block, subprogram or package. However, you cannot declare a cursor variable in a package. Unlike a package variable, a cursor variable does not have a persistent state. (A cursor variable creates a pointer to an item instead of the item itself.)

In the example in the slide, you open a cursor variable by passing it to a package procedure that declares a cursor variable as one of its formal parameters. Note that the type of the formal parameter is IN OUT, as the subprogram opens the cursor variable. That way, the subprogram can pass an open cursor back to the calling program.

Alternatively you can use a standalone procedure. In this case you define the REF CURSOR type in a separate package, and then reference that type in a standalone procedure and open the cursor variable.

Oracle Supplied Packages

Several packaged procedures are provided with the Oracle Server to allow either PL/SQL access to certain SQL features or to extend the functionality of the database.

- DBMS_ALERT
- DBMS_APPLICATION_INFO
- DBMS_DDL
- DBMS_DESCRIBE
- DBMS_JOB
- DBMS_LOCK
- DBMS_MAIL
- DBMS_OUTPUT

Using 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_DDL	Recompiles procedure, functions, and packages and analyzes indexes, tables, and clusters
DBMS_DESCRIBE	Returns a description of the arguments for a given stored procedure
DBMS_JOB	Schedules periodic execution of PL/SQL code
DBMS_LOCK	Requests, converts, and releases userlocks, which are managed by the RDBMS lock management services
DBMS_MAIL	Sends messages from the Oracle Server directly to an Oracle*Mail identifier
DBMS_OUTPUT	Outputs values and messages from triggers, stored procedures, or functions

Oracle Supplied Packages

Several packaged procedures are provided with the Oracle Server to allow either PL/SQL access to certain SQL features or to extend the functionality of the database.

- DBMS_PIPE
- DBMS_SESSION
- DBMS_SHARED_POOL
- DBMS_SQL
- DBMS_TRANSACTION
- DBMS_UTILITY
- UTL_FILE

6-15

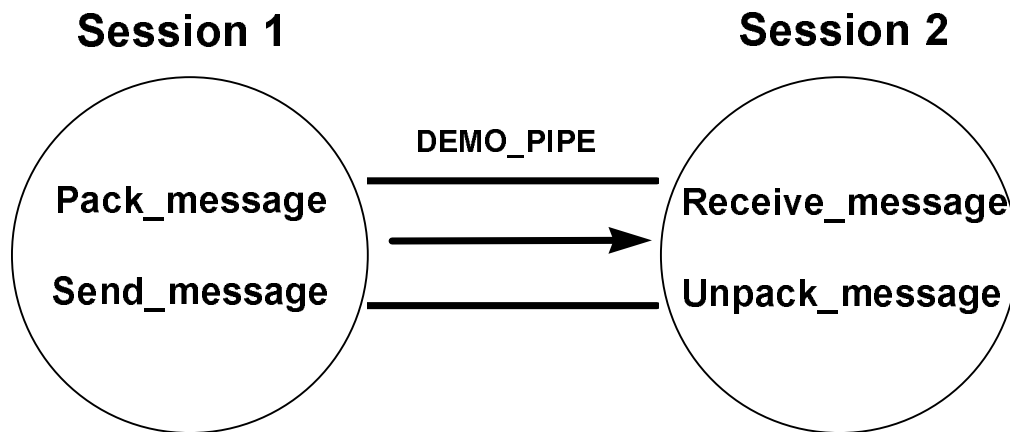
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Using Supplied Packages (continued)

Package	Description
DBMS_PIPE	Sends messages from one session to another in the same instance
DBMS_SESSION	Provides access to SQL alter session statements, and other session informations
DBMS_SHARED_POOL	Keeps objects in shared memory so that they will not be aged out with the normal LRU mechanism
DBMS_SQL	Uses dynamic SQL to access the database
DBMS_TRANSACTION	Controls logical transactions and improves the performance of short, non-distributed transactions by creating them as discrete
DBMS_UTILITY	Analyzes objects in a particular schema, checks whether the server is running in parallel mode, and returns the time
UTL_FILE	Adds file input/output capabilities to PL/SQL

Using DBMS_PIPE



6-16

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

The DBMS_PIPE package performs the following functions:

- It allows two or more sessions connected to the same instance to communicate through a pipe just like pipes used in UNIX.
- Each pipe works asynchronously.
- Depending on your security requirements, you can use either a public pipe or a private pipe.
- Once buffered information is read by one user, it is emptied from the buffer, and is not available for other readers of the same pipe.
- To send a message, first make one or more calls to PACK_MESSAGE to build your message. Then call SEND_MESSAGE to send the message on the named pipe.
- To receive a message from a pipe, first call RECEIVE_MESSAGE, and then call UNPACK_MESSAGE to access the individual items in the message.

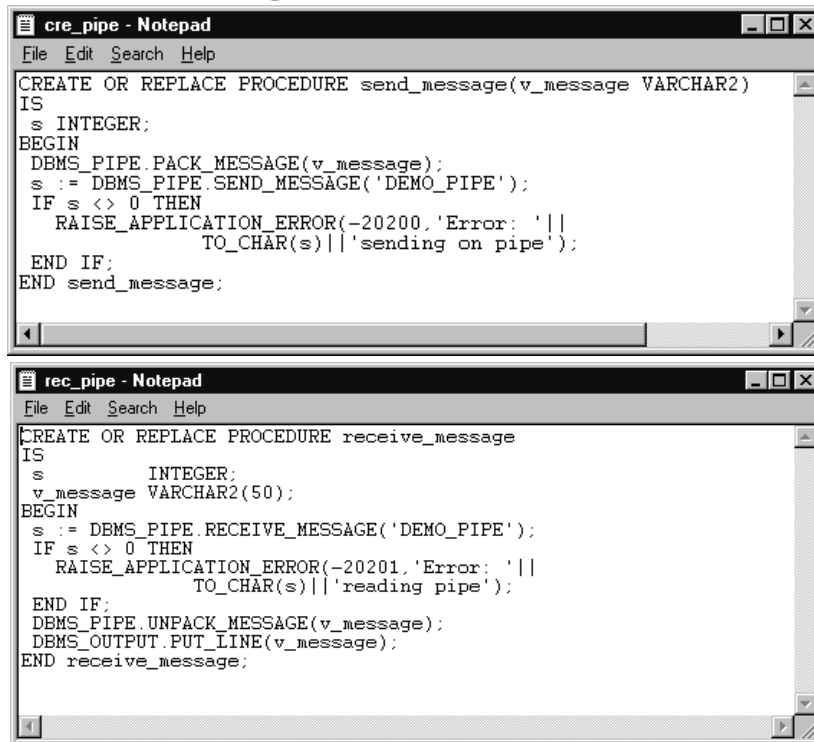


Oracle pipes buffer information in the SGA, which means that information is lost if you shut down your instance.



For more information, see *Oracle8 Server Application Developer's Guide*.

Using DBMS_PIPE



```
cre_pipe - Notepad
File Edit Search Help
CREATE OR REPLACE PROCEDURE send_message(v_message VARCHAR2)
IS
  s INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE(v_message);
  s := DBMS_PIPE.SEND_MESSAGE('DEMO_PIPE');
  IF s <> 0 THEN
    RAISE_APPLICATION_ERROR(-20200,'Error: '||
      TO_CHAR(s)||'sending on pipe');
  END IF;
END send_message;

rec_pipe - Notepad
File Edit Search Help
CREATE OR REPLACE PROCEDURE receive_message
IS
  s INTEGER;
  v_message VARCHAR2(50);
BEGIN
  s := DBMS_PIPE.RECEIVE_MESSAGE('DEMO_PIPE');
  IF s <> 0 THEN
    RAISE_APPLICATION_ERROR(-20201,'Error: '||
      TO_CHAR(s)||'reading pipe');
  END IF;
  DBMS_PIPE.UNPACK_MESSAGE(v_message);
  DBMS_OUTPUT.PUT_LINE(v_message);
END receive_message;
```

6-17

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

The first code box shows an example of a procedure that a PL/SQL program can call to place debugging information in a pipe. The second code box shows an example of a procedure that can be used to output the information. The pipe is named DEMO_PIPE.

DBMS_PIPE Package

- **PACK_MESSAGE**
- **PACK_MESSAGE_RAW**
- **PACK_MESSAGE_ROWID**
- **SEND_MESSAGE**
- **RECEIVE_MESSAGE**

DBMS_PIPE Package

The DBMS_PIPE package sends messages from one session to another in the same instance.

Function or Procedure	Description
PACK_MESSAGE	Packs an item into the local message buffer, that is, to be sent by the SEND_MESSAGE function (VARCHAR2, NUMBER, or DATE type item)
PACK_MESSAGE_RAW	Packs a raw item into the local message buffer
PACK_MESSAGE_ROWID	Packs a rowid item into the local message buffer
SEND_MESSAGE	Sends a message contained in the local message buffer to the named pipe
RECEIVE_MESSAGE	Retrieves a message from the named pipe and put it into the local message buffer to be unpacked by the UNPACK procedure
UNPACK_MESSAGE	Unpacks an item from the local message buffer (VARCHAR2, NUMBER, or DATE type item)
UNPACK_MESSAGE_RAW	Unpacks a raw item from the local message buffer
UNPACK_MESSAGE_ROWID	Unpacks a row of item from the local message buffer

DBMS_PIPE Package

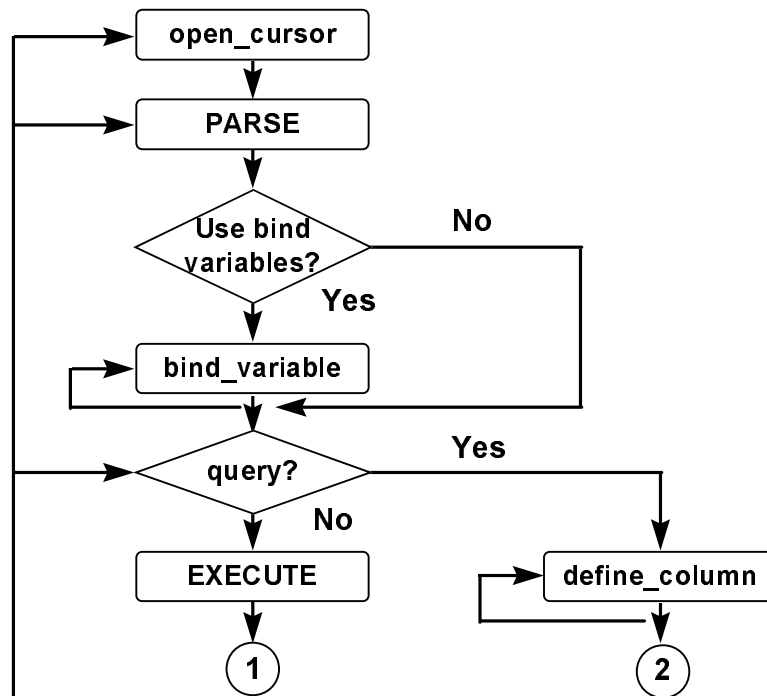
- **UNPACK_MESSAGE**
- **UNPACK_MESSAGE_RAW**
- **UNPACK_MESSAGE_ROWID**
- **NEXT_ITEM_TYPE**
- **UNIQUE_SESSION_NAME**
- **PURGE**

DBMS_PIPE Package (continued)

The DBMS_PIPE package sends messages from one session to another in the same instance.

Function or Procedure	Description
PACK_MESSAGE	Packs an item into the local message buffer, that is, to be sent by the SEND_MESSAGE function (VARCHAR2, NUMBER, or DATE type item)
PACK_MESSAGE_RAW	Packs a raw item into the local message buffer
PACK_MESSAGE_ROWID	Packs a rowid item into the local message buffer
SEND_MESSAGE	Sends a message contained in the local message buffer to the named pipe
RECEIVE_MESSAGE	Retrieves a message from the named pipe and put it into the local message buffer to be unpacked by the UNPACK procedure
UNPACK_MESSAGE	Unpacks an item from the local message buffer (VARCHAR2, NUMBER, or DATE type item)
UNPACK_MESSAGE_RAW	Unpacks a raw item from the local message buffer
UNPACK_MESSAGE_ROWID	Unpacks a row of item from the local message buffer

DBMS_SQL Execution Flow



6-20

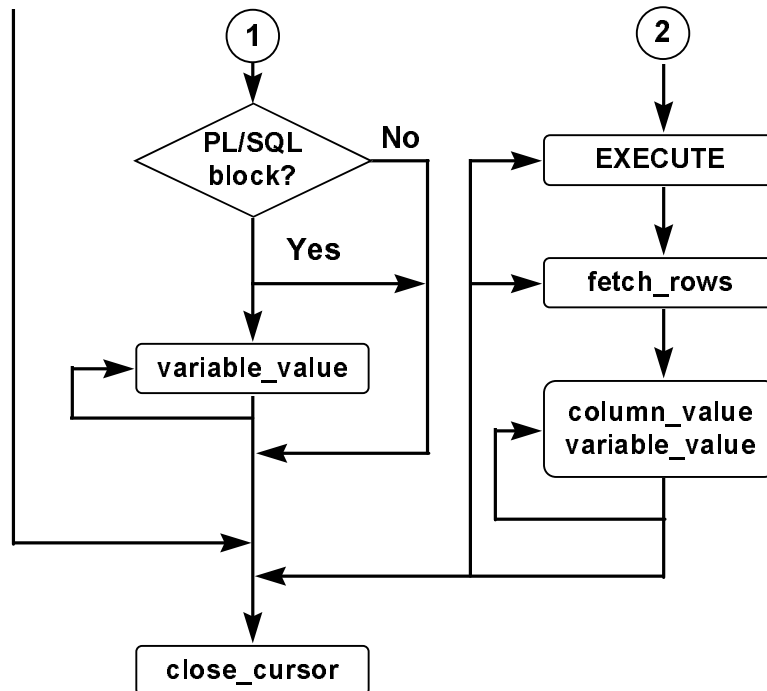
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Typical Flow of Procedure Calls

1. **OPEN_CURSOR**: To process a SQL statement, you must open a cursor where the handle is received as a cursor ID number.
2. **PARSE**: Every SQL statement must be parsed to check syntax, check privileges, and allocate a private SQL area for the statement.
3. **BIND_VARIABLE**: For each input data supplied at runtime you must call **BIND_VARIABLE** to supply the value of a variable to a placeholder.
4. **DEFINE_COLUMN**: For a query, you must specify the variables that are to receive the **SELECT** values, similar to the way an **INTO** clause does for a static query.
5. **EXECUTE**: Execute your SQL statement.
6. **FETCH_ROWS**: Retrieve the rows that satisfy the query.
7. **VARIABLE_VALUE** or **COLUMN_VALUE**: Call **COLUMN_VALUE** to determine the value of a column retrieved by the **FETCH_ROWS** call. Call **VARIABLE_VALUE** to retrieve the values assigned to PL/SQL output variables.
8. **CLOSE_CURSOR**: Close a cursor to deallocate memory when it is no longer needed.

DBMS_SQL Execution Flow



6-21

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

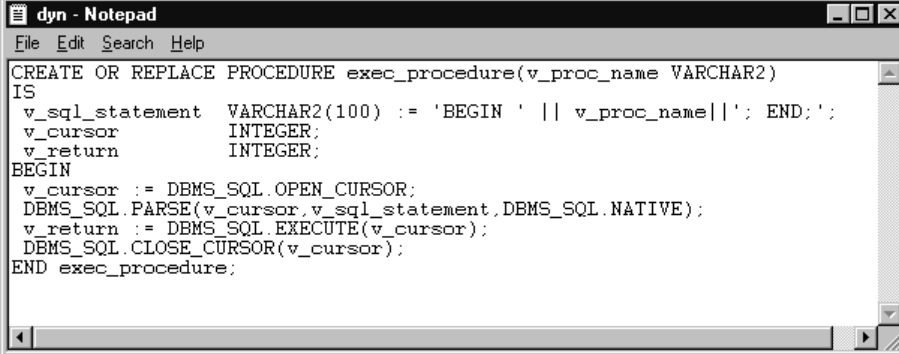
Using the DBMS_SQL package allows the following functions:

- Executing DDL
- Performing dynamic SQL

Practical Uses of the DBMS_SQL Package

- Execute a procedure that takes a table name, column name, and column value as parameter values.
- Execute a procedure that takes the names of a source and destination table, and copies rows from the source table to the destination table.

Using DBMS_SQL



```
dyn - Notepad
File Edit Search Help
CREATE OR REPLACE PROCEDURE exec_procedure(v_proc_name VARCHAR2)
IS
  v_sql_statement VARCHAR2(100) := 'BEGIN ' || v_proc_name || '; END;';
  v_cursor         INTEGER;
  v_return         INTEGER;
BEGIN
  v_cursor := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(v_cursor, v_sql_statement, DBMS_SQL.NATIVE);
  v_return := DBMS_SQL.EXECUTE(v_cursor);
  DBMS_SQL.CLOSE_CURSOR(v_cursor);
END exec_procedure;
```

Sending a message on a pipe

```
SQL> EXECUTE send_message('Testing dynamic SQL');
```

Use dynamic SQL to execute a procedure

```
SQL> EXECUTE exec_procedure('receive_message');
```

Testing dynamic SQL

The DBMS_SQL Package: Characteristics

- Allows developers not to have to embed SQL statements in source programs, but store them in character strings that are input to, or built by, the program at runtime. Doing this permits them to create more general purpose procedures, for example, create a procedure that operates on a table whose name is not known until runtime.
- Allows parsing of any DML or DDL statement, which solves the problem of not being able to parse DDL statements directly using PL/SQL. For example, you can drop a table from within a stored procedure.
- Using this package to execute DDL statements can result in a program hang.
- 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.
- OCI uses bind by addresses, whereas this package uses bind by values.
- DBMS_SESSION.SET_ROLE procedure call is ineffective with embedded SQL statements in stored procedures, but not with DBMS_SQL procedures.

DBMS_SQL Package

- **OPEN_CURSOR**
- **PARSE**
- **BIND_VARIABLE**
- **DEFINE_COLUMN**
- **EXECUTE**
- **FETCH_ROWS**
- **EXECUTE_AND_FETCH**

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: 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
DEFINE_COLUMN	Specifies the variables that are to receive the SELECT values, such as making an INTO clause in a static query (only for the SELECT statement)
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 it in a loop.)
EXECUTE_AND_FETCH	Executes and retrieves a row for the specified cursor. More efficient than the EXECUTE and FETCH_ROWS for a single iteration.)

DBMS_SQL Package

- **COLUMN_VALUE**
- **COLUMN_VALUE_CHAR**
- **COLUMN_VALUE_RAW**
- **COLUMN_VALUE_ROWID**
- **VARIABLE_VALUE**
- **VARIABLE_VALUE_CHAR**
- **VARIABLE_VALUE_RAW**

6-24

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Function or Procedure	Description
COLUMN_VALUE	Returns the value of the cursor element for a given position in a given cursor, for NUMBER, DATE, VARCHAR2, and MLSLABEL datatypes (Is used to access the data fetched by FETCH ROWS call.)
COLUMN_VALUE_CHAR	Returns the same information for CHAR
COLUMN_VALUE_RAW	Returns the same information for RAW
COLUMN_VALUE_ROWID	Returns the same information for ROWID
VARIABLE_VALUE	Returns the value of the named variable for a given cursor, for NUMBER, DATE, VARCHAR2, and MLSLABEL datatypes (Is used to retrieve the values assigned to the output variables of PL/SQL procedures once they are executed.)
VARIABLE_VALUE_CHAR	Returns the same information for CHAR
VARIABLE_VALUE_RAW	Returns the same information for RAW.
VARIABLE_VALUE_ROWID	Returns the same information for ROWID
IS_OPEN	Checks if the specified cursor is currently opened or not
CLOSE_CURSOR	Closes the specified cursor

DBMS_SQL Package

- **VARIABLE_VALUE_ROWID**
- **IS_OPEN**
- **CLOSE_CURSOR**

Rather than embedding SQL statements in source programs, DBMS_SQL allows developers to store them in character strings that are input to, or built by, the program at runtime. Thus developers can create more general purpose procedures: for example, a procedure that operates on a table whose name is not known until runtime.

DBMS_SQL also allows parsing of any DML or DDL statement, which solves the problem of not being able to parse DDL statements directly using PL/SQL, for example, dropping a table from within a stored procedure.

DBMS_SQL Package

- **LAST_ERROR_POSITION**
- **LAST_ROW_COUNT**
- **LAST_ROW_ID**
- **LAST_SQL_FUNCTION_CODE**

DBMS_SQL Functions Locating Errors	Description
LAST_ERROR_POSITION	Returns the byte offset in the SQL statement text where the error occurred (first character is 0)
LAST_ROW_COUNT	Returns the cumulative count of the number of rows fetched
LAST_ROW_ID	Returns the ROWID of the last row fetched.
LAST_SQL_FUNCTION_CODE	Returns the SQL function code for the statement. (Use this function immediately after the execution of the SQL statement; otherwise, the return value is not pertinent.)

Using DBMS_SQL: Example

```
CREATE OR REPLACE PROCEDURE exec_select
(p_table_name      IN      VARCHAR2,
 p_column_name     IN      VARCHAR2,
 p_column_value    IN      VARCHAR2,
 p_name            OUT     NUMBER,
 p_salary          OUT     NUMBER)
IS
  mycursor          INTEGER;
  myreturn          INTEGER;
  --
  v_ename           emp.ename%TYPE;
  v_sal             emp.sal%TYPE;
  v_column_value    VARCHAR(30);
  --
  sql_statement VARCHAR(200) :=
    'SELECT ename,sal FROM '||p_table_name||
    'WHERE ' ||p_column_name||'=:col_value';
  --
```

The example above illustrates how a SQL statement can be stored in a character string, and parsed using DBMS_SQL.

Note: The example is continued on the next two pages.

Using DBMS_SQL: Example

```
BEGIN
/*
  Assign column-value parameter to a variable as cannot
  reference a parameter in call to bind_variable
*/
  v_column_value := upper(p_column_value);
  --
mycursor := dbms_sql.open_cursor;
  --
dbms_sql.parse(mycursor,sql_statement,dbms_sql.native);
  --
dbms_sql.bind_variable(mycursor,'col_value',v_column_value);
  --
dbms_sql.define_column(mycursor,1,v_ename,30);
dbms_sql.define_column(mycursor,2,v_sal);
  --
myreturn := dbms_sql.execute(mycursor);
myreturn := dbms_sql.fetch_rows(mycursor);
IF myreturn = 0 THEN
  RAISE no_data_found;
END IF;
```

Using DBMS_SQL: Example

```
--
dbms_sql.column_value(mycursor,1,v_ename);
dbms_sql.column_value(mycursor,2,v_sal);
myreturn := dbms_sql.fetch_rows(mycursor);
IF myreturn > 1 THEN
    RAISE too_many_rows;
END IF;
--
p_name := v_ename;
p_salary := v_sal;
--
dbms_sql.close_cursor(mycursor);
EXCEPTION
    WHEN others THEN
        dbms_output.put('error - sqlcode is : ');
        dbms_output.put_line(to_char(sqlcode));
        dbms_output.put_line('message is : '||sqlerrm);
        p_name := null;
    END;
/
```

Invoking the General Purpose Procedure

```
VAR v_out1 VARCHAR2(10)
VAR v_out2 NUMBER
SET serveroutput ON
EXECUTE exec_select('&table_name' , '&column_name' , -
'&column_value' , :v_out1 , :v_out2)
PRINT v_out1
PRINT v_out2
```

The above example invokes the previously defined general purpose procedure.

DBMS_OUTPUT Package

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

6-31

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

The DBMS_OUTPUT package outputs values and messages from triggers, stored procedures, and functions.

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 screen for debugging purposes.
- This package allows developers to closely follow the execution of a function or procedure by sending messages and values to the output buffer.

DBMS_DDL Package

- **ALTER_COMPILE**
- **ANALYZE_OBJECT**

6-32

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

The DBMS_DDL package recompiles procedures, functions and packages, and analyzes indexes, tables, and clusters.

Function or Procedure	Description
ALTER_COMPILE	Recompiles procedures, functions, or packages
ANALYZE_OBJECT	Analyzes indexes, tables or clusters. Estimate is for all rows or a percentage of rows

The use of DBMS_DDL is not allowed in triggers, in procedures called from SQL*Forms or in remote sessions.

Practical Uses

- You can recompile your modified procedures, functions, and packages using DBMS_DDL.ALTER_COMPILE.
- 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.ANALYZE_SCHEMA.)



This package gives developers access to COMPILER and ANALYZE SQL statements through PL/SQL environments.

Summary

Additional features of packages:

- **Overloading**
- **Forward Referencing**
- **One-time-only procedures**
- **Purity level of package functions**

Summary

- You can take advantage of a number of preconfigured packages supplied by the Oracle Server.
- All packages are installed in the script *catproc.sql* or each one can be installed individually.
 - DBMS_ALERT
 - DBMS_APPLICATION_INFO
 - DBMS_DDL

DBMS Packages and the Script to Execute

DBMS_ALERT	<i>dbmsalrt.sql</i>
DBMS_APPLICATION_INFO	<i>dbmsutil.sql</i>
DBMS_DDL	<i>dbmsutil.sql</i>
DBMS_LOCK	<i>dbmslock.sql</i>
DBMS_MAIL	<i>dbmsmail.sql</i>
DBMS_OUTPUT	<i>dbmsotpt.sql</i>
DBMS_PIPE	<i>dbmspipe.sql</i>
DBMS_SESSION	<i>dbmsutil.sql</i>
DBMS_SHARED_POOL	<i>dbmsspool.sql</i>
DBMS_SQL	<i>dbmssql.sql</i>
DBMS_TRANSACTION	<i>dbmsutil.sql</i>
DBMS_UTILITY	<i>dbmsutil.sql</i>

Summary

- DBMS_LOCK
- DBMS_OUTPUT
- DBMS_PIPE
- DBMS_SESSION

You can use packages as libraries of routines when developing a database application.



For more information, see *Oracle8 Server Application Developer's Guide*.

Practice Overview

- **Using overloaded subprograms**
- **Creating a one-time-only procedure**
- **Using DBMS_SQL for dynamic SQL**

Practice 6

1. Create a new package to implement a new business rule.
 - a. Create a procedure called `chk_dept_job` to verify whether a given combination of department number and job is a valid one. In this case “valid” means that it must be a combination that currently exists in the EMP table.

Notes:

- 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.

```
SQL> execute chk_pack.chk_dept_job(20,'CLERK')
```

- c. Test your `CHK_DEPT_JOB` package procedure by executing the following command.

```
SQL> execute chk_pack.chk_dept_job(40,'CLERK')
```

2. Create two functions, each called `PRINT_IT` to print a date, or a number, or a boolean value depending on how the function was invoked.

Notes:

- 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 the number, use “999,999.00” as the output format.
- a. Test the first version of `PRINT_IT` with the following command.

```
SQL> variable todays_date varchar2(20)
SQL> execute :todays_date := over_load.print_it(sysdate)
PL/SQL procedure successfully completed.
SQL> print todays_date
TODAYS_DATE
-----
January,29/1998
```

- b. Test the second version of `PRINT_IT` with the following command.

```
SQL> variable g_emp_sal number
SQL> execute :g_emp_sal := over_load.print_it('33,600')
PL/SQL procedure successfully completed.
SQL> print g_emp_sal
G_EMP_SAL
-----
33600
```

Practice 6 (continued)

3. 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.
 - a. Test the `DROP_TABLE` procedure by creating a new table called `EMP_DUP` as a copy of the `EMP` table, then executing the `DROP_TABLE` procedure to drop the `EMP_DUP` table.

7

Creating Database Triggers

Objectives

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

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

Lesson Aim

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

Overview of Triggers

- **A trigger is a PL/SQL block that executes implicitly whenever a particular event takes place.**
- **A trigger can be either a database trigger or an application trigger.**

Database triggers execute implicitly when an INSERT, UPDATE, or DELETE statement (triggering statement) is issued against the associated table, no matter which user is connected or which application is used.

Application triggers execute implicitly whenever a particular event occurs within an application. An example of an application that uses triggers extensively is one developed with Developer/2000 Form Builder.

Note: Database triggers can be defined only on tables, not on views. However, if a DML operation is issued against a view, triggers on the base table(s) of a view are fired.

