



Integrated Cloud Applications & Platform Services



Oracle Database 12c R2: Develop PL/SQL Program Units

Student Guide - Volume I

D80170GC20

Edition 2.0 | April 2017 | D98644

Learn more from Oracle University at education.oracle.com

ORACLE®

Oracle Internal & Oracle Academy Only

Author

Jayashree Sharma

**Technical Contributors
and Reviewers**

Bryan Roberts
Miyuki Osato
Nancy Greenberg
Suresh Rajan
Drishya

Editors

Anwesha Ray
Raj Kumar
Kavita Saini

Publishers

Giri Venugopal
Sumesh Koshy
Veena Narasimhan

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

1 Introduction

- Lesson Objectives 1-2
- Lesson Agenda 1-3
- Course Objectives 1-4
- Course Road Map 1-5
- Lesson Agenda 1-8
- Human Resources (HR) Schema for This Course 1-9
- Course Agenda 1-10
- Class Account Information 1-12
- Appendices and Practices Used in This Course 1-13
- Lesson Agenda 1-14
- Oracle Database 12c: Focus Areas 1-15
- Oracle Database 12c 1-16
- Lesson Agenda 1-18
- PL/SQL Development Environments 1-19
- Oracle SQL Developer 1-20
- Specifications of SQL Developer 1-21
- SQL Developer 4.1.3 Interface 1-22
- Coding PL/SQL in SQL*Plus 1-23
- Lesson Agenda 1-24
- Oracle SQL and PL/SQL Documentation 1-25
- Additional Resources 1-26
- Summary 1-27
- Practice 1 Overview: Getting Started 1-28

2 Creating Procedures

- Course Road Map 2-2
- Objectives 2-3
- Lesson Agenda 2-4
- Modularized Program Design 2-5
- Modularizing Code with PL/SQL 2-6
- Benefits of Modularization 2-7
- What Are PL/SQL Subprograms? 2-8
- Lesson Agenda 2-9
- Procedures 2-10

What Are Procedures? 2-11
Creating Procedures: Overview 2-12
Creating Procedures 2-13
Creating Procedures Using SQL Developer 2-14
Compiling Procedures 2-15
Calling Procedures 2-16
Calling Procedures Using SQL Developer 2-17
Procedures 2-18
What Are Parameters and Parameter Modes? 2-19
Formal and Actual Parameters 2-20
Procedural Parameter Modes 2-21
Comparing the Parameter Modes 2-22
Using the IN Parameter Mode: Example 2-23
Using the OUT Parameter Mode: Example 2-24
Using the IN OUT Parameter Mode: Example 2-25
Passing Parameters to Procedures 2-26
Passing Actual Parameters: Creating the raise_sal Procedure 2-27
Passing Actual Parameters: Examples 2-28
Using the DEFAULT Option for the Parameters 2-29
Lesson Agenda 2-30
Handled Exceptions 2-31
Handled Exceptions: Example 2-32
Exceptions Not Handled 2-33
Exceptions Not Handled: Example 2-34
Removing Procedures: Using the DROP SQL Statement or SQL Developer 2-35
Viewing Procedure Information Using the Data Dictionary Views 2-36
Viewing Procedures Information Using SQL Developer 2-37
Quiz 2-38
Summary 2-39
Practice 2 Overview: Creating, Compiling, and Calling Procedures 2-40

3 Creating Functions

Course Road Map 3-2
Objectives 3-3
Lesson Agenda 3-4
Functions 3-5
Creating Functions syntax 3-6
Tax Calculation 3-8
The Difference Between Procedures and Functions 3-9
Creating Functions: Overview 3-10
Invoking a Stored Function: Example 3-11

Using Different Methods for Executing Functions	3-12
Creating and Compiling Functions Using SQL Developer	3-14
Lesson Agenda	3-15
Using a Function in a SQL Expression: Example	3-16
Calling User-Defined Functions in SQL Statements	3-17
Restrictions When Calling Functions from SQL Expressions	3-18
Side Effects of Function Execution	3-19
Controlling Side Effects	3-20
Guidelines to Control Side Effects	3-21
Lesson Agenda	3-22
Passing Parameters to Functions	3-23
Named and Mixed Notation from SQL: Example	3-24
Viewing Functions Using Data Dictionary Views	3-25
Viewing Functions Information Using SQL Developer	3-26
Lesson Agenda	3-27
Removing Functions: Using the DROP SQL Statement or SQL Developer	3-28
Quiz	3-29
Practice 3-1: Overview	3-30
Summary	3-31

4 Debugging Subprograms

Course Road Map	4-2
Objectives	4-3
Lesson Agenda	4-4
Before Debugging PL/SQL Subprograms	4-5
Lesson Agenda	4-9
Debugging a Subprogram: Overview	4-10
Lesson Agenda	4-12
The Debugging – Log Tab Toolbar	4-13
Tracking Data and Execution	4-15
Lesson Agenda	4-16
Debugging a Procedure Example: Creating a New emp_list Procedure	4-17
Debugging a Procedure Example: Creating a New get_location Function	4-18
Setting Breakpoints and Compiling emp_list for Debug Mode	4-19
Compiling the get_location Function for Debug Mode	4-20
Debugging emp_list and Entering Values for the PMAXROWS Parameter	4-21
Debugging emp_list: Step Into (F7) the Code	4-22
Viewing the Data	4-23
Modifying the Variables While Debugging the Code	4-24
Debugging emp_list: Step Over Versus Step Into	4-25
Debugging emp_list: Step Out of the Code (Shift + F7)	4-26

Debugging emp_list: Step to End of Method 4-27
Debugging a Subprogram Remotely: Overview 4-28
Summary 4-29
Practice 4 Overview: Introduction to the SQL Developer Debugger 4-30

5 Creating Packages

Course Road Map 5-2
Objectives 5-3
Lesson Agenda 5-4
DBMS_OUTPUT.PUT_LINE 5-5
What Is a Package? 5-6
Advantages of Packages 5-7
How Do You Create PL/SQL Packages? 5-8
Components of a PL/SQL Package 5-9
Application Program Interface 5-10
Lesson Agenda 5-11
Creating the Package Specification: Using the CREATE PACKAGE Statement 5-12
Creating Package Specification: Using SQL Developer 5-13
Creating the Package Body: Using SQL Developer 5-14
Example of a Package Specification: comm_pkg 5-15
Creating the Package Body 5-16
Example of a Package Body: comm_pkg 5-17
Invoking the Package Subprograms: Examples 5-18
Invoking Package Subprograms: Using SQL Developer 5-19
Creating and Using Bodiless Packages 5-20
Viewing Packages by Using the Data Dictionary 5-21
Viewing Packages by Using SQL Developer 5-22
Removing Packages 5-23
Removing Package Bodies 5-24
Guidelines for Writing Packages 5-25
Quiz 5-26
Summary 5-27
Practice 5 Overview: Creating and Using Packages 5-28

6 Working with Packages

Course Road Map 6-2
Objectives 6-3
Lesson Agenda 6-4
Why Overloading of Subprogram? 6-5
Overloading Subprograms in PL/SQL 6-6
Overloading Procedures Example: Creating the Package Specification 6-8

Overloading Procedures Example: Creating the Package Body	6-9
Restrictions on Overloading	6-10
STANDARD package	6-11
Overloading and the STANDARD Package	6-12
Lesson Agenda	6-13
Package Instantiation and Initialization	6-14
Initializing Packages in Package Body	6-15
Using User-Defined Package Functions in SQL	6-16
User-Defined Package Function in SQL: Example	6-17
Lesson Agenda	6-18
Package State	6-19
Seriously Reusable Packages	6-20
Memory Architecture	6-21
Seriously Reusable Packages	6-23
Persistent State of Packages	6-24
Persistent State of Package Variables: Example	6-25
Persistent State of a Package Cursor: Example	6-26
Executing the CURS_PKG Package	6-28
Quiz	6-29
Summary	6-30
Practice 6 Overview: Working with Packages	6-31

7 Using Oracle-Supplied Packages in Application Development

Course Road Map	7-2
Objectives	7-3
Lesson Agenda	7-4
Using Oracle-Supplied Packages	7-5
Examples of Some Oracle-Supplied Packages	7-6
Lesson Agenda	7-7
How the DBMS_OUTPUT Package Works	7-8
Using the UTL_FILE Package	7-9
Some of the UTL_FILE Procedures and Functions	7-10
File Processing Using the UTL_FILE Package: Overview	7-11
Using the Available Declared Exceptions in the UTL_FILE Package	7-12
FOPEN and IS_OPEN Functions: Example	7-13
Using UTL_FILE: Example	7-16
What Is the UTL_MAIL Package?	7-18
Setting Up and Using the UTL_MAIL: Overview	7-19
Summary of UTL_MAIL Subprograms	7-20
Installing and Using UTL_MAIL	7-21
The SEND Procedure Syntax	7-22

The SEND_ATTACH_RAW Procedure	7-23
Sending Email with a Binary Attachment: Example	7-24
The SEND_ATTACH_VARCHAR2 Procedure	7-26
Sending Email with a Text Attachment: Example	7-27
Quiz	7-29
Summary	7-30
Practice 7 Overview: Using Oracle-Supplied Packages in Application Development	7-31

8 Using Dynamic SQL

Objectives	8-2
Course Road Map	8-3
Lesson Agenda	8-4
What is Dynamic SQL?	8-5
When do you use Dynamic SQL?	8-6
Using Dynamic SQL	8-7
Execution Flow of SQL Statements	8-8
Dynamic SQL implementation	8-9
Lesson Agenda	8-10
Native Dynamic SQL (NDS)	8-11
Using the EXECUTE IMMEDIATE Statement	8-12
Dynamic SQL with a DDL Statement: Examples	8-13
Dynamic SQL with DML Statements	8-14
Dynamic SQL with a Single-Row Query: Example	8-15
Executing a PL/SQL Anonymous Block Dynamically	8-16
BULK COLLECT INTO clause	8-17
OPEN FOR clause	8-18
Using BULK COLLECT and OPEN FOR clause	8-19
Summarizing Methods for Using Dynamic SQL	8-20
Lesson Agenda	8-22
Using the DBMS_SQL Package	8-25
Using the DBMS_SQL Package Subprograms	8-26
Using DBMS_SQL with a DML Statement: Deleting Rows	8-27
Using DBMS_SQL with a Parameterized DML Statement	8-29
Quiz	8-30
Summary	8-31
Practice 8 Overview: Using Dynamic SQL	8-32

9 Creating Triggers

Objectives	9-2
Course Road Map	9-3

Lesson Agenda	9-4
What are Triggers?	9-6
Defining Triggers	9-7
Why do you use Triggers?	9-8
Trigger Event Types	9-9
Available Trigger Types	9-10
Trigger Event Types and Body	9-11
Lesson Agenda	9-12
Creating DML Triggers by Using the CREATE TRIGGER Statement	9-13
Creating DML Triggers by Using SQL Developer	9-14
Specifying the Trigger Execution Time	9-15
Creating a DML Statement Trigger Example: SECURE_EMP	9-16
Testing Trigger SECURE_EMP	9-17
Using Conditional Predicates	9-18
Multiple Triggers of the Same Type	9-19
CALL Statements in Triggers	9-20
Lesson Agenda	9-21
Statement-Level Triggers Versus Row-Level Triggers	9-22
Creating a DML Row Trigger	9-24
Correlation names and Pseudorecords	9-25
Correlation Names and Pseudorecords	9-26
Using OLD and NEW Qualifiers	9-27
Using OLD and NEW Qualifiers: Example	9-28
Using the WHEN Clause to Fire a Row Trigger Based on a Condition	9-30
Trigger-Firing Sequence: Single-Row Manipulation	9-31
Trigger-Firing Sequence: Multirow Manipulation	9-32
Summary of the Trigger Execution Model	9-33
Lesson Agenda	9-34
INSTEAD OF Triggers	9-35
Creating an INSTEAD OF Trigger: Example	9-36
Creating an INSTEAD OF Trigger to Perform DML on Complex Views	9-37
Lesson Agenda	9-39
The Status of a Trigger	9-40
System Privileges Required to Manage Triggers	9-41
Managing Triggers by Using the ALTER and DROP SQL Statements	9-42
Managing Triggers by Using SQL Developer	9-43
Viewing Trigger Information	9-44
Using USER_TRIGGERS	9-45
Testing Triggers	9-46

Quiz 9-47

Summary 9-48

Practice 9 Overview: Creating Statement and Row Triggers 9-49

10 Creating Compound, DDL, and Event Database Triggers

Objectives 10-2

Course Road Map 10-3

Lesson Agenda 10-4

What is a Compound Trigger? 10-5

Working with Compound Triggers 10-6

Why Compound Triggers? 10-7

Compound Trigger Structure 10-8

Compound Trigger Structure for Views 10-9

Compound Trigger Restrictions 10-10

Lesson Agenda 10-11

Mutating Tables 10-12

Mutating Table: Example 10-13

Using a Compound Trigger to Resolve the Mutating Table Error 10-15

Lesson Agenda 10-18

Creating Triggers on DDL Statements 10-19

Creating Triggers on DDL Statements -Example 10-20

Lesson Agenda 10-21

Creating Database Triggers 10-22

Creating Triggers on System Events 10-23

LOGON and LOGOFF Triggers: Example 10-24

Lesson Agenda 10-25

Guidelines for Designing Triggers 10-26

Quiz 10-27

Summary 10-28

Practice 10 Overview: Creating Compound, DDL, and Event Database

Triggers 10-29

11 Design Considerations for the PL/SQL Code

Objectives 11-2

Course Road Map 11-3

Lesson Agenda 11-4

Standardizing Constants and Exceptions 11-5

Standardizing Exceptions 11-6

Standardizing Exception Handling 11-7

Standardizing Constants 11-8

Local Subprograms 11-9

Lesson Agenda	11-10
Definer's and Invoker's Rights	11-11
Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER	11-12
Granting Privileges to Invoker's Rights Unit	11-13
Lesson Agenda	11-14
Autonomous Transactions	11-15
Features of Autonomous Transactions	11-16
Using Autonomous Transactions: Example	11-17
Lesson Agenda	11-19
Using the NOCOPY Hint	11-20
Effects of the NOCOPY Hint	11-21
When Does the PL/SQL Compiler Ignore the NOCOPY Hint?	11-22
Using the PARALLEL_ENABLE Hint	11-23
Using the Cross-Session PL/SQL Function Result Cache	11-24
Declaring and Defining a Result-Cached Function: Example	11-25
Using the DETERMINISTIC Clause with Functions	11-26
Using the RETURNING Clause	11-27
Lesson Agenda	11-28
Using Bulk Binding	11-29
Bulk Binding: Syntax and Keywords	11-30
Bulk Binding FORALL: Example	11-31
Using BULK COLLECT INTO with Queries	11-34
Using BULK COLLECT INTO with Cursors	11-35
Using BULK COLLECT INTO with a RETURNING Clause	11-36
Quiz	11-37
Summary	11-38
Practice 11 Overview: Design Considerations for PL/SQL Code	11-39