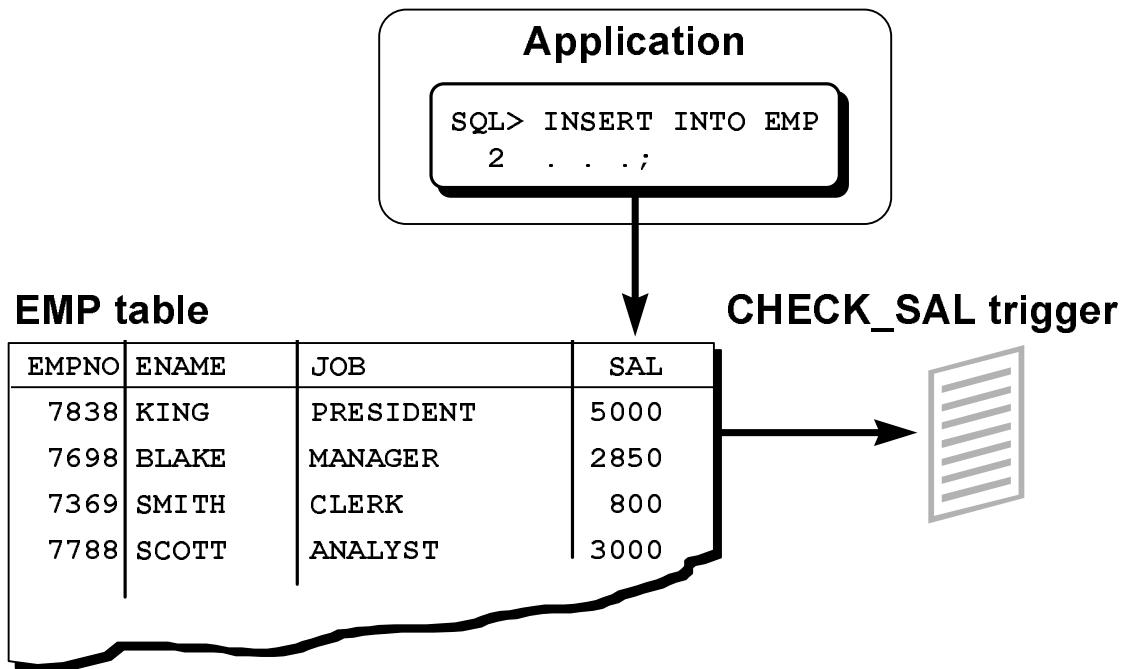
Designing Triggers: Guidelines

- **Perform related actions**
- **Use triggers for global operations**
- **Do not reinvent the wheel**
- **Watch out for overkill**

Use the following guidelines when designing database triggers

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Only use database triggers 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.
- 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.

Database Trigger: Example



7-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

In the example on the slide, a database trigger checks salary values. Values that are out of range according to the job category can be rejected, or could be allowed and recorded in an audit table.

Creating Triggers

- **Trigger timing: BEFORE or AFTER**
- **Triggering event: INSERT or UPDATE or DELETE**
- **Table name: On table**
- **Trigger type: Row or statement**
- **When clause: Restricting condition**
- **Trigger body: DECLARE
BEGIN
END;**

7-6

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Database Trigger

Before coding the trigger body, decide on the components of the trigger: 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
Triggering event	Which data manipulation operation on the table 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



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.

Trigger Components

Trigger Timing: When should the trigger fire?

- **BEFORE:** The code in the trigger body will execute before the triggering DML event.
- **AFTER:** The code in the trigger body will execute after the triggering DML event.

BEFORE Triggers

This type of trigger is frequently used in the following situations:

- When the trigger action should determine whether that triggering statement should be allowed to complete. This allows 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.

AFTER Triggers

This type of trigger is frequently used in the following situations:

- When you want the triggering statement to complete before executing the triggering action.
- If a BEFORE trigger is already present, and an AFTER trigger can perform different actions on the same triggering statement.

Trigger Components

Trigger Timing: When should the trigger fire?

- **INSTEAD OF:** The code in the trigger body will execute instead of the the triggering statement. Used for VIEWS that are not otherwise modifiable.

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 normal INSERT, UPDATE, and DELETE statements against the view and the INSTEAD OF trigger works invisibly in the background performing the action coded in the trigger body directly on the underlying table(s).

Trigger Components

Triggering Event:

What DML operation will cause the trigger to execute?

- **INSERT**
- **UPDATE**
- **DELETE**
- **Any combination of the above**

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, you can include a column list to identify which column(s) must be changed to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement, as they always affect entire rows.
- The triggering event can contain multiple DML statements. In this way, you can differentiate what code to execute depending on the statement that caused the trigger to fire.

Trigger Components

Trigger Type:

How many times should the trigger body execute when the triggering event takes place?

- **Statement:** The trigger body executes once for the triggering event. This is the default.
- **Row:** The trigger body executes once for each row affected by the triggering event.

7-10

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Statement Triggers and Row Triggers

You can specify the number of times the trigger action is to 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 data of rows that are affected or 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 the table is affected by the triggering event. If the triggering event affects no row(s), a row trigger is not executed at all.

Row triggers are useful if the trigger action depends on data of rows that are affected or data provided by the triggering event itself.

Trigger Components

Trigger Body:

What action should the trigger perform?

- The trigger body is defined with an anonymous PL/SQL block.

```
[DECLARE]
BEGIN
[EXCEPTION]
END;
```

7-11

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Trigger Body

The trigger action defines what needs to be done when the triggering event is issued. It can contain SQL and PL/SQL statements, define PL/SQL constructs such as variables, cursors, exceptions and so on.

Additionally, row triggers have access to the old and new column values of the row being processed by the trigger, using correlation names.

Firing Sequence of Database Triggers on a Single Row

DEPT table

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

→ **BEFORE statement trigger**

→ **BEFORE row trigger**
→ **AFTER row trigger**

→ **AFTER statement trigger**

Creating Row or Statement Triggers?

Create a statement trigger or a row trigger based upon the requirement to fire the trigger 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.

Statement and Row Triggers

Example 1

```
SQL> INSERT INTO dept (deptno, dname, loc)
2  VALUES (50, 'EDUCATION', 'NEW YORK');
```

Example 2

```
SQL> UPDATE emp
2  SET sal = sal * 1.1
3  WHERE deptno = 30;
```

Example 1

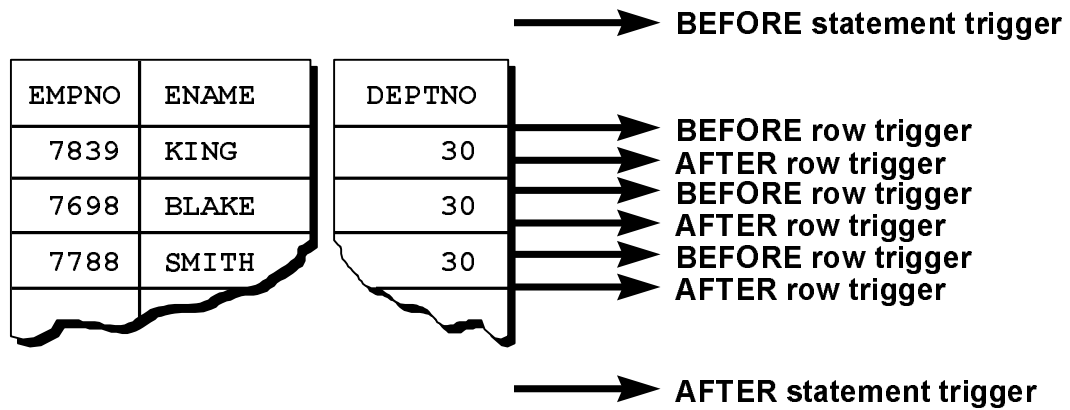
This SQL statement does not differentiate statement triggers from row triggers, since exactly one row is inserted into the table using this syntax.

When the triggering data manipulation statement affects multiple rows, the statement trigger fires once and the row trigger fires once for each row affected by the statement.

Example 2

This SQL statement would cause 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.

Firing Sequence of Database Triggers on Multiple Rows



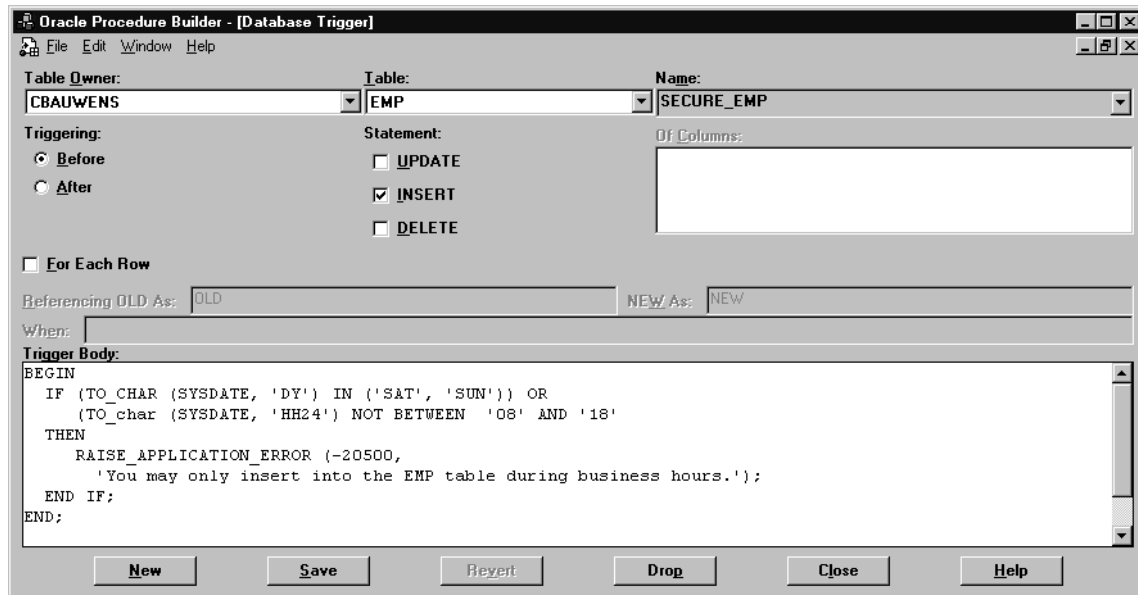
Syntax for Creating Statement Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing event1 [OR event2 OR event3]
ON table_name
PL/SQL block;
```

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 column] DELETE
<i>table name</i>	Indicates the table associated with the trigger
<i>PL/SQL block</i>	Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, ending with END

Creating Statement Triggers Using Procedure Builder



Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Creating a Statement Trigger Using 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 by clicking on the corresponding list item buttons.
5. Click New to start creating the trigger.
6. Select the Triggering radio button to choose the timing component.
7. Select the Statement check box(es) to choose the event(s) component.
8. 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.

Before Statement Trigger: Example

```
SQL> CREATE OR REPLACE TRIGGER secure_emp
  2  BEFORE INSERT ON emp
  3  BEGIN
  4    IF (TO_CHAR (sysdate, 'DY') IN ('SAT', 'SUN'))
  5      OR (TO_CHAR (sysdate, 'HH24') NOT BETWEEN
  6        '08' AND '18'
  7      THEN RAISE_APPLICATION_ERROR (-20500,
  8        'You may only insert into EMP during normal
  9        hours. ');
 10  END IF;
 11  END;
 12  /
```

7-17

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example: Create a BEFORE Statement Trigger

You can create a BEFORE statement trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

Create a trigger to restrict inserts into the EMP table to certain business hours, Monday through Friday.

If a user attempted to insert a row into the EMP table on Saturday, for example, the user will see the message, the trigger will fail, and the triggering statement will be rolled back.

RAISE_APPLICATION_ERROR is a server-side built-in procedure that prints a message 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.

Example

```
SQL> INSERT INTO emp (empno, ename, deptno)
      2 VALUES          (7777, 'BAUWENS', 40);
INSERT INTO emp (empno, ename, deptno)
      *
ERROR at line 1:
ORA-20500: You may only insert into EMP during
normal hours.
ORA-06512: at "SCOTT.SECURE_EMP", line 4
ORA-04088: error during execution of trigger
'SCOTT.SECURE_EMP'
```

Example

Attempt to insert a row into the EMP table during nonbusiness hours.

Using Conditional Predicates

```
SQL>CREATE OR REPLACE TRIGGER secure_emp
 2 BEFORE INSERT OR UPDATE OR DELETE ON emp
 3 BEGIN
 4   IF (TO_CHAR (sysdate,'DY') IN ('SAT','SUN')) OR
 5   (TO_CHAR (sysdate, 'HH24') NOT BETWEEN '08' AND '18') THEN
 6     IF DELETING THEN
 7       RAISE_APPLICATION_ERROR (-20502,
 8       'You may only delete from EMP during normal hours. ');
 9     ELSIF INSERTING THEN
10       RAISE_APPLICATION_ERROR (-20500,
11       'You may only insert into EMP during normal hours. ');
12     ELSIF UPDATING ('SAL') THEN
13       RAISE_APPLICATION_ERROR (-20503,
14       'You may only update SAL during normal hours. ');
15     ELSE
16       RAISE_APPLICATION_ERROR (-20504,
17       'You may only update EMP during normal hours. ');
18     END IF;
19   END IF;
20 END;
21 /
```

7-19

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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 EMP table to certain business hours, Monday through Friday.



Also use BEFORE statement triggers to initialize global variables or flags, and to validate complex business rules.

After Statement Trigger: Example

```
SQL>CREATE OR REPLACE TRIGGER check_salary_count
 2 AFTER UPDATE OF sal ON emp
 3 DECLARE
 4   v_salary_changes NUMBER;
 5   v_max_changes     NUMBER;
 6 BEGIN
 7   SELECT upd, max_upd
 8   INTO   v_salary_changes, v_max_changes
 9   FROM   audit_table
10  WHERE  user_name = user
11  AND    table_name = 'EMP'
12  AND    column_name = 'SAL';
13  IF v_salary_changes > v_max_changes THEN
14    RAISE_APPLICATION_ERROR (-20501,
15    'You may only make a maximum of ' ||
16    TO_CHAR (v_max_changes) ||
17    ' changes to the SAL column');
18  END IF;
19 END;
20 /
```

7-20

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

Create an AFTER statement trigger in order to audit the triggering operation or perform a calculation after an operation has completed.

Suppose you have a user-defined audit table (see next page) that lists users and counts their data manipulation operations. After any user has updated the SAL column in the EMP table, use the audit table to ensure that the number of salary changes does not exceed the maximum permitted for that user.

User Audit Table

USER_NAME	TABLERNAME	COLUMN_NAME	INS	UPD	DEL
SCOTT	EMP	SAL	1	1	1
SCOTT	EMP			1	
JONES	EMP		0	0	0

Continuation

MAX_INS	MAX_UPD	MAX_DEL
5	5	5
5	0	0

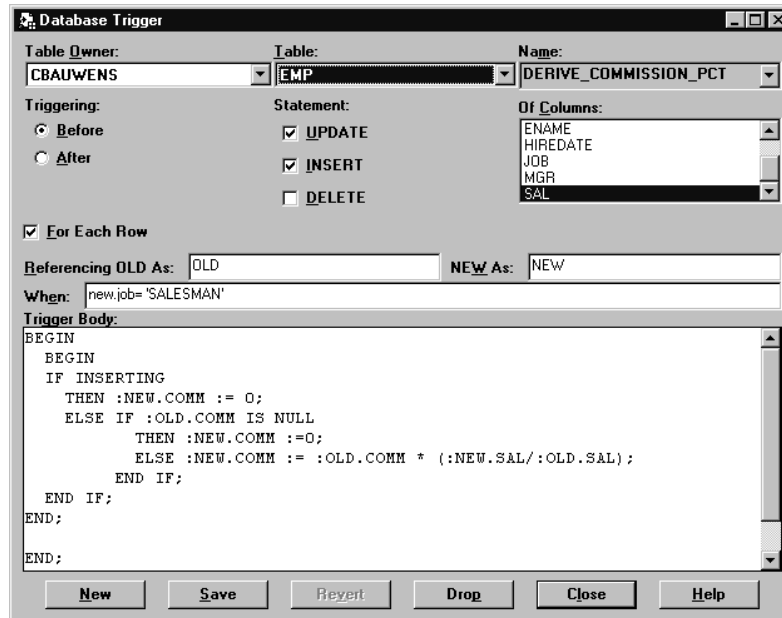
Creating a Row Trigger

```
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]
PL/SQL block;
```

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
<i>event</i>	Identifies the data manipulation operation that causes the trigger to fire: INSERT UPDATE [OF column] DELETE
<i>table name</i>	Indicates the table associated with trigger
<i>referencing</i>	Specifies correlation names for the old and new values of the current row (The default are OLD and NEW.)
<i>FOR EACH ROW</i>	Designates the trigger to be a row trigger
<i>WHEN</i>	Specifies the trigger restriction (This conditional predicate is evaluated for each row to determine whether or not the trigger body is executed.)
<i>PL/SQL block</i>	Is the trigger body that defines the action performed by the trigger, beginning with either DECLARE or BEGIN, ending with END

Creating Row Triggers Using Procedure Builder



7-23

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Steps to Create a Row Trigger Using 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 by clicking on the corresponding list items.
5. Click New to start creating the trigger.
6. Choose the Triggering radio button to choose the timing component.
7. Choose the Statement check box(es) to choose the event(s) component.
8. Choose the For Each Row check box to designate the trigger as a row trigger.
9. Enter Referencing OLD as and NEW as if you want to modify the correlation names. Enter a When condition to restrict the execution of the trigger. These components are optional and available only with ROW triggers.
10. Enter the trigger code.
11. Click Save. The trigger code will now be compiled by the PL/SQL engine in the server. Once successfully compiled, the trigger is stored in the database and automatically enabled.

After Row Trigger: Example

```
SQL>CREATE OR REPLACE TRIGGER audit_emp
 2 AFTER DELETE OR INSERT OR UPDATE ON emp
 3 FOR EACH ROW
 4 BEGIN
 5   IF DELETING THEN
 6     UPDATE audit_table SET del = del + 1
 7     WHERE user_name = user AND table_name = 'EMP'
 8     AND column_name IS NULL;
 9   ELSIF INSERTING THEN
10     UPDATE audit_table SET ins = ins + 1
11     WHERE user_name = user AND table_name = 'EMP'
12     AND column_name IS NULL;
13   ELSIF UPDATING ('SAL') THEN
14     UPDATE audit_table SET upd = upd + 1
15     WHERE user_name = user AND table_name = 'EMP'
16     AND column_name = 'SAL';
17   ELSE /* The data manipulation operation is a general UPDATE. */
18     UPDATE audit_table SET upd = upd + 1
19     WHERE user_name = user AND table_name = 'EMP'
20     AND column_name IS NULL;
21   END IF;
22 END;
23 /
```

7-24

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

Create a row trigger to keep a running count of data manipulation operations by different users on database tables.



If a trigger routine does not have to take place before the triggering operation, create an AFTER row trigger rather than a BEFORE row trigger.

Using Old and New Qualifiers

```
SQL>CREATE OR REPLACE TRIGGER audit_emp_values
  2 AFTER DELETE OR INSERT OR UPDATE ON emp
  3 FOR EACH ROW
  4 BEGIN
  5     INSERT INTO audit_emp_values (user_name,
  6         timestamp, id, old_last_name, new_last_name,
  7         old_title, new_title, old_salary, new_salary)
  8     VALUES (USER, SYSDATE, :old.empno, :old.ename,
  9         :new.ename, :old.job, :new.job,
 10         :old.sal, :new.sal);
 11 END;
 12 /
```

Example

Create a trigger on the EMP table to add rows to a user table, AUDIT_EMP_VALUES, logging a user's activity against the EMP 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.

User Audit_Emp_Values Table

USER_NAME	TIMESTAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME
EGRAVINA	12-NOV-97	7950	NULL	HUTTON
NGREENBE	10-DEC-97	7844	MAGEE	TURNER

Continuation

OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
NULL	ANALYST	NULL	3500
CLERK	SALESMAN	1100	1100

7-26

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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 only available in ROW triggers.
- Prefix these qualifiers with a colon in every SQL and PL/SQL statement.
- There is no colon prefix if they are referenced in the WHEN restricting condition.

Restricting a Row Trigger

```
SQL>CREATE OR REPLACE TRIGGER derive_commission_pct
 2 BEFORE INSERT OR UPDATE OF sal ON emp
 3 FOR EACH ROW
 4 WHEN (new.job = 'SALESMAN')
 5 BEGIN
 6   IF INSERTING THEN :new.comm := 0;
 7   ELSE /* UPDATE of salary */
 8     IF :old.comm IS NULL THEN
 9       :new.comm :=0;
10   ELSE
11     :new.comm := :old.comm * (:new.sal/:old.sal);
12   END IF;
13 END IF;
14 END;
15 /
```

7-27

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

To restrict the trigger action to those rows that satisfy a certain condition, provide a WHEN clause.

Create a trigger on the EMP table to calculate an employee's commission when a row is added to the EMP table or an employee's salary is modified.



The NEW qualifier does not need to be prefixed with a colon in the WHEN clause.

Differentiating Between Triggers and Stored Procedures

Triggers	Procedure
Use CREATE TRIGGER	Use CREATE PROCEDURE
Data dictionary contains source and p-code	Data dictionary contains source and p-code
Implicitly invoked	Explicitly invoked
COMMIT, SAVEPOINT, ROLLBACK not allowed	COMMIT, SAVEPOINT, ROLLBACK allowed

7-28

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Database Triggers and Stored Procedures

There are differences between database triggers and stored procedures:

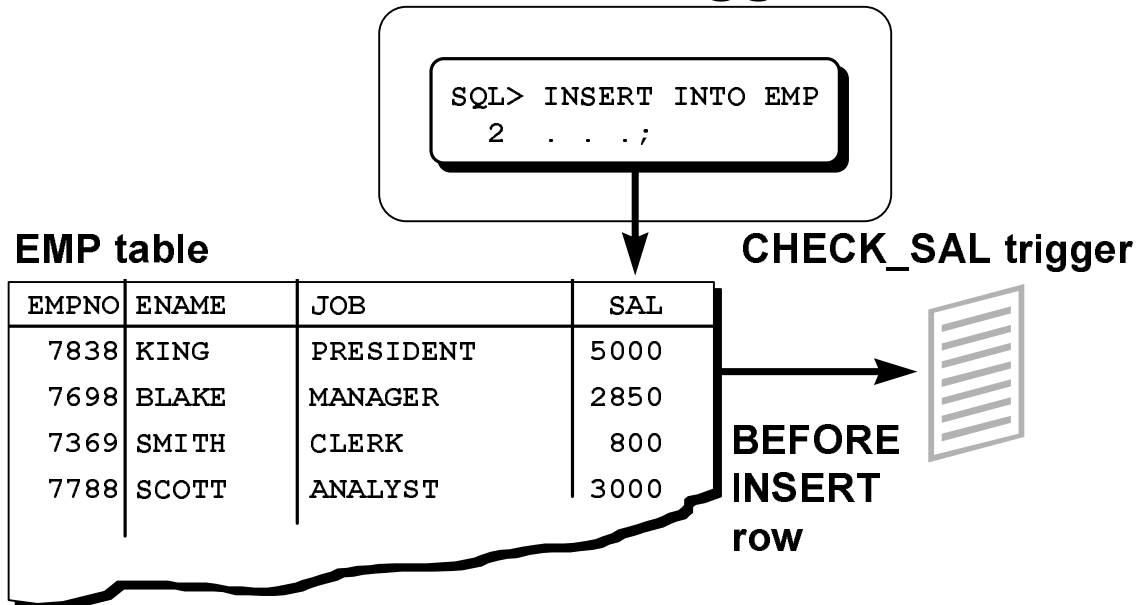
Database Trigger	Stored Procedure
Invoked implicitly.	Invoked explicitly.
COMMIT, ROLLBACK, and SAVEPOINT statements are not allowed within the trigger body.	COMMIT, ROLLBACK, and SAVEPOINT statements are permitted within the procedure body.



Triggers are fully compiled when the **CREATE TRIGGER** command is issued, and the p-code is stored in the Data Dictionary. Hence, firing the trigger no longer requires the opening of a shared cursor to run the trigger action. Instead, the trigger is executed directly.

If errors occur during the compilation of a trigger, the trigger is still created.

Differentiating Between Triggers and Form Builder Triggers



7-29

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Differences Between a Database Trigger and a Form Builder Trigger

Do not confuse database triggers with Developer/2000 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 data manipulation statement	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 not 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 Re-enable a database trigger

```
ALTER TRIGGER trigger_name DISABLE | ENABLE
```

Disable or Re-enable all triggers for a table

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS
```

Recompile a trigger for a table

```
ALTER TRIGGER trigger_name COMPILE
```



Trigger Modes: Enabled or Disabled

- When a trigger is first created, it is enabled automatically.
- For enabled triggers the Oracle Server checks integrity constraints and guarantees that triggers cannot compromise integrity constraints. In addition the Oracle Server provides read-consistent views for queries and constraints, manages the dependencies, and provides two-phase commit 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, through utilities such as SQL*Loader. You may also want to disable the trigger when it references a database object that is currently unavailable, due to a failed network connection, disk crash, off-line datafile, or off-line 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 RECOMPILE option, the trigger recompiles regardless of whether it is valid or invalid.

Removing Triggers

To remove a trigger from the database,
use the **DROP TRIGGER** syntax:

```
DROP TRIGGER trigger_name
```

Trigger Test Cases

- **Test each of the triggering data operations, as well as non-triggering 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.**

7-32

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Testing Triggers

Ensure the trigger works properly by testing a number of cases separately.



Take advantage of the DBMS_OUTPUT procedures to debug triggers.

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 the DML statement and perform integrity constraint checking
 - c. Execute all AFTER ROW triggers
3. Complete deferred integrity constraint checking
4. Execute all AFTER STATEMENT triggers

7-33

Copyright © Oracle Corporation, 1998. All rights reserved.

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, including 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 be undergoing 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).

Rules Governing Triggers

- **Rule 1: Do *not* change data in the primary key, foreign key, or unique key columns of a constraining table.**
- **Rule 2: Do *not* read data from a mutating table.**

Reading and writing data using triggers is subject to certain rules.

Table

ORACLE®

Constraining Table

A constraining table is a table that the triggering event might need to read, either directly, for a SQL statement, or indirectly, for a declarative referential integrity constraint. Tables are not considered constraining for STATEMENT triggers.

Example

Try changing data in a constraining table.

When the value of DEPTNO changes in the DEPT parent table, trying to cascade the update to the corresponding rows in the EMP child table produces a runtime error.

Constraining Table: Example

```
SQL>CREATE OR REPLACE TRIGGER cascade_updates
  2 AFTER UPDATE OF deptno on DEPT
  3 FOR EACH ROW
  4 BEGIN
  5     UPDATE emp
  6     SET     emp.deptno = :new.deptno
  7     WHERE  emp.deptno = :old.deptno;
  8 END;
  9 /
```

Constraining Table Example

This trigger attempts to cascade the update of the parent key in the DEPT table to the foreign key for child records in the EMP table.

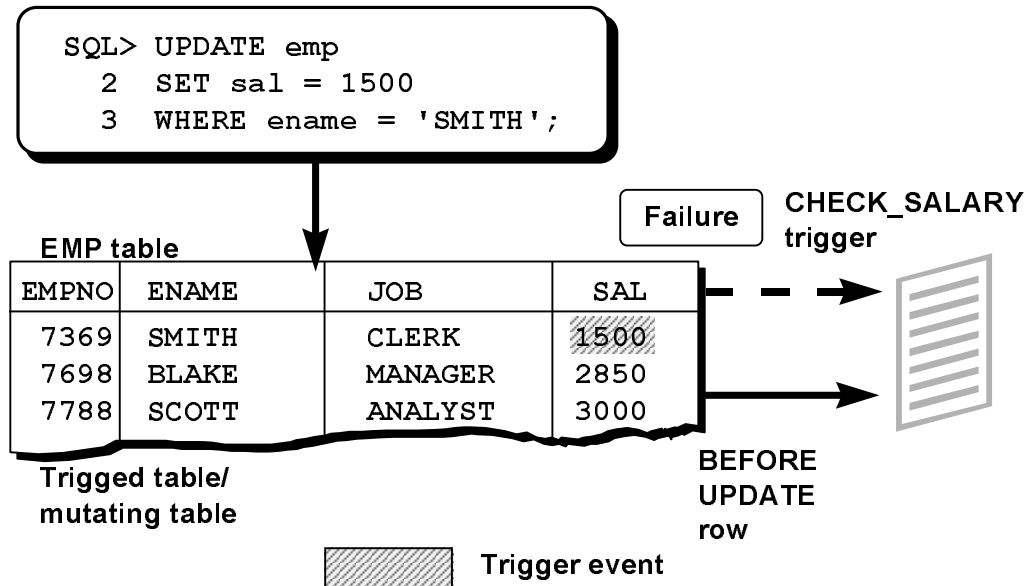
Constraining Table: Example

```
SQL> UPDATE dept
  2  SET    deptno = 1
  3  WHERE  deptno = 30;
*
ERROR at line 1:
ORA-04091: table DEPT is mutating, trigger/function
may not see it
```

Constraining Table Example (continued)

An error results when the user tries to modify the DEPT table. The triggered table, DEPT, references EMP through a FOREIGN KEY constraint. Therefore, EMP is said to be a constraining table. The CASCADE_UPDATES trigger tries to change data in the constraining table, which is not allowed.

Reading Data from a Mutating Table



7-38

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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

```
SQL>CREATE OR REPLACE TRIGGER check_salary
  2 BEFORE INSERT OR UPDATE OF sal, job ON emp
  3 FOR EACH ROW
  4 WHEN (new.job <> 'PRESIDENT')
  5 DECLARE
  6     v_minsalary emp.sal%TYPE;
  7     v_maxsalary emp.sal%TYPE;
```

Mutating Table Example

This trigger, CHECK_SALARY, guarantees that whenever a new employee is added to the EMP table or an existing employee's salary or job title is changed, the employee's salary falls within the established salary range for the employee's job.

Mutating Table: Example

```
8 BEGIN
9     SELECT MIN(sal), MAX(sal)
10    INTO   v_minsalary, v_maxsalary
11   FROM    emp
12  WHERE    job = :new.job;
13  IF :new.sal < v_minsalary OR
14     :new.sal > v_maxsalary THEN
15      RAISE_APPLICATION_ERROR(-20505,
16                              'Out of range');
17  END IF;
18 END;
19 /
```

Mutating Table: Example

```
SQL> UPDATE emp
  2  SET sal = 1500
  3  WHERE ename = 'SMITH';
*
ERROR at line 2
ORA_4091 : Table EMP is mutating, trigger/function
may not see it
ORA_06512: at line 4
ORA_04088: error during execution of trigger
'check_salary'
```

Mutating Table Example (continued)

Try to read from a mutating table.

Trying to restrict the salary within a range between the minimum existing value and the maximum existing value produces a runtime error. The EMP table is mutating, or in a state of change; therefore the trigger cannot read from it.

Implementation of Triggers

- **Security**
- **Auditing**
- **Data integrity**
- **Referential integrity**
- **Table replication**
- **Derived data**
- **Event logging**

Develop database triggers in order to enhance features that cannot otherwise be implemented 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

```
SQL> GRANT SELECT, INSERT, UPDATE, DELETE
      2  ON      emp
      3  TO      CLERK;    -- database role
SQL> GRANT CLERK TO SCOTT;
```

Security

Develop schemas and roles within Oracle 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

```
SQL>CREATE OR REPLACE TRIGGER secure_emp
 2 BEFORE INSERT OR UPDATE OR DELETE ON emp
 3 DECLARE
 4   v_dummy VARCHAR2(1);
 5 BEGIN
 6   IF TO_CHAR (sysdate, 'DY' IN ('SAT','SUN'))
 7   THEN RAISE_APPLICATION_ERROR (-20506,
 8     'You may only change data during normal business
 9     hours. ');
10   END IF;
11   SELECT COUNT(*) INTO v_dummy FROM holiday
12   WHERE holiday_date = TRUNC (sysdate);
13   IF v_dummy > 0 THEN RAISE_APPLICATION_ERROR (-20507,
14     'You may not change data on a holiday. ');
15   END IF;
16 END;
17 /
```

7-44

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Security (continued)

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.

Auditing Using the Server Facility

```
SQL> AUDIT INSERT, UPDATE, DELETE  
2  ON      emp  
3  BY ACCESS  
4  WHENEVER SUCCESSFUL;
```

Audit Data Operations

Audit data operations within Oracle8.

- Audit data retrieval, data manipulation, and data definition statements.
- Write the audit trail to the 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.

Auditing Using a Trigger

```
SQL>CREATE OR REPLACE TRIGGER audit_emp_values
 2 AFTER DELETE OR INSERT OR UPDATE ON emp
 3 FOR EACH ROW
 4 BEGIN
 5     IF audit_emp_package.g_reason IS NULL THEN
 6         RAISE_APPLICATION_ERROR (-20059, 'Specify a reason
 7         for the data operation with the procedure
 8         SET_REASON before proceeding. ');
 9     ELSE
10         INSERT INTO audit_emp_values (user_name, timestamp, id,
11         old_last_name, new_last_name, old_title, new_title,
12         old_salary, new_salary, comments)
13         VALUES (user, sysdate, :old.empno, :old.ename,
14         :new.ename, :old.job, :new.job, :old.sal,
15         :new.sal, :audit_emp_package.g_reason);
16     END IF;
17 END;
18 /

SQL>CREATE TRIGGER cleanup_audit_emp
 2 AFTER INSERT OR UPDATE OR DELETE ON emp
 3 BEGIN
 4     audit_emp_package.g_reason := NULL;
 5 END;
 6 /
```

Audit Data Values

Audit actual data values with triggers.

- 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.

Enforce Data Integrity Within the Server

```
SQL> ALTER TABLE emp ADD  
2 CONSTRAINT ck_salary CHECK (sal >= 500);
```

Data Integrity

Enforce data integrity within the Oracle Server.

Develop triggers to handle more complex data integrity rules.

Enforce standard data integrity rules: not null, not unique, primary key, and foreign key.

- Provide constant default values.
- Enforce static constraints.
- Enable and disable dynamically.

Example

Ensure that the salary is at least \$500.

Protect Data Integrity with a Trigger

```
SQL>CREATE OR REPLACE TRIGGER check_salary
 2 BEFORE UPDATE OF sal ON emp
 3 FOR EACH ROW
 4 WHEN (new.sal < old.sal) OR
 5      (new.sal > old.sal * 1.1)
 6 BEGIN
 7   RAISE_APPLICATION_ERROR (-20508,
 8   'Do not decrease salary nor increase by
 9   more than 10%.');
10 END;
11 /
```

Data Integrity (continued)

Protect data integrity with a trigger. 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

Ensure that the salary is never decreased nor increased more than 10 percent at a time.

Enforce Referential Integrity Within the Server

```
SQL> ALTER TABLE emp
  2  ADD CONSTRAINT emp_deptno_fk
  3  FOREIGN KEY (deptno) REFERENCES dept(deptno)
  4  ON DELETE CASCADE;
```

Referential Integrity

Enforce referential integrity within the server.

- Restrict updates and deletes
- Cascade deletes
- Enable and disable dynamically

Example

When a department is removed from the DEPT parent table, cascade the delete to the corresponding rows in the EMP child table.

Protect Referential Integrity with a Trigger

```
SQL>CREATE OR REPLACE TRIGGER cascade_updates
2 AFTER UPDATE OF deptno ON dept
3 FOR EACH ROW
4 BEGIN
5     UPDATE emp
6     SET     emp.deptno = :new.deptno
7     WHERE  emp.deptno = :old.deptno;
8 END;
9 /
```

Referential Integrity (continued)

Develop triggers to implement nonstandard referential integrity.

- Cascade updates.
- Set to NULL on updates and deletes.
- Set to a default value on updates and deletes.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.
- Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency.

Example

Enforce referential integrity with a trigger. When the value of DEPTNO changes in the DEPT parent table, cascade the update to the corresponding rows in the EMP child table. *This example works only if there is no referential integrity between EMP and DEPT within the table definitions.*

Replicate a Table Within the Server

```
SQL> CREATE SNAPSHOT emp_copy AS  
2  SELECT * FROM emp@ny;
```

Copy 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, replicate tables using triggers.

Example

In San Francisco, create a snapshot of the remote EMP table in New York.

Replicate a Table with a Trigger

```
SQL>CREATE OR REPLACE TRIGGER emp_replica
 2 BEFORE INSERT OR UPDATE ON emp
 3 FOR EACH ROW
 4 BEGIN /*Only proceed if user init. data operation,
 5        NOT the casc. trigger.*/
 6     IF INSERTING THEN
 7         IF :new.flag IS NULL THEN
 8             INSERT INTO emp@sf VALUES (:new.empno,
 9             :new.ename,...,'B');
10         :new.flag = 'A';
11     ELSE /* Updating. */
12         IF :new.flag = :old.flag THEN
13             UPDATE emp@sf SET ename = :new.ename, ...,
14                             FLAG = :new.flag
15             WHERE empno = :new.empno;
16         END IF;
17         IF :old.flag = 'A' THEN :new.flag := 'B';
18         ELSE :new.flag := 'A';
19         END IF;
20     END IF;
21 END;
22 /
```

7-52

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Table Replication

Replicate a table with a trigger.

- Copy tables synchronously, in real time.
- Usually 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 EMP table to San Francisco.

Compute Derived Data Within the Server

```
SQL> UPDATE dept
2> SET total_sal = (SELECT SUM(salary)
3>                  FROM emp
4>                  WHERE emp.deptno = dept.deptno) ;
```

Derive Data

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, with triggers, keep running computations of derived data.

Example

Keep the salary total for each department within a special TOTAL_SALARY column of the DEPT table.

Compute Derived Values with a Trigger

```
SQL>CREATE OR REPLACE PROCEDURE increment_salary
 2  (v_id      IN dept.deptno%TYPE,
 3   v_salary  IN dept.total_salary%TYPE)
 4  IS
 5  BEGIN
 6      UPDATE dept
 7      SET    total_sal = NVL (total_sal,0)+ v_salary
 8      WHERE  deptno = v_id;
 9  END increment_salary;
10  /
```

```
SQL>CREATE OR REPLACE TRIGGER compute_salary
 2  AFTER INSERT OR UPDATE OF sal OR DELETE ON emp
 3  FOR EACH ROW
 4  BEGIN
 5  IF DELETING THEN increment_salary(:old.deptno, -1 * :old.sal);
 6  ELIF UPDATING THEN increment_salary(:new.deptno,
 7                                     :new.sal-:old.sal);
 8  ELSE /*inserting*/ increment_salary(:new.deptno, :new.sal);
 9  END IF;
10  END;
11  /
```

7-54

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Derive Data (continued)

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 DEPT table.

Log Events with a Trigger

```
SQL>CREATE OR REPLACE TRIGGER notify_reorder_rep
 2 AFTER UPDATE OF amount_in_stock, reorder_point ON inventory
 3 FOR EACH ROW
 4 WHEN new.amount_in_stock <= new.reorder_point
 5 DECLARE
 6     v_descrip          product.descrip%TYPE;
 7     v_msg_text         VARCHAR2(2000);
 8 BEGIN
 9     SELECT descrip INTO v_descrip
10 FROM   PRODUCT WHERE prodid = :new.product_id;
11 v_msg_text := 'It has come to my personal attention that,
12             due to recent '
13             CHR(10) || 'transactions, our inventory for product # ' ||
14             TO_CHAR(:new.product_id) || '--'
15             || v_name || '-- has fallen' || CHR(10) || CHR(10) ||
16             'Yours,' || CHR(10) || user || '.';
17     dbms_mail.send ('Inventory', user, null, null, 'Low
18 Inventory', null, v_msg_text);
19 END;
20 /
```

7-55

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Within the server, log events by querying data and performing operations manually, in order to send electronic mail when inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package, DBMS_MAIL to send the e-mail.

Logging Events Within the Server

- Explicitly query data to determine whether an operation is necessary.
- In a second step, perform the operation, such as sending electronic mail.

Using Triggers to Log Events

- Implicitly perform the operation transparent to the user, such as firing off an automatic electronic memo.
- Modify data and perform its dependent operation in a single step.
- With triggers, log events automatically as data is changing.

Logging Events Transparently

In the trigger code:

- CHR(10) is a carriage return.
- Reorder_point is not null.

Benefits of Database Triggers

Improved data security

- **Provide value-based security checks**
- **Provide value-based auditing**

Improved data integrity

- **Enforce dynamic data integrity constraints**
- **Enforce complex referential integrity constraints**
- **Ensure related operations are performed together implicitly**

Summary

Procedure

Package

Trigger

7-57

Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

Construct	Usage
Procedure	PL/SQL programming block 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 Overview

- **Creating statement and row triggers**
- **Creating advanced triggers to add to the capabilities of the Oracle database**

Practice 7

1. DML will only be allowed on tables during normal office hours of 8:45 in the morning until 5:30 in the afternoon, Monday through Friday.
 - a. Create a stored procedure called `SECURE_DML` that will fail outside of these hours, returning the message:
“You may only make data changes during normal office hours”
2. Create a trigger on the `PRODUCT` table which calls the above procedure.
 - a. Test the procedure by temporarily modifying the hours in the procedure and attempting to insert a new record into the `PRODUCT` table. After testing, reset the procedure hours as specified in step 1.
3. A sales person’s commission should change for any new orders or changes to existing orders. Their commission is stored in the `COMM` column of the `EMP` table. A sales person is assigned to a particular customer in the `CUSTOMER` table.
 - a. Create a procedure that will update the relevant sales person’s commission. Use parameters to accept the customer id, old order total and new order total from the calling trigger. The procedure will then need to find the appropriate employee number from the `CUSTOMER` table and update the sales person’s record in the `EMP` table, adding the new commission to the existing value. Assume for this practice a fixed commission rate of 5%.
 - b. Create a trigger on the `ORD` table which will call the procedure, passing the required parameters.
 - c. Use two packaged procedures to alter the customer’s order, `UPD_ITEM` and `ADD_ITEM` in the `ITEM_PACK` package. There is a script in your working directory, *p7_3.sql* that will create `ITEM_PACK`.
 - d. After modifying order 601, verify that `WARD`’s commission has increased by 0.03. The original commission was 500.

Practice 7 (continued)

4. A number of business rules applies to the EMP and DEPT tables. A partial package is provided in file *p7_4.sql* to which you should add any necessary procedures/functions that are to be called from triggers you may create for the following elements.

- a. Decide how to implement each rule: by means of declarative constraints or using triggers.

Which constraints or triggers are needed and are there any problems to be expected?

- b. Implement the business rules guarded by triggers.

Business Rules

1. Sales persons should always receive commission. Employees who are not sales persons should never receive a commission.
2. The EMP table should contain exactly one PRESIDENT. Test your answer.
3. An employee should never be manager of more than five employees. Test your answer.
4. Salaries may only be increased, never decreased. Test your answer.
5. If a department moves to another location, each employee of that department automatically receives a salary raise of 2%.

8

Managing Subprograms

Objectives

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

- **Describe system privilege requirements**
- **Describe object privilege requirements**
- **Track procedural dependencies**
- **Predict the effect of changing a database object upon stored procedures and functions**
- **Manage procedural dependencies**
- **Debug subprograms**

8-2

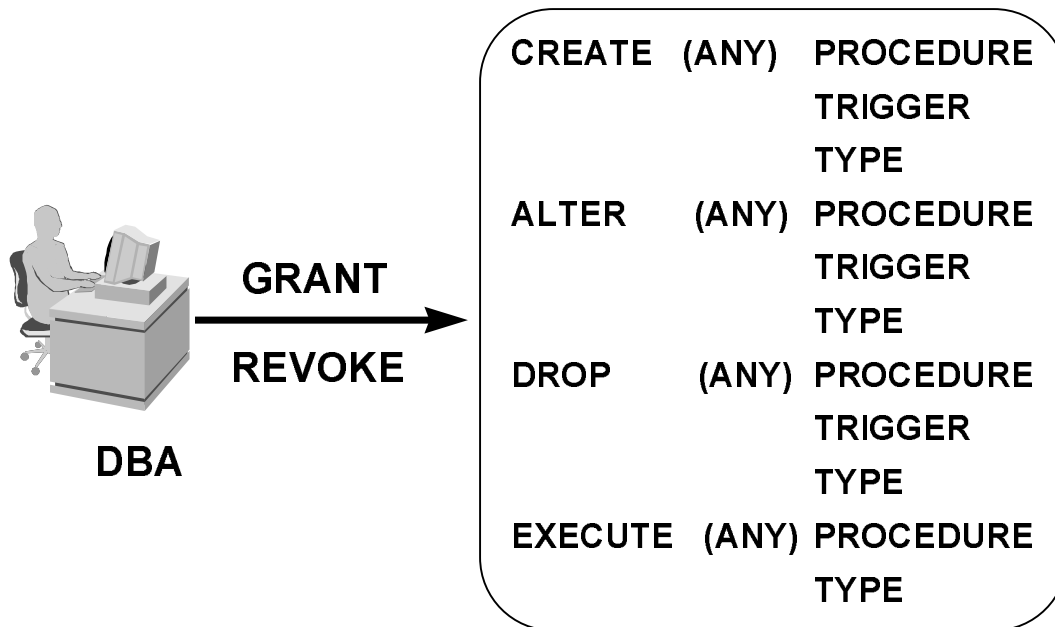
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Lesson Aim

This lesson introduces you to object dependencies, and system and object privilege requirements. You also learn how to debug subprograms.

System Privileges Requirements



8-3

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

In order to create, alter, or drop a PL/SQL subprogram, you must be granted the appropriate system privilege. Unless the ANY keyword is used, the privilege is for your own schema.

You need the EXECUTE privilege to invoke the PL/SQL subprogram if you are not the owner. The object privileges to the schema objects referenced in the PL/SQL subprogram must be granted explicitly (not through a role). The PL/SQL subprogram executes under the security domain of the owner.

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

Example

DIRECT ACCESS

```
SQL> GRANT select, insert
      2  ON      emp
      3  TO      scott;
Grant Succeeded.
```

INDIRECT ACCESS

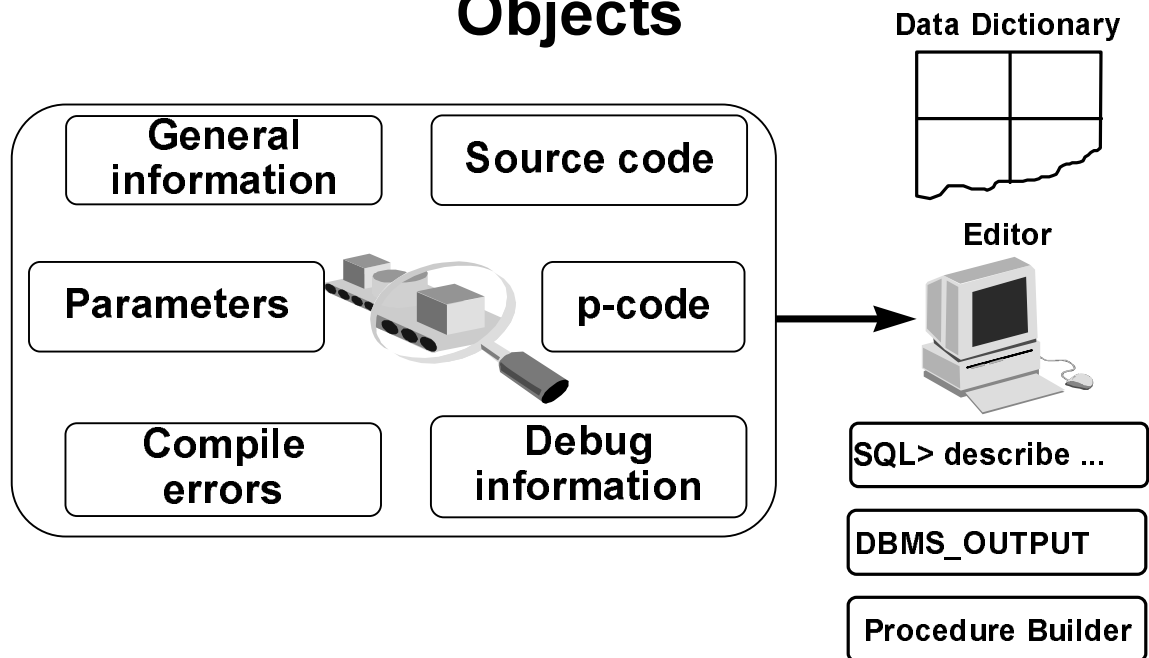
```
SQL> GRANT execute
      2  ON      hire_emp
      3  TO      green;
Grant Succeeded.
```

Provide Indirect Access to Data

Suppose the EMP table is located within the personnel schema, and there is a developer named Scott and an end user named Green. Ensure that Green may only access the EMP table by way of the HIRE_EMP procedure created by Scott, which queries and inserts employee records.

From the personnel schema, provide object privileges on the EMP table for the procedure.

Managing Stored PL/SQL Objects



8-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Stored Information	Description	Access Method
General	Object information	USER_OBJECTS data dictionary view
Source code	Text of the procedure	USER_SOURCE and USER_TRIGGERS data dictionary views SQL*Plus: Editor Procedure Builder: Stored Program Unit Editor
Parameters	Mode: IN/OUT/IN OUT, datatype	DESCRIBE—SQL*Plus
p-code	Compiled object code	Not accessible
Compile errors	PL/SQL syntax errors	USER_ERRORS data dictionary view SHOW ERRORS: SQL*Plus Procedure Builder: Compiler
Runtime debug information	User-specified debug variables and messages	DBMS_OUTPUT Procedure Builder: Interpreter

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
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

8-6

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



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

Also, you should examine the ALL_OBJECTS and DBA_OBJECTS views, each of which contains the additional column OWNER, for the owner of the object.

List All Procedures and Functions

```
SQL>SELECT    object_name, object_type
2 FROM        user_objects
3 WHERE        object_type in ('PROCEDURE', 'FUNCTION')
4 ORDER BY    object_name
5 /
```

OBJECT_NAME	OBJECT_TYPE
-----	-----
ADD_DEPT	PROCEDURE
ADD_ONE	PROCEDURE
FIRE_EMP	PROCEDURE
GET_SAL	FUNCTION
HIRE_EMP	PROCEDURE
LOG_EXECUTION	PROCEDURE
PROCESS_EMP	PROCEDURE
QUERY_EMP	PROCEDURE
REMOVE_AND_PROCESS_EMP	PROCEDURE
REMOVE_DEPT	PROCEDURE

Example

Display the names of all the procedures and functions you have created.

Note: The output has been formatted.

USER_SOURCE

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

8-8

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



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 column OWNER, for the owner of the object.

If you have lost the source file, use SQL*Plus to regenerate it from USER_SOURCE, or use the Procedure Builder Stored Program Unit Editor.

List the Code of Procedures and Functions

```
SQL> SELECT      text
      2 FROM        user_objects
      3 WHERE       name = 'QUERY_EMP'
      4 ORDER BY   line;
```

```
TEXT
-----
PROCEDURE QUERY_EMP
(v_empno      IN      emp.empno%TYPE,
v_name_job    OUT     VARCHAR2,
v_salary      OUT     emp.sal%TYPE,
v_commission  OUT     emp.comm%TYPE)
IS
BEGIN
SELECT ename ||', ' ||job, sal, comm
INTO    v_name_job, v_salary, v_commission
FROM EMP
WHERE empno = v_empno;
END query_emp;
```

8-9

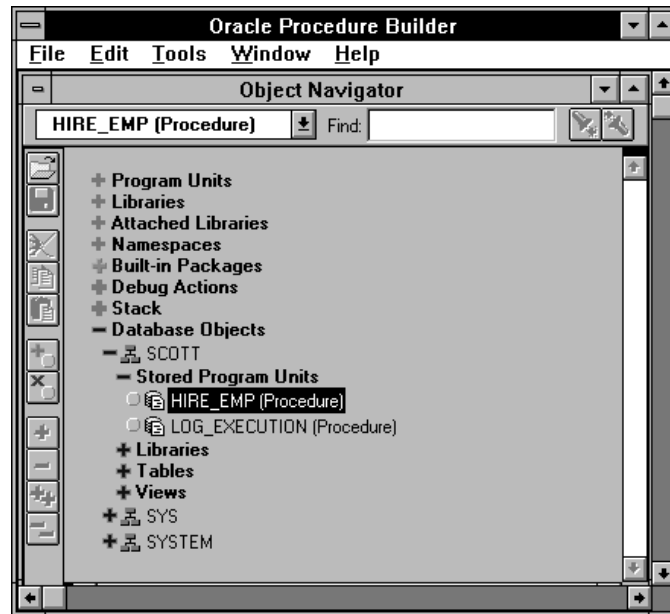
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

Display the complete text for the QUERY_EMP procedure.

List Code of Stored Procedures in Procedure Builder



8-10

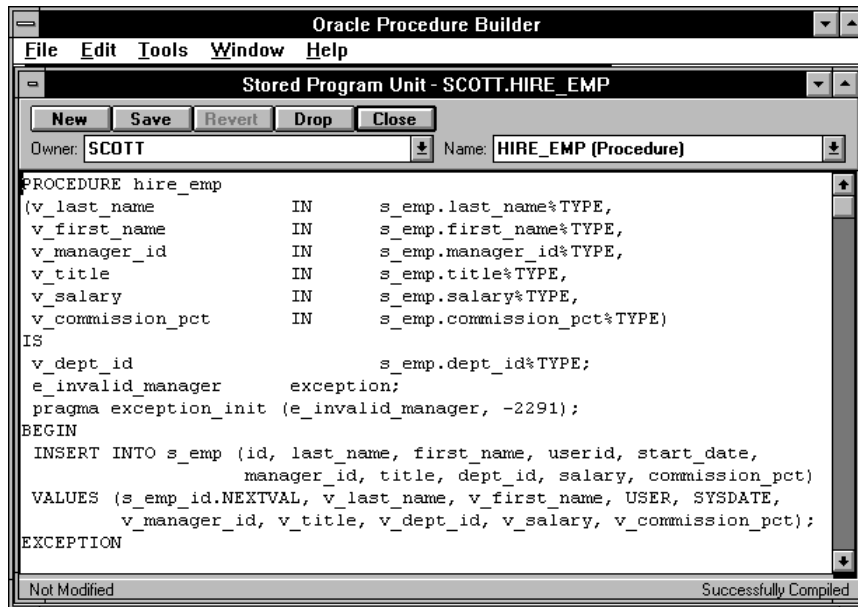
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

To obtain the text of a stored procedure using Procedure Builder:

1. Select File—>Connect and enter your username, password, and database.
2. Click the Expand button and select Database Objects.
3. Click the Expand button and select the schema of the procedure owner.
4. Click the Expand button and select Stored Program Units.
5. Double-click the icon of the stored procedure, and the stored program unit editor appears on the screen.
6. Click Edit—>Export and enter the name of your file in the Open dialog box.
7. Click OK and you have a file containing your stored procedure text (*.pls* extension).

List Code of Stored Procedures in Procedure Builder



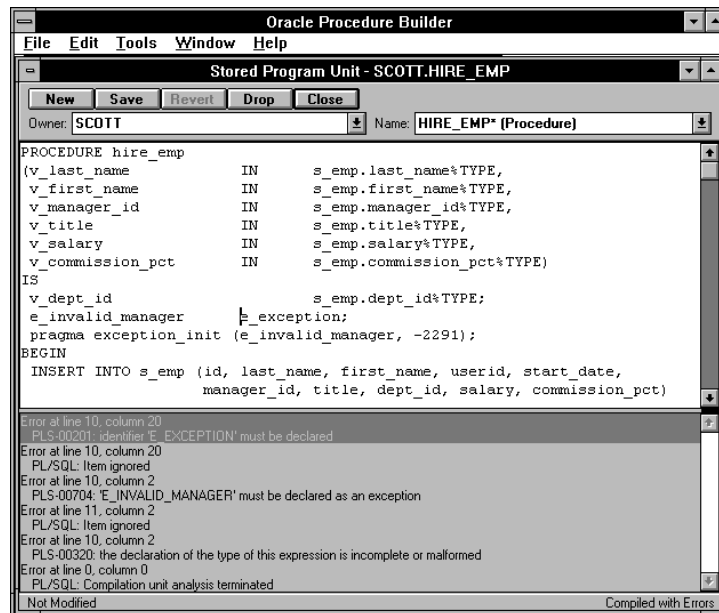
8-11

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

The example above shows the PL/SQL Program Unit Editor with the code for the HIRE_EMP procedure.