12 Tuning the PL/SQL Compiler

Objectives	12-2
Course Road Map	12-3
Lesson Agenda	12-4
Optimizing PL/SQL Compiler Performance	12-5
Initialization Parameters for PL/SQL Compilation	12-6
Using the Initialization Parameters for PL/SQL Compilation	12-8
Displaying the PL/SQL Initialization Parameters	12-9
Displaying and Setting PL/SQL Initialization Parameters	12-10
Changing PL/SQL Initialization Parameters: Example	12-11
Lesson Agenda	12-12
PL/SQL Compile-Time Warnings	12-13
Benefits of Compiler Warnings	12-14

Categories of PL/SQL Compile-Time Warning Messages	12-15
Enabling Warning Messages	12-16
Setting Compiler Warning Levels: Using PLSQL_WARNINGS, Examples	12-17
Enabling Compiler Warnings: Using PLSQL_WARNINGS in SQL Developer	12-18
Viewing the Current Setting of PLSQL_WARNINGS	12-19
Viewing Compiler Warnings	12-20
SQL*Plus Warning Messages: Example	12-21
Defining PLSQL_WARNINGS for Program Units	12-22
Lesson Agenda	12-23
Using the DBMS_WARNINGS Package	12-24
Using the DBMS_WARNING Package Subprograms	12-25
The DBMS_WARNING Procedures: Syntax, Parameters, and Allowed Values	12-26
The DBMS_WARNING Procedures: Example	12-27
The DBMS_WARNING Functions: Syntax, Parameters, and Allowed Values	12-28
The DBMS_WARNING Functions: Example	12-29
Using DBMS_WARNING: Example	12-30
Quiz	12-32
Summary	12-33
Practice 12 Overview: Tuning PL/SQL Compiler	12-34

13 Managing Dependencies

Objectives	13-2
Course Road Map	13-3
Lesson Agenda	13-4
What are Dependencies in a Schema?	13-5
How Dependencies Work?	13-6
Dependent and Referenced Objects	13-8
Querying Object Dependencies: Using the USER_DEPENDENCIES View	13-10
Querying an Object's Status	13-11
Categorizing Dependencies	13-12
Lesson Agenda	13-13
Direct Dependencies	13-14
Indirect Dependencies	13-15
Displaying Direct and Indirect Dependencies	13-16
Lesson Agenda	13-18
Fine-Grained Dependency Management	13-19
Fine-Grained Dependency Management: Example 1	13-21
Fine-Grained Dependency Management: Example 2	13-23
Guidelines for Reducing Invalidation	13-24
Object Revalidation	13-25

Lesson Agenda	13-26
Remote Dependencies	13-27
Managing Remote Procedure Dependencies	13-28
Setting the REMOTE_DEPENDENCIES_MODE Parameter	13-29
Timestamp Checking	13-30
Signature Checking	13-35
Lesson Agenda	13-36
Revalidating PL/SQL Program Units	13-37
Unsuccessful Recompilation	13-38
Successful Recompilation	13-39
Recompiling Procedures	13-40
Lesson Agenda	13-41
Packages and Dependencies: Subprogram References the Package	13-42
Packages and Dependencies: Package Subprogram References Procedure	13-43
Quiz	13-44
Summary	13-45
Practice 13 Overview: Managing Dependencies in Your Schema	13-46

14 Oracle Cloud Overview

Lesson Objectives	14-2
Lesson Agenda	14-3
Introduction to Oracle Cloud	14-4
Oracle Cloud Services	14-5
Cloud Deployment Models	14-6
Lesson Agenda	14-7
Evolving from On-premises to Exadata Express	14-8
What is in Exadata Express?	14-9
Exadata Express for Users	14-10
Exadata Express for Developers	14-11
Getting Started with Exadata Express	14-12
Oracle Exadata Express Cloud Service	14-13
Getting Started with Exadata Express	14-14
Managing Exadata	14-15
Service Console	14-16
Web Access through Service Console	14-17
Client Access Configuration through Service Console	14-18
Database Administration through Service Console	14-19
SQL Workshop	14-20
Connecting through Database Clients	14-22
Enabling SQL*Net Access for Client Applications	14-23
Downloading Client Credentials	14-24

Connecting Oracle SQL Developer 14-25
Connecting Oracle SQLcl 14-26
Summary 14-27

A Commonly Used SQL Commands

Objectives A-2
Basic SELECT Statement A-3
SELECT Statement A-4
WHERE Clause A-5
ORDER BY Clause A-6
GROUP BY Clause A-7
Data Definition Language A-8
CREATE TABLE Statement A-9
ALTER TABLE Statement A-10
DROP TABLE Statement A-11
GRANT Statement A-12
Privilege Types A-13
REVOKE Statement A-14
TRUNCATE TABLE Statement A-15
Data Manipulation Language A-16
INSERT Statement A-17
UPDATE Statement Syntax A-18
DELETE Statement A-19
Transaction Control Statements A-20
COMMIT Statement A-21
ROLLBACK Statement A-22
SAVEPOINT Statement A-23
Joins A-24
Types of Joins A-25
Qualifying Ambiguous Column Names A-26
Natural Join A-27
Equijoin A-28
Retrieving Records by Using Equijoins A-29
Adding Search Conditions by Using the AND and WHERE Operators A-30
Retrieving Records with Nonequijoins A-31
Retrieving Records by Using the USING Clause A-32
Retrieving Records by Using the ON Clause A-33
Left Outer Join A-34
Right Outer Join A-35
Full Outer Join A-36
Self-Join: Example A-37

Cross Join A-38

Summary A-39

B Managing PL/SQL Code

Objectives B-2

Agenda B-3

Conditional Compilation B-4

How Does Conditional Compilation Work? B-5

Using Selection Directives B-6

Using Predefined and User-Defined Inquiry Directives B-7

The PLSQL_CCFLAGS Parameter and the Inquiry Directive B-8

Displaying the PLSQL_CCFLAGS Initialization Parameter Setting B-9

The PLSQL_CCFLAGS Parameter and the Inquiry Directive: Example B-10

Using Conditional Compilation Error Directives to Raise User-Defined Errors B-11

Using Static Expressions with Conditional Compilation B-12

DBMS_DB_VERSION Package: Boolean Constants B-13

DBMS_DB_VERSION Package Constants B-14

Using Conditional Compilation with Database Versions: Example B-15

Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text B-17

Agenda B-18

Obfuscation B-19

Benefits of Obfuscating B-20

What's New in Dynamic Obfuscating Since Oracle 10g? B-21

Nonobfuscated PL/SQL Code: Example B-22

Obfuscated PL/SQL Code: Example B-23

Dynamic Obfuscation: Example B-24

PL/SQL Wrapper Utility B-25

Running the PL/SQL Wrapper Utility B-26

Results of Wrapping B-27

Guidelines for Wrapping B-28

DBMS_DDL Package Versus wrap Utility B-29

Summary B-30

C Implementing Triggers

Objectives C-2

Controlling Security Within the Server C-3

Controlling Security with a Database Trigger C-4

Enforcing Data Integrity Within the Server C-5

Protecting Data Integrity with a Trigger C-6

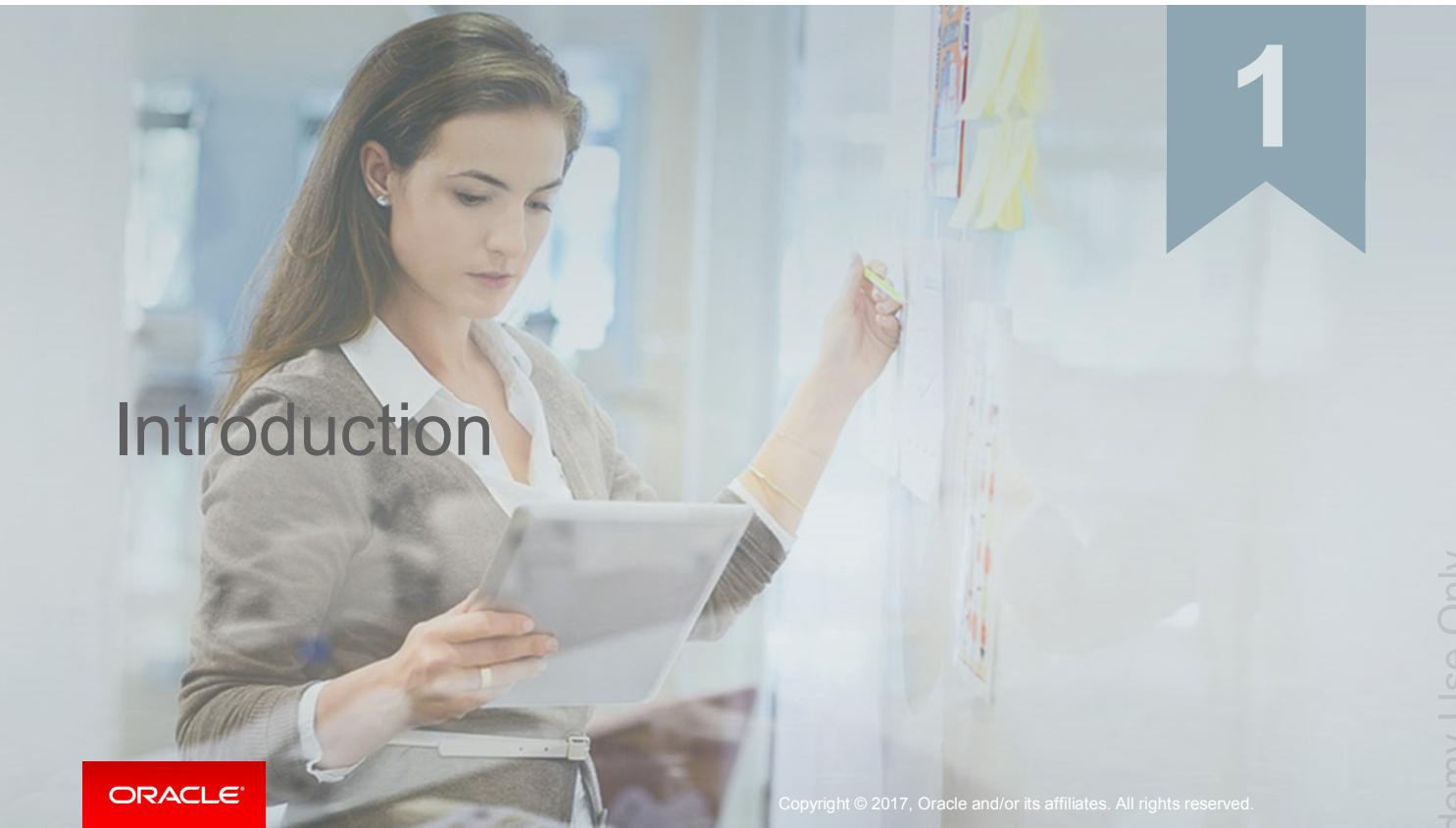
Enforcing Referential Integrity Within the Server C-7

Protecting Referential Integrity with a Trigger C-8

Replicating a Table Within the Server	C-9
Replicating a Table with a Trigger	C-10
Computing Derived Data Within the Server	C-11
Computing Derived Values with a Trigger	C-12
Logging Events with a Trigger	C-13
Summary	C-15

D Using the DBMS_SCHEDULER and HTP Packages

Objectives	D-2
Generating Webpages with the HTP Package	D-3
Using the HTP Package Procedures	D-4
Creating an HTML File with SQL*Plus	D-5
The DBMS_SCHEDULER Package	D-6
Creating a Job	D-8
Creating a Job with Inline Parameters	D-9
Creating a Job Using a Program	D-10
Creating a Job for a Program with Arguments	D-11
Creating a Job Using a Schedule	D-12
Setting the Repeat Interval for a Job	D-13
Creating a Job Using a Named Program and Schedule	D-14
Managing Jobs	D-15
Data Dictionary Views	D-16
Summary	D-17



1

Introduction

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Lesson Objectives

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

- Discuss the goals of the course
- Describe the HR database schema that is used in this course
- Identify the available user interface environments that can be used in this course
- Reference the available appendices, documentation, and other resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This lesson gives you a high-level overview of the course and its flow. You learn about the database schema and the tables that the course uses. The course discusses the fundamentals of PL/SQL. You learn how to write PL/SQL code blocks, and are introduced to the concepts of procedures and functions.

You are also introduced to a programming tool, SQL Developer 4.1.3.

Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Objectives

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

- Identify the programming extensions that PL/SQL provides to SQL
- Write PL/SQL code to interface with the database
- Design PL/SQL anonymous blocks that execute efficiently
- Use PL/SQL programming constructs and conditional control statements
- Handle runtime errors
- Describe stored procedures and functions



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This course presents the basics of PL/SQL. You learn about PL/SQL syntax, blocks, and programming constructs, and also about the advantages of integrating SQL with those constructs. You learn how to write PL/SQL program units and execute them efficiently. In addition, you learn how to use SQL Developer as a development environment for PL/SQL. You also learn how to design reusable program units such as procedures and functions.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL code

▶ Lesson 2: Creating Stored Procedures

▶ Lesson 3: Creating Functions

▶ Lesson 4: Debugging Subprograms

▶ Lesson 5: Creating Packages

▶ Lesson 6: Working with Packages

▶ Lesson 7: Using Oracle Supplied Packages in Application Development

▶ Lesson 8: Using Dynamic SQL



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Introducing PL/SQL, Programming with PL/SQL, and Working with PL/SQL Code.

In the first unit, we have four lessons:

1. Introduction to PL/SQL
2. Declaring PL/SQL Variables
3. Writing Anonymous PL/SQL Blocks
4. Using SQL Statements in PL/SQL Blocks.

You learn to write code in PL/SQL and define some anonymous PL/SQL blocks in this unit.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

▶ Lesson 9: Creating Triggers

▶ Lesson 10: Creating Compound DDL and Event Based Triggers

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 2, you learn to program with PL/SQL by using control structures, composite data types, and explicit cursors.

Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

Unit 3: Working with PL/SQL Code

▷ Lesson 11: Design Considerations for PL/SQL Code

▷ Lesson 12: Using PL/SQL Compiler

▷ Lesson 13: Managing Dependencies

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 3, you learn how to handle exceptions that might occur during the execution of PL/SQL blocks. You are also introduced to the concept of procedures and functions in PL/SQL.

Lesson Agenda

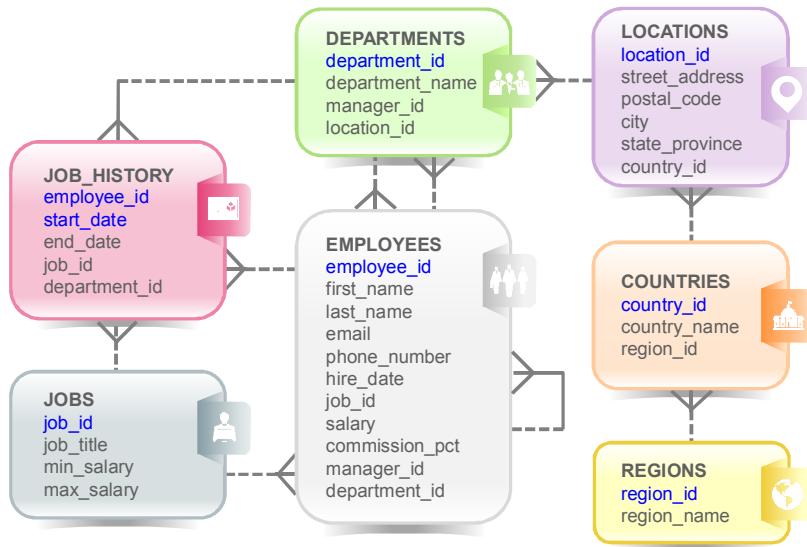
- Course objectives and course agenda
- The schema and appendices used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Human Resources (HR) Schema for This Course



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The Human Resources (HR) schema is part of the Oracle Sample Schemas that can be installed in an Oracle database. You will use the HR schema in all the practice lessons of the course. The fields marked in blue are the primary key attributes for the respective entities.