Detecting Compile Errors Using the Stored Program Unit Editor



8-12

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

When you click Save in the Stored Program Unit Editor, the procedure compiles. If compilation errors are detected, they are displayed in the compilation text pane in the bottom of the window, and the cursor moves to the location of the first error.

If no compilation errors are detected, "Successfully Compiled" appears in the compilation text pane.

USER_ERRORS

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
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

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

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

Detecting Compilation Errors: Example

```
SQL>CREATE OR REPLACE PROCEDURE log_execution
  2 IS
  3 BEGIN
  4 INPUT INTO LOG_TABLE (user_id, log_date)  -- wrong
  5 VALUES (user, sysdate);
  6 END;
  7 /
```

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

List Compilation Errors Using USER_ERRORS

```
SQL> SELECT   line||'/'||position POS,text
2 FROM       user_errors
3 WHERE      name= 'LOG_EXECUTION'
4 ORDER BY   line;
```

POS	TEXT
3/7	PLS-00103: Encountered the symbol "INT0" when expecting one of the following: := . (@ % ; Replacing "INT0" with `:=`.
4/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (* % & = - + ; < / > in mod not rem an exponent (**) <> or != or ~= >= <= <> and or like between etc. Replacing "VALUES" with "(".
4/23	PLS-100103: Encountered the symbol ";" when expecting one of the following:) , * & = - + < / > in mod not rem => .. an exponent (**) < or != ~= >+ <+ <> and or like between etc.) was inserted before ";" to continue.

List Compilation Errors Using SHOW ERRORS

```
SQL>SHOW ERRORS PROCEDURE log_execution
Errors for PROCEDURE LOG_EXECUTION:
```

LINE/COL	ERROR
3/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . (@ % ; Replacing "INTO" with ":=".
4/1	PLS-00203: Encountered the symbol "VALUES" when expecting one of the following: . (* % & = - + ; < / > in mod not rem an exponent (**) <> or != or ~= >= <= <> and or like between etc. Replacing "VALUES" with "(".
4/23	PLS-100103: Encountered the symbol ";" when expecting one of the following:) , * & = - + < / > in mod not rem => .. an exponent (**) <> or != ~= >= <= <> and or like between etc.) was inserted before ";" to continue.



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

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

8-17

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®



If you have lost the source file, use SQL*Plus to regenerate it from USER_TRIGGERS or use the Procedure Builder Stored Program Unit Editor.

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

Describing PL/SQL Objects in SQL*Plus

```
SQL>DESCRIBE HIRE_EMP
PROCEDURE HIRE_EMP
Argument Name Type           In/Out Default?
-----
V_ENAME   VARCHAR2(25)           IN
V_MANAGER_ID NUMBER(7)       IN
V_TITLE   VARCHAR2(25)           IN
V_SALARY  NUMBER(11,2)           IN
V_COMMISSION NUMBER(4,2)         IN
```

```
SQL>DESCRIBE ADD_DEPT
PROCEDURE ADD_DEPT
Argument Name Type           In/Out Default?
-----
V_NAME     VARCHAR2(25)           IN      DEFAULT
V_REGION_ID NUMBER(7)       IN      DEFAULT
```

Describing Procedures and Functions

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

Examples

Display the argument list for the HIRE_EMP procedure, and display the argument list for the ADD_DEPT procedure, which has defaults.

The DEFAULT column only says there is a default value, not the actual value itself.

Describing PL/SQL Objects in SQL*Plus

```
SQL>DESCRIBE GET_SAL
FUNCTION GET_SAL RETURNS NUMBER
Argument Name Type      In/Out      Default?
-----
V_ID      NUMBER(7)    IN
```

Example

Display the parameter list for the GET_SAL function.

Understanding Dependencies

Dependent Objects

View
Procedure
Function
Package Specification
Package Body
Database Trigger



Referenced Objects

Table
View
Sequence
Synonym
Procedure
Function
Package Specification

8-20

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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.

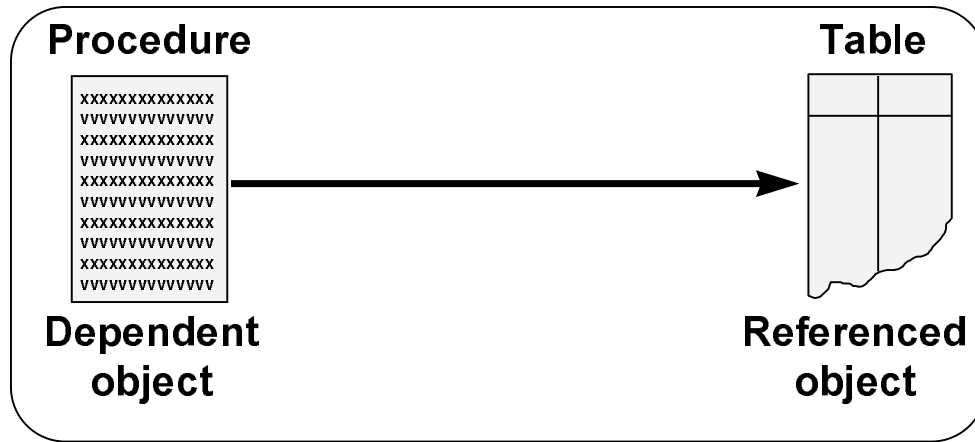
Dependency Issues

If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. In the scenario above, 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) which is recorded in the Data dictionary.

Status	Significance
VALID	The object has been compiled and can be immediately used when referenced.
INVALID	The object must be compiled before it can be used.

Direct Dependency



8-21

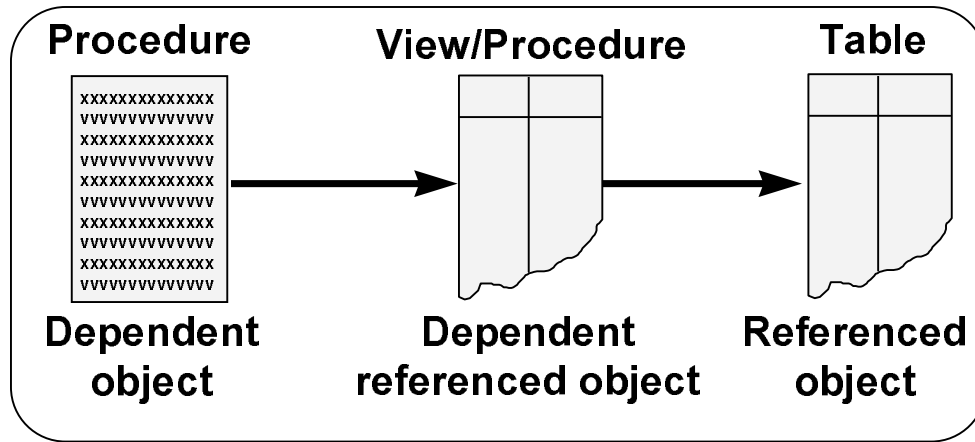
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

A procedure or a function can directly reference the following objects:

- Table
- View
- Sequence
- Procedure
- Function

Indirect Dependency



8-22

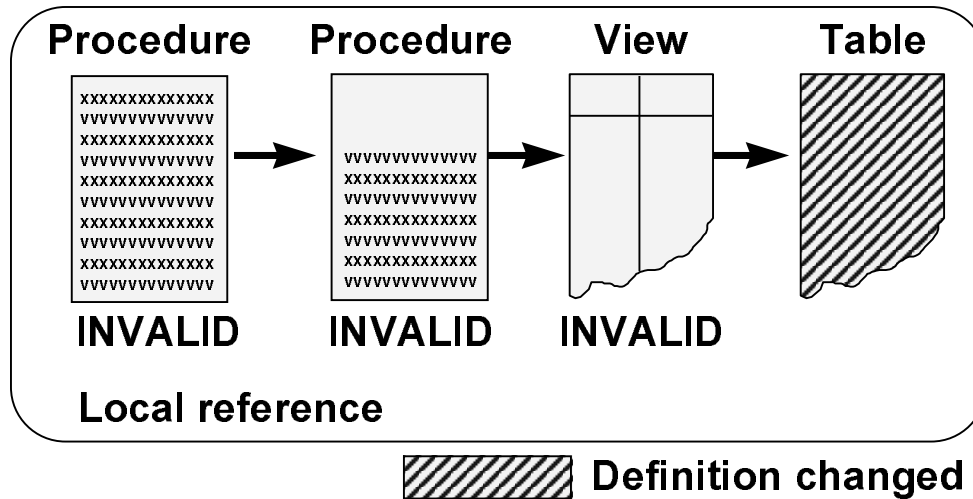
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

A procedure or function can indirectly reference the following objects through an intermediate view, procedure, or function:

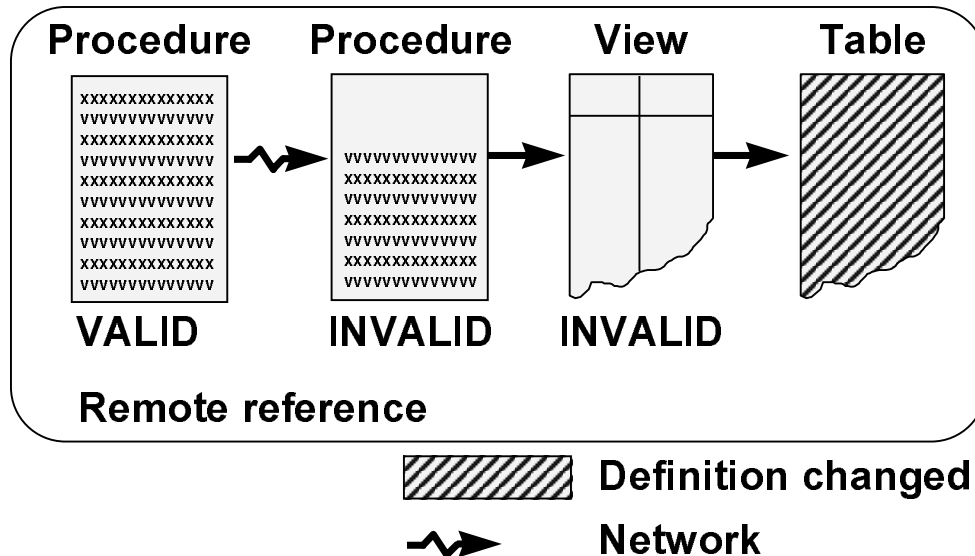
- Table
- View
- Sequence
- Procedure
- Function

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.

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 the first time around.

A Scenario of Local Dependencies

ADD_EMP procedure

```
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
```

NEW_EMP view

EMPNO	ENAME
7839	KING
7698	BLAKE
7782	CLARK
7566	JONES

QUERY_EMP procedure

```
xxxxxxxxxxxxxxxxxxxxx
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvvvvv
```

EMP table

EMPNO	ENAME	HIREDATE	JOB
7839	KING	17-NOV-81	PRESIDENT
7698	BLAKE	01-MAY-81	MANAGER
7782	CLARK	09-JUN-81	MANAGER
7566	JONES	02-APR-81	MANAGER

8-25

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

The QUERY_EMP procedure directly references the EMP table. The ADD_EMP procedure updates the EMP table indirectly by way of the NEW_EMP view. For each of the following cases, will the ADD_EMP procedure be invalidated, and will it successfully recompile?

- The internal logic of the QUERY_EMP procedure is modified.
- A new column is added to the EMP table.
- The NEW_EMP view is dropped.

Displaying Direct Dependencies using USER_DEPENDENCIES

```
SQL> SELECT name, type, referenced_name, referenced_type
2  FROM    user_dependencies
3  WHERE   referenced_name IN ( 'EMP' , 'NEW_EMP' );
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
QUERY_EMP	PROCEDURE	EMP	TABLE
ADD_EMP	PROCEDURE	NEW_EMP	VIEW
NEW_EMP	VIEW	EMP	TABLE

Display Direct Dependencies 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, for the owner of the object.

Column	Column Description
NAME	Name of dependent object
TYPE	Type of dependent object (PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY)
REFERENCED_OWNER	Schema of referenced object
REFERENCED_NAME	Name of referenced object
REFERENCED_TYPE	Type of referenced object
REFERENCED_LINK_NAME	Database link used to access referenced object

Displaying Direct and Indirect Dependencies

Populate the DEPTREE and IDEPTREE views by invoking the DEPTREE_FILL procedure.

```
SQL> @UTLDTREE -- consult your DBA
SQL> EXECUTE deptree_fill ('TABLE', 'SCOTT', 'EMP')
PL/SQL procedure successfully completed.
SQL> SELECT * FROM deptree;
SQL> SELECT * FROM ideptree;
```

Direct and Indirect Dependencies Using Views Provided by Oracle

Display indirect dependencies from additional user views called DEPTREE and IDEPTREE, provided by Oracle.

1. Make sure the UTLDTREE.SQL script has been executed (ask your DBA).
2. Populate the DEPTREE_TEMPTAB table with information for a particular referenced object, by invoking the DEPTREE_FILL procedure. There are three arguments for this procedure:

where:	<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.

Example

Populate the DEPTREE and IDEPTREE views by invoking the DEPTREE_FILL procedure.

1. Display a tabular representation of all dependent objects, direct and indirect, by querying the DEPTREE view.
2. Display an indented tree representation of the same information by querying the IDEPTREE view, which contains the single column DEPENDENCIES.

DEPTREE View

```
SQL> SELECT nested_level, type, name  
2 FROM deptree  
3 ORDER BY seq#;
```

NESTED_LEVEL	TYPE	NAME
0	TABLE	EMP
1	VIEW	NEW_EMP
2	PROCEDURE	ADD_EMP
1	PROCEDURE	QUERY_EMP

Example

Display dependencies using DEPTREE.

IDEPTREE View

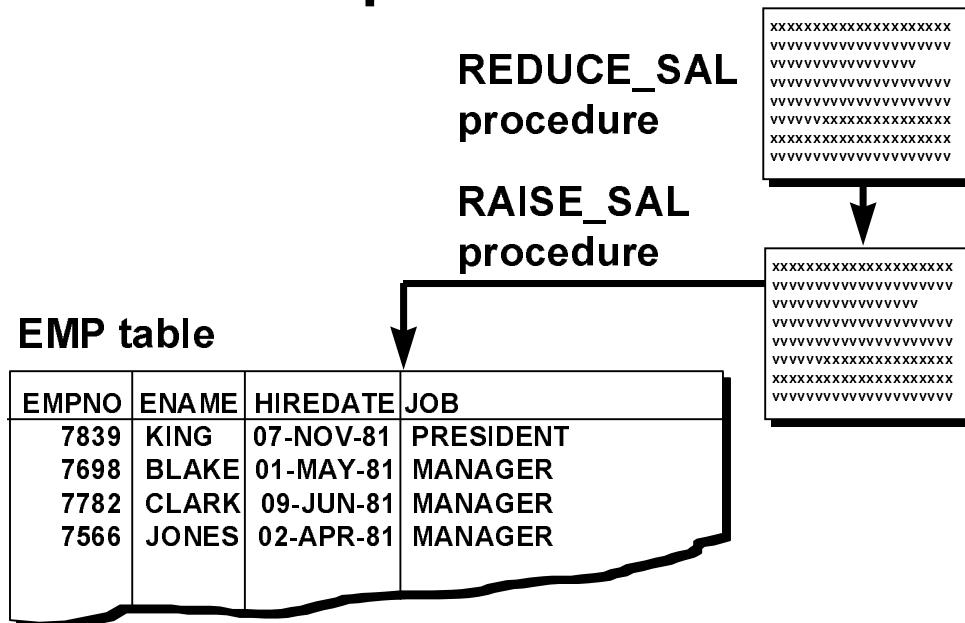
```
SQL> SELECT * FROM ideptree;

DEPENDENCIES
-----
TABLE  schema_name.EMP
      VIEW  schema_name.NEW_EMP
          PROCEDURE  schema_name.ADD_EMP
          PROCEDURE  schema_name.QUERY_EMP
```

Example

Display dependencies using IDEPTREE.

Another Scenario of Local Dependencies



8-30

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

Predict the effect that a change in the definition of a procedure has upon the recompilation of a dependent procedure.

Suppose that the **RAISE_SAL** procedure updates the **EMP** table directly, and that the **REDUCE_SAL** procedure updates the **EMP** table indirectly by way of **RAISE_SAL**. In each of the following cases, will the **REDUCE_SAL** procedure successfully recompile?

- The internal logic of the **RAISE_SAL** procedure is modified.
- One of the formal parameters to the **RAISE_SAL** procedure is eliminated.

A Scenario of Local Naming Dependencies

**QUERY_EMP
procedure**

```

XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVV
VVVVVVVVVVVVVVVVVVV
VVVVVVVVVVVVVVVVVVV
VVVVVVVVVVVVVVVVVVV
VVVVVVXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXX
VVVVVVVVVVVVVVVVVVV

```



EMP public synonym

EMPNO	ENAME	HIREDATE	JOB
7839	KING	17-NOV-81	PRESIDENT
7698	BLAKE	01-MAY-81	MANAGER
7782	CLARK	09-JUN-81	MANAGER
7566	JONES	02-APR-81	MANAGER

**EMP
table**

EMPNO	ENAME	HIREDATE	JOB
7839	KING	17-NOV-81	PRESIDENT
7698	BLAKE	01-MAY-81	MANAGER
7782	CLARK	09-JUN-81	MANAGER
7566	JONES	02-APR-81	MANAGER

8-31

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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.

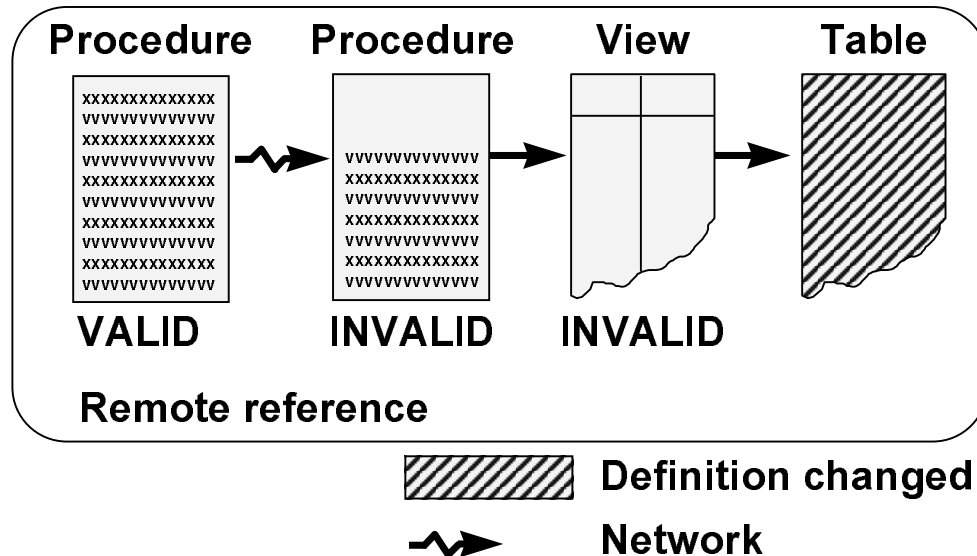
Example

Predict the effect that the name of a new object has upon a dependent procedure.

Suppose that originally, your QUERY_EMP procedure referenced a public synonym called EMP. However, you have just created a new table called EMP within your own schema. Will this change invalidate the procedure? Which of the two EMP objects will QUERY_EMP reference?

You can track security dependencies within the USER_TAB_PRIVS data dictionary view.

Remote Dependencies



8-32

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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 runtime 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**

OR

- **SIGNATURE checking**

TIMESTAMP Checking

Each PL/SQL program unit carries a timestamp 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 timestamp 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 timestamp and the signature are recorded. The signature of a PL/SQL construct contains information about the following:

- The name of the construct (procedure, function, package)
- The base types of the parameters of the construct
- The modes of the parameters (IN, OUT, IN OUT)
- The number of the parameters

The recorded timestamp in the calling program unit is compared to the current timestamp in the called remote program unit. If the timestamps 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.

Setting the Remote Dependency Mode Parameter

- As an *init.ora* parameter

REMOTE_DEPENDENCIES_MODE = value

- At system level

**ALTER SYSTEM SET
REMOTE_DEPENDENCIES_MODE = value**

- At session level

**ALTER SESSION SET
REMOTE_DEPENDENCIES_MODE = value**

8-34

Copyright © Oracle Corporation, 1998. All rights reserved.

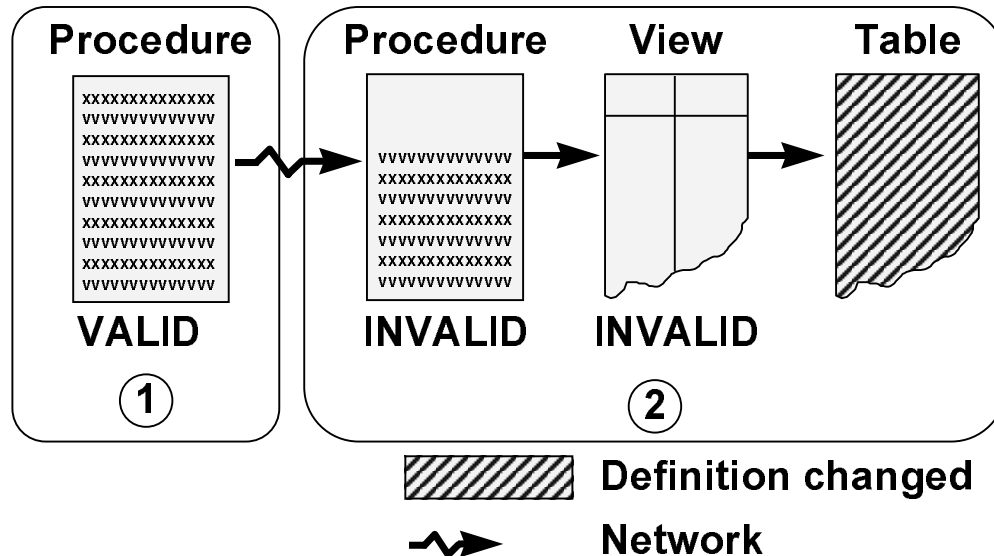
ORACLE®

where: value is “TIMESTAMP” or “SIGNATURE”

Specify the value of the REMOTE_DEPENDENCIES_MODE parameter using one of the three methods above.

Note: The calling site determines the dependency model.

Remote Dependencies and Timestamp Mode: Scenario



8-35

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

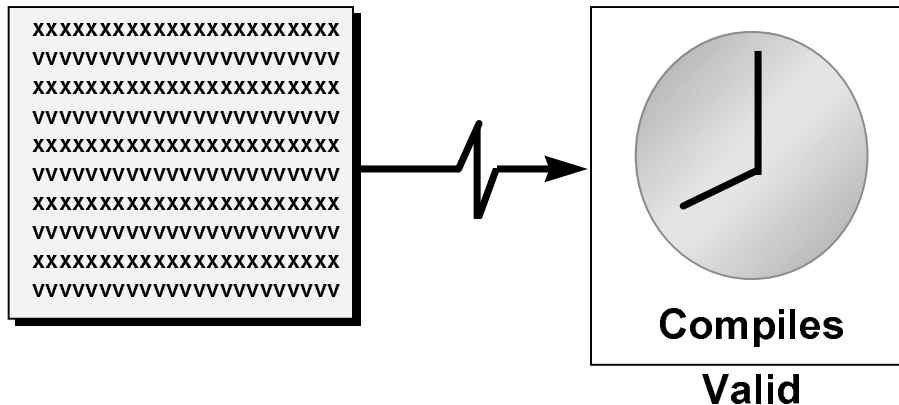
Automatic Recompilation of Local and Remote Objects Using Timestamp Mode

- When remote objects change, we strongly recommend that you recompile local dependent objects manually, rather than relying on the automatic remote dependency mechanism, in order to avoid disrupting production.
- The automatic remote dependency mechanism is different from the automatic local dependency mechanism already discussed. The first time an invalid subprogram is invoked, you will get an execution error; the second time it is invoked, implicit automatic recompilation will take place.

Remote Procedure B Compiled at 8:00

Local procedure A

Remote procedure B



8-36

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Local Procedures Referencing Remote Procedures

A local procedure that references a remote procedure will be invalidated by the Oracle Server if the remote procedure is recompiled after the local procedure is compiled.

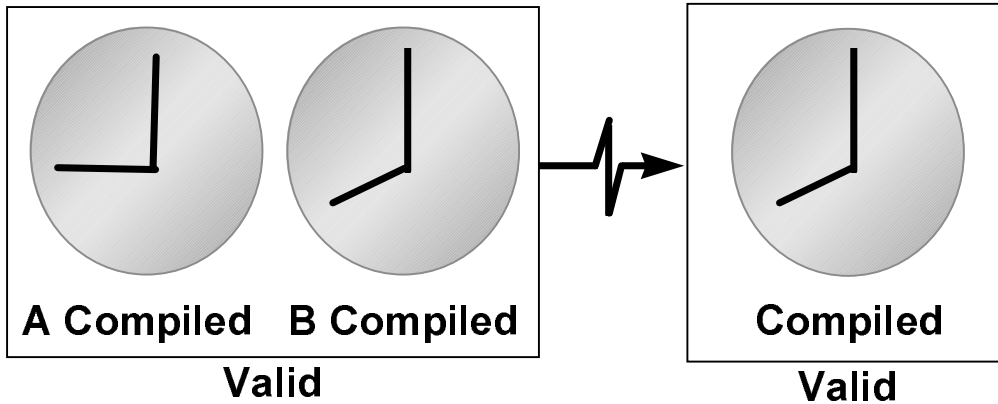
Automatic Remote Dependency Mechanism

- The Oracle Server records the timestamp within the p-code of every procedure at compile time.
- When a procedure compiles, the Oracle Server records the timestamp of that compilation within the p-code of the procedure.
- When a local procedure referencing a remote procedure compiles, the Oracle Server also records the timestamp of the remote procedure into the p-code of the local procedure.

Local Procedure A Compiles at 9:00

Local procedure A

Remote procedure B

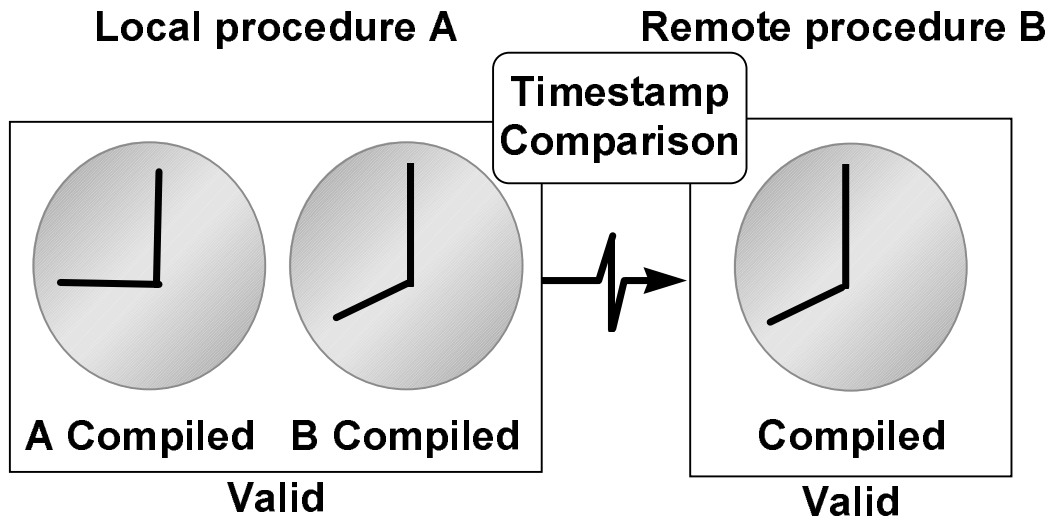


8-37

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Procedure A Is Invoked at 10:00; Procedure B Has Not Recompiled Since 8:00



8-38

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Automatic Remote Dependency

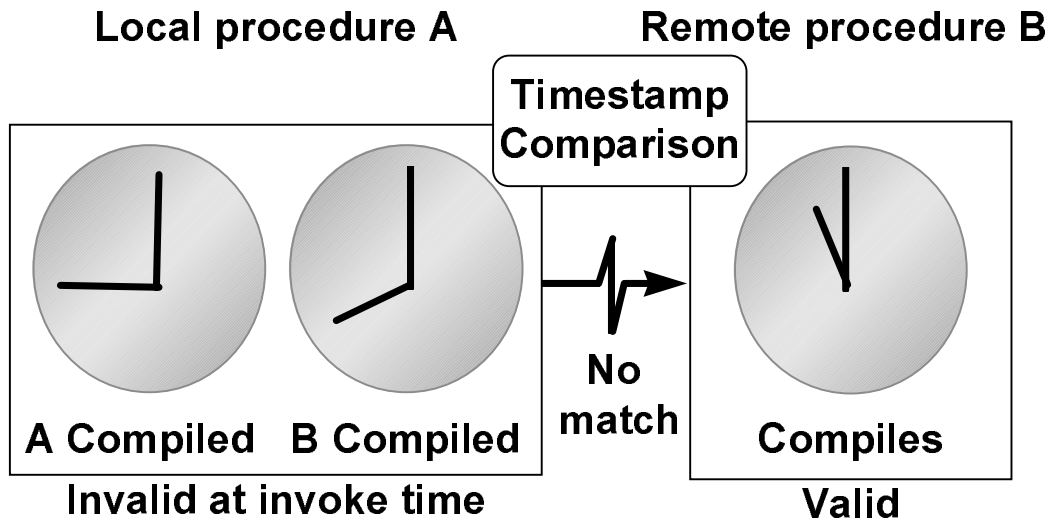
When the local procedure is invoked at runtime, the Oracle Server compares the two timestamps.

- If the timestamps are equal, indicating that the remote procedure has not recompiled, the Oracle Server executes the local procedure.
- If the timestamps 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.

- If the first time a local procedure is invoked, it returns a runtime error indicating that its remote timestamp has changed, you should develop a strategy to reinvoke the local procedure.

Procedure A Is Invoked at 12:00; Procedure B Has Recompiled at 11:00 (Local Time)



Recompiling a PL/SQL Program Unit

- **Handled automatically through implicit runtime recompilation**
- **Handled through explicit recompilation with the ALTER statement**

```
ALTER PROCEDURE [PACKAGE.] procedure_name COMPILE
```

```
ALTER FUNCTION [PACKAGE.] function_name COMPILE
```

```
ALTER PACKAGE [PACKAGE.] package_name COMPILE PACKAGE
```

```
ALTER PACKAGE [PACKAGE.] package_name COMPILE BODY
```

```
ALTER TRIGGER TRIGGER_NAME [ENABLE | DISABLE | COMPILE | [DEBUG] ]
```

8-40

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Recompiling PL/SQL Objects

If the recompilation is successful, the object becomes valid. If not, the Oracle Server returns an error and the object remains invalid.

When you recompile a PL/SQL object, the Oracle Server first recompiles any invalid 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, will also be invalidated.

Packages

The COMPILE PACKAGE option recompiles both the package specification and the body, regardless of whether it is invalid. The COMPILE BODY option only recompiles 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

A trigger is enabled by default. When disabled, the trigger does not fire when the triggering statement is issued.

Recompilation of Procedures

Recompiling dependent procedures and functions will be unsuccessful when:

- **The referenced object is dropped or renamed**
- **The datatype of the referenced column is changed**
- **A referenced view is replaced by a view with different columns**
- **The parameter list of a referenced procedure is modified**

Recompilation of Procedures

Recompiling dependent procedures and functions will be successful if:

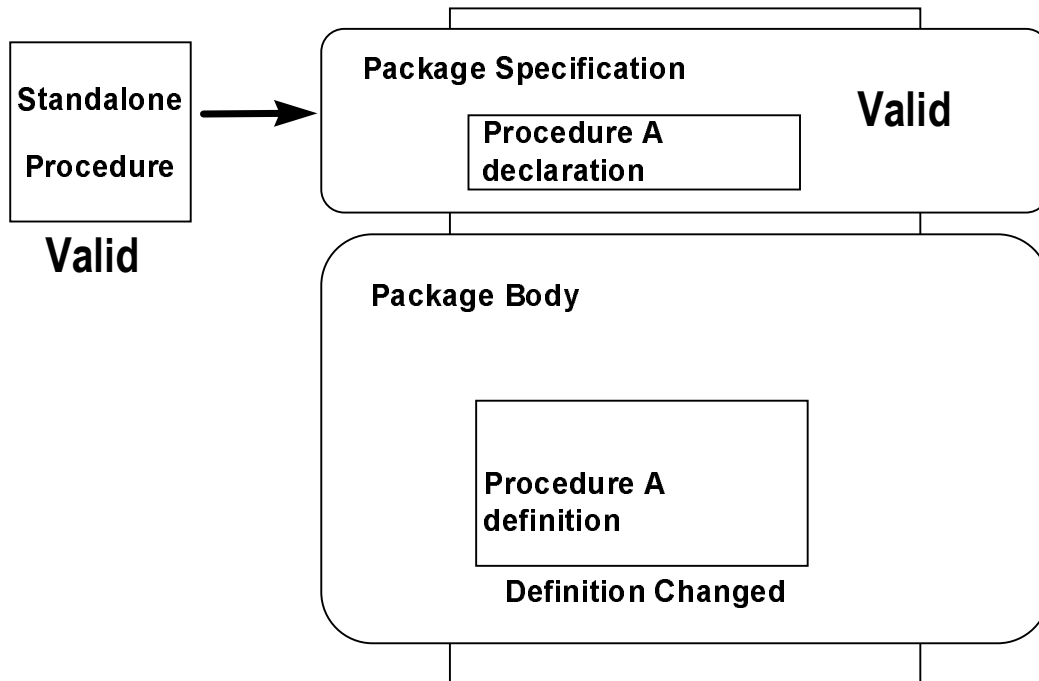
- **The referenced table has new columns**
- **The datatype of referenced columns has not changed**
- **The PL/SQL body of a referenced procedure has been modified and recompiled successfully**

Recompilation of Procedures

Minimize dependency failures by:

- **Declaring records using the %ROWTYPE attribute**
- **Declaring variables with the %TYPE attribute**
- **Querying with the SELECT * notation**
- **Including a column list with INSERT statements**

Packages and Dependencies



8-44

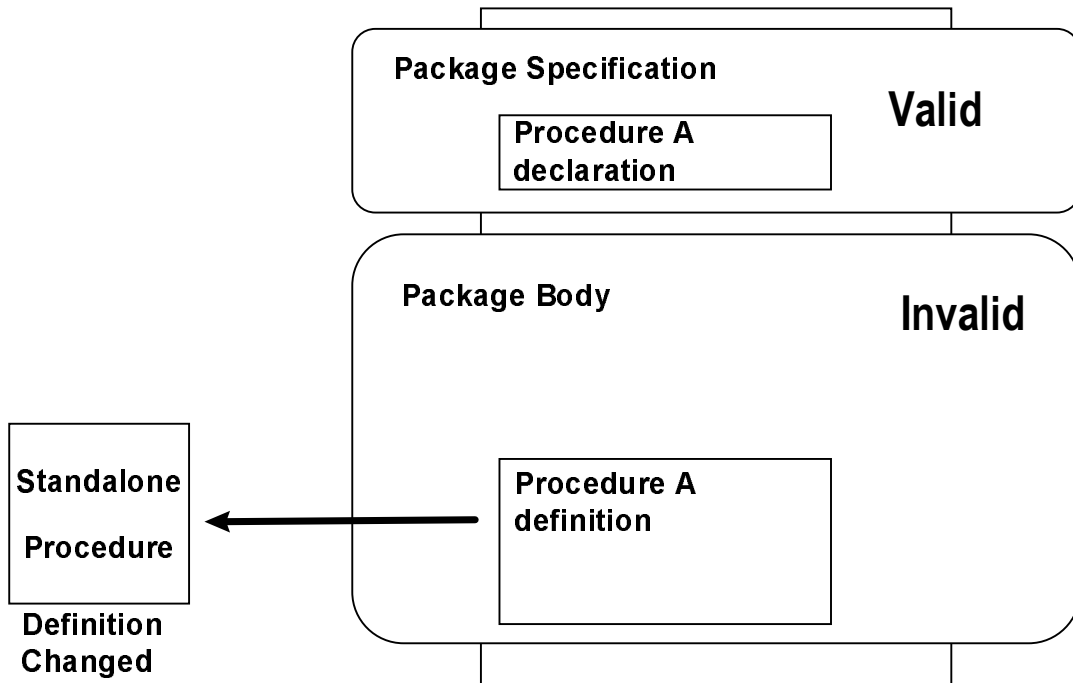
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Greatly simplify dependency management with packages when referencing a package procedure or function from a standalone procedure or function.

- If the package body changes and the package specification does not change, the standalone procedure referencing a package construct remains valid.
- If the package specification changes, the outside procedure referencing a package construct is invalidated as well as the package body.

Packages and Dependencies



8-45

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Expect no improvement in dependency management when referencing a standalone procedure or function from a package: the entire package body depends upon the procedure. Therefore, it is recommended to bring the procedure into the package.

If the standalone procedure referenced within the package changes, the entire package body is invalidated, but the package specification remains valid.

Debugging Using The DBMS_OUTPUT Package

The DBMS_OUTPUT package:

- **Accumulates information into a buffer**
- **Allows retrieval of the information from the buffer**

You can use DBMS_OUTPUT procedures, an Oracle supplied package, to output values and messages from a stored procedure or function. This is done by accumulating information into a buffer and then allowing the retrieval of the information from the buffer.

Benefits

- Allow developers to closely follow the execution of a function or procedure by sending messages and values to the output buffer.
- Qualify every reference to these procedures with the DBMS_OUTPUT prefix.
- Within SQL*Plus or SQL*DBA, use SET SERVEROUTPUT ON or OFF instead of using the ENABLE or DISABLE procedures.

How to Use DBMS_OUTPUT

1. Enable SERVEROUTPUT

```
SQL> SET SERVEROUTPUT ON
```

2. Put text into the buffer

```
SQL> EXECUTE DBMS_OUTPUT.PUT ('testing 1 2 3')  
PL/SQL procedure successfully completed.
```

3. Display from the buffer

```
SQL> EXECUTE DBMS_OUTPUT.NEW_LINE  
testing 1 2 3  
PL/SQL procedure successfully completed.
```

or

```
SQL> EXECUTE DBMS_OUTPUT.PUT_LINE  
      ('testing 1 2 3 testing 4 5 6')  
testing 1 2 3 testing 4 5 6  
PL/SQL procedure successfully completed.
```

8-47

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Use DBMS_OUTPUT procedures to display diagnostic messages and values of variables.

Suggested Diagnostic Information

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

Debugging a Stored Procedure Using DBMS_OUTPUT

```
CREATE OR REPLACE PROCEDURE remove_and_process_emp
(v_id IN s_emp.id%TYPE)
IS
  v_no_of_employees NUMBER; v_dept__id s_emp.dept_id%TYPE;
BEGIN
  ...../* SELECT statement to count the number of employees */ .....
  dbms_output.put_line('no of employees in dept ' || v_dept_id || ' is '
    || v_no_of_employees);
  IF v_no_of_employees = 1 THEN
    dbms_output.put_line('As there is only 1 person left in the dept. ');
    dbms_output.put_line('fire_emp then remove department');
    fire_emp (v_id);
    remove_dept (v_dept_id);
  ELSE
    dbms_output.put_line('As there is more than 1 person left in the dept, ');
    dbms_output.put_line('fire_emp only, will be fired');
    fire_emp (v_id);
  END IF;
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20200, 'Employee does not exist. ');
END remove_and_process_emp;
```

8-48

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example

The example above illustrates how to record diagnostic information about the REMOVE_AND_PROCESS_EMP procedure.

Debug Output

Sample output from the previous procedure

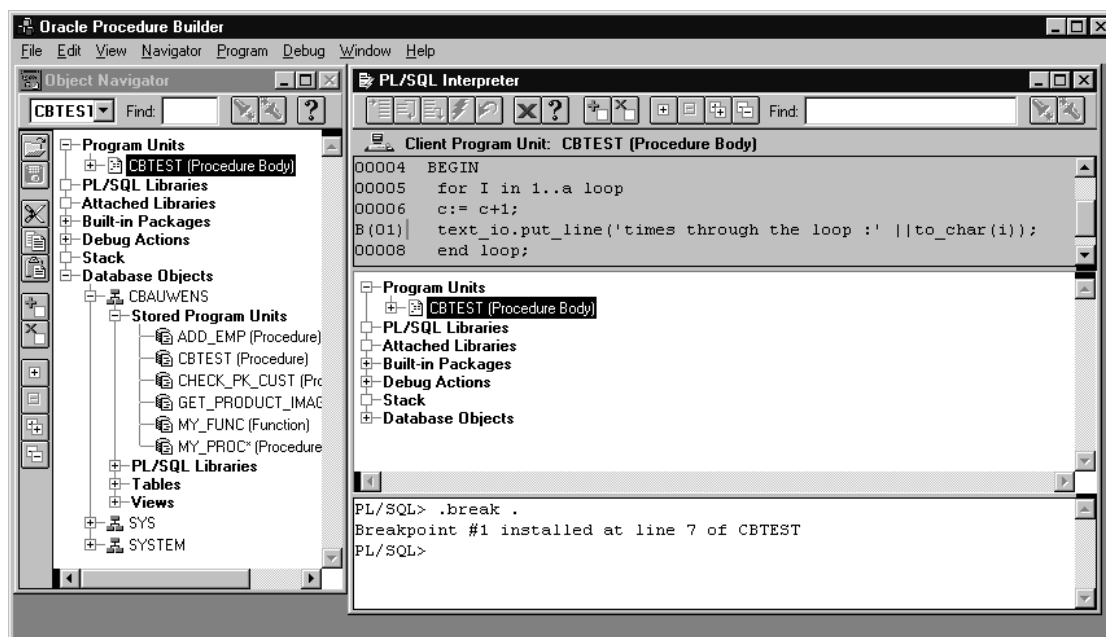
```
SQL> SET SERVEROUTPUT ON
SQL> EXECUTE remove_and_process_emp (22)
no of employees in dept 44 is 2
as there is more than 1 person left in the dept,
fire_emp only, will be fired.

PL/SQL procedure successfully completed.
```

Example

The sample output displays sample diagnostic information returned when executing the REMOVE_AND_PROCESS_EMP procedure, at runtime.

Debugging Subprograms Using Procedure Builder



8-50

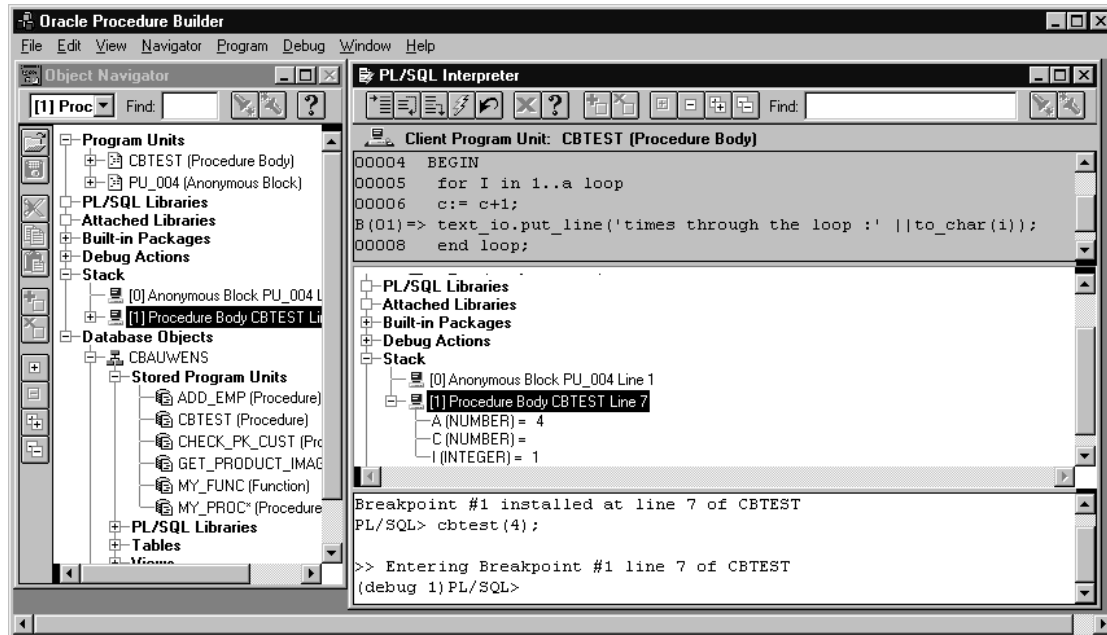
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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, click on Program—>PL/SQL Interpreter in the menu.
2. In the menu, click on View—>Navigator Pane.
3. From the Navigator pane, expand either the Program Units or the Database objects node.
4. Click on the subprogram you want to debug.

Creating Breakpoints



8-51

Copyright © Oracle Corporation, 1998. All rights reserved.

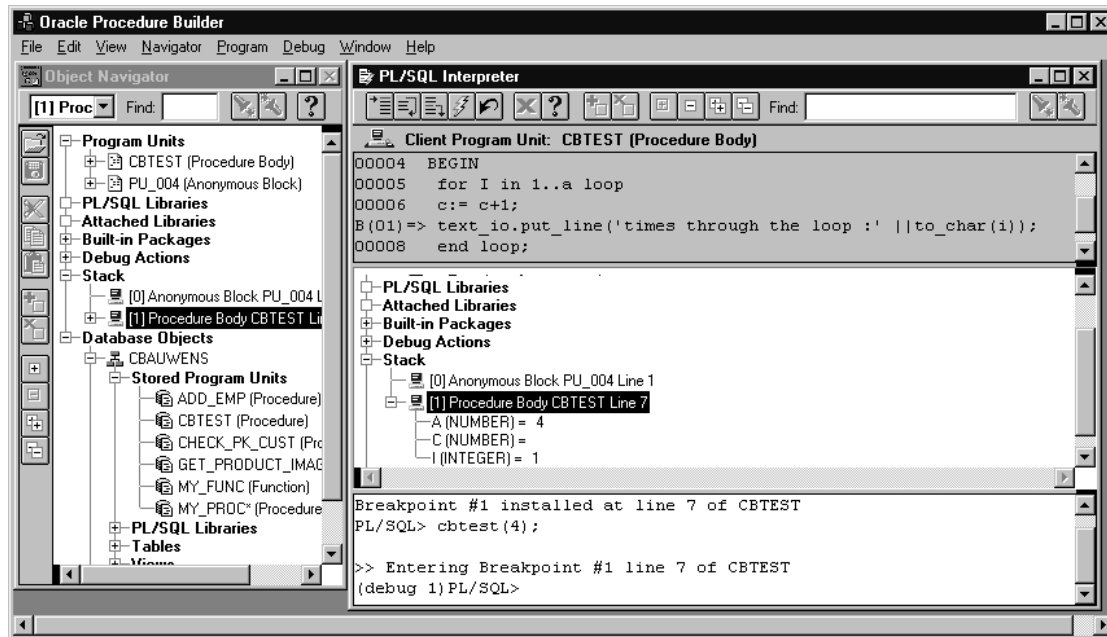
ORACLE®

You insert breakpoints in a program unit to suspend execution at a specific source line. When PL/SQL encounters a breakpoint in a program unit, it suspends execution at the line just before the breakpoint and passes control to the Interpreter.

To create a breakpoint in a program unit, double-click the line to create the breakpoint.

In the example you see that the line number has changed from 00007 to the breakpoint B(01).

Using Debugging Levels



8-52

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

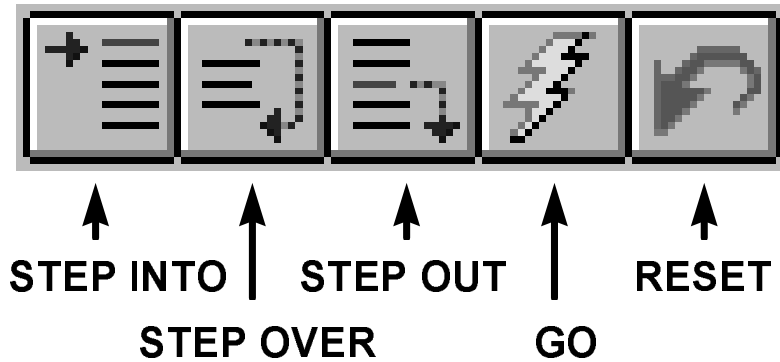
To start the Debugger, execute the program unit you want to debug:

```
PL/SQL> cbtest(4);
```

When a debug action interrupts program execution, the Interpreter takes control and establishes what is known as a debug level. At a debug level, you can enter commands and PL/SQL statements to inspect and modify the state of the interrupted program unit.

In the example above, the stack node has been expanded to look at the value of the variables used in the subprogram.

Controlling Program Unit Execution



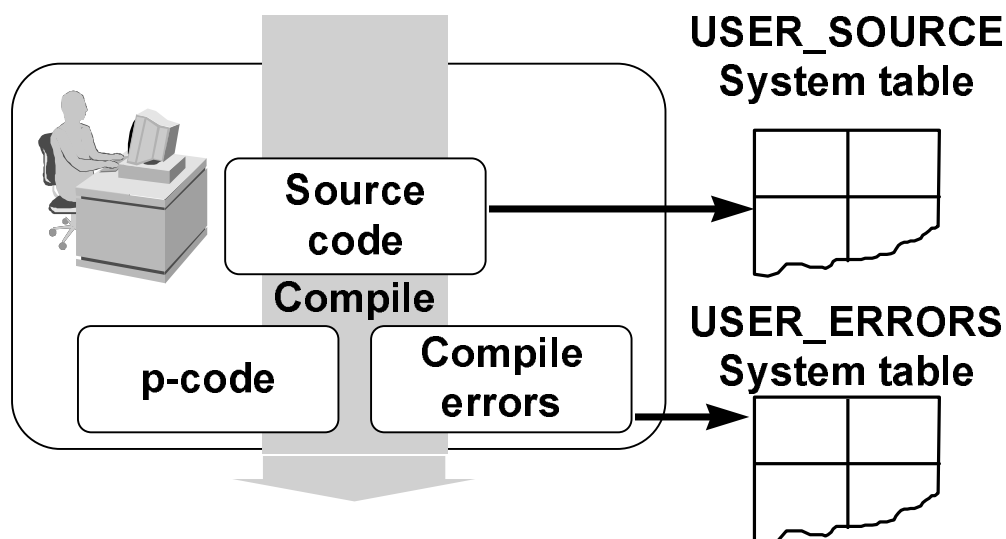
8-53

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

- Use STEP INTO/OVER/OUT to temporarily resume execution of an interrupted program. Control returns to the Interpreter after the specified set of statements is executed.
- Use GO to resume program execution indefinitely, until either the execution terminates or is interrupted again due to a debug action.
- Use RESET to return control to an outer debug level without continuing execution in the current debug level.

Summary



8-54

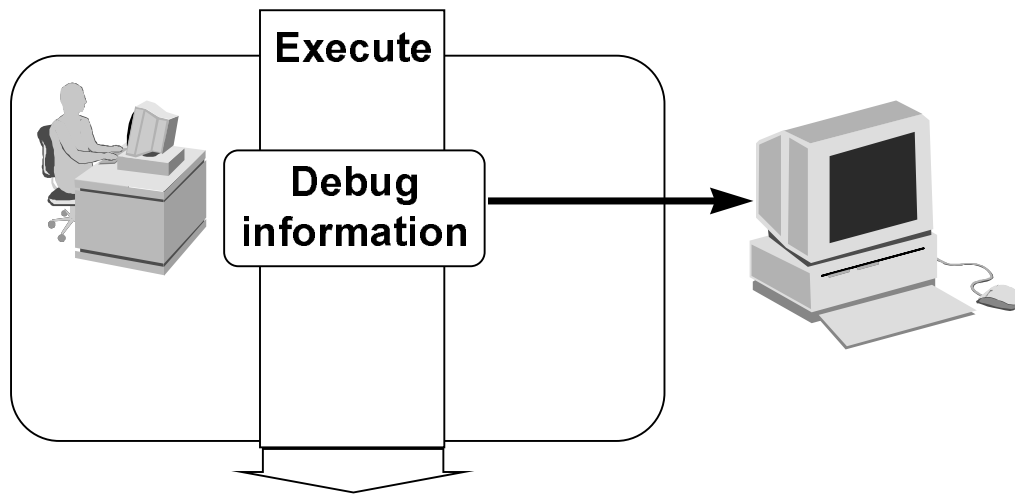
Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Take advantage of various data dictionary views, SQL commands, SQL*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	SQL*Plus command	Provides the declaration of the object
USER_ERRORS	Data dictionary view	To see any compilation errors
SHOW_ERRORS	SQL*Plus command	To see any compilation errors, per procedure or function
DBMS_OUTPUT	Oracle-supplied package	User-specified debugging, giving variable values and messages
GRANT	SQL command	Providing the security privileges for the owner who creates the procedure and the user who runs it, enabling them to perform their respective operations

Summary



8-55

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Suggested SQL*Plus Scripts

- 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.

Summary

Avoid disrupting production by:

- **Keeping track of dependent procedures**
- **Recompiling procedures manually as soon as possible after the definition of a database object changes**

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 implement a strategy of reinvoking it a second time.
Procedure depends upon a remote object other than a procedure.	No.

Practice Overview

- **Get code back into a spool file from USER_SOURCE.**
- **Track object dependencies.**

Practice 8

1. Answer the following questions.
 - a. Can a table or a synonym be invalid?
 - b. Assume the following scenario:
 - The standalone 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 specification is not altered in the package specification.
 - Is the standalone procedure MY_PROC invalidated?
2. Suppose you have lost the code for the NEW_EMP procedure and the VALID_DEPTNO function you created in Lesson 4. Create a SQL*Plus spool file to query the appropriate Data Dictionary view to regenerate the code.
3. Execute the *utldtree.sql* script. Print a tree structure showing all dependencies involving your NEW_EMP procedure and your VALID_DEPTNO function. Query the ideptree view to see your results.

If you have time:

4. Dynamically validate invalid objects.
 - a. Make a copy of your EMP table, called EMP_COP.
 - b. Alter your EMP table and add the column TOTSAL(NUMBER(9,2)).
 - c. Create a script file to print the name, type, and status of dependent objects.
 - d. Create a procedure called COMPILE_OBJ that recompiles all invalid objects in your schema.

Make use of the ALTER_COMPILE procedure in the DBMS_DDL package.

- e. Run the previous script file again and check the status column value.

9

Working with Object Types

Objectives

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

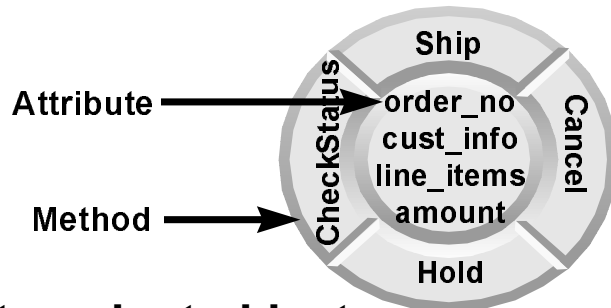
- **Describe object types**
- **Create transient objects**
- **Manipulate objects in object tables**

Lesson Aim

PL/SQL8 provides support for the object features in the Oracle Server. Objects require special functionality from the database language. This lesson discusses the new elements of PL/SQL8 for object support.

An Object Type

- Is a user-defined composite datatype
- Encapsulates a data structure along with the methods needed to manipulate it



- Is a transient object

Object Types

An object, such as a car, an order, a person, has specific attributes and behaviors. An object type allows you to maintain this perspective. It is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures to manipulate the data.

- The variables that make up the data structure are called *attributes*.
- The functions and procedures that characterize the behavior are called *methods*.

The data structure formed by a set of attributes is public (visible to client programs). You should not manipulate it directly. Instead, use the methods provided.

Transient Objects

For persistent objects, the associated object instances are stored in the database. Transient objects differ from persistent objects in the way that they are declared, initialized, used, and deleted *programmatically*. When the program unit is completed, the transient object no longer exists, but the type exists in the database.

Transient objects are defined as an instance of a persistent object type, therefore transient object attributes cannot be PL/SQL datatypes. Most Oracle Server datatypes however can be used.

The Structure of an Object Type

**Public
interface**

Specification

Attribute declarations

Method specifications

**Private
implementation**

Body

Method bodies

9-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Object Type Structure

Like an Oracle database package, an object type has two parts:

- The object specification is the interface to your applications. It declares the data structure (set of attributes) along with the operations (methods) to access, use, or manipulate the data. All declarations are public (visible outside the object type).
- The object body implements the specification. It fully defines the methods. You can debug, enhance, or replace the body without changing the specification and without affecting the client program.



Restrictions

- In the object type specification, all attributes must be declared before any methods.
- If the object type specification declares only attributes, the object type body is unnecessary.
- You cannot declare attributes in the object type body.
- The object type body can contain private declarations defining methods. These can only be used within the (public) methods defined with the object.

Creating an Object Type Specification

Syntax