A description of each of the tables in the HR schema is as follows:

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

Course Agenda

- Day 1:
 1. Introduction
 2. Creating Procedures
 3. Creating Functions
 4. Debugging Subprograms
 5. Creating Packages
- Day 2:
 1. Working with Packages
 2. Using Oracle Supplied Packages in Application Development
 3. Using Dynamic SQL
 4. Creating Triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Agenda

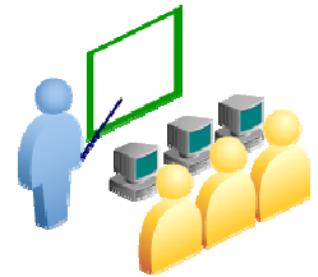
- Day 3:
 1. Creating Compound, DDL and Event Database Triggers
 2. Design Considerations for PL/SQL Code
 3. Tuning PL/SQL Compiler
 4. Managing Dependencies



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Class Account Information

- A cloned HR account ID is set up for you.
- Your account ID is ora61.
- The password matches your account ID.
- Each machine has its own complete environment, and is assigned the same account.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Appendices and Practices Used in This Course

- Appendix A: Commonly Used SQL Commands
- Appendix B: Managing PL/SQL Code
- Appendix C: Implementing Triggers
- Appendix D: Using DBMS_SCHEDULER and HTP Packages
- Activity Guide: Practices and Solutions



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

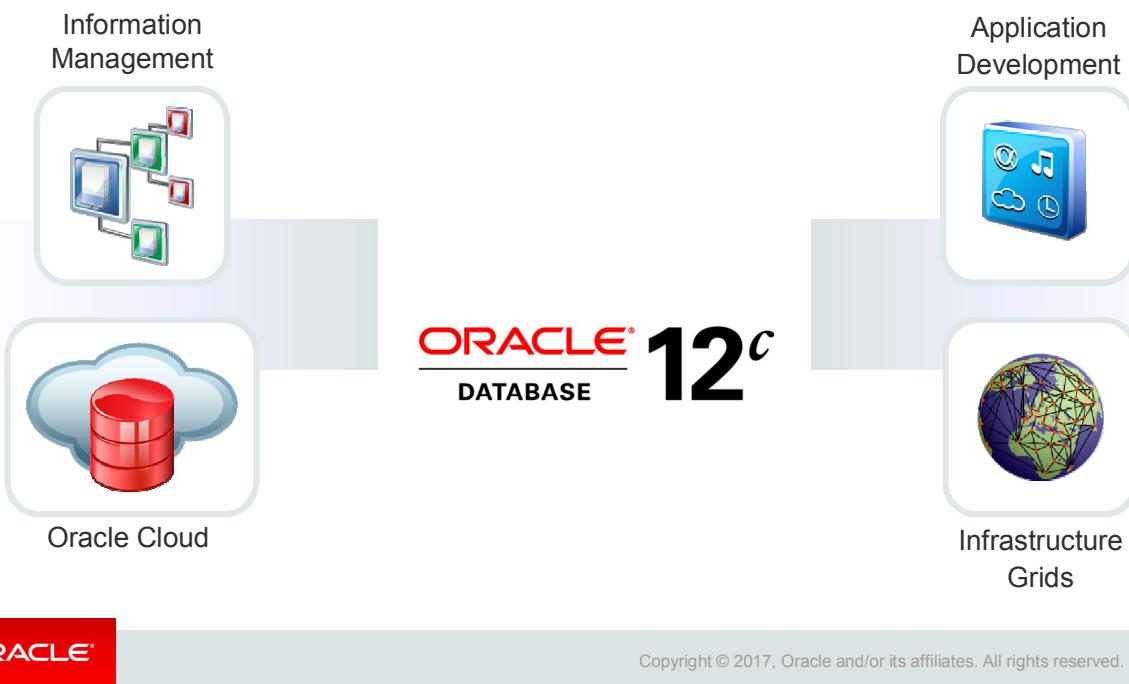
- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Oracle Database 12c: Focus Areas



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

By using Oracle Database 12c, you can utilize the following features across focus areas:

- With the **Infrastructure Grid** technology of Oracle, you can pool low-cost servers and storage to form systems that deliver the highest quality of service in terms of manageability, high availability, and performance. Oracle Database 12c also helps you to consolidate and extend the benefits of grid computing and manage changes in a controlled and cost-effective manner.
- Oracle Database 12c enables **Information Management** by providing capabilities in content management, information integration, and information lifecycle management areas. You can manage content of advanced data types such as Extensible Markup Language (XML), text, spatial, multimedia, medical imaging, and semantic technologies using the features provided by Oracle.
- With Oracle Database 12c, you can manage all the major **Application Development** environments such as PL/SQL, Java/JDBC, .NET, Windows, PHP, SQL Developer, and Application Express.
- You can now plug into **Oracle Cloud** with Oracle Database 12c. This will help you to standardize, consolidate, and automate database services on the cloud.

Oracle Database 12c



High Availability

ORACLE[®] DATABASE 12^c

Manageability



Performance



Security



Information Integration

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Imagine you have an organization that needs to support multiple terabytes of information for users who demand fast and secure access to business applications round the clock. The database systems must be reliable and must be able to recover quickly in the event of any kind of failure. Oracle Database 12c is designed to help organizations manage infrastructure grids easily and deliver high-quality service:

- **Manageability:** By using some of the change assurance, management automation, and fault diagnostics features, the database administrators (DBAs) can increase their productivity, reduce costs, minimize errors, and maximize quality of service. Some of the useful features that promote better management are the Database Replay facility, the SQL Performance Analyzer, the Automatic SQL Tuning facility, and Real-Time Database Operations Monitoring.

Enterprise Manager Database Express 12c is a web-based tool for managing Oracle databases. It greatly simplifies database performance diagnostics by consolidating the relevant database performance screens into a view called Database Performance Hub. DBAs get a single, consolidated view of the current real-time and historical view of the database performance across multiple dimensions such as database load, monitored SQL and PL/SQL, and Active Session History (ASH) on a single page for the selected time period.

- **High availability:** By using the high availability features, you can reduce the risk of down time and data loss. These features improve online operations and enable faster database upgrades.
- **Performance:** By using capabilities such as SecureFiles, Result Caches, and so on, you can greatly improve the performance of your database. Oracle Database 12c enables organizations to manage large, scalable, transactional, and data warehousing systems that deliver fast data access using low-cost modular storage.
- **Security:** Oracle Database 12c helps in protecting your information with unique secure configurations, data encryption and masking, and sophisticated auditing capabilities.
- **Information integration:** You can utilize Oracle Database 12c features to integrate data throughout the enterprise in a better way. You can also manage the changing data in your database by using Oracle Database 12c's advanced information lifecycle management capabilities.

Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL Development Environments

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

- Oracle SQL Developer (used in this course)
- Oracle SQL*Plus



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use various tools provided by Oracle to write PL/SQL code. In this course, we primarily focus on SQL Developer. We also look at the usage of SQL*Plus.

- **Oracle SQL Developer** is an integrated development environment, which allows database users and administrators to perform their database tasks in fewer clicks and keystrokes. For developers, it provides powerful editors for working with SQL, PL/SQL, stored Java procedures, and XML. You can run queries, generate execution plans, export data to different formats such as XML and HTML, and so on.
- **Oracle SQL*Plus:** SQL*Plus is an interactive and batch query tool that is installed with every installation of Oracle Database. It has a command-line user interface, a Windows Graphical User Interface (GUI), and the *i*SQL*Plus web-based user interface. SQL*Plus has its own commands and environment, and it provides access to Oracle Database.

You can use SQL*Plus to generate reports interactively or as batch processes, and to output the results to a text file, a screen, or to an HTML file for browsing on the Internet. You can generate reports dynamically by using the HTML output facility of SQL*Plus, or by using the dynamic reporting capability of *i*SQL*Plus to run a script from a web page.

Note: The code and screen examples presented in the course notes were generated from the output in the SQL Developer environment.

Oracle SQL Developer

- Oracle SQL Developer is a free graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.
- You use SQL Developer in this course.



SQL Developer

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

What is Oracle SQL Developer?

Oracle SQL Developer is a free graphical tool that is designed to improve productivity and simplify the performance of everyday database tasks. The user interface of SQL Developer allows you to browse, create, and manage database objects.

Using SQL Developer, you can easily create and maintain stored procedures, test SQL statements, and view optimizer plans, with just a few clicks.

SQL Developer, the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When you are connected, you can perform operations on the objects in the database.

Specifications of SQL Developer

- Is developed in Java
- Supports the Windows, Linux, and Mac OS X platforms
- Enables default connectivity by using the JDBC Thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Connects to Oracle Database on Cloud also



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Oracle SQL Developer is developed in Java, leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on the Windows, Linux, and Mac operating system (OS) X platforms.

Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver; therefore, no Oracle Home is required. SQL Developer does not require an installer. All you need to do is simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions, including Express Edition. SQL Developer allows you to connect to Oracle Database Service on cloud also.

SQL Developer 4.1.3 Interface



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The SQL Developer interface contains three main navigation tabs, from left to right:

- **Connections tab:** By using this tab, you can browse database objects and users to which you have access.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.
- **Start page:** The Start page gives you some links that are helpful when you use SQL Developer to create applications.

General Navigation and Use

SQL Developer uses the left pane for navigation to find and select objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures and functions. You can start the connection creation wizard by clicking on “+” on the Connections tab.

Coding PL/SQL in SQL*Plus



A screenshot of the Oracle SQL*Plus command-line interface. A red arrow points from a small icon of a smartphone displaying 'SQL> ...' towards the terminal window. The terminal window shows the following session:

```
oracle@EDRSR9P1:~/Desktop
File Edit View Search Terminal Help
Copyright (c) 1982, 2012, Oracle. All rights reserved.

Enter user-name: ora41
Enter password:
Last Successful login time: Mon Sep 2012 21:55:44 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

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

Procedure created.

SQL> execute hello
Hello Class!

PL/SQL procedure successfully completed.

SQL>
```

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

SQL*Plus is:

- Shipped with the database
- Installed on a client and on the database server system
- Accessed by using an icon or the command line

When you code PL/SQL subprograms by using SQL*Plus, remember the following:

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

Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Oracle SQL and PL/SQL Documentation

- *Oracle Database PL/SQL Language Reference*
- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SQL Developer User's Guide*
- *Oracle Database 2 Day Developer's Guide*
- *Getting Started with Oracle Cloud*



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Additional Resources

For additional information about the new Oracle SQL and PL/SQL new features, refer to the following:

- Oracle Database: New Features self study
- What's New in PL/SQL in Oracle Database on the Oracle Technology Network (OTN):
 - <http://www.oracle.com/technetwork/database/features/plsql/index.html>
- The online SQL Developer Home page and the SQL Developer tutorial available at:
 - http://www.oracle.com/technology/products/database/sql_developer/index.html
 - <http://download.oracle.com/oll/tutorials/SQLDeveloper/index.htm>



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Discuss the goals of the course
- Describe the HR database schema that is used in the course
- Identify the available user interface environments that can be used in this course
- Reference the available appendixes, documentation, and other resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Practice 1 Overview: Getting Started

This practice covers the following topics:

- Starting SQL Developer
- Creating a new database connection
- Browsing the `HR` schema tables
- Setting a SQL Developer preference



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this practice, you use SQL Developer to execute SQL statements to examine data in the `HR` schema. You also create a simple anonymous block.

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus environment that is available in this course.



2

Creating Procedures

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

▶ Lesson 2: Creating Procedures

▶ Lesson 3: Creating Functions

▶ Lesson 4: Debugging Subprograms

▶ Lesson 5: Creating Packages

▶ Lesson 6: Working with Packages

▶ Lesson 7: Using Oracle-Supplied Packages in Application Development

▶ Lesson 8: Using Dynamic SQL

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Working with Subprograms, Working with Triggers, and Working with PL/SQL Code.

In the first unit, we have six lessons: Creating Stored Procedures, Creating Functions, Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms, packages and use them in Application Development.

Objectives

After completing this lesson, you should be able to:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure and display its information



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

Lesson Agenda

- Modularizing code with PL/SQL
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedure's information



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Modularized Program Design



Hey Alice, I need your help in updating the database with new organizational changes



Sure tell me your requirements



I need a program which can help me change the salary of the employee, change the job_id and promote the employee

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Here the HR manager comes up with a requirement for which Alice has to write a code. The HR manager wants to put the organizational changes in the database. The organizational changes are:

- **Salary Hike** – Employees get a salary hike but practically every employee wouldn't get a salary hike and even when they get the amount of hike might not be uniform.
- **Promotion** – Employees got a promotion but practically every employee wouldn't get a promotion.

However it is clear that these two functions on the database are independent of each other, because an employee who gets a hike may not get a promotion and vice versa may also be true.

Therefore it would be efficient if Alice implement them as independent modules. When implemented as two independent functions they can be used together also if there's a requirement, a situation where an employee gets both salary hike and promotion.

PL/SQL provides for this kind of modular design where you write a subprogram to implement an independent functionality and when required you can merge the functionality of more than one modules together.

Modularizing Code with PL/SQL

- PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:
 - Anonymous blocks
 - Procedures and functions
 - Packages
 - Database triggers



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

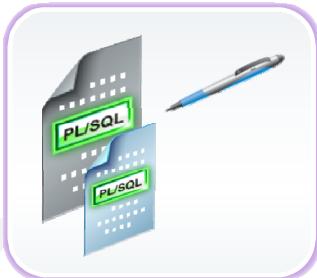
Modularization converts large blocks of code into smaller groups of code called modules. When you create modules, you should ensure that each module is representing a logically independent function of the application. The modules can be reused by the same program or shared with other programs. It is easier to maintain and debug code that comprises of smaller modules than it is to maintain code in a single large program. These modules are implemented as subprograms in PL/SQL.

You create a subprogram with a declarative section, executable section, and an exception section. A PL/SQL block can't exist without an executable section, but it can exist without a declarative section and exception section. Apart from anonymous blocks, all other PL/SQL blocks mentioned in the slide can be compiled and stored in the database as independent objects. As you store them in the database, you can independently invoke them and use them in other PL/SQL blocks, thus reducing the effort of rewriting the same code.

When you reuse a tried and tested PL/SQL subprogram in your application, it brings in certain amount of assured performance into the application.