```
CREATE TYPE type_name AS OBJECT
[ (attribute1 datatype,
  attribute2 datatype,
  . . .) ]
[ (MEMBER procedure1 | function1 spec,
  procedure2 | function2 spec,
  . . .) ]
```

Currently you cannot define object types in a PL/SQL block, subprogram, or package. However you can define them in SQL*Plus or Enterprise Manager using the CREATE TYPE statement.

Attributes

When you define attributes, the following rules apply:

- An attribute is declared with a name and datatype.
- The datatype can be almost any Oracle Server datatype.
- You cannot initialize an attribute in its declaration using the assignment operator or the DEFAULT clause.
- You cannot impose the NOT NULL constraint on an attribute.

The collection of attributes form the data structure. The kind of data structure depends on the real-world object being modeled. To represent a college student you need several VARCHAR2 variables to hold name, address and so on, and a VARRAY variable to hold courses taken, credits and grades.

Simple object types have attributes with datatypes that are Oracle Server datatypes only. For complex object types, however, the datatype of an attribute can be another object type. This allows you to build a complex object from simpler object types.

Creating an Object Type Body

Syntax

```
CREATE TYPE BODY type_name AS  
[MEMBER procedure1 | function1 body,  
  procedure2 | function2 body,  
  . . .]
```

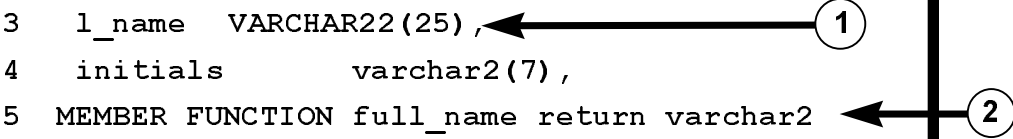
Methods

A method is a subprogram declared in the object type specification using the keyword **MEMBER**. Methods have two parts: a specification and a body.

- The specification consists of a method name, an optional parameter list and, for functions, a return type.
- The body is the actual PL/SQL code that executes to perform a specific operation.
- For each method specification in an object type specification, there must be a corresponding method body in the object type body.

Example: Creating an Object Type Specification

```
SQL> CREATE TYPE name_typ AS OBJECT(  
  2   f_name  VARCHAR2(25) ,  
  3   l_name  VARCHAR2(25) ,  
  4   initials    varchar2(7) ,  
  5   MEMBER FUNCTION full_name return varchar2  
  6   PRAGMA RESTRICT_REFERENCES( full_name,  
  7   WNDS, WNPS, RNPS );
```




1. Attributes
2. Method specification

Like package PL/SQL functions, member functions can be called from SQL statements, but the function must be free of side effects. In the example above, the member function FULL_NAME uses the PRAGMA RESTRICT_REFERENCES to specify the purity level.

Example: Creating an Object Type Body

```
SQL> CREATE TYPE BODY name_typ AS
  2  MEMBER FUNCTION full_name RETURN VARCHAR2
  3  IS
  4  BEGIN
  5      RETURN (l_name || ' ' || f_name );
  6  END full_name;
  7  END;
```

A diagram consisting of a horizontal arrow pointing left from a circle containing the number '1' to the 'BEGIN' statement on line 4 of the SQL code.

1. Method body

Example: Creating a Complex Object Type Specification

```
SQL> CREATE TYPE emp_typ AS OBJECT(  
2   emp_id    NUMBER(7) ,  
3   name      name_typ, --datatype is object type  
4   street    VARCHAR2(25) ,  
5   city      VARCHAR2(15) ,  
6   state     CHAR(2) ,  
7   zip       INTEGER,  
8   MEMBER FUNCTION get_name RETURN VARCHAR2  
9   PRAGMA RESTRICT_REFERENCES (get_name ,  
10  WNDS, RNDS, WNPS, RNPS) ,  
11  MEMBER PROCEDURE set_l_name (v_name VARCHAR2)) ;
```

The example above shows a complex object type: *emp_typ*. As you can see, the datatype of the attribute name is an object type.

Example: Creating a Complex Object Type Body

```
SQL> CREATE TYPE BODY emp_typ AS
  2  MEMBER FUNCTION get_name RETURN VARCHAR2
  3  IS
  4      BEGIN
  5          RETURN (name.l_name || ' ' || name.f_name );
  6  END;
  7  MEMBER PROCEDURE set_l_name (v_name VARCHAR2)
  8  IS
  9      BEGIN
 10          name.l_name := v_name;
 11  END;
 12 END;
```

Obtaining Descriptions of Object Type Methods from the Data Dictionary

Information about objects can be found in:

- USER_OBJECTS
- ALL_OBJECTS

Information about object type methods can be found in:

- USER_METHOD_PARAMS
- ALL_METHOD_PARAMS
- USER_METHOD_RESULTS
- ALL_METHOD_RESULTS
- USER_TYPE_METHODS
- ALL_TYPE_METHODS

Describing Object Type Methods with the Data Dictionary

The object type specification and body are stored in the data dictionary.

The views available to developers (replace % by ALL or USER) are the following:

- %_METHOD_PARAMS: Describes the method parameters of types owned by or accessible to the user.
- %_METHOD_RESULTS: Describes the method results of types owned by or accessible to the user.
- %_TYPE_METHODS: Describes the methods of types owned by or accessible to the user.

Example: Calling Object Methods

```
SQL> CREATE TABLE name_table OF name_typ;
SQL> INSERT INTO name_table
      2 VALUES('Marilyn','Monroe','MM');
SQL> SELECT nt.f_name,nt.name_typ.full_name()
      2 FROM name_table nt;
```

Because the objects being referenced in a SQL statement are unnamed rows in an object table, qualify the name of the method for the object type and pass the table name (or table alias).

Using the Constructor Method

```
SQL> CREATE TYPE tv_type AS OBJECT (  
2   tv_category  VARCHAR2(20) ,  
3   screen_size  NUMBER(4)) ;
```

```
DECLARE  
    v_new_tv      tv_type := tv_type('WEB tv', 32) ;  
    v_alt_tv      tv_type ;  
BEGIN  
    v_alt_tv := tv_type('Big Screen', 72) ;  
END;
```

Using Constructor Methods

Every object type has a constructor method. This is an implicitly defined function that has:

- The same name as the object type
- Formal parameters that exactly match the attributes of the object type (the number, order, and datatypes are the same)
- A return value of the given object type

The constructor can be invoked any place an object of that object types is allowed. You use the constructor to initialize and return an instance of that object type.

```
SQL> CREATE TABLE tv OF tv_type;  
SQL> INSERT INTO tv VALUES(tv_type('Color tv', '28'));
```

Manipulating Objects

Object type

```
SQL> CREATE TYPE emp_typ AS OBJECT(  
2   emp_id    NUMBER(7),  
3   name      name_typ, --datatype is object type  
4   street    VARCHAR2(25),  
5   city      VARCHAR2(15),  
6   state     CHAR(2),  
7   zip       INTEGER,  
8   MEMBER FUNCTION get_name RETURN VARCHAR2  
9   PRAGMA RESTRICT_REFERENCES(get_name,  
10  WNDS, RNDS, WNPS, RNPS),  
11  MEMBER PROCEDURE set_l_name (v_name VARCHAR2));
```

Object table

```
SQL> CREATE TABLE person_tab OF emp_typ;
```

In the next section we will look at selecting, inserting, updating, and deleting objects.

We use the object type `emp_typ` and create an object table `person_tab`. The table `person_tab` will be populated and store objects of type `emp_type` in its rows. Each column in a row corresponds to an attribute of the object type, and in addition to the Oracle Server ROWID, has an object identifier (OID) which uniquely identifies the object stored in that row and serves as a reference to the object.

Manipulating Objects: Selecting

To return a result set of rows:

```
BEGIN
  INSERT INTO employees --object table of type emp_typ
    SELECT * FROM person_tab p
    WHERE p.l_name LIKE '%BLAKE';
END;
```

To return a result set of objects:

```
BEGIN
  INSERT INTO employees --object table of type emp_typ
    SELECT VALUE(p) FROM person_tab p
    WHERE p.l_name LIKE '%BLAKE';
END;
```

The first example returns a result set of rows containing the attributes of emp_typ objects.

To return a result set of objects, you must use the VALUE operator. VALUE takes a correlation variable as a parameter. In this example it is a table alias associated with a row in the object table. We can now use VALUE to return a specific person object:

```
DECLARE . . .
  p1 emp_typ;
  p2 emp_typ;
  . . .
BEGIN
  SELECT VALUE(p) INTO p1 FROM person_tab p
  WHERE p.l_name = 'ALLEN';
  p2 := p1;
  ...
END;
```

We can use the local PL/SQL variables to access and update the objects they hold.

Manipulating Objects: Inserting

Adding objects to an object table:

```
BEGIN
  INSERT INTO person_tab
  VALUES (1234, 'Ellen', 'Gravina', . . .);
END;
```

Using the constructor for the object type:

```
BEGIN
  INSERT INTO person_tab
  VALUES (emp_typ(4567, 'Christian', 'Bauwens', . . .));
END;
```

The constructor is also useful with complex object types.

The example below creates a relational table with a column of the type EMP_TYP.

```
CREATE TABLE department
(dept_name varchar2(20),
 manager   emp_typ,
 location  VARCHAR2(20));
```

Next we insert a row into the table. The constructor emp_typ() provides a value for column manager.

```
BEGIN
  INSERT INTO department
  VALUES ('Education', emp_typ(7788, 'John', 'Deer', . . .), . . .,
  'Seattle', . . .);
END;
```

Manipulating Objects: Updating

Updating attributes of an object table:

```
BEGIN
  UPDATE person_tab p
  SET    p.street = '25 Elk Street'
  WHERE  p.l_name = 'Gravina';
END;
```

Using the constructor for the object type:

```
BEGIN
  UPDATE person_tab p
  SET    p = emp_typ(5678, 'Charlie', 'Brown', . . .)
  WHERE  p.l_name = 'Bauwens';
END;
```

Manipulating Objects: Deleting

Removing objects from an object table:

```
BEGIN
  DELETE FROM person_tab p
  WHERE  p.street = '25 Elk Street';
END;
```

Summary

- **Object types give developers the ability to construct datatypes that were not previously defined.**
- **An object type has three components:**
 - **Attribute specification**
 - **Method specification**
 - **Method implementation**
- **Object types can be simple or complex.**

Practice Overview

- **Identifying the components of object types**
- **Defining an object type**
- **Manipulating object types**

Practice 9

1. Create an object type called `BANK_ACCOUNT`. The attributes are: account number, balance, and status. The methods are: `open_acc`, `verify_acct`, `deposit`, and `withdraw`.

Implementation requirements:

- The status can be “open” or “closed.”
 - Use the sequence `ACCT_SEQ` to generate account numbers.
 - Open an account with an initial deposit
 - Check for wrong account number or closed account.
 - Only allow withdrawal if the account has sufficient funds.
2. Create an object table `ACCOUNT`.

Add a few objects of type `BANK_ACCOUNT` to the table.

Update the balance for one of the accounts you created.

10

Manipulating Large Objects

Objectives

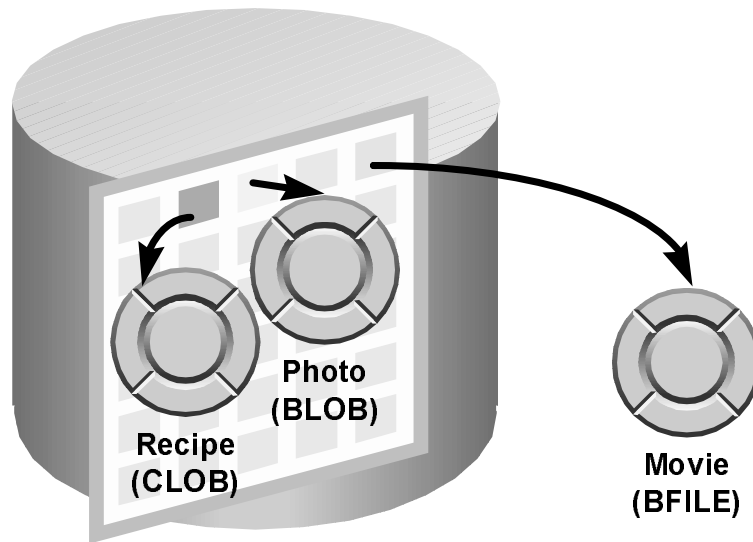
After completing this lesson, you should be able to do the following:

- **Compare and contrast LONG and large object (LOB) datatypes**
- **Create and maintain LOB datatypes**
- **Differentiate between internal and external LOBs**
- **Utilize the DBMS_LOB PL/SQL package**

Lesson Aim

Databases have long been used to store large objects. However, the mechanisms built into the database have never been as useful as the new LOB datatypes provided in Oracle8. This lesson discusses the new datatypes and their characteristics, comparing and contrasting with earlier datatypes. Examples, syntax, and issues regarding the LOB types are presented also.

Overview



10-3

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Overview

There are four Large Object (LOB) datatypes in Oracle8:

- BLOB represents a binary large object.
- CLOB represents a character large object.
- NCLOB represents a fixed-width multibyte character large object.
- BFILE represents a binary file stored outside the database.

LOBs are characterized in two ways: 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 LOB:

- Internal LOBs (CLOB, NCLOB, BLOB): Stored in the database
- External files (BFILE): Stored outside the database



The Oracle Server will not convert data between the types. For example, if the user creates a table T with a CLOB column and a table S with a BLOB column, the data is not directly transferable between these two columns.

BFILE can only be accessed in read only mode from an Oracle server.

Contrasting LONG and LOB Datatypes

LONG, LONG RAW	LOB
Single column per table	Multiple 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
No object type support	Supports object types
Sequential access to data	Random access to data

10-4

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Contrasting LONG and LOB Datatypes

LONG and LONG RAW datatypes were previously used for unstructured data, such as binary images, documents, or geographical information. These datatypes are superseded by the LOB datatypes. LOB datatypes are distinct from LONG and LONG RAW, and they are not interchangeable. LOBs will not support the LONG application programming interface (API), and vice versa.

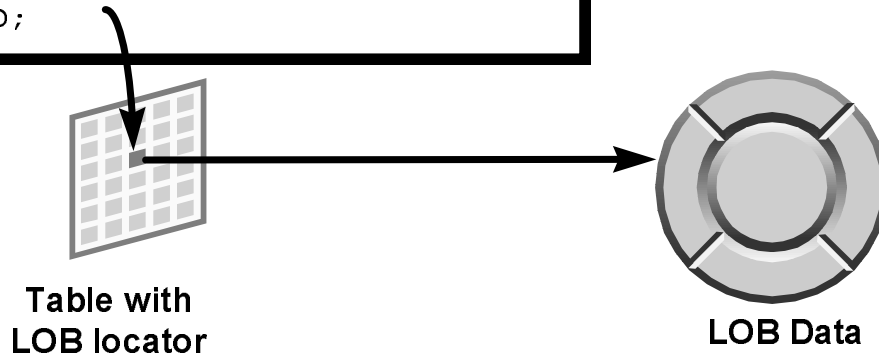
It is beneficial to discuss LOB functionality in comparison to the older types. Below, LONGs refers to LONG and LONG RAW, and LOBs refers to all LOB datatypes.

- LOBs allow multiple LOB columns per table or attributes in an object type; LONGs only one.
- The maximum size of LONGs is 2 gigabytes; LOBs can be up to 4 GB.
- Upon retrieval, LOBs return the locator; LONGs return the data.
- LOBs store a locator in the table and the data in a different segment, unless it is less than 4000 bytes; LONGs store all data in the same datablock. In addition, LOBs allow data to be stored in a separate segment and tablespace, or in a host file.
- LOBs support object type attributes (except NCLOBs); LONGs do not.
- LOBs support random piecewise access to the data through a file-like interface; LONGs are restricted to sequential piecewise access.

Anatomy of a LOB

Program with LOB locator handle

```
DECLARE
  lobloc BLOB;
BEGIN
  SELECT col1 INTO lobloc
  FROM LOB_Table WHERE col2=123;
END;
```



10-5

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Anatomy of a LOB

There are two distinct parts of a LOB:

- LOB value: What the real object being stored is made up of, data
- LOB locator: An indicator of the LOB value location stored in the database

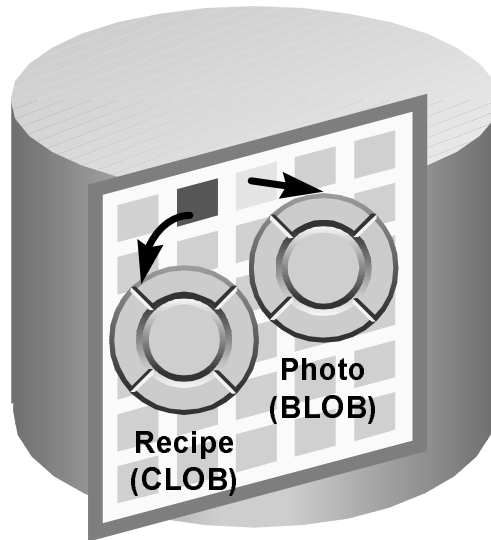
In addition, a program to access and manipulate LOBs requires a pointer or LOB locator to be declared.

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.

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

Internal LOBs



10-6

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

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 SQL bind variable
- A program host variable
- A PL/SQL variable, parameter, or result

Internal LOBs can take advantage of Oracle features such as:

- Concurrency mechanisms
- Redo logging
- Recovery mechanisms

The BLOB datatype is interpreted by the Oracle Server as a bitstream, similar to the LONG RAW datatype.

CLOB is interpreted as a single-byte character stream.

NCLOB is interpreted as a fixed-width, multiple-byte character stream, based on the byte-length of the database national character set.

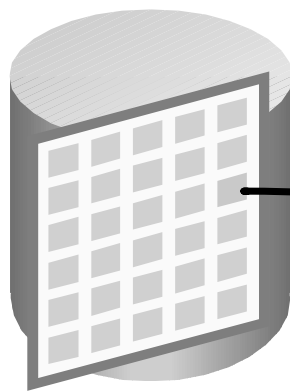
Creating a Table with LOBs: Example

```
SQL> CREATE TABLE employee
2  (emp_id      NUMBER,
3  emp_name    VARCHAR2 (35) ,
4  resume      CBLOB,
5  picture     BLOB) ;
```

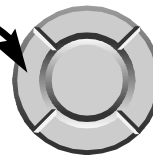
LOB columns are defined by way of SQL data definition language (DDL) as in the CREATE TABLE statement above. The contents of a LOB column is stored in the LOB segment while the column in the table only contains a reference to that specific storage area, called the LOB locator. In PL/SQL you can define a variable of type LOB, which will again only contain the value of the LOB locator.

External LOBs

Oracle8 provides a BFILE datatype to support an external or file-based large object as one of the following:



- **Attributes in an object type**
- **Column values in a table**



Movie (BFILE)

10-8

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

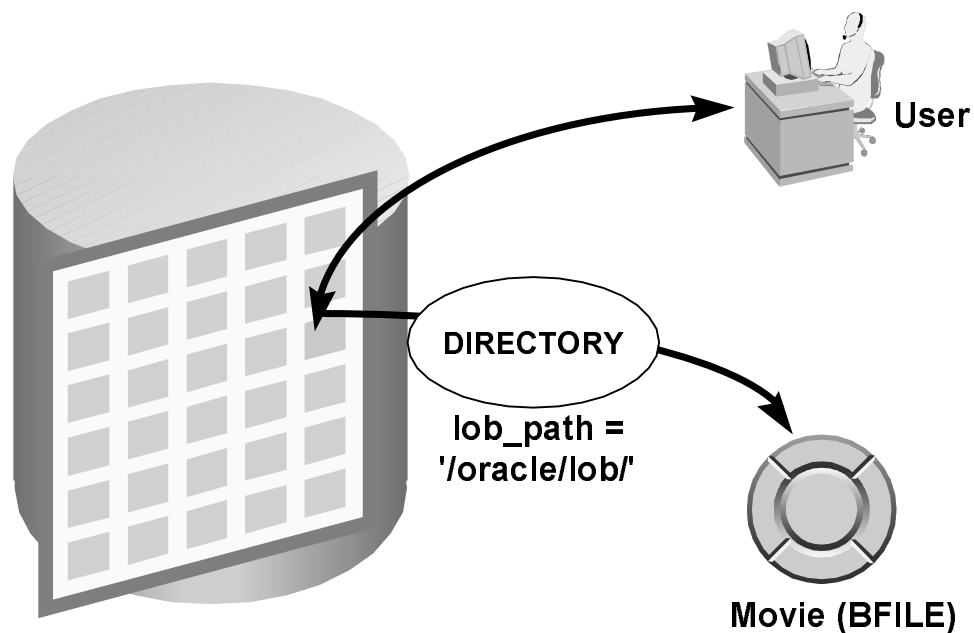
External LOBs

The BFILE datatype is provided by the Oracle Server to enable access to the external file system for a database user. Oracle8 SQL enables:

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

A BFILE column stores a BFILE locator which is nothing but a pointer to a file in the file system. The locator contains the directory alias, the filename, and some state information.

The Directory Alias



10-9

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

A New Database Item: DIRECTORY

A **DIRECTORY** object specifies an alias for a directory on the server's file system. By granting suitable privileges secured access can be provided to files in the corresponding directories on a user-by-user basis (certain directories can be made read-only, inaccessible, and so on).

Managing LOBs

- **To fully interact with LOBs, file-like interfaces are provided in:**
 - **PL/SQL package DBMS_LOB**
 - **Oracle Call Interface**
- **The Oracle Server provides some support for LOB management through SQL.**

Managing LOBs

The general method to manage an internal LOB is the following:

1. Create and populate the table containing the LOB datatype.
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 or OCI calls using the LOB locator as a reference to the LOB value.
5. Commit.

Managing BFILEs

- **Create OS directory and supply files**
- **Create Oracle table**
- **Interact with BFILE using**
 - **PL/SQL package DBMS_LOB**
 - **Oracle Call Interface**

Managing BFILES

The method in which one manages the BFILE and DIRECTORY objects follows:

1. Create the OS directory (as Oracle user) and set permissions so the Oracle Server has the READ privilege.
2. Load files into the the OS directory.
3. Create a table containing the BFILE datatype in the Oracle Server.
4. Create the DIRECTORY item and grant permissions to it through the GRANT command.
5. Insert rows into the table, associating the OS files with the corresponding row/column intersection.
6. Declare and initialize the LOB locator in program.
7. Select the row and column containing the BFILE into the LOB locator.
8. Read the BFILE with OCI or DBMS_LOB function, using the locator as a reference to the file.

Example: Populating LOBs Using SQL

To populate a LOB column:

- **INSERT** row into table, providing either a value, a **NULL**, or the empty function for the LOB.

```
SQL> INSERT INTO employee VALUES  
2> (7898, 'Marilyn Monroe', EMPTY_CLOB(), NULL);
```

- **UPDATE** row, modifying the column value to a blank **RESUME**.

```
SQL> UPDATE employee SET resume =  
2> (SELECT resume FROM employee  
3> WHERE emp_name='Default')  
4> AND emp_id = 4508;
```

10-12

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example: Populating LOBs

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.

You can directly insert a value into a LOB column by using variables in SQL or in PL/SQL, 3GL-embedded SQL, or OCI.

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 datatype (**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.

Populating LOBs with PL/SQL

```
DECLARE          -- populate through the PL/SQL package
  lobloc CLOB;   --will serve as the LOB locator
  text VARCHAR2(2000);
  amount NUMBER ;
  offset INTEGER;
BEGIN
  text := 'the text to be written into the LOB';
  SELECT resume INTO lobloc      -- get LOB handle
  FROM employee
  WHERE emp_id = 5887 FOR UPDATE;
  offset := DBMS_LOB.GETLENGTH(lobloc) + 1;
  amount := length(text);
  DBMS_LOB.WRITE ( lobloc, amount, offset, text );
  COMMIT;
  DBMS_OUTPUT.PUT_LINE('You have just written ' ||
    to_char(amount) || ' characters into the LOB');
END;
```

10-13

Copyright © Oracle Corporation, 1998. All rights reserved.

ORACLE®

Example: Populating LOBs

In the example above, the LOBLOC variable serves as the LOB locator, and the AMOUNT variable is set to the length of the text we 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 DBMS_LOB.WRITE is called to write the text into the LOB value at the specified offset.

Note: The DBMS_LOB package will be discussed later in this lesson.

Example: Removing LOBs

- **Delete rows containing LOBs.**

```
SQL>DELETE FROM person_tab  
2 WHERE pname = 'Opie Griffith';
```

- **Disassociate a LOB value from a row.**

```
SQL>UPDATE person_tab SET resume = EMPTY_CLOB()  
2 WHERE pname = 'Laura Roshto';
```

Example: Removing LOBs

A LOB instance can be deleted (destroyed) using appropriate SQL DML statements. The DELETE SQL statement will delete a row and its associated internal LOB value. To preserve the row, and destroy only the reference to the LOB, update the row, replacing the LOB column value with NULL or the empty string (), or use the EMPTY_B/C/NCLOB() function.

A LOB is destroyed when the row containing the LOB column is deleted; the table is dropped or truncated; or implicitly when the whole LOB data is updated, as in the following examples:

```
SQL> DELETE FROM lob_table_1 WHERE key = 21;  
SQL> DROP lob_table_1;  
SQL> TRUNCATE lob_table_1;  
SQL> UPDATE lob_table_1 SET blob_col = EMPTY_BLOB()  
2 WHERE key = 12;
```

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

Working with the DBMS_LOB Package

- Much of the work done with LOBs will require use of the provided PL/SQL package, DBMS_LOB.
- DBMS_LOB defines the constants
 - DBMS_LOB.LOBMAXSIZE = 4294967294
 - file_readonly constant binary_integer:=0

Working with the DBMS_LOB Package

In order to load the DBMS_LOB package, the DBA must login as SYS and submit *dbmslob.sql* and *prvtlob.plb* scripts, or execute the *catproc.sql* script. Then, users can be granted appropriate privileges to use the package.

The package does not support any concurrency control mechanism for BFILE operations.

You are 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. These constants are used in the procedures and functions of DBMS_LOB, for example to achieve the maximum possible level of purity so that they can be used in SQL expressions.

The DBMS_LOB Package

Mutators	Observers
APPEND	COMPARE
COPY	FILEGETNAME
ERASE	INSTR
TRIM	GETLENGTH
WRITE	READ
FILECLOSE	SUBSTR
FILECLOSEALL	FILEEXISTS
FILEOPEN	FILEISOPEN

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, FILEISOPEN

The DBMS_LOB Package

- **NULL arguments get NULL returns**
- **Offsets**
 - **BLOB, BFILE: Measured in bytes**
 - **CLOB, NCLOB: Measured in characters**
- **No negative values for parameters**

Using the DBMS_LOB Routines (continued)

All functions in the DBMS_LOB package return NULL if any of the input parameters are NULL. All mutator procedures in the DBMS_LOB package raise an exception if the destination LOB/BFILE is input as NULL.

For Oracle8, only positive, absolute offsets are allowed. They represent the number of bytes/characters from the beginning of LOB data to start the operation from. Negative offsets and ranges observed in SQL8 string functions and operators are not allowed. Corresponding exceptions will be raised upon violation. The default value for an offset is 1, which indicates the first byte/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 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
```

10-18

Copyright © Oracle Corporation, 1998. All rights reserved. ORACLE®

READ

Call the READ procedure to piecewise read and return 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 value 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. The Oracle Server will raise the appropriate memory exceptions otherwise.

Note: BLOB and BFILE return RAW; the others return VARCHAR2.

WRITE

Call the WRITE procedure to piecewise write 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 if they match, and will write AMOUNT bytes of the buffer contents into the LOB.

Summary

- **Four built-in types for large objects:
BLOB, CLOB, NCLOB, BFILE**
- **LOBs replace LONG and LONG RAW:
New and old types are not compatible**
- **Two storage options for LOBs:**
 - **Stored in the Oracle Server (internal LOBs)**
 - **Stored in external host files (external LOBs)**
- **DBMS_LOB PL/SQL package provides routines for LOB management.**

Practice Overview

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

Create an object type with both BLOB and CLOB attributes and create a table of that type.

Use the DBMS_LOB package to populate and interact with the data: Declare and work with the BFILE datatype.

Practice 10

1. Create a table, PERSONNEL, using the following attributes and datatypes:

Column Name	Datatype	Length
id	number	6
last_name	varchar2	35
large	clob	n/a
picture	blob	n/a

2. Insert two records in the PERSONNEL table. Use the empty function for the CLOB, and provide NULL as the value for the BLOB.
3. Examine and execute the script *labs\p10_3.sql*.
4. Update the PERSONNEL table.

- Populate the CLOB for the first record using the following query in a SQL UPDATE statement:

```
SELECT resume  
FROM res_table;
```

- Populate the CLOB for the second record using PL/SQL and the DBMS_LOB package.

A

Cursor Variables

Managing Cursor Variables

What Are Cursor Variables?

Like a cursor, a cursor variable points to the current row in the result set of a multi row query. Cursor variables however are like C pointers, they hold the memory location of an item instead of the item itself. In this way, cursor variables differ from cursors the way constants differ from variables. A cursor is static, a cursor variable is dynamic. In PL/SQL, a cursor variable has a datatype REF CURSOR, where REF stands for reference, and CURSOR stands for the class of the object.

Using Cursor Variables

To execute a multi row query, the Oracle Server opens a work area called a cursor to store processing information. To access the information, you either explicitly name the work area, or you use a cursor variable which points to the work area. Whereas a cursor always refers to the same work area, a cursor variable can refer to different work areas. Hence, cursors and cursor variables are not interoperable.

Primarily you use a cursor variable to pass query result sets between PL/SQL stored subprograms and various clients. None of them owns the result set, they simply share a pointer to the query work area which stores the result set. You can declare a cursor variable on the client side, open and fetch from it on the server side, and then continue to fetch from it on the client side.

Working with Cursor Variables

There are four steps for handling a cursor variable:

- 1 Define and declare the cursor variable.
- 2 Open the cursor variable.
- 3 Fetch rows from the result set one at the time.
- 4 Close the cursor variable.

The next section contains detailed information on each of the steps.

Step 1: Defining a Cursor Variable

To create a cursor variable, you first need to define a REF CURSOR type, and then declare a variable of that type.

- Defining the REF CURSOR TYPE

```
TYPE  ref_type_name IS REF CURSOR  [RETURN return_type];
```

where *ref_type_name* a type specifier in subsequent declarations.
 return_type represents a row in a database table.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). A strong REF CURSOR type definition specifies a return type, a weak definition does not. PL/SQL lets you associate a strong type with type-compatible queries only, whereas a weak type can be associated with any query. This makes strong REF CURSOR types less error prone, but weak REF CURSOR types more flexible.

In the example below, the first definition is strong, the second is said to be weak.

```
DECLARE  
    TYPE emp_cur_typ IS REF CURSOR RETURN emp%ROWTYPE;  
    TYPE general_purpose_typ IS REF CURSOR;  
    . . .
```

- Declaring a cursor variable

```
cursor_variable_name      ref_type_name;
```

where *cursor_variable_name* name of the cursor variable.
 ref_type_name name of the REF CURSOR type.

Cursor variables follow the same scoping and instantiation rules as all other PL/SQL variables.

In the example below, we declare the cursor variable *emp_cur_var*.

```
DECLARE  
    TYPE emp_cur_typ IS REF CURSOR RETURN emp%ROWTYPE;  
    emp_cur_var  emp_cur_typ;
```

Step 1: Defining a Cursor Variable (continued)

Here are some more examples of cursor variable declarations:

- Use %TYPE to provide the datatype of a record variable.

```
DECLARE
    emp_rec emp%rowtype;          --record variable
    TYPE emp_cur_typ IS REF CURSOR RETURN emp_rec%TYPE;
    emp_cur_var emp_cur_typ;      --cursor variable
```

- Specify a user-defined RECORD type in the RETURN clause.

```
DECLARE
    TYPE emp_rec_typ IS RECORD
        (empno      NUMBER(4) ,
         ename       VARCHAR2(10) ,
         sal         NUMBER(7,2));
    TYPE emp_cur_typ IS REF CURSOR RETURN emp_rec_typ;
    emp_cur_var emp_cur_typ;
```

- Declare a cursor variable as the formal parameter of a stored procedure or a function.

```
DECLARE
    TYPE emp_cur_typ IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE use_emp_cur_var(emp_cur_var IN OUT emp_cur_typ)
    IS . . .
```

Step 2: Opening a Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multi row query, it executes the query and identifies the result set. The syntax is:

```
OPEN    cursor_variable_name FOR select_statement;
```

where *cursor_variable_name* the cursor variable to be used.
 select_statement the SQL SELECT statement associated with the
 cursor variable.

Other OPEN-FOR statements can open the same cursor variable for different queries. You do not need to close a cursor variable before reopening it. Keep in mind that when you reopen a cursor variable for a different query, the previous query is lost.

In the example below, the packaged procedure declares a variable used to select one of several alternatives in an IF THEN ELSE statement. When called, the procedure opens the cursor variable for the chosen query.

```
CREATE OR REPLACE PACKAGE emp_data AS
    TYPE emp_cur_typ IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_emp_cur_var(emp_cur_var IN OUT emp_cur_typ,
                               your_choice IN NUMBER);
END emp_data;

CREATE OR REPLACE PACKAGE BODY emp_data AS
    PROCEDURE open_emp_cur_var(emp_cur_var IN OUT emp_cur_typ,
                               your_choice IN NUMBER) IS
    BEGIN
        IF your_choice = 1 THEN
            OPEN emp_cur_var FOR SELECT * FROM emp;
        ELSIF your_choice = 2 THEN
            OPEN emp_cur_var FOR SELECT * FROM EMP WHERE sal > 2000;
        ELSIF your_choice = 3 THEN
            OPEN emp_cur_var FOR SELECT * FROM EMP WHERE deptno = 30;
        END IF;
    END open_emp_cur_var;
END emp_data;
```

Step 3: Fetching from a Cursor Variable

The FETCH statement retrieves rows from the result set one at a time. The syntax is:

```
FETCH    cursor_variable_name
INTO     variable_name1 [,variable_name2,. . .] | record_name;
```

PL/SQL makes sure that the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. For each query column value returned there must be a type-compatible variable in the INTO clause. Also the number of query column values must equal the number of variables. In case of a mismatch in number or type, the error occurs at compile time for strongly typed cursor variables, and at runtime for weakly typed cursor variables.

Note: When you declare a cursor variable as the formal parameter of a subprogram that fetches from a cursor variable, you must specify the IN (or IN OUT) mode. If the subprogram also opens the cursor variable, you must specify the IN OUT mode.

Step 4: Closing a Cursor Variable

The CLOSE statement disables a cursor variable. After that the result set is undefined. The syntax is:

```
CLOSE cursor_variable_name;
```

In the example below, the cursor is closed when the last row is processed.

```
. . .
LOOP
    FETCH emp_cur_var INTO emp_rec;
    EXIT WHEN emp_cur_var%NOTFOUND;
    . . .
END LOOP;
CLOSE emp_cur_var;
. . .
```

Restrictions with Cursor Variables

- You cannot declare cursor variables in a package because they do not have a persistent state.
- You cannot use remote program calls to pass cursor variables from one server to another.
- You cannot include the FOR UPDATE clause in the query associated with a cursor variable.
- You cannot assign NULL values to a cursor variable.
- You cannot use REF CURSOR types to specify column types in a CREATE TABLE or a CREATE VIEW statement.
- You cannot use REF CURSOR types to specify the element type of a collection type (nested table, index-by table, or varray).
- You cannot use cursor variables with dynamic SQL.

B

Practice Solutions

Practice 2 Solutions

1. Load and execute a loop counter.
 - a. Launch Procedure Builder. Your instructor will give you the login information.
 - ◆ Locate the Procedure Builder icon.
 - b. From the menu, load the *labs\p2loop.pls* file.
 - ◆ Click on File—>Load.
 - ◆ Browse for the .pls file mentioned above
 - c. Examine the code using the Program Unit Editor.
 - ◆ In the Program Units node, double click on the p2loop program unit icon.
 - d. Compile the procedure. Correct any errors.
 - ◆ The error messages pane clearly shows there are a number of problems with the code:
 - Two dots are required in the FOR LOOP clause
 - The semi colon is missing at the end of the TEXT_IO.PUT_LINE command
 - The FOR LOOP statement needs END LOOP;
 - e. Execute the procedure from the interpreter pane. Pass a numeric value into the procedure as demonstrated below.

```
PL/SQL> my_loop (4) :
```

- ◆ Close the Program Unit Editor
 - ◆ From the menu, click on Program—>PL/SQL Interpreter
 - ◆ Click in the PL/SQL Interpreter window pane
 - ◆ At the PL/SQL> prompt enter the code shown above
2. Create a client-side procedure named MY_MESSAGE to print the message you provide at design time. The executable part in your procedure should contain the following line of code

```
PROCEDURE my_message IS  
BEGIN  
    TEXT_IO.PUT_LINE('Hello world') ;  
END;
```

- ◆ In the Object Navigator, click on the Program Units node.
- ◆ Click on the Create button in the vertical toolbar.
- ◆ Enter the name of the procedure in the New Program Unit dialog box.
- ◆ Click on OK.
- ◆ Type in the code shown above.

Compile and execute your procedure from the interpreter window pane as shown below.

```
PL/SQL> my_message;
```

Practice 2 Solutions (continued)

3. Add a new department to your DEPT table using the interpreter pane. Verify the insert.
- ◆ Enter your INSERT INTO statement at the PL/SQL> prompt in the Interpreter pane.

```
SQL> INSERT INTO dept VALUES (50, 'MIS', 'HOUSTON');
```

- ◆ Query the DEPT table.

```
SQL> SELECT * FROM dept WHERE deptno = 50;
```

4. Insert a new department using a procedure.

- a. From the menu, load the *labs\p2ins.pls* file.
- b. Replace the comment inside the procedure with your INSERT statement.

```
SQL> INSERT INTO dept VALUES (60, 'HR', 'DENVER');
```

- c. Execute the procedure from the interpreter pane.
- d. Verify the insert.

```
SQL> SELECT * FROM dept WHERE deptno = 60;
```

Practice 3 Solutions

1. Create and invoke the ADD_PROD procedure and consider the results.

a. Create a procedure called ADD_PROD to insert a new product into the PRODUCT table.

```
CREATE OR REPLACE PROCEDURE add_prod
(v_prodid IN product.prodid%TYPE,
v_descrip IN product.descrip%TYPE)
IS
BEGIN
INSERT INTO product (prodid, descrip)
VALUES      (v_prodid, v_descrip);
COMMIT;
END add_prod;
/
```

b. Compile the code, invoke the procedure and then query the PRODUCT table to view the results.

```
SQL> START p3_1.sql
Procedure created.

SQL> EXECUTE add_prod (9999, 'SP TENNIS BALLS')
PL/SQL procedure successfully completed.
```

c. Invoke the procedure again, passing a prodid of 100860. What happens and why?

```
SQL> EXECUTE add_prod (100860, 'SP ADULT TENNIS RACKET')
begin add_prod(100860, 'SP ADULT TENNIS RACKET'); end;

*
ERROR at line 1
ORA-00001:      unique constraint(SCOTT.PRODUCT_PRIMARY_KEY)
                violated
ORA-06512:      at "SCOTT.ADD_PROD", line 6
ORA-06512:      at line 1
```

There is a primary key integrity constraint on the *prodid* column.

Practice 3 Solutions (continued)

2. Create a procedure called UPD_PROD to modify a product in the PRODUCT table.
 - a. Create a procedure called UPD_PROD to update the product description. Include the necessary exception handling.

```
CREATE OR REPLACE PROCEDURE upd_prod
  (v_prodid  IN product.prodid%TYPE,
   v_descrip IN product.descrip%TYPE)
IS
BEGIN
  UPDATE product
  SET   descrip = v_descrip
  WHERE prodid = v_prodid;

  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202,'No products updated.');
```

- b. Compile the code, invoke the procedure and then query the product table to view the results. Also check the exception handling by trying to update a product which does not exist.

```
SQL> START p3_2.sql
Procedure created.

SQL> EXECUTE upd_prod (9999, 'SP TENNIS NETS')
PL/SQL procedure successfully completed.
```

Practice 3 Solutions (continued)

3. Create a procedure called DEL_PROD to delete a product from the PRODUCT table.
 - a. Create a procedure called DEL_PROD to delete a product from the PRODUCT table. Include the necessary exception handling.

```
CREATE OR REPLACE PROCEDURE del_prod
  (v_prodid IN product.prodid%TYPE)
IS
BEGIN
  DELETE FROM product
  WHERE prodid = v_prodid;

  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20203, 'No products deleted. ');
  END IF;
END DEL_PROD;
/
```

- b. Compile the code, invoke the procedure and query the PRODUCT table to view the result. Also, check the exception handling by trying to delete a product that does not exist.

```
SQL> START p3_3.sql
Procedure created.

SQL> EXECUTE del_prod (9999)
PL/SQL successfully completed.
```

Practice 3 Solutions (continued)

4. Create a procedure to query the EMP table, retrieving the salary and job title for employee 7839.
 - a. Create a procedure which returns a value from the SAL and JOB columns for a specified employee (use EMPNO).

```
CREATE OR REPLACE PROCEDURE query_emp
  (v_empno IN emp.empno%TYPE,
   v_sal   OUT emp.sal%TYPE,
   v_job   OUT emp.job%TYPE)
IS
BEGIN
  SELECT  sal, job
  INTO    v_sal, v_job
  FROM    emp
  WHERE   empno = v_empno;
END query_emp;
/
```

- b. Compile the code, invoke the procedure and display the salary and job title for employee 7839.

```
SQL> START p3_4.sql
Procedure created.

SQL> VARIABLE g_salNUMBER
SQL> VARIABLE g_job VARCHAR2 (15)
SQL> EXECUTE query_emp (7839, :g_sal, :g_job)
PL/SQL procedure successfully completed.
```

Practice 3 Solutions (continued)

```
SQL> PRINT g_sal  
      G_SAL  
      ----  
      5000  
  
SQL> PRINT g_job  
      G_JOB  
      -----  
      PRESIDENT
```

c. Invoke the procedure again, passing an EMPNO of 9898. What happens and why?

```
SQL> execute query_emp (9898, :g_sal, :g_job)  
begin query_emp (9898, :g_sal, :g_job); end;  
  
*  
ERROR at line 1:  
ORA-01403: no data found  
ORA-06512: at "SCOTT.QUERY_EMP", line 7  
ORA-06512: at line 1
```

There is no employee in the EMP table with an EMPNO of 9898. The SELECT statement retrieved no data from the database, resulting in a fatal PL/SQL error, NO_DATA_FOUND.

Practice 4 Solutions

1. Create and invoke the Q_PROD function to return a product description.
 - a. Create a function called Q_PROD to return a product description to a host variable.

```
CREATE OR REPLACE FUNCTION q_prod
(v_prodid IN product.prodid%TYPE)
RETURN VARCHAR2
IS
    v_descrip product.descrip%TYPE;
BEGIN
    SELECT    descrip
    INTO      v_descrip
    FROM      product
    WHERE     prodid = v_prodid;
    RETURN    (v_descrip);
END q_prod;
/
```

- b. Compile the code, invoke the function, and then query the host variable to view the result.

```
SQL> START p4_1.sql
Function created.

SQL> VARIABLE g_descrip VARCHAR2 (30)
SQL> EXECUTE :g_descrip := q_prod (101863)
PL/SQL procedure successfully completed.

SQL> PRINT g_descrip
G_DESCRIP
-----
SP JUNIOR RACKET
```

Practice 4 Solutions (continued)

2. Create the function ANNUAL_COMP to return an annual salary when passed an employee's monthly salary and commission. Be sure the function addresses NULL values.

- a. Create and invoke the function ANNUAL_COMP, passing in values for monthly salary and commission. The function should return the annual salary, as defined by:

```
(sal*12) + comm
```

Address NULL values in the function.

```
CREATE OR REPLACE FUNCTION annual_comp
  (v_sal  IN emp.sal%TYPE,
   v_comm IN emp.comm%TYPE)
  RETURN NUMBER
IS
BEGIN
  RETURN (NVL(v_sal,0) * 12 + NVL(v_comm,0));
END annual_comp;
/
```

- b. Use the stored function in a SELECT statement against the EMP table.

```
SQL> SELECT empno, ename,
  2   annual_comp(sal, comm) "Annual Compensation"
  3 FROM   emp;
```

EMPNO	ENAME	Annual Compensation
7839	KING	60000
7698	BLAKE	34200
7782	CLARK	29400
7566	JONES	35700
.	.	.

14 rows selected.

Practice 4 Solutions (continued)

3. Create a procedure, NEW_EMP, to insert a new employee into the EMP table. The procedure should contain a call to the function VALID_DEPTNO to check if the department number specified for the new employee exists in the DEPT table.
 - a. Create a function VALID_DEPTNO to validate a specified department number. The function should return a BOOLEAN.

```
CREATE OR REPLACE FUNCTION valid_deptno
(v_deptno IN dept.deptno%TYPE)
RETURN BOOLEAN
IS
    v_dummy  VARCHAR2 (1) ;
BEGIN
    SELECT  'x'
    INTO    v_dummy
    FROM    dept
    WHERE   deptno = v_deptno;
    RETURN (TRUE) ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN (FALSE) ;
END valid_deptno;
/
```

Practice 4 Solutions (continued)

- b. Then create the procedure NEW_EMP to add an employee to the EMP table. A new record should be added to EMP if the function returns TRUE. If the function returns FALSE, the procedure should alert the user with an appropriate message.

Define the following DEFAULT values . The default commission is 0, the default salary is 1000, the default number is 30, the default job is SALESMAN and the default manager number is 7839. For the employee's ID number, use the sequence SEQ_EMPNO. Run the `\labs\cre_seq.sql` file to create the sequence.

```
CREATE OR REPLACE PROCEDURE new_emp
(v_ename   emp.ename%TYPE,
v_job      emp.job%TYPE      DEFAULT 'SALESMAN',
v_mgr      emp.mgr%TYPE      DEFAULT 7839,
v_sal      emp.sal%TYPE      DEFAULT 1000,
v_comm     emp.comm%TYPE     DEFAULT 0,
v_deptno   emp.deptno%TYPE   DEFAULT 30)
IS
BEGIN
    IF valid_deptno(v_deptno) THEN
        INSERT INTO emp
        VALUES (seq_empno.NEXTVAL, v_ename, v_job, v_mgr,
                TRUNC (SYSDATE, 'DD'), v_sal, v_comm, v_deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE ('Invalid department number.  Try
                                again.');
```

- c. Test your NEW_EMP procedure by adding a new employee named HARRIS to department 99. Let all other parameters default. What was the result?

```
SQL> execute new_emp(v_ename => 'HARRIS',v_deptno => 99)
Invalid department number.  Try again.
PL/SQL procedure successfully completed.
```

- d. Test your NEW_EMP procedure by adding a new employee named HARRIS to department 30. Let all other parameters default. What was the result?

```
SQL> execute new_emp(v_ename => 'HARRIS',v_deptno => 30)
PL/SQL procedure successfully completed.
```

Practice 5 Solutions

1. Create a package specification and body called PROD_PACK that contains your ADD_PROD, UPD_PROD, DEL_PROD procedures, and your Q_PROD function.

- a. Make all the constructs public.

Note: Consider whether you still need the standalone procedures and functions you just packaged.

```
CREATE OR REPLACE PACKAGE prod_pack IS
    PROCEDURE add_prod
        (v_prodid  IN product.prodid%TYPE,
         v_descrip IN product.descrip%TYPE) ;
    PROCEDURE upd_prod
        (v_prodid  IN product.prodid%TYPE,
         v_descrip IN product.descrip%TYPE) ;
    PROCEDURE del_prod
        (v_prodid IN  product.prodid%TYPE) ;
    FUNCTION q_prod
        (v_prodid IN  product.prodid%TYPE)
        RETURN VARCHAR2 ;
END prod_pack ;
/
```

Practice 5 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY prod_pack IS
  PROCEDURE add_prod
    (v_prodid  IN product.prodid%TYPE,
     v_descrip IN product.descrip%TYPE)
  IS
  BEGIN
    INSERT INTO product (prodid, descrip)
    VALUES              (v_prodid, v_descrip);
  END add_prod;

  PROCEDURE upd_prod
    (v_prodid  IN product.prodid%TYPE,
     v_descrip IN product.descrip%TYPE)
  IS
  BEGIN
    UPDATE product
    SET    descrip = v_descrip
    WHERE  prodid = v_prodid;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR(-20202, 'No products updated. ');
    END IF;
  END upd_prod;

  PROCEDURE del_prod
    (v_prodid IN product.prodid%TYPE)
  IS
  BEGIN
    DELETE FROM    product
    WHERE          prodid = v_prodid;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR (-20203, 'No products deleted. ');
    END IF;
  END DEL_PROD;
/
```

Practice 5 Solutions (continued)

```
FUNCTION q_prod
  (v_prodid IN product.prodid%TYPE)
  RETURN VARCHAR2
IS
  v_descrip product.descrip%TYPE;
BEGIN
  SELECT  descrip
  INTO    v_descrip
  FROM    product
  WHERE   prodid = v_prodid;
  RETURN  (v_descrip);
END q_prod;
END prod_pack;
/
```

b. Invoke your DEL_PROD procedure.

```
SQL> execute prod_pack.del_prod(100860)
PL/SQL procedure successfully completed.
```

c. Query the PRODUCT table to see the result.

```
SQL> SELECT * from product
      2 WHERE prodid = 100860;
no rows selected
```

Practice 5 Solutions (continued)

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_DEPTNO function as a private construct.

```
CREATE OR REPLACE PACKAGE emp_pack IS
PROCEDURE new_emp
    (v_ename    emp.ename%TYPE,
     v_job      emp.job%TYPE      DEFAULT 'SALESMAN',
     v_mgr      emp.mgr%TYPE      DEFAULT 7839,
     v_sal      emp.sal%TYPE      DEFAULT 1000,
     v_comm     emp.comm%TYPE     DEFAULT 0,
     v_deptno   emp.deptno%TYPE   DEFAULT 30);
END emp_pack;
/
```


Practice 5 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY emp_pack IS
    FUNCTION valid_deptno
        (v_deptno IN dept.deptno%TYPE)
        RETURN BOOLEAN
    IS
        v_dummy  VARCHAR2(1);
    BEGIN
        SELECT 'x'
        INTO    v_dummy
        FROM    dept
        WHERE   deptno = v_deptno;
        RETURN (TRUE);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN(FALSE);
    END valid_deptno;
    PROCEDURE new_emp
        (v_ename  emp.ename%TYPE,
         v_job     emp.job%TYPE      DEFAULT 'SALESMAN',
         v_mgr     emp.mgr%TYPE      DEFAULT 7839,
         v_sal     emp.sal%TYPE      DEFAULT 1000,
         v_comm    emp.comm%TYPE     DEFAULT 0,
         v_deptno  emp.deptno%TYPE   DEFAULT 30)
    IS
    BEGIN
        IF valid_deptno(v_deptno) THEN
            INSERT INTO emp
            VALUES (seq_empno.NEXTVAL, v_ename, v_job, v_mgr,
                    TRUNC (SYSDATE, 'DD'), v_sal, v_comm, v_deptno);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Invalid department number. Try again. ');
        END IF;
    END new_emp;
END emp_pack;
/
```

Practice 5 Solutions (continued)

- b. Invoke the NEW_EMP procedure using 99 as a department number.

```
SQL> execute emp_pack.new_emp (v_ename => 'HARRIS' ,  
                               v_deptno=>99)  
  
Invalid department number try again.
```

- c. Invoke the NEW_EMP procedure using 30 as a department number.

```
SQL> execute emp_pack.new_emp (v_ename => 'HARRIS' ,  
                               v_deptno=>30)  
  
PL/SQL procedure successfully completed.
```

3. Create a package called CHK_PACK that contains the procedures CHK_HIREDATE and CHK_DEPT_MGR. Make both constructs public.

- a. The procedure CHK_HIREDATE checks whether an employee's hiredate is within the following range: [sysdate - 50 years, sysdate + 3 months].

Notes:

- 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 hiredate should be treated as an invalid hiredate.

- b. The procedure CHK_DEPT_MGR checks the department and manager combination for a given employee. This means that the manager number provided must be equal to the manager number supervising the employee's department.

Notes:

- If the department number/manager combination is invalid, you should raise an application error with an appropriate message.
- Make sure you handle the case where there is no manager for the department.

```
CREATE OR REPLACE PACKAGE chk_pack IS  
  PROCEDURE chk_hiredate(v_date in emp.hiredate%type);  
  PROCEDURE chk_dept_mgr(v_empno in emp.empno%type,  
                          v_mgr in emp.mgr%type);  
  
END chk_pack;  
/
```

Practice 5 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY chk_pack IS
  PROCEDURE chk_hiredate(v_date in emp.hiredate%type)
  IS
    v_low date := sysdate - (50 * 365);
    v_high date := add_months(sysdate,3);
  BEGIN
    IF TRUNC(v_date) NOT BETWEEN v_low and v_high
      OR v_date IS NULL THEN
      RAISE_APPLICATION_ERROR(-20200,'Not a valid hiredate');
    END IF;
  END chk_hiredate;
  PROCEDURE chk_dept_mgr(v_empno in emp.empno%type,
                        v_mgr in emp.mgr%type)
  IS
    v_empnr emp.empno%type;
    v_deptno emp.deptno%type;
  BEGIN
    SELECT deptno --pick up the department number
    INTO v_deptno
    FROM emp
    WHERE empno = v_empno;
    SELECT empno /*check valid combination
                deptno/mgr for given employee */
    INTO v_empnr
    FROM emp
    WHERE deptno = v_deptno
    AND empno = v_mgr
    AND job = 'MANAGER';
    IF v_empnr IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('No manager for this department');
    END IF;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20201,'Not the mgr for this
        department');
    WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR(-20202,'Other error occurred');
  END chk_dept_mgr;
END chk_pack;
/
```

Practice 5 Solutions (continued)

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

```
SQL> execute chk_pack.chk_hiredate('01-JAN-47')
      begin chk_pack.chk_hiredate('01-JAN-47'); end;

*
ERROR at line 1:
ORA-20200: Not a valid hiredate
ORA-06512: at "SCOTT.CHK_PACK", line 9
ORA-06512: at line 1
```

- d. Test the CHK_HIREDATE procedure with the following command.

```
SQL> execute chk_pack.chk_hiredate(NULL)
      begin chk_pack.chk_hiredate(NULL); end;

*
ERROR at line 1:
ORA-20200: Not a valid hiredate
ORA-06512: at "SCOTT.CHK_PACK", line 9
ORA-06512: at line 1
```

- e. Test the CHK_HIREDATE procedure with the following command.