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

Benefits of Modularization



Easy maintenance



Improved data integrity



Improved data security



Improved performance

Subprograms:
Stored procedures and
functions

ORACLE

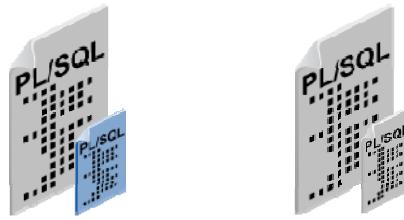
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Procedures and functions have many benefits due to modularizing of code:

- **Easy maintenance** - When you make changes and recompile the code of the subprogram stored in the database, the changes would be in effect in all the following calls to the subprogram. A change in one place is sufficient, you need not make changes in every module which uses the subprogram.
- **Data integrity** is maintained through subprograms. Each subprogram usually represents a logical function. A subprogram is executed completely or doesn't execute at all which implies a logical function executes completely or it doesn't execute at all, thus leaving your database in a consistent state.
- **Improved performance** is assured when you reuse a tried and tested code. The compiled subprograms in the database therefore ensure better performance in your application.
- **Improved data security** is assured because PL/SQL subprograms reside on the server, where the implementation isn't exposed to the client. The internal details, the structure of the database are hidden from the end user. The client can use the functionality of the sub program but can't modify the internal structure of the database.

What Are PL/SQL Subprograms?

- A PL/SQL subprogram is a named PL/SQL block.
- It enables modularization in the application.
- A subprogram consists of a specification and a body.
- Subprograms can be grouped into PL/SQL packages.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You write a subprogram to implement a logical function of the application. For example, you can write a subprogram to reflect the promotion of an employee.

Syntactically a subprogram has the following components:

- A subprogram consists of a specification (spec) and a body.
- The specification is the name of the subprogram and the description of parameters if it is defined to accept any.

To define a subprogram, you must provide both the specifications and the body. You can either declare a subprogram first or define it later.

- **Subprogram Types:** PL/SQL has three types of subprograms: procedures, functions, and triggers.
- **Procedures** are used to perform an action on the database.
- **Functions** are used to perform an operation and return a value.
- **Triggers** are used to execute actions in response to some event in the database. Procedures and functions can be invoked in the PL/SQL block to perform an action whereas triggers are defined to respond to an event in the database.

Lesson Agenda

- Modularizing code with PL/SQL
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Procedures



Thanks, Alice.



I understand your requirements.
I will come up with a solution

Alice concludes that she will create two different procedures `raise_sal` and `promotion`.

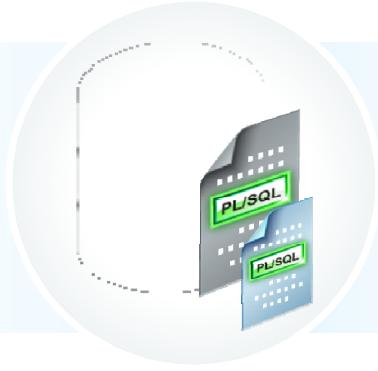
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Alice started the task of creating two procedures – `raise_sal` and `promotion`. Let us first understand how to write procedures and then discuss how can we use them to solve this problem.

What Are Procedures?

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



ORACLE®

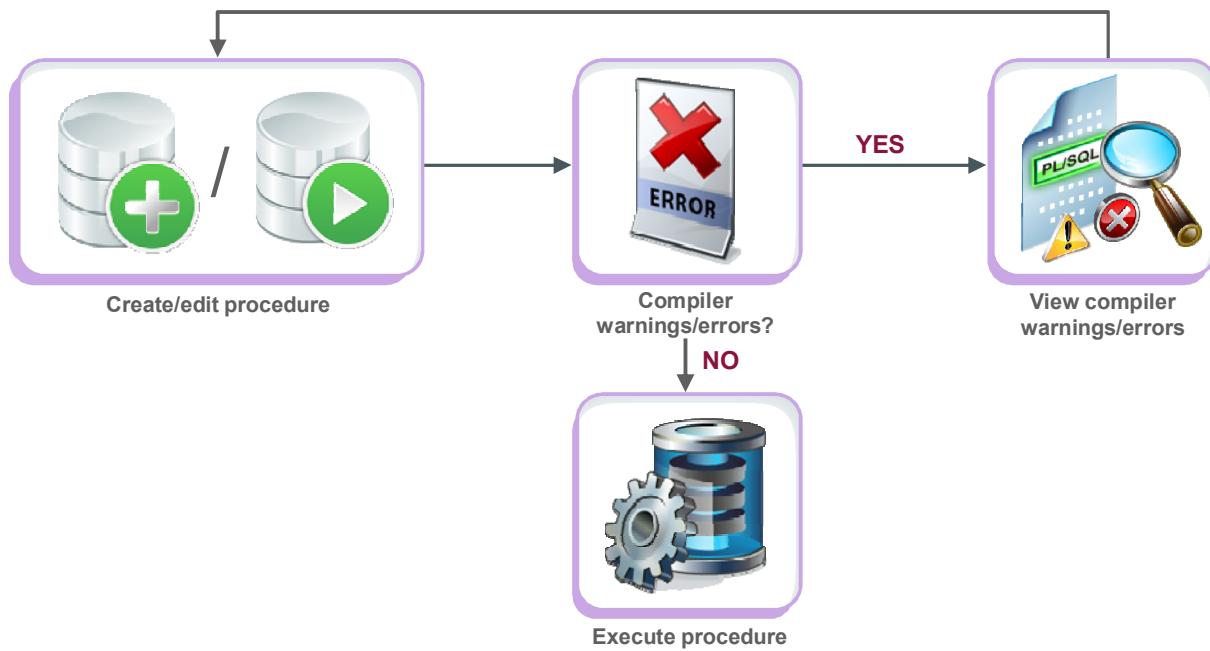
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

Procedures promote reusability and maintainability. When validated, they can be used in any number of applications.

Creating Procedures: Overview



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To develop a procedure by using a tool such as SQL Developer, perform the following steps:

1. Create the procedure using SQL Developer or SQL Plus.
2. Compile the procedure. The procedure is created in the database on compilation. The `CREATE PROCEDURE` statement creates and stores the source code and the compiled *m-code* in the database. The compilation process can be GUI based as in SQL Developer or as in case of command line interface such as SQL Plus ,you can compile the procedure like any other command.
3. If compilation errors exist, then the *m-code* is not generated and you must edit the source code to make corrections. You cannot invoke a procedure that contains compilation errors. You can view the compilation errors in Compiler log of the SQL Developer, SQL*Plus when you compile the code.
4. After successful compilation, execute the procedure to perform the desired action. You can run the procedure using SQL Developer GUI or use the `EXECUTE` command in SQL*Plus.

Creating Procedures

- Use the CREATE clause to create a stand-alone procedure that is stored in the Oracle database.
- Use the OR REPLACE option to overwrite an existing procedure.

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

PL/SQL block



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

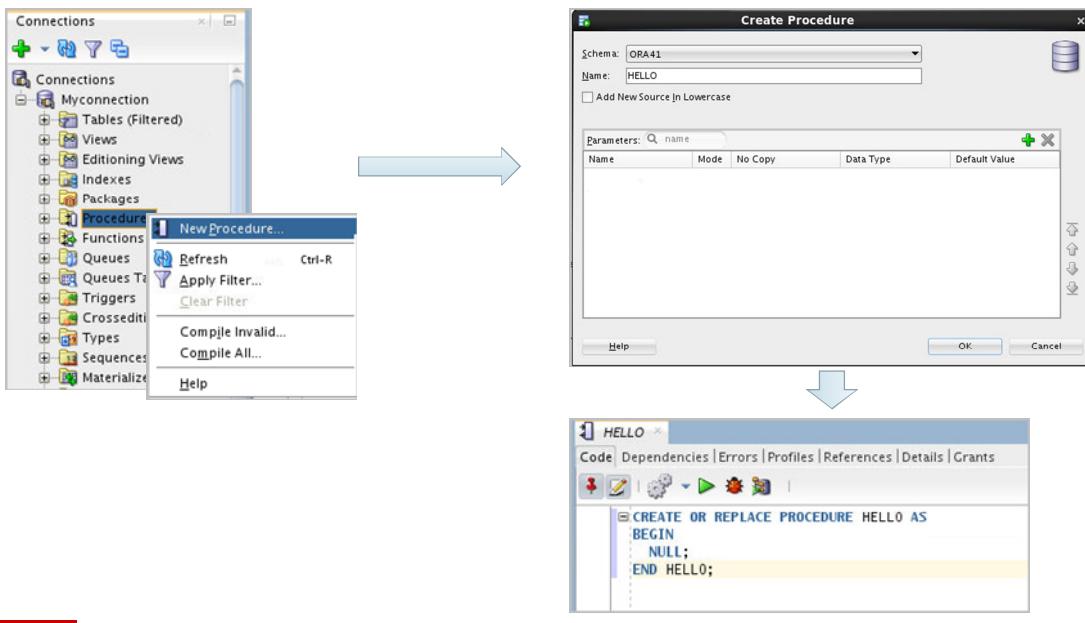
You can use the CREATE PROCEDURE SQL statement to create stand-alone procedures that are stored in an Oracle database. A procedure performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the BEGIN-END block that contains its code and handles any exceptions.

- Procedures start with BEGIN, optionally preceded by the declaration of local variables and end with either END or END *procedure_name*.
- The REPLACE option indicates that if the procedure exists, it is dropped and replaced with the new version created by the statement. The REPLACE option doesn't drop any of the privileges associated with the procedure.

Other Syntactic Elements

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

Creating Procedures Using SQL Developer



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To create a procedure by using SQL Developer, perform the following steps:

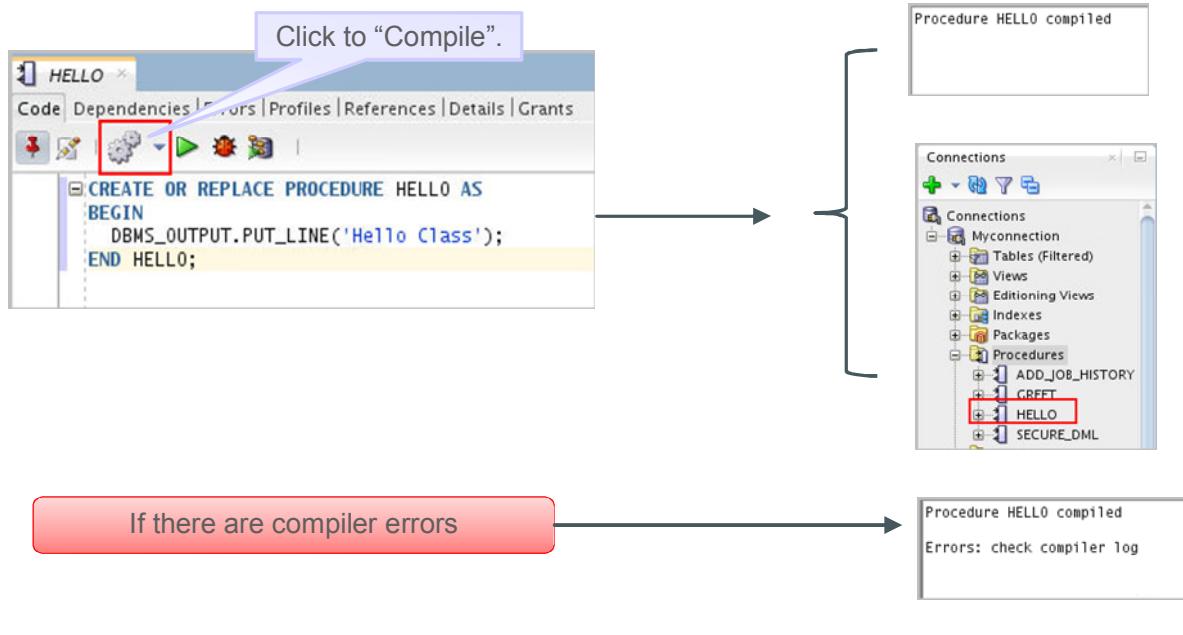
1. Right-click the **Procedures** in the **Connections** tab.
2. Select **New Procedure** from the shortcut menu. The **Create PL/SQL Procedure** dialog box is displayed.
3. Specify the information for the new procedure.
4. Click OK to create the subprogram and have it displayed in the Editor window, where you can enter the details.

Note: You can also write the code of the procedure in a SQL worksheet and execute it like any other SQL statement.

The components of the **Create PL/SQL Procedure** dialog box are as follows:

- **Schema:** The database schema in which to create the PL/SQL subprogram
- **Name:** The name of the subprogram that must be unique within a schema
- **Add New Source in Lowercase:** If this option is selected, new text appears in lowercase regardless of the case in which you enter it. This option affects only the appearance of the code, because PL/SQL isn't case-sensitive in its execution.
- **Parameters tab:** To add a parameter, click the Add (+) icon. For each parameter in the procedure to be created, specify the parameter name, data type, mode, and optionally the default Value. Use the Remove (X) icon and the arrow icons to delete and to move a parameter up or down in the list, respectively.

Compiling Procedures



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can compile procedures by clicking the compile button. It provides you with two options: **Compile** and **Compile for Debug**. You choose the appropriate one and run compilation.

If the compilation goes through successfully, you see the response 'Procedure HELLO Compiled' in the **Messages** tab and a Procedure object is created in **Connections** tab.

If there are compiler errors you see the response 'Procedure HELLO Compiled' Errors: check compiler log in the **Messages** tab. You can check the **Compiler Log** for description on errors.

You can fix the errors and recompile the code.

Calling Procedures

- You can call procedures from:
 - Anonymous blocks
 - Another named PL/SQL block
 - Command prompt using EXECUTE

```
--invoking procedure from anonymous block
BEGIN
  hello;
END;
/
```

```
--invoking procedure in another procedure
CREATE OR REPLACE PROCEDURE greet IS
BEGIN
  hello;
END greet;
/
```

```
--invoking procedure with EXECUTE
EXECUTE greet;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can invoke procedures by using:

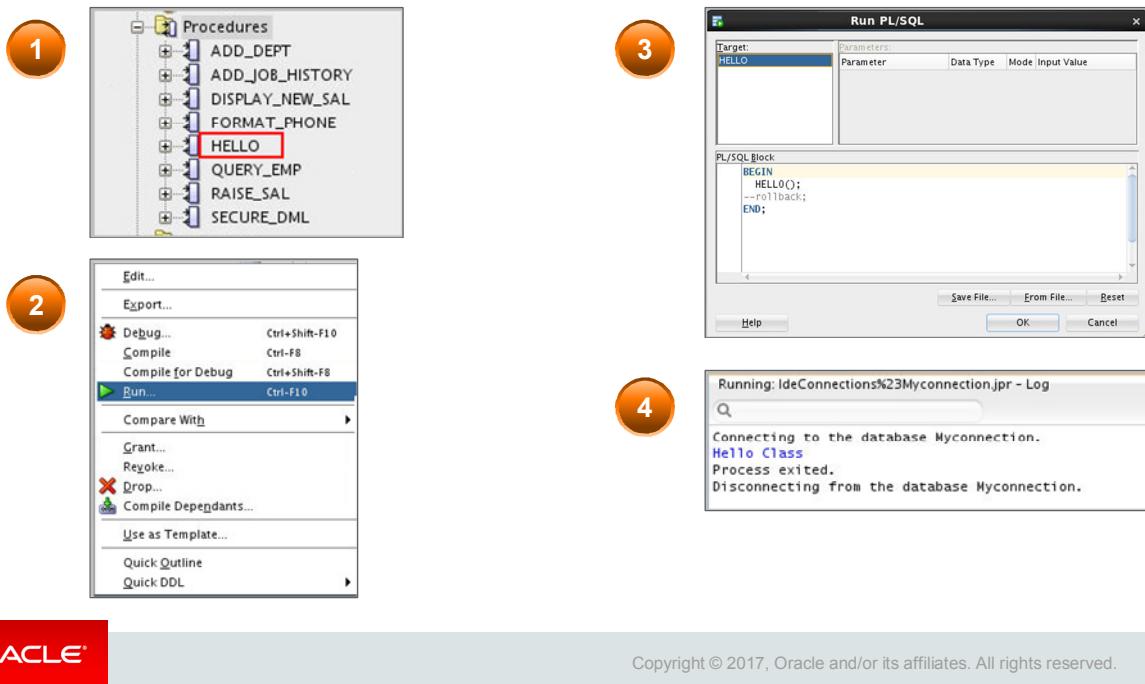
- Anonymous blocks
- Another procedure or PL/SQL subprogram
- EXECUTE command with procedure name at the command prompt

Examples in the slide have been illustrated using all the three options.

The example in the slide shows you how to invoke a procedure `hello` from anonymous PL/SQL block. You can also see that the procedure `hello` is invoked from the procedure `greet`.

The third code box shows invoking the procedure from the command prompt using the EXECUTE command.

Calling Procedures Using SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide example, the `hello` procedure is called to raise the current salary of employee 176 (\$8,600) by 10 percent as follows:

1. Right-click the procedure name in the **Procedures** node, and then click **Run**. The **Run PL/SQL** dialog box is displayed.
2. You see that the procedure `hello` is enclosed in an anonymous PL/SQL block for execution.
3. Click **OK**. SQL Developer runs the procedure and displays "Hello Class".

Results	Script Output	Explain	Autotrace	DBMS Output	OWA Output
Results:					
	EMPLOYEE_ID	SALARY			
1	176	9460			

Procedures

```
CREATE OR REPLACE PROCEDURE raise_sal AS
v_raise_percent NUMBER := 10;
BEGIN
UPDATE employees
SET
salary = salary * (1+1/v_raise_percent);
END salary_hike;
```

Alice created the procedure `raise_sal`, but she had questions for the HR manager before creating the `promotion` procedure



There is a problem here Alice!



I have created this procedure where you can increment the salary of all the employees by 10%. You can change the percentage to some other value too



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Alice here created a procedure `raise_sal` which would update the salary of all the employees by 10%, where you can change the percentage of hike by initializing the variable with a different value. However things don't work like this practically. The increment an employee gets depend on factors like employee's performance, the job he/she hold and other factors and remember every employee may not get an increment.

Our HR manager expressed the same to Alice. Now Alice has to modify this procedure to accept input such as `employee_id` and raise percentage as input and perform the expected operation on the database.

We will further look into the concept of parameters. Parameters enable you to provide a different input each time you execute the procedure. Alice can use parameters to pass input to the increment procedure and perform the required operations.

What Are Parameters and Parameter Modes?

- Means of communication between the calling environment and the Procedure
- Each parameter is associated with a mode:
 - An `IN` parameter mode (the default) provides values for a subprogram to process.
 - An `OUT` parameter mode returns a value to the caller.
 - An `IN OUT` parameter mode supplies an input value, which may be returned (output) as a modified value.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Parameters are defined with one of the three parameter-passing modes: `IN`, `OUT`, or `IN OUT`.

- An `IN` parameter passes a constant value from the calling environment into the procedure.
- An `OUT` parameter passes a value from the procedure to the calling environment.
- An `IN OUT` parameter passes a value from the calling environment to the procedure and a possibly different value from the procedure back to the calling environment using the same parameter.

Formal and Actual Parameters

- Formal parameters are place holders for data in a subprogram specification.
- Actual parameters are literal values, variables or expressions substituted in the parameter list while invoking the subprogram.

```
-- Procedure definition
Formal_parameters
CREATE PROCEDURE raise_sal(p_id
NUMBER) IS
BEGIN
...
END raise_sal;
```

```
--Procedure calling, Actual
parameters (arguments)
execute raise_sal(100)
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Formal parameters are declared in the parameter list of a subprogram specification. In the procedure definition of `raise_sal`, the variable `p_id` identifier represent the formal parameter.

The actual parameters can be literal values, variables, and expressions that are provided in the parameter list while calling a procedure. In the procedure call, the `raise_sal` procedure is invoked with a value of 100, which is the actual parameter.

Actual parameters:

- Are associated with formal parameters during the subprogram call
- Can also be expressions, as in the following example:
`raise_sal(v_emp_id, raise+100);`

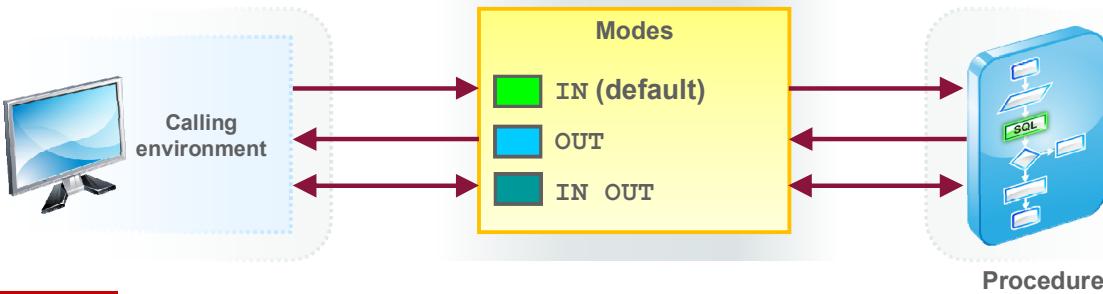
where `v_emp_id` and `raise` are variables with some initialized values

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

Procedural Parameter Modes

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

```
CREATE PROCEDURE proc_name(param_name [mode] datatype,...)
...
```



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

Comparing the Parameter Modes

IN	OUT	IN OUT
It is the default mode.	It must be specified.	It must be specified.
Value is passed into subprogram.	Value is returned to the calling environment.	Value is passed into sub-program; value is returned to calling environment.
Formal parameter acts as a constant.	It is uninitialized variable.	It is initialized variable.
Actual parameter can be a literal, expression, constant, or initialized variable.	It must be a variable.	It must be a variable.
It can be assigned a default value.	It cannot be assigned a default value.	It cannot be assigned a default value.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

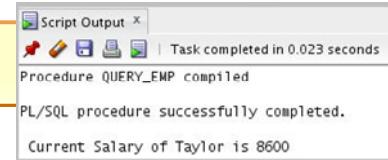
A formal parameter of `IN` mode cannot be assigned a value and cannot be modified in the body of the procedure. By default, the `IN` parameter is passed by reference. An `IN` parameter can be assigned a default value in the formal parameter declaration, in which case the caller need not provide a value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

An `OUT` or `IN OUT` parameter must be assigned a value before returning to the calling environment. The `OUT` and `IN OUT` parameters cannot be assigned default values.

Using the IN Parameter Mode: Example

```
CREATE OR REPLACE PROCEDURE query_emp
(p_id IN employees.employee_id%TYPE) AS
  v_name employees.last_name%TYPE;
  v_salary employees.salary%TYPE;
BEGIN
  SELECT last_name, salary INTO v_name,v_salary
  FROM employees
  WHERE employee_id = p_id;
  DBMS_OUTPUT.PUT_LINE(' Current Salary of '||v_name || ' is '|| v_salary);
END query_emp;
```

EXECUTE query_emp(176)



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows an example for using IN mode of procedure parameters. IN mode implies input to the procedure.

In the example, the procedure accepts an employee ID as input parameter. This input parameter is used to retrieve data from the employees table inside the procedure and display the current value of the employee salary.

Here in the slide we invoke the procedure by using the EXECUTE command.

You can also invoke a procedure from another procedure by making a direct call inside an executable section of the calling block. Invoke the procedure with actual parameters in the PL/SQL block as shown below:

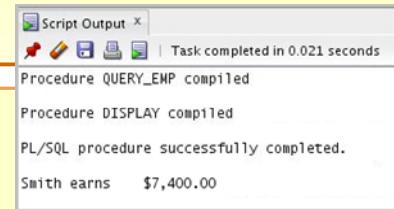
```
...
BEGIN
  query_emp(176);
END;
```

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

Using the OUT Parameter Mode: Example

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

```
CREATE OR REPLACE PROCEDURE display AS
  v_emp_name employees.last_name%TYPE;
  v_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, v_emp_name, v_emp_sal);
  DBMS_OUTPUT.PUT_LINE(v_emp_name||' earns '|| to_char(v_emp_sal, '$999,999.00'));
END display;
EXECUTE display;
```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

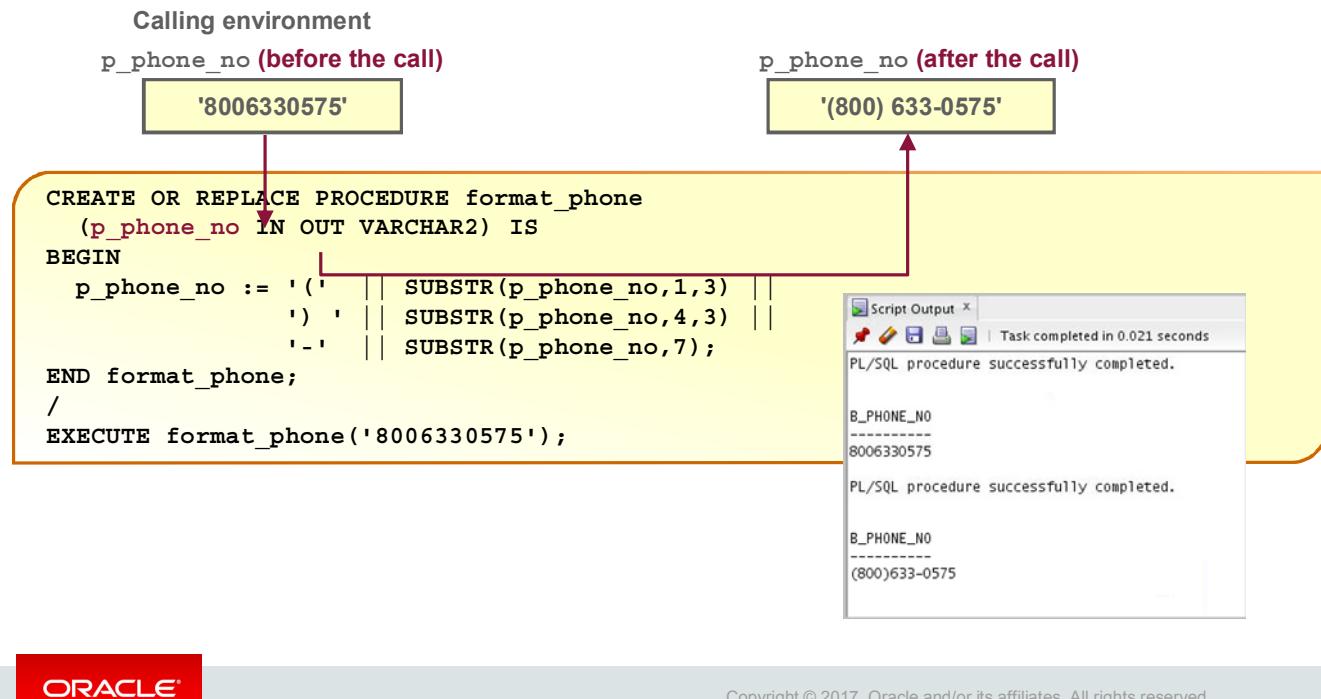
The slide shows the usage of OUT parameters in a procedure.

The first piece of code is the definition of a procedure `query_emp` which has one parameter set to `IN` mode and two parameters set to `OUT` mode. The parameter set to `IN` mode is the value of the `employee_id` and the parameters set to `OUT` mode are the values of `last_name` and `salary` of the employee. We use this procedure in another procedure `display` defined in the second code box.

The `query_emp` procedure is invoked in the `display` procedure by passing only the input parameter. There are place holders for the output parameter, declared as variables in the `display` procedure. The output parameters of the `query_emp` procedure are retrieved into the `display` procedure and used in the output message.

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

Using the IN OUT Parameter Mode: Example



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using an `IN OUT` parameter, you can pass a value into a procedure that can be updated and returned as output. The actual parameter value supplied from the calling environment can return either the original unchanged value or a new value that is set within the procedure.

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

The following code uses the `b_phone_no` host variable of SQL*Plus to provide the input value passed to the `FORMAT_PHONE` procedure. The procedure is executed and returns an updated string in the `b_phone_no` host variable. The output of the following code is displayed in the slide:

```

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

```

Passing Parameters to Procedures

- When calling a subprogram, you can write the actual parameters using the following notations:
 - Positional: Lists the actual parameters in the same order as the formal parameters
 - Named: Lists the actual parameters in arbitrary order and uses the association operator (`=>`) to associate a named formal parameter with its actual parameter
 - Mixed: Lists some of the actual parameters as positional and some as named



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When calling a subprogram, you can write the actual parameters by using the following notations:

- Positional:** You list the actual parameter values in the same order in which the formal parameters are declared. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the procedure execution may give wrong results or it might result in an error.
- Named:** You list the actual values in an arbitrary order and use the association operator to associate each actual parameter with its formal parameter by name. The PL/SQL **association operator** is an “equal” sign followed by an “is greater than” sign, without spaces: `=>`. The order of the parameters is not significant. This notation is more verbose, but makes your code easier to read and maintain. You can sometimes avoid changing your code if the procedure’s parameter list changes, for example, if the parameters are reordered or a new optional parameter is added.
- Mixed:** You list the first parameter values by their position and the remainder by using the special syntax of the named method. You can use this notation to call procedures that have some required parameters, followed by some optional parameters.

Passing Actual Parameters: Creating the `raise_sal` Procedure

```
CREATE OR REPALCE PROCEDURE raise_sal
(p_id IN employees.employee_id%TYPE,
 p_percent IN NUMBER) AS
BEGIN
  UPDATE employees
  SET salary = salary * (1 + p_percent/100)
  WHERE employee_id = p_id;
END raise_sal;
```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can see in the slide that Alice has modified the `raise_sal` procedure to accept input parameters. The procedure accepts the `employee_id` and the percentage of salary raise that has to be given to the specific employee.

Based on the values of the input parameters an `UPDATE` command is executed in the executable section of the `raise_sal` procedure.