```
SQL> execute chk_pack.chk_hiredate('01-JAN-98')
PL/SQL procedure successfully completed.
```

Practice 6 Solutions

1. Create a new package to implement a new business rule.
 - a. Create a procedure called `CHK_DEPT_JOB` to verify whether a given combination of department number and job is a valid one. In this case “valid” means that it must be a combination that currently exists in the `EMP` table.

Notes:

- 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 chk_pack2 IS
    PROCEDURE chk_dept_job(v_deptno in emp.deptno%type,
                           v_job      in emp.job%type);
END chk_pack2;
/
```

Practice 6 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY chk_pack2 IS
    i number :=1;
    TYPE emp_table_type IS TABLE OF VARCHAR2(50)
    INDEX BY BINARY_INTEGER;
    deptno_job emp_table_type;

    CURSOR emp_cur IS
    SELECT deptno,job
    FROM emp;

    PROCEDURE chk_dept_job(v_deptno in emp.deptno%type,
                           v_job      in emp.job%type)
    IS
        v_deptno_job      varchar2(50);
    BEGIN
        v_deptno_job := to_char(v_deptno) || v_job;
        FOR k in 1..i-1 LOOP
            IF v_deptno_job = deptno_job(k) THEN
                EXIT;
            ELSIF (v_deptno_job != deptno_job(k) and
                  k >= i-1) THEN
                RAISE_APPLICATION_ERROR(-20500,'Not a valid job for
                this dept');
            END IF;
        END LOOP;
    END chk_dept_job;

    BEGIN          -- one-time-only-procedure
        FOR emp_rec in emp_cur LOOP
            deptno_job(i) := to_char(emp_rec.deptno) || emp_rec.job;
            i := i + 1;
        END LOOP;
    END chk_pack2;
/
```

Practice 6 Solutions (continued)

- b. Test your CHK_DEPT_JOB package procedure by executing the following command.

```
SQL> execute chk_pack.chk_dept_job(20,'CLERK')
```

- c. Test your CHK_DEPT_JOB package procedure by executing the following command.

```
SQL> execute chk_pack.chk_dept_job(40,'CLERK')
```

Practice 6 Solutions (continued)

2. Create two functions, each called PRINT_IT to print a date, or a number depending on how the function is invoked.

Notes:

- 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 the number, use “999,999.00” as the output format.

```
CREATE OR REPLACE PACKAGE over_load is
    FUNCTION print_it(v_arg date)
        RETURN VARCHAR2;
    FUNCTION print_it(v_arg VARCHAR2)
        RETURN NUMBER;
END over_load;
/
```

```
CREATE OR REPLACE PACKAGE BODY over_load is
    FUNCTION print_it(v_arg date)
        RETURN VARCHAR2
    IS
    BEGIN
        RETURN to_char(v_arg, 'FmMonth,dd/yyyy') ;
    END print_it;

    FUNCTION print_it(v_arg VARCHAR2)
        RETURN NUMBER
    IS
    BEGIN
        RETURN to_number(v_arg, '999,999.00') ;
    END print_it;

END over_load;
/
```


Practice 6 Solutions (continued)

- a. Test the first version of PRINT_IT with the following command.

```
SQL> variable todays_date varchar2(20)
SQL> execute :todays_date := over_load.print_it(sysdate)
PL/SQL procedure successfully completed.
SQL> print todays_date
TODAYS_DATE
-----
January, 29/1998
```

- b. Test the second version of PRINT_IT with the following command.

```
SQL> variable g_emp_sal number
SQL> execute :g_emp_sal := over_load.print_it('33,600')
PL/SQL procedure successfully completed.
SQL> print g_emp_sal
G_EMP_SAL
-----
33600
```

Practice 6 Solutions (continued)

3. 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
    (v_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'||
        v_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;
/
```

- a. Test the DROP_TABLE procedure by creating a new table called EMP_DUP as a copy of the EMP table, then executing the DROP_TABLE procedure to drop the EMP_DUP table.

```
SQL> create table emp_dup as
    2  select * from emp;
Table created.
SQL> execute drop_table('emp_dup')
PL/SQL procedure successfully completed.
SQL> select * from emp_dup;
select * from emp_dup
            *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Practice 7 Solutions

1. DML will only be allowed on tables during normal office hours of 8:45 in the morning until 5:30 in the afternoon, Monday through Friday.
 - a. Create a stored procedure called SECURE_DML that will fail outside of these hours returning the message:
"You may only make changes 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 only make changes during normal office hours');
    END IF;
END secure_dml;
/
```

2. Create a trigger on the PRODUCT table which calls the above procedure.

```
CREATE OR REPLACE TRIGGER secure_prod
BEFORE INSERT OR UPDATE OR DELETE ON product
BEGIN
    secure_dml;
END secure_prod;
/
```

- a. Test the procedure by temporarily modifying the hours in the procedure and attempting to insert a new record into the PRODUCT table. After testing, reset the procedure hours as specified in step 1.

```
SQL> insert into product
      2 values (99999,'My Product');
insert into product
      *

ERROR at line 1:
ORA-20205: You may only make changes during normal office hours.
ORA-06512: at "SCOTT.SECURE_DML", line 6
ORA-06512: at "SCOTT.SECURE_PROD", line 2
ORA-04088: error during execution of trigger 'SCOTT.SECURE_PROD'
```

Practice 7 Solutions (continued)

3. A sales person's commission should change for any new orders or changes to existing orders. Their commission is stored in the COMM column of the EMP table. A sales person is assigned to a particular customer in the CUSTOMER table.
- a. Create a procedure that will update the relevant sales person's commission. Use parameters to accept the customer id, old order total and new order total from the calling trigger. The procedure will then need to find the appropriate employee number from the CUSTOMER table and update the sales person's record in the EMP table, adding the new commission to the existing value. Assume for this exercise a fixed commission rate of 5%.

```
CREATE OR REPLACE PROCEDURE update_comm
    (v_custid    IN    NUMBER,
     v_old_tot   IN    NUMBER,
     v_new_tot   IN    NUMBER)
IS
    v_repid    NUMBER;
    v_comm     NUMBER;
BEGIN
    SELECT repid
    INTO    v_repid
    FROM    customer
    WHERE   custid = v_custid;

    v_comm := (NVL(v_new_tot, 0) - NVL(v_old_tot, 0)) * .05;
    UPDATE  emp
    SET      comm = comm + v_comm
    WHERE   empno = v_repid;
END update_comm;
/
```

- b. Create a trigger on the ORD table which will call the procedure, passing the required parameters.

```
CREATE OR REPLACE TRIGGER update_emp_comm
    AFTER INSERT OR UPDATE OR DELETE ON ord
    FOR EACH ROW
BEGIN
    update_comm (:new.custid, :old.total, :new.total);
END;
/
```

Practice 7 Solutions (continued)

- c. Use two packaged procedures to alter the customer's order, UPD_ITEM and ADD_ITEM in the ITEM_PACK package. There is a script in your working directory, *p7_3.sql* that will create ITEM_PACK.

```
SQL> START p7_3.sql
Package created.

Package body created.
```

The package contains two public procedures, ADD_ITEM and UPD_ITEM.

```
ADD_ITEM (v_ordid    IN NUMBER,
          v_itemid   IN NUMBER,
          v_prodid    IN NUMBER,
          v_actualprice IN NUMBER,
          v_qty       IN NUMBER DEFAULT 1);
```

```
UPD_ITEM (v_ordid    IN NUMBER,
          v_itemid   IN NUMBER,
          v_qty       IN NUMBER DEFAULT 1);
```

Use these two procedures to alter the customer's order.

```
SQL> EXECUTE ITEM_PACK.ADD_ITEM (610, 4, 102130, 3.4,1)
PL/SQL procedure successfully completed.

SQL> EXECUTE ITEM_PACK.UPD_ITEM (610, 2, -1)
PL/SQL procedure successfully completed.
```

- d. After modifying order 601, verify that WARD's commission has increased by 0.03. The original commission was 500.

```
SQL> SELECT ename, comm FROM emp WHERE ename = 'WARD';
ENAME          COMM
-----
WARD           500.03
```

Practice 7 Solutions (continued)

4. A number of business rules applies to the EMP and DEPT tables. A partial package is provided in file *labs\p7_4.sql* to which you should add any necessary procedures/functions that are to be called from triggers you may create for the following elements.
 - a. Decide how to implement each rule: by means of declarative constraints or using triggers.
Which constraints or triggers are needed and are there any problems to be expected?
 - b. Implement the business rules guarded by triggers.

Business Rules

1. Sales persons should always receive commission. Employees who are not sales persons should never receive a commission.
 - ◆ Implement rule 1 with a constraint.

```
ALTER TABLE emp
ADD CONSTRAINT emp_comm_chk CHECK ((job = 'SALESMAN' and
comm IS NOT NULL) OR (job != 'SALESMAN' and comm IS NULL));
```

2. The EMP table should contain exactly one PRESIDENT. Test your answer.

```
CREATE OR REPLACE TRIGGER check_pres_title
  BEFORE INSERT OR UPDATE OF job ON emp
BEGIN
  mgr_constraints_pkg.check_president;
END check_pres_title;
/

SQL> insert into emp (empno, ename, job, deptno)
  2 values (7800, 'HARRIS', 'PRESIDENT', 20);
insert into emp (empno, ename, job, deptno)
*
ERROR at line 1:
ORA-20001: President title already exists
ORA-06512: at "SCOTT.MGR_CONSTRAINTS_PKG", line 55
ORA-06512: at "SCOTT.CHECK_PRES_TITLE", line 2
ORA-04088: error during execution of trigger
'SCOTT.CHECK_PRES_TITLE'
```

Practice 7 Solutions (continued)

3. An employee should never be manager of more than five employees. Test your answer.

```
CREATE OR REPLACE TRIGGER check_max_mgr
  AFTER INSERT OR UPDATE OF mgr ON emp
BEGIN
  mgr_constraints_pkg.check_mgr;
END check_max_mgr;
/
SQL> insert into emp (empno, ename, job, mgr, deptno)
  2  values (7800, 'HARRIS', 'CLERK', 7698, 20);
insert into emp (empno, ename, job, mgr, deptno)
*
ERROR at line 1:
ORA-20000: Max number of emps exceeded for 7698
ORA-06512: at "SCOTT.MGR_CONSTRAINTS_PKG", line 45
ORA-06512: at "SCOTT.CHECK_MAX_MGR", line 2
ORA-04088: error during execution of trigger 'SCOTT.CHECK_MAX_MGR'
```

4. Salaries may only be increased, never decreased. Test your answer.

```
CREATE OR REPLACE TRIGGER check_sal
  BEFORE UPDATE OF sal ON emp
  FOR EACH ROW
  WHEN (new.sal < old.sal)
BEGIN
  RAISE_APPLICATION_ERROR(-20002, 'Salary may not be reduced');
END check_sal;
/
SQL> update emp
  2  set      sal = 1100
  3  where empno = 7934;
update emp
*
ERROR at line 1:
ORA-20002: Salary may not be reduced
ORA-06512: at "SCOTT.CHECK_SAL", line 2
ORA-04088: error during execution of trigger 'SCOTT.CHECK_SAL'
```

Practice 7 Solutions (continued)

5. If a department moves to another location, each employee of that department automatically receives a salary raise of 2%.

```
CREATE OR REPLACE TRIGGER change_location
BEFORE UPDATE OF loc ON dept
FOR EACH ROW
BEGIN
    mgr_constraints_pkg.new_location(:old.deptno);
END change_location;
/
```

```
SQL> select ename, sal, deptno
2   from emp
3  where deptno = 30;
```

ENAME	SAL	DEPTNO
-----	-----	-----
BLAKE	2850	30
MARTIN	1250	30
ALLEN	1600	30
...		

```
SQL> update dept
2   set loc = 'HOUSTON'
3  where deptno = 30;
SQL> select ename, sal, deptno
2   from emp
3  where deptno = 30;
```

ENAME	SAL	DEPTNO
-----	-----	-----
BLAKE	2907	30
MARTIN	1275	30
ALLEN	1632	30
...		

Practice 8 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.
 - b. Assume the following scenario:
 - The standalone 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 specification is not altered in the package specification.
 - Is the standalone procedure MY_PROC invalidated?
 - ◆ Although the package body is recompiled, the standalone procedure MY_PROC that depends on the packaged procedure MY_PROC_PACK, is not invalidated because the package specification is not altered.
2. Suppose you have lost the code for the NEW_EMP procedure and the VALID_DEPTNO function you created in Lesson 4. Create a SQL*Plus spool file to query the appropriate Data Dictionary view to regenerate the code.

```
SET ECHO OFF HEADING OFF FEEDBACK OFF VERIFY OFF
COL DUMMY NOPRINT
COL LINE NOPRINT
SET PAGESIZE 0
SPOOL RECREATE.SQL
SELECT text,line
FROM USER_SOURCE
WHERE  name = '&the_object'
UNION
SELECT 'CREATE OR REPLACE ',1 dummy
FROM DUAL
ORDER BY line
/
SELECT '/'
FROM DUAL
/
SPOOL OFF
SET PAGESIZE 24
SET FEEDBACK ON VERIFY ON HEADING ON ECHO ON
```

Practice 8 Solutions (continued)

3. Execute the *utldtree.sql* script. Print a tree structure showing all dependencies involving your NEW_EMP procedure and your VALID_DEPTNO function. Query the ideptree view to see your results.

```
SQL> exec deptree_fill('PROCEDURE','SCOTT','NEW_EMP')
PL/SQL procedure successfully completed.
SQL> select * from ideptree;
DEPENDENCIES
-----
PROCEDURE SCOTT.NEW_EMP

SQL> exec deptree_fill('FUNCTION','SCOTT','VALID_DEPTNO')
PL/SQL procedure successfully completed.
SQL> select * from ideptree;
DEPENDENCIES
-----
FUNCTION SCOTT.VALID_DEPTNO
      PROCEDURE SCOTT.NEW_EMP
```

If you have time:

4. Dynamically validate invalid objects.
- Make a copy of your EMP table, called EMP_COP.
 - Alter your EMP table and add the column TOTSAL(NUMBER(9,2)).
 - Create a script file to print the name, type, and status of dependent objects.

```
CREATE TABLE emp_cop AS
SELECT * FROM emp;
```

```
ALTER TABLE emp
ADD (totsal NUMBER(9,2));
```

```
SELECT object_name, object_type, status
FROM user_objects
WHERE status = 'INVALID'
/
```

Practice 8 Solutions (continued)

- d. Create a procedure called `COMPILE_OBJ` that recompiles all invalid objects 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');
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;
/
```

- e. Run the previous script file again and check the status column value.

Practice 9 Solutions

1. Create an object type called BANK_ACCOUNT. The attributes are: account number, balance, and status. The methods are: open_acc, verify_acct, deposit, and withdraw.

Implementation requirements:

- The status can be "open" or "closed."
- Use the sequence ACCT_SEQ to generate account numbers.
- Open an account with an initial deposit.
- Check for wrong account number or closed account.
- Only allow withdrawal if the account has sufficient funds.

```
CREATE TYPE bank_account AS OBJECT
(
    acct_number INTEGER(8) ,
    balance NUMBER,
    status VARCHAR2(10) ,
    MEMBER PROCEDURE open_acct(amount IN REAL) ,
    MEMBER PROCEDURE verify_acct(num IN INTEGER) ,
    MEMBER PROCEDURE deposit(num IN, amount IN REAL)
    MEMBER PROCEDURE withdraw(num IN INTEGER, amount IN REAL) )
/
```

```
CREATE TYPE BODY bank_account AS
MEMBER PROCEDURE open_acct(amount IN REAL) IS
BEGIN
    IF NOT amount > 0 THEN
        RAISE_APPLICATION_ERROR(-20400,'Need real amount');
    END IF;
    SELECT acct_seq.NEXTVAL INTO acct_number FROM dual;
    status := 'open';
    balance := amount;
END open_acct;
MEMBER PROCEDURE verify_acct(num IN INTEGER) IS
BEGIN
    IF num != acct_number THEN
        RAISE_APPLICATION_ERROR(-20401,'Wrong acct number');
    ELSIF status = 'closed' THEN
        RAISE_APPLICATION_ERROR(-20402,'Acct closed');
    END IF;
END verify_acct;
```

Practice 9 Solutions (continued)

```
MEMBER PROCEDURE deposit(num IN INTEGER, amount IN REAL) IS
BEGIN
    verify_acct(num);
    IF NOT amount > 0 THEN
        RAISE_APPLICATION_ERROR(-20403,'Invalid amount');
    END IF;
    balance := balance + amount;
END deposit;
MEMBER PROCEDURE withdraw(num IN, amount IN REAL) IS
BEGIN
    verify_acct(num);
    IF amount <= balance THEN
        balance := balance - amount;
    ELSE
        RAISE_APPLICATION_ERROR(-20404,'Insufficient funds');
    END IF;
END withdraw;
END;
```

2. Create an object table ACCOUNT.

Add a few objects of type BANK_ACCOUNT to the table.

Update the balance for one of the accounts you created.

```
SQL> CREATE TABLE account OF bank_account;
SQL> INSERT INTO account
    2  VALUES (10283746, 4859.75, 'open');
SQL> UPDATE account
    2  SET status = 'hold'
    3  WHERE acct_number = 10283746;
```

Practice 10 Solutions

1. Create a table PERSONNEL using the following attributes and datatypes:

Column Name	Datatype	Length
id	number	6
last_name	varchar2	35
large	clob	n/a
picture	blob	n/a

```
SQL> CREATE TABLE personnel
2  (id NUMBER(6) constraint personnel_id_pk PRIMARY KEY,
3  last_name VARCHAR2 (35) ,
4  large CLOB,
5  picture BLOB)
6  /
```

2. Insert two records in the PERSONNEL table. Use the empty function for the CLOB, and provide NULL as the value for the BLOB.

```
SQL> INSERT INTO  personnel
2  VALUES (2034, 'Allen' ,empty_clob() ,NULL) ;

SQL> INSERT INTO  personnel
2  VALUES (2035, 'Bond' ,empty_clob() ,NULL) ;
```

3. Execute the script *labs\p10_3.sql*.
 - ◆ This script creates the res_table table. You will be prompted for two character strings.

Practice 10 Solutions (continued)

4. Update the PERSONNEL table.

- Populate the CLOB for the first record using the following query in a SQL UPDATE statement:

```
SQL> SELECT resume
      2 FROM res_table;
```

- Populate the CLOB for the second record using PL/SQL and the DBMS_LOB package.

```
SQL> UPDATE personnel
      2 SET large = (SELECT resume
      3                FROM   res_table
      4                WHERE  id = 1)
      5 WHERE last_name = 'Allen'
      6 /
```

```
DECLARE
  lobloc CLOB;
  text VARCHAR2(2000);
  amount NUMBER;
  offset INTEGER;
BEGIN
  SELECT resume INTO text
  FROM res_table
  WHERE id =2;
  SELECT large INTO lobloc
  FROM personnel
  WHERE last_name = 'Bond' FOR UPDATE;
  offset := 1;
  amount := length(text);
  DBMS_LOB.WRITE ( lobloc, amount, offset, text );
END;
/
```

C

Table Descriptions and Data

EMP Table

```
SQL> DESCRIBE emp
```

Name	Null?	Type
-----	-----	----
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
DEPTNO	NOT NULL	NUMBER(2)

```
SQL> SELECT * FROM emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----	-----
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

DEPT Table

```
SQL> DESCRIBE dept
```

Name	Null?	Type
-----	-----	----
DEPTNO	NOT NULL	NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

```
SQL> SELECT * FROM dept;
```

DEPTNO	DNAME	LOC
-----	-----	-----
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SALGRADE Table

```
SQL> DESCRIBE salgrade
```

Name	Null?	Type
GRADE		NUMBER
LOSAL		NUMBER
HISAL		NUMBER

```
SQL> SELECT * FROM salgrade;
```

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

ORD Table

```
SQL> DESCRIBE ord
```

Name	Null?	Type
-----	-----	----
ORDID	NOT NULL	NUMBER(4)
ORDERDATE		DATE
COMMPLAN		VARCHAR2(1)
CUSTID	NOT NULL	NUMBER(6)
SHIPDATE		DATE
TOTAL		NUMBER(8,2)

```
SQL> SELECT * FROM ord;
```

ORDID	ORDERDATE	C	CUSTID	SHIPDATE	TOTAL
-----	-----	-	-----	-----	-----
610	07-JAN-87	A	101	08-JAN-87	101.4
611	11-JAN-87	B	102	11-JAN-87	45
612	15-JAN-87	C	104	20-JAN-87	5860
601	01-MAY-86	A	106	30-MAY-86	2.4
602	05-JUN-86	B	102	20-JUN-86	56
604	15-JUN-86	A	106	30-JUN-86	698
605	14-JUL-86	A	106	30-JUL-86	8324
606	14-JUL-86	A	100	30-JUL-86	3.4
609	01-AUG-86	B	100	15-AUG-86	97.5
607	18-JUL-86	C	104	18-JUL-86	5.6
608	25-JUL-86	C	104	25-JUL-86	35.2
603	05-JUN-86		102	05-JUN-86	224
620	12-MAR-87		100	12-MAR-87	4450
613	01-FEB-87		108	01-FEB-87	6400
614	01-FEB-87		102	05-FEB-87	23940
616	03-FEB-87		103	10-FEB-87	764
619	22-FEB-87		104	04-FEB-87	1260
617	05-FEB-87		105	03-MAR-87	46370
615	01-FEB-87		107	06-FEB-87	710
618	15-FEB-87	A	102	06-MAR-87	3510.5
621	15-MAR-87	A	100	01-JAN-87	730

PRODUCT Table

```
SQL> DESCRIBE product
```

Name	Null?	Type
PRODID	NOT NULL	NUMBER(6)
DESCRIP		VARCHAR2(30)

```
SQL> SELECT * FROM product;
```

PRODID	DESCRIP
100860	ACE TENNIS RACKET I
100861	ACE TENNIS RACKET II
100870	ACE TENNIS BALLS-3 PACK
100871	ACE TENNIS BALLS-6 PACK
100890	ACE TENNIS NET
101860	SP TENNIS RACKET
101863	SP JUNIOR RACKET
102130	RH: "GUIDE TO TENNIS"
200376	SB ENERGY BAR-6 PACK
200380	SB VITA SNACK-6 PACK

ITEM Table

```
SQL> DESCRIBE item
```

Name	Null?	Type
-----	-----	-----
ORDID	NOT NULL	NUMBER (4)
ITEMID	NOT NULL	NUMBER (4)
PRODID		NUMBER (6)
ACTUALPRICE		NUMBER (8, 2)
QTY		NUMBER (8)
ITEMTOT		NUMBER (8, 2)

```
SQL> SELECT * FROM item;
```

ORDID	ITEMID	PRODID	ACTUALPRICE	QTY	ITEMTOT
-----	-----	-----	-----	-----	-----
610	3	100890	58	1	58
611	1	100861	45	1	45
612	1	100860	30	100	3000
601	1	200376	2.4	1	2.4
602	1	100870	2.8	20	56
604	1	100890	58	3	174
604	2	100861	42	2	84
604	3	100860	44	10	440
603	2	100860	56	4	224
610	1	100860	35	1	35
610	2	100870	2.8	3	8.4
613	4	200376	2.2	200	440
614	1	100860	35	444	15540
614	2	100870	2.8	1000	2800
612	2	100861	40.5	20	810
612	3	101863	10	150	1500
620	1	100860	35	10	350
620	2	200376	2.4	1000	2400
620	3	102130	3.4	500	1700
613	1	100871	5.6	100	560
613	2	101860	24	200	4800
613	3	200380	4	150	600
619	3	102130	3.4	100	340
617	1	100860	35	50	1750
617	2	100861	45	100	4500
614	3	100871	5.6	1000	5600

Continued on next page

ITEM Table (continued)

ORDID	ITEMID	PRODID	ACTUALPRICE	QTY	ITEMTOT
616	1	100861	45	10	450
616	2	100870	2.8	50	140
616	3	100890	58	2	116
616	4	102130	3.4	10	34
616	5	200376	2.4	10	24
619	1	200380	4	100	400
619	2	200376	2.4	100	240
615	1	100861	45	4	180
607	1	100871	5.6	1	5.6
615	2	100870	2.8	100	280
617	3	100870	2.8	500	1400
617	4	100871	5.6	500	2800
617	5	100890	58	500	29000
617	6	101860	24	100	2400
617	7	101863	12.5	200	2500
617	8	102130	3.4	100	340
617	9	200376	2.4	200	480
617	10	200380	4	300	1200
609	2	100870	2.5	5	12.5
609	3	100890	50	1	50
618	1	100860	35	23	805
618	2	100861	45.11	50	2255.5
618	3	100870	45	10	450
621	1	100861	45	10	450
621	2	100870	2.8	100	280
615	3	100871	5	50	250
608	1	101860	24	1	24
608	2	100871	5.6	2	11.2
609	1	100861	35	1	35
606	1	102130	3.4	1	3.4
605	1	100861	45	100	4500
605	2	100870	2.8	500	1400
605	3	100890	58	5	290
605	4	101860	24	50	1200
605	5	101863	9	100	900
605	6	102130	3.4	10	34
612	4	100871	5.5	100	550
619	4	100871	5.6	50	280

CUSTOMER Table

```
SQL> DESCRIBE customer
```

Name	Null?	Type
-----	-----	----
CUSTID	NOT NULL	NUMBER (6)
NAME		VARCHAR2 (45)
ADDRESS		VARCHAR2 (40)
CITY		VARCHAR2 (30)
STATE		VARCHAR2 (2)
ZIP		VARCHAR2 (9)
AREA		NUMBER (3)
PHONE		VARCHAR2 (9)
REPID	NOT NULL	NUMBER (4)
CREDITLIMIT		NUMBER (9,2)
COMMENTS		LONG

CUSTOMER Table (continued)

```
SQL> SELECT * FROM customer;
```

CUSTID	NAME	ADDRESS
100	JOCKSPORTS	345 VIEWRIDGE
101	TKB SPORT SHOP	490 BOLI RD.
102	VOLLYRITE	9722 HAMILTON
103	JUST TENNIS	HILLVIEW MALL
104	EVERY MOUNTAIN	574 SURRY RD.
105	K + T SPORTS	3476 EL PASEO
106	SHAPE UP	908 SEQUOIA
107	WOMENS SPORTS	VALCO VILLAGE
108	NORTH WOODS HEALTH AND FITNESS SUPPLY CENTER	98 LONE PINE WAY

CITY	ST	ZIP	AREA	PHONE	REPID	CREDITLIMIT
BELMONT	CA	96711	415	598-6609	7844	5000
REDWOOD CITY	CA	94061	415	368-1223	7521	10000
BURLINGAME	CA	95133	415	644-3341	7654	7000
BURLINGAME	CA	97544	415	677-9312	7521	3000
CUPERTINO	CA	93301	408	996-2323	7499	10000
SANTA CLARA	CA	91003	408	376-9966	7844	5000
PALO ALTO	CA	94301	415	364-9777	7521	6000
SUNNYVALE	CA	93301	408	967-4398	7499	10000
HIBBING	MN	55649	612	566-9123	7844	8000

COMMENTS

Very friendly people to work with -- sales rep likes to be called Mike.
 Rep called 5/8 about change in order - contact shipping.
 Company doing heavy promotion beginning 10/89. Prepare for large orders during winter
 Contact rep about new line of tennis rackets.
 Customer with high market share (23%) due to aggressive advertising.
 Tends to order large amounts of merchandise at once. Accounting is considering raising their credit limit
 Support intensive. Orders small amounts (< 800) of merchandise at a time.
 First sporting goods store geared exclusively towards women. Unusual promotional style

PRICE Table

```
SQL> DESCRIBE price
```

Name	Null?	Type
PRODID	NOT NULL	NUMBER(6)
STDPRICE		NUMBER(8,2)
MINPRICE		NUMBER(8,2)
STARTDATE		DATE
ENDDATE		DATE

```
SQL> SELECT * FROM price;
```

PRODID	STDPRICE	MINPRICE	STARTDATE	ENDDATE
100871	4.8	3.2	01-JAN-85	01-DEC-85
100890	58	46.4	01-JAN-85	
100890	54	40.5	01-JUN-84	31-MAY-84
100860	35	28	01-JUN-86	
100860	32	25.6	01-JAN-86	31-MAY-86
100860	30	24	01-JAN-85	31-DEC-85
100861	45	36	01-JUN-86	
100861	42	33.6	01-JAN-86	31-MAY-86
100861	39	31.2	01-JAN-85	31-DEC-85
100870	2.8	2.4	01-JAN-86	
100870	2.4	1.9	01-JAN-85	01-DEC-85
100871	5.6	4.8	01-JAN-86	
101860	24	18	15-FEB-85	
101863	12.5	9.4	15-FEB-85	
102130	3.4	2.8	18-AUG-85	
200376	2.4	1.75	15-NOV-86	
200380	4	3.2	15-NOV-86	