Passing Actual Parameters: Examples

```
-- Passing parameters using the positional notation.
SELECT * FROM EMPLOYEES WHERE employee_id = 176;
EXECUTE raise_sal ('176', 10)
SELECT * FROM EMPLOYEES WHERE employee_id = 176;
```

The screenshot shows three windows from Oracle SQL Developer:

- Query Result:** Shows a single row from the EMPLOYEES table with EMPLOYEE_ID 176, FIRST_NAME Jonathon, LAST_NAME Taylor, and SALARY 8600.
- Script Output:** Shows the message "PL/SQL procedure successfully completed."
- Query Result:** Shows the same row again, but with the SALARY value updated to 9460.

```
-- Passing parameters using the named notation.
EXECUTE raise_sal( p_percent =>10,p_id => 176)
```



Thanks Alice. This will help. Please find a solution to update the promotion data also in the database



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Passing actual parameters by position is shown in the first call to execute `raise_sal` in the first code example in the slide. The first actual parameter supplies the value 176 for the `p_id` formal parameter. The second actual parameter value of 10 is assigned by position to the `p_percent` formal parameter.

Passing parameters using the named notation is shown in the second code example in the slide. The `p_percent` actual parameter, which is declared as the second formal parameter, is referenced by name in the call, where it is associated with the actual value of 10. The `p_id` parameter is associated with the value 176. The order of the actual parameters is irrelevant if all parameter values are specified with the names.

Note: You must provide a value for each parameter unless the formal parameter is assigned a default value. We discuss specifying default values for formal parameters in the following slides.

Now the HR manager can either execute multiple `EXECUTE` commands for updating the salaries of different employees or ask Alice to write a PL/SQL block to for all the `UPDATES` by providing the employee ids and their corresponding salary raise percentage.

Using the DEFAULT Option for the Parameters

- Defines default values for parameters
- Provides flexibility by combining the positional and named parameter-passing syntax

```
CREATE OR REPLACE PROCEDURE raise_sal
(p_id IN employees.employee_id%TYPE,
 p_percent IN NUMBER DEFAULT 10) AS
BEGIN
  UPDATE employees
  SET salary = salary * (1 + p_percent/100)
  WHERE employee_id = p_id;
END raise_sal;
```

```
EXECUTE raise_sal(176) or
EXECUTE raise_sal(176,10)
EXECUTE raise_sal (p_id => 176)
```

An enhancement !



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This is an enhancement provided by Alice where the HR manager can apply a default value for raise percentage. Wherever there are exceptions to the default value the HR manager can explicitly enter the value while invoking the procedure.

You can assign a default value to an `IN` parameter through one of the following options:

- Through an assignment operator (`:=`)
- Through `DEFAULT` option, as shown for the `p_percent` parameter in the slide

When default values are assigned to formal parameters, you can call the procedure without supplying an actual parameter value for the parameter. Thus, you can pass different numbers of actual parameters to a subprogram, either by accepting or by overriding the default values as required.

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

The second code box in the slide shows three ways of invoking the `raise_sal` procedure:

- The first example assigns the default value for the `p_percent` parameter.
- The second example illustrates assigning values to both the parameters
- The last example uses the named notation to assign value to the parameter `p_id`

Lesson Agenda

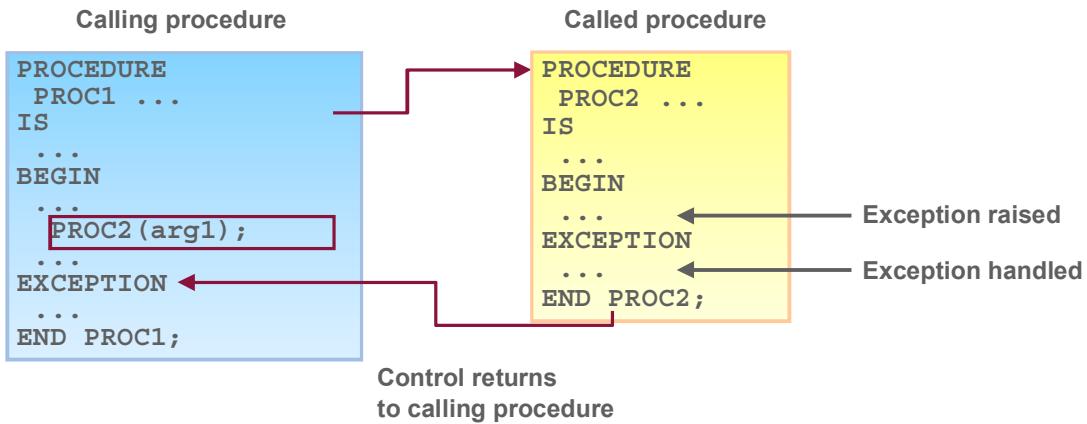
- Modularizing code with PL/SQL
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedure's information



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Handled Exceptions



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When you develop procedures that are called from other procedures, you should be aware of the effects that handled and unhandled exceptions have on the transaction and the calling procedure. When an exception is raised in a called procedure, the control immediately goes to the exception section of that block. An exception is considered handled if the exception section provides a handler for the exception raised.

When an exception occurs and is handled, the following code flow takes place:

1. The exception is raised.
2. Control is transferred to the exception handler.
3. The block is terminated.
4. The calling program/block continues to execute unaffected.

If a transaction was started in the calling block, then the transaction is unaffected. A DML operation performed within the procedure before the exception is rolled back.

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

Handled Exceptions: Example

```

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

```

```

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

```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.



The two procedures in the slide are the following:

- The `add_department` procedure creates a new department record by allocating a new department number from an Oracle sequence, and sets the `department_name`, `manager_id`, and `location_id` column values using the `p_name`, `p_mgr`, and `p_loc` parameters, respectively.
- The `create_departments` procedure creates more than one department by using calls to the `add_department` procedure.

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

```

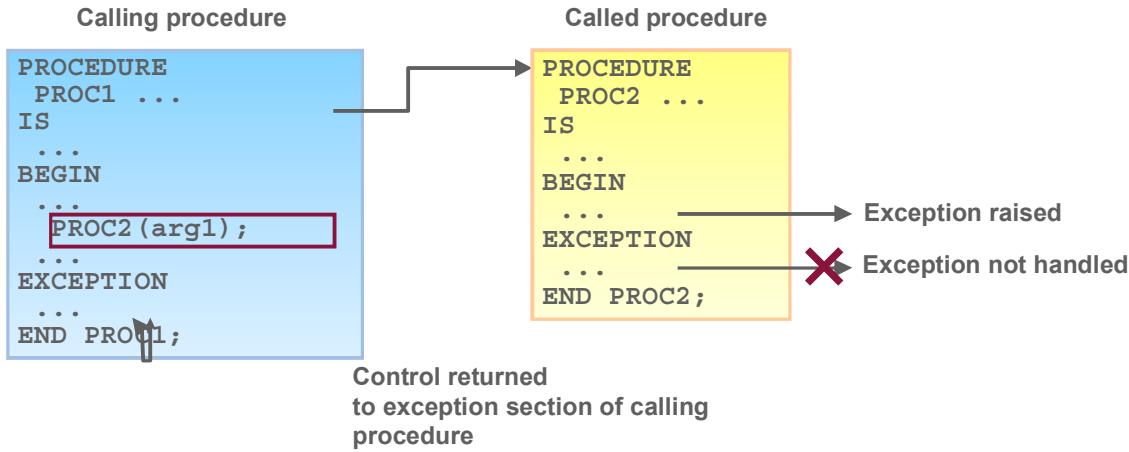
PL/SQL procedure successfully completed.

Added Dept: Media
Err: adding dept: Editing
Added Dept: Advertising

```

The `Editing` department with a `manager_id` of 99 is not inserted because a foreign key integrity constraint violation on `manager_id` ensures that no manager has an ID of 99. Because the exception was handled in the `add_department` procedure, the `create_department` procedure continues to execute. A query on the `DEPARTMENTS` table where the `location_id` is 1800 shows that `Media` and `Advertising` are added but the `Editing` record is not.

Exceptions Not Handled



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

If an exception is not handled, then all the DML statements in the calling procedure and the called procedure are rolled back along with any changes to any host variables.

Exceptions Not Handled: Example

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

```
Script Output X
| Task completed in 0.076 seconds
Error starting at line : 292 in command -
EXECUTE create_departments_noex
Error report -
ORA-02291: integrity constraint (ORA61.DEPT_MGR_FK) violated - parent key not found
ORA-06512: at "ORA61.ADD_DEPARTMENT_NOEX", line 4
ORA-06512: at "ORA61.CREATE_DEPARTMENTS_NOEX", line 4
ORA-06512: at line 1
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause: A foreign key value has no matching primary key value.
*Action: Delete the foreign key or add a matching primary key.
Added Dept: Media
```

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

Removing Procedures: Using the DROP SQL Statement or SQL Developer

- Using the DROP statement:

```
DROP PROCEDURE raise_sal;
```

- Using SQL Developer:



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When a stored procedure is no longer required, you can use the `DROP PROCEDURE` SQL statement followed by the procedure's name to remove it as follows:

```
DROP PROCEDURE procedure_name
```

You can also use SQL Developer to drop a stored procedure as follows:

- Right-click the procedure name in the **Procedures** node, and then click **Drop**. The **Drop** dialog box is displayed.
- Click **Apply** to drop the procedure.

Note

- Whether successful or not, executing a data definition language (DDL) command such as `DROP PROCEDURE` commits any pending transactions that cannot be rolled back.
- You might have to refresh the **Procedures** node before you can see the results of the drop operation. To refresh the **Procedures** node, right-click the procedure name in the **Procedures** node, and then click **Refresh**.

Viewing Procedure Information Using the Data Dictionary Views

```
DESCRIBE user_source
```

Name	Null Type
<hr/>	
NAME	VARCHAR2(128)
TYPE	VARCHAR2(12)
LINE	NUMBER
TEXT	VARCHAR2(4000)
ORIGIN_CON_ID	NUMBER

```
SELECT text
  FROM user_source
 WHERE name = 'RAISE_SAL' AND type = 'PROCEDURE'
 ORDER BY line;
```

TEXT
1 PROCEDURE raise_sal
2 (p_id IN employees.employee_id%TYPE,
3 p_percent IN NUMBER) AS
4 BEGIN
5 UPDATE employees
6 SET salary = salary * (1 + p_percent/100)
7 WHERE employee_id = p_id;
8 END raise_sal;



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

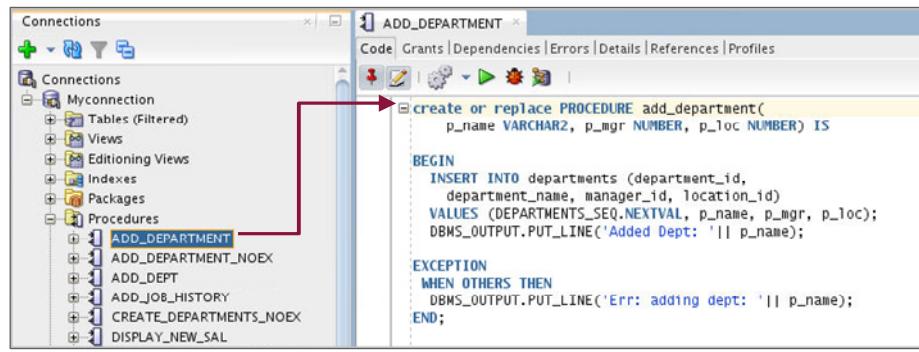
The query example shows all the columns provided by the `USER_SOURCE` table:

- The `NAME` column holds the name of the subprogram in uppercase text.
- The `TYPE` column holds the subprogram type, such as `PROCEDURE` or `FUNCTION`.
- The `LINE` column stores the line number for each source code line.
- The `TEXT` column holds PL/SQL source code.
- The `ORIGIN_CON_ID` holds the ID of the container where the data originates. The value of the container id can range from 0-n based on database configuration.

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

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

Viewing Procedures Information Using SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To view a procedure's code in SQL Developer, perform the following steps:

1. Click the **Procedures** node in the **Connections** tab.
2. Click the procedure's name.
3. The procedure's code is displayed in the **Code** tab as shown in the slide.

Quiz



Which of the following is an invalid procedure call for a procedure?

ADD (a IN NUMBER DEFAULT 10, b IN NUMBER DEFAULT 20, result OUT NUMBER)

- a. ADD
- b. ADD (ex)
- c. ADD (10, 20, ex)
- d. ADD (b=>45, a=>35, ex)



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a

Options b and c are valid procedure calls with the parameter specified through OUT mode but option a does not have the parameter with the OUT mode as defined in the procedure specification.

Summary

In this lesson, you should have learned how to:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure and display its information



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Practice 2 Overview: Creating, Compiling, and Calling Procedures

This practice covers the following topics:

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



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Practice 2: Overview

In this practice, you create, compile, and invoke procedures that issue DML and query commands. You also learn how to handle exceptions in procedures.

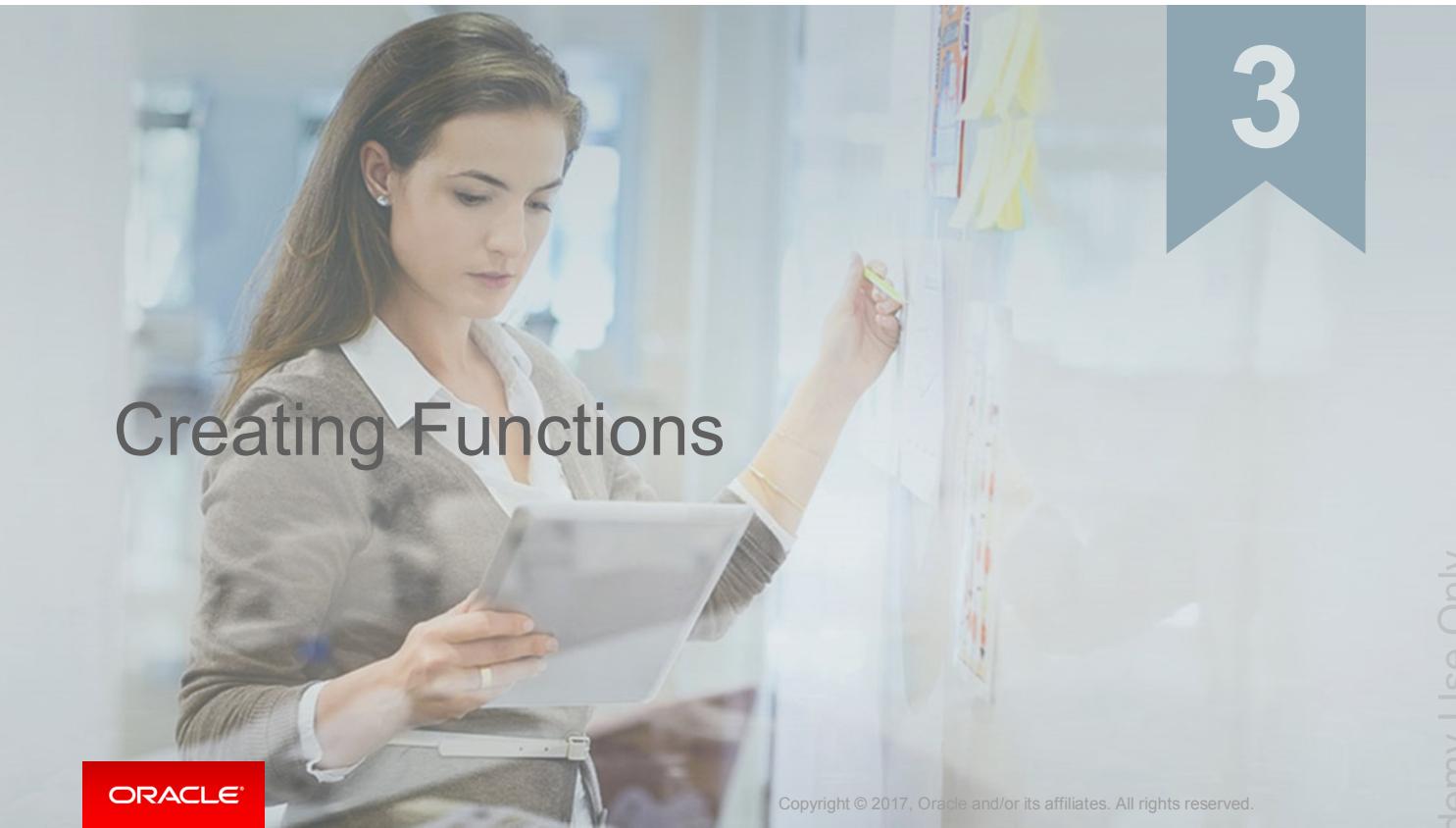
If you encounter compilation errors when you execute procedures, you can use the **Compiler-Log** tab in SQL Developer.

Note: It is recommended to use SQL Developer for this practice.

Important

All practices in this course and the practice solutions assume that you create objects such as procedures, functions, and so on using the SQL Worksheet area in SQL Developer. When you create an object in the SQL Worksheet area, you need to refresh the object node in order for the new object to be displayed in the Navigator tree. To compile the newly created object, you can right-click the object name in the Navigator tree, and then select **Compile** from the shortcut menu. For example, after you enter the code to create a procedure in the SQL Worksheet area, click the **Run Script** icon (or press F5) to run the code. This creates and compiles the procedure.

Alternatively, you can create objects such as procedures using the **PROCEDURES** node in the **Navigator tree**, and then compile the procedure. Creating objects using the **Navigator tree** automatically displays the newly created object.



3

Creating Functions

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

▶ Lesson 2: Creating Procedures

Lesson 3: Creating Functions

▶ Lesson 4: Debugging Subprograms

▶ Lesson 5: Creating Packages

▶ Lesson 6: Working with Packages

▶ Lesson 7: Using Oracle-Supplied Packages in Application Development

▶ Lesson 8: Using Dynamic SQL

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Working with Subprograms, Working with Triggers, and Working with PL/SQL Code.

In the first unit, we have seven lessons – Creating Stored Procedures, Creating Functions, Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms, packages, and use them in Application Development.

Objectives

After completing this lesson, you should be able to:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Explain the basic functionality of the SQL Developer debugger



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Lesson Agenda

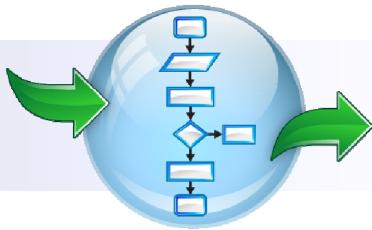
- Creating and invoking functions
- Functions in SQL expressions
- Passing parameters to functions
- Removing stored functions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Functions



- Named PL/SQL block like procedure
- Accepts input as parameters
- Processes the input and returns a value
- Stored as a database object
- Enables modularization in the application
- Can be invoked as part of an expression

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You know about procedures: they are named PL/SQL blocks stored on the database. Functions are similar to procedures in most of its features except for the fact that they return a single value.

Functions are named PL/SQL blocks that are used to implement modularization in the application. You should define functions to represent a logical functionality of the application. Procedures do the same job except for the fact that functions return a value to the calling environment. In case of procedures, you can see the output of the procedure by accessing the `OUT` parameters of the procedure and procedures may or may not have an output, but functions will always return a value.

Each function has a `RETURN` statement in its PL/SQL block. There can be more than one return statements in a PL/SQL block; however, a function will execute only one of these return statements based on the function logic.

Creating Functions syntax

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Syntax for Creating Functions

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

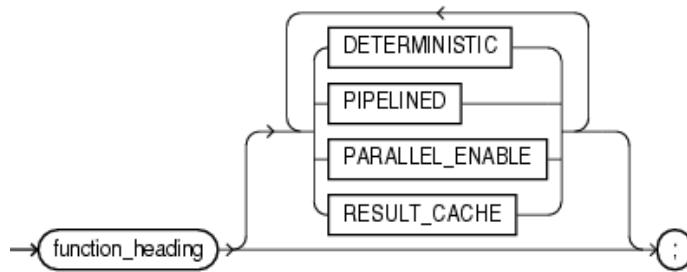
You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

The type of value returned from the function can be a scalar data type or a composite data type.

You should consider the following points about the CREATE FUNCTION statement:

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

The function syntax shown in the slide is good enough to create an usable function; however, a function declaration may have following additional options to improve performance in an operational application. These options are not required to implement the basic functionality of a function.



DETERMINISTIC option	A deterministic option in the function declaration implies that each time you give certain values as parameters, the function would return the same value. The value returned by the function is determined by the values of the input parameters. The function do not contain any nondeterministic entities . This is helpful for caching the results and improving the performance
PARALLEL_ENABLE option	Enables the function for parallel execution, which implies that there are no dependencies on the execution of other functions or modules. Each parallel session using the function will have private copies of the function's local variables.
PIPELINED option	A pipelined table function returns a row to its invoker immediately after processing that row and continues to process other rows. It is used only with table functions.
RESULT_CACHE option	Stores function results in the PL/SQL function result cache (appears only in declaration)
RESULT_CACHE clause	Stores function results in the PL/SQL function result cache (appears only in definition)

Tax Calculation



I could update the database with the promotions and salary raise. Thanks Alice. Now I need your help in calculating the tax applicable for each employee.

Sure, tell me the formula for calculation.



I will mail you the formula. It's a bit complicated. The code you write should help me in arriving at a value based on input. It should be good to use in mathematical formulas. I may have to calculate taxes in many places.

Got it



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The HR manager in this scenario wants to calculate the tax applicable for an employee. He wants to use this in some mathematical expression. For instance to calculate the payable salary for an employee, the tax per month must be deducted from the salary value in the `EMPLOYEES` table for that employee.

$$\text{net salary} = \text{salary} - \text{tax applicable}$$

The HR manager requires a program construct that can substitute the value of tax applicable in the mathematical expression.

Functions are useful in such scenarios. You can write a function for tax and pass the `employee_id` as the input parameter. The function should pick up the value of the salary from the database, calculate the applicable tax, and return the value of the tax into the mathematical expression. The expression would in turn calculate the net salary.

The Difference Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a RETURN statement without a value	Must contain at least one RETURN statement
Cannot invoke from SQL expressions	Can be invoked from a SQL expression



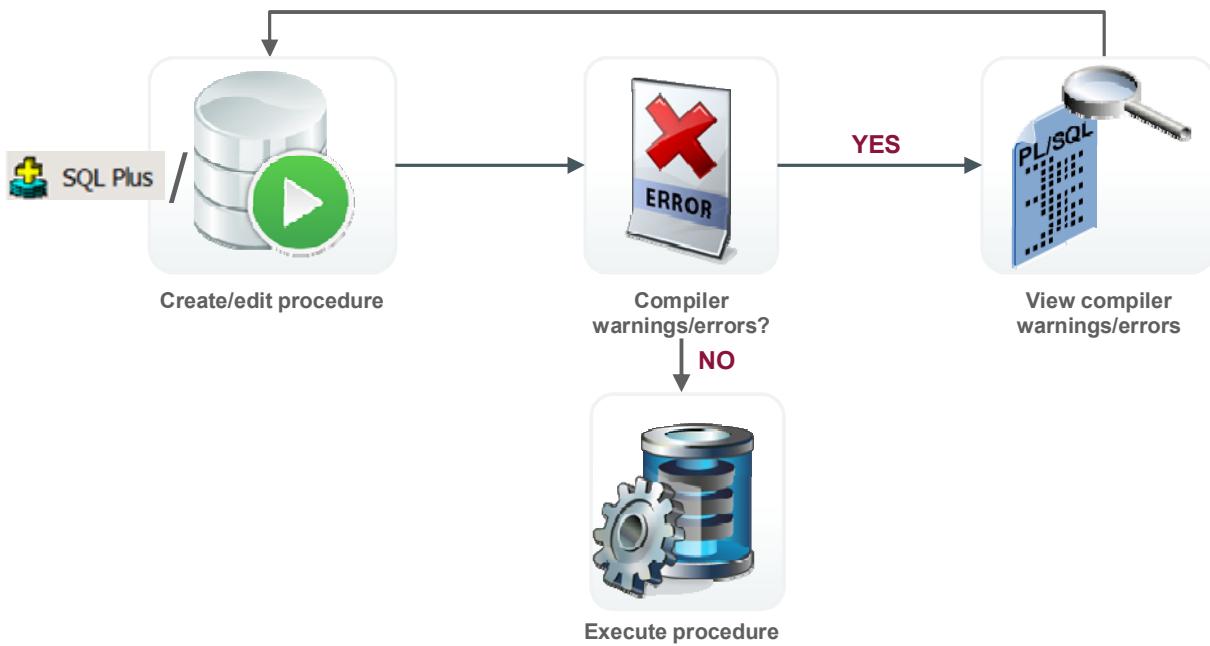
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

Creating Functions: Overview



ORACLE

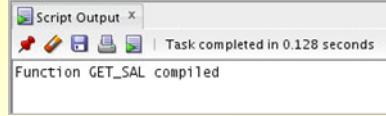
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To develop a function by using a tool such as SQL Developer, perform the following steps:

1. You can create the function using SQL Developer or SQL Plus.
2. Compile the function. The function is created in the database on compilation. The `CREATE FUNCTION` statement creates and stores the source code and the compiled *m-code* in the database. The compilation process can be GUI based as in SQL Developer or as in the case of command-line interface such as SQL Plus ,you can compile the function like any other command.
3. If compilation errors exist, then the *m-code* isn't generated and you must edit the source code to make corrections. You can't invoke a function that contains compilation errors. You can view the compilation errors in the **Compiler Log** of SQL Developer, SQL*Plus when you compile the code.
4. After successful compilation, execute the procedure to perform the desired action. You can run the procedure using SQL Developer GUI or use the `EXECUTE` command in SQL*Plus.

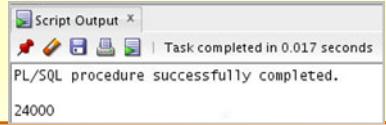
Invoking a Stored Function: Example

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



```
-- Invoke the function as an expression or as
-- a parameter value.

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this slide, you create a `get_sal` function to retrieve the value of the salary of an employee. The code in the first code box creates the function. When the function is successfully compiled, you see the response ‘Function GET_SAL compiled’ as shown in the slide.

The function has one input parameter and returns a `NUMBER` value. It accepts the `employee_id` and retrieves the salary of the employee with the given `employee_id`.

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

You can incorporate a function call as part of a PL/SQL expression because the function will return a value to the calling environment. You can incorporate multiple function calls in a single PL/SQL expression resulting in a simpler and maintainable code.

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

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

Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables  
VARIABLE b_salary NUMBER  
EXECUTE :b_salary := get_sal(100)
```

PL/SQL procedure successfully completed.

B_SALARY

24000

```
-- As a PL/SQL expression, get the results using a local  
-- variable  
SET SERVEROUTPUT ON  
DECLARE  
    sal employees.salary%type;  
BEGIN  
    sal := get_sal(100);  
    DBMS_OUTPUT.PUT_LINE('The salary is: '|| sal);  
END;  
/
```

PL/SQL procedure successfully completed.

The salary is: 24000



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Functions can be invoked in the following ways:

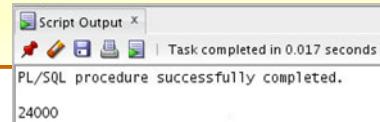
- **As part of PL/SQL expressions:** You can use host or local variables to hold the returned value from a function. The first example in the slide uses a host variable and the second example uses a local variable in an anonymous block.

Note: The benefits and restrictions that apply to functions when used in a SQL statement are discussed in the following slides.

Using Different Methods for Executing Functions

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

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



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

```
SELECT job_id, get_sal(employee_id)
FROM employees
WHERE department_id = 60;
```

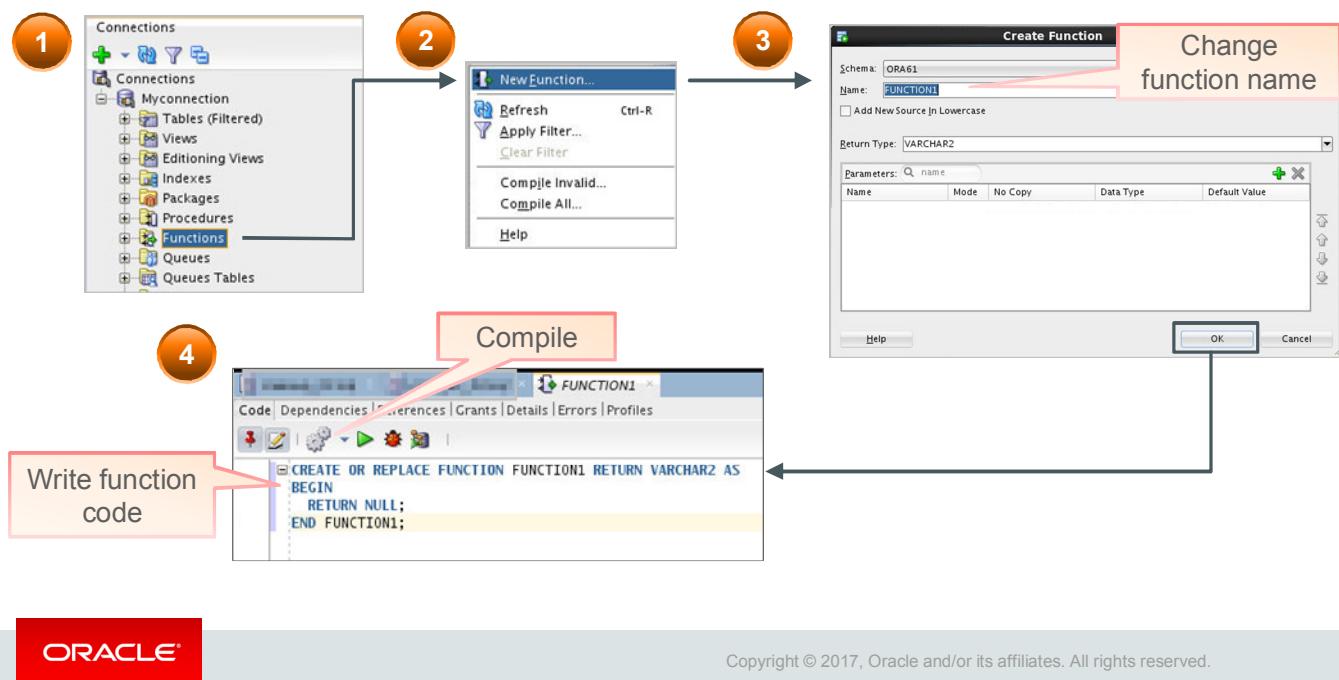
JOB_ID	GET_SAL(EMPLOYEE_ID)
1 IT_PROG	9000
2 IT_PROG	6000
3 IT_PROG	4800
4 IT_PROG	4800
5 IT_PROG	4200

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- **As a parameter to another subprogram:** The first example in the slide demonstrates this usage. The `get_sal` function with all its arguments is nested in the parameter required by the `DBMS_OUTPUT.PUT_LINE` procedure. This comes from the concept of nesting functions as discussed in the course titled *Oracle Database: SQL Fundamentals I*.
- **As an expression in a SQL statement:** The second example in the slide shows how a function can be used as a single-row function in a SQL statement.

Creating and Compiling Functions Using SQL Developer



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can create a new function in SQL Developer by using the following steps:

1. Right-click the **Functions** node.
2. Select **New Function** from the shortcut menu. The **Create PL/SQL Function** dialog box is displayed.
3. Select the schema, function name, and the parameters list (using the + icon), and then click **OK**. The code editor for the function is displayed.
4. Enter the function's code.
5. To compile the function, click the **Compile** icon.

Note

- To create a new function in SQL Developer, you can also enter the code in the SQL Worksheet, and then click the **Run Script** icon.
- For additional information about creating functions in SQL Developer, access the appropriate online Help topic titled “Create PL/SQL Subprogram Function or Procedure.”

Lesson Agenda

- Creating and invoking functions
- Functions in SQL expressions
- Passing parameters to functions
- Removing stored functions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using a Function in a SQL Expression: Example

```
CREATE OR REPLACE FUNCTION tax(p_id IN employees.employee_id%type)
  RETURN NUMBER IS
  v_sal employees.salary%type;
BEGIN
  select salary into v_sal
  from employees
  where employee_id = p_id;
  RETURN (v_sal * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(employee_id)
FROM   employees
WHERE  department_id = 100;
```

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
1	108 Greenberg	12008	960.64
2	109 Faviet	9000	720
3	110 Chen	8200	656
4	111 Sciarra	7700	616
5	112 Urman	7800	624
6	113 Popp	6900	552



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows how to create a `tax` function to calculate income tax. The function accepts an `employee_id` as input parameter and calculates the tax applicable for that employee based on the salary of that employee.

To execute the code shown in the slide example in SQL Developer, enter the code in the SQL Worksheet, and then click the **Run Script** icon. The `tax` function is invoked as an expression in the `SELECT` clause along with the employee ID, last name, and salary for employees in a department with ID 100. The return result from the `tax` function is displayed with the regular output from the query.

Calling User-Defined Functions in SQL Statements

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A PL/SQL user-defined function can be called from any SQL expression where a built-in single-row function can be called as shown in the following example:

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

EMPLOYEE_ID	TAX(EMPLOYEE_ID)
1	1920
2	1360
3	1360
4	1120
5	1080
6	1040
7	960.64
8	960.64
9	960
10	920

Note: You can use SQL functions with `CONNECT BY` and `START WITH` clauses in hierarchical SQL queries.

Restrictions When Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
 - Be stored in the database
 - Accept only `IN` parameters with valid SQL data types and PL/SQL-specific data types
 - Return valid SQL data types and PL/SQL-specific data types
- When calling functions in SQL statements:
 - Be the owner of the function
 - Have the `EXECUTE` privilege
 - Enable the `PARALLEL_ENABLE` option in function specification



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The user-defined PL/SQL functions that are callable from SQL expressions must meet the following requirements:

- The function must be stored in the database.
- The function parameters must be `IN` and of valid SQL data types and PL/SQL-specific data types.
- The functions must return data types that are valid SQL data types and PL/SQL-specific data types such as `BOOLEAN`, `RECORD`, or `TABLE`.

The following restrictions apply when calling a function in a SQL statement:

- Parameters must use positional notation or named notation.
- You must own or have the `EXECUTE` privilege on the function.
- You may need to enable the `PARALLEL_ENABLE` option in the function specification to allow a parallel execution of the SQL statement using the function. Each parallel session using the function will have private copies of the function's local variables.

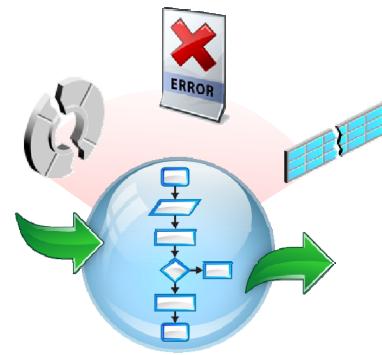
Other restrictions on a user-defined function include the following:

- You cannot invoke the function from the `CHECK` constraint clause of a `CREATE TABLE` or `ALTER TABLE` statement.
- You cannot use a function to define the default value of a column.

Side Effects of Function Execution

A subprogram may have side effects when it performs one of the following operations:

- Updates its own OUT or IN OUT parameter
- Modifies a global variable
- Modifies a public variable in a package
- Updates a database table
- Modifies the database



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A function executes with or without input parameters and returns a value. A function is expected to return one value of scalar data type or composite data type. However, developers can also use OUT or IN OUT parameters, which also reflect the output of the function. Functions can update global variables, perform update and other DML operations on the database.

If the function definition has any of the operations mentioned in the slide, then those updates and modifications are side effects of the function execution besides the value returned.

These side effects might sometimes produce undesirable effects in the database. Execution of such functions may result in inconsistent data in the database or an error. Therefore, it is essential to follow some guidelines while writing code for functions.

Controlling Side Effects

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

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

```
FUNCTION DML_CALL_SQL compiled
Error starting at line 127 in command:
UPDATE employees
  SET salary = dml_call_sql(2000)
 WHERE employee_id = 170
Error report:
SQL Error: ORA-04091: table ORA61.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA61.DML_CALL_SQL", line 4
04091. 00000 -  "table %S.%S is mutating, trigger/function may not see it"
*Cause:   A trigger (or a user defined plsql function that is referenced in
this statement) attempted to look at (or modify) a table that was
in the middle of being modified by the statement which fired it.
*Action: Rewrite the trigger (or function) so it does not read that table.
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The code example in the slide demonstrates that, when a function is called from an UPDATE or DELETE statement, the function cannot query or modify database tables modified by that statement. The `dml_sql_call` function is invoked in the UPDATE statement on the EMPLOYEES table and the function definition has an INSERT statement on the EMPLOYEES table.

You can see that `dml_call_sql` function in the slide contains an INSERT statement that inserts a new record into the EMPLOYEES table and returns the input salary value incremented by 100. This function is invoked in the UPDATE statement that modifies the salary of employee 170 to the amount returned from the function. The UPDATE statement fails with an error indicating that the table is mutating.

Therefore, a function invoked in an UPDATE statement on a table, which in turn has DML statements for the same table, would lead to an error.

Guidelines to Control Side Effects

Functions called from:

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide discusses the guidelines to be followed to avoid undesirable side effects of a function execution.

Additional restrictions apply when a function is called in expressions of SQL statements:

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

Lesson Agenda

- Creating and Invoking functions
- Functions in SQL expressions
- Passing parameters to functions
- Removing stored functions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Passing Parameters to Functions

Functions accept :

- Named parameters
- Positional parameters
- Mixed parameters



Let us see a variant of the tax function. This new function would accept the `employee_id` and nontaxable allowances as input parameters and calculate the tax applicable for the employee. The nontaxable allowances per month has to be deducted from the salary and tax has to be calculated on the remaining amount.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL allows arguments in a subroutine call to be specified using positional, named, or mixed notation.

Named and Mixed Notation from SQL: Example

```
CREATE OR REPLACE FUNCTION tax_new(p_id IN employees.employee_id%type,
p_allowances NUMBER default 500)
RETURN NUMBER IS
v_sal employees.salary%type;
BEGIN
    select salary into v_sal
    from employees
    where employee_id = p_id;
    v_sal := v_sal-p_allowances;
    RETURN (v_sal * 0.08);
END tax_new;
```

```
EXECUTE DBMS_OUTPUT.PUT_LINE(tax_new(100));
EXECUTE DBMS_OUTPUT.PUT_LINE(tax_new(100,1500));
EXECUTE DBMS_OUTPUT.PUT_LINE(tax_new(p_allowances =>1500, p_id=>100));
EXECUTE DBMS_OUTPUT.PUT_LINE(tax_new(p_id=>100));
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows a tax calculation function which accepts two parameters as input. The input parameters `p_id` and `p_allowances` represent the `employee_id` and allowances payable to the employee, respectively.

The second code box shows various valid function calls for the function `tax_new`.

The first function call `tax_new(100)` uses positional parameter, where the first parameter is the `employee_id`. The `p_allowances` parameter is substituted with the default value of 500.

The second function call also uses positional parameters where the first value represents the `employee_id` and the second value represents the allowances for the employee.

The third function call uses named parameters where the values of the parameters are mentioned along with the names.

The fourth function call has the parameter `employee_id` with the named notation and the parameter `p_allowances` is provided through the default value.

Viewing Functions Using Data Dictionary Views

DESCRIBE USER_SOURCE

Name	Null	Type
NAME		VARCHAR2(128)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)
ORIGIN_CON_ID		NUMBER

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

```
Query Result X
SQL | Fetched 50 rows in 0.016 seconds
TEXT
1 FUNCTION tax_new(p_id IN employees.employee_id%type,
2 FUNCTION dml_call_sql(p_sal NUMBER)
3 FUNCTION tax(p_id IN employees.employee_id%type)
4 FUNCTION get_sal
5 FUNCTION f(
```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

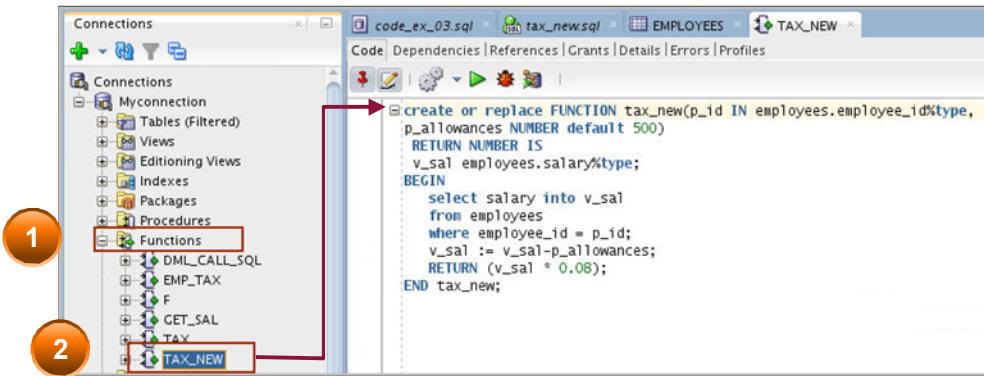
The query example shows all the columns provided by the `USER_SOURCE` table:

- The `NAME` column holds the name of the subprogram in uppercase text.
- The `TYPE` column holds the subprogram type, such as `PROCEDURE` or `FUNCTION`.
- The `LINE` column stores the line number for each source code line.
- The `TEXT` column holds PL/SQL source code.
- The `ORIGIN_CON_ID` holds the ID of the container where the data originates. The value of the container ID can range from 0 - n based on database configuration.

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

You can also use the `USER_OBJECTS` data dictionary view to display a list of your function names.

Viewing Functions Information Using SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To view a function's code in SQL Developer, perform the following steps:

1. Click the **Functions** node in the **Connections** tab.
2. Click the function's name.
3. The function's code is displayed in the **Code** tab as shown in the slide.

Lesson Agenda

- Creating and Invoking functions
- Functions in SQL expressions
- Passing parameters to functions
- Removing stored functions



ORACLE®

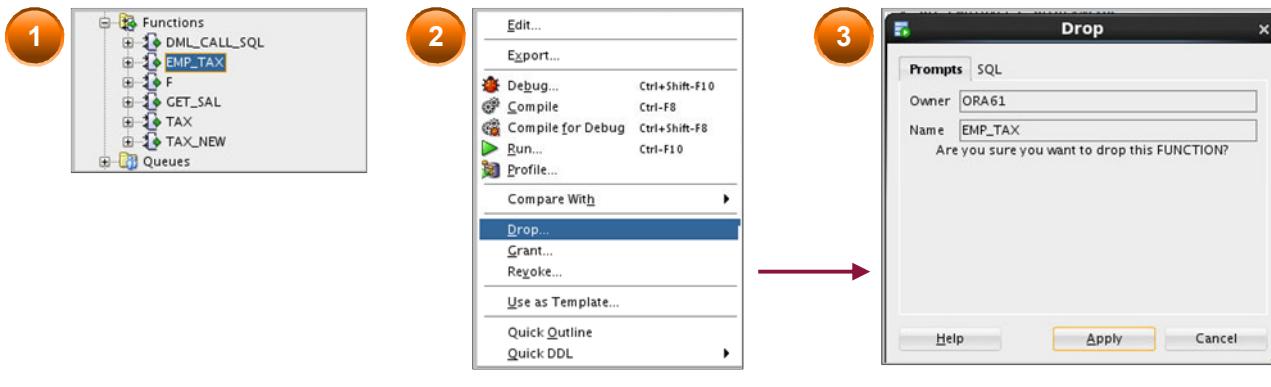
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Removing Functions: Using the DROP SQL Statement or SQL Developer

- Using the DROP statement:

```
DROP FUNCTION function_name;
```

- Using SQL Developer:



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using the `DROP` statement

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

Using `CREATE OR REPLACE` Versus `DROP` and `CREATE`

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

Using SQL Developer

To drop a function in SQL Developer, right-click the function name in the **Functions** node, and then select **Drop**. The **Drop** dialog box is displayed. To drop the function, click **Apply**.

Quiz



Which of the following statements are true about a PL/SQL stored function?

- a. Can be invoked as part of an expression
- b. Must contain a RETURN clause in the header
- c. Must return a single value
- d. Must contain at least one RETURN statement
- e. Does not contain a RETURN clause in the header



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, c, d

Practice 3-1: Overview

This practice covers the following topics:

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

It is recommended to use SQL Developer for this practice.

Summary

In this lesson, you should have learned how to:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Explain the basic functionality of the SQL Developer debugger



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

A function can be created or dropped.

A function is invoked as a part of an expression.



4

Debugging Subprograms

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

- ▶ Lesson 2: Creating Procedures
- ▶ Lesson 3: Creating Functions
- ▶ Lesson 4: Debugging Subprograms**
- ▶ Lesson 5: Creating Packages
- ▶ Lesson 6: Working with Packages
- ▶ Lesson 7: Using Oracle-Supplied Packages in Application Development
- ▶ Lesson 8: Using Dynamic SQL

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Working with Subprograms, Working with Triggers, and Working with PL/SQL Code.

In the first unit, we have seven lessons: Creating Stored Procedures, Creating Functions ,Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms, packages and use them in Application Development.

Objectives

After completing this lesson, you should be able to:

- Describe the functionality of the SQL Developer debugger
- Debug Subprograms
- Use various features of SQL Developer debugger for debugging



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Lesson Agenda

- Before debugging PL/SQL subprograms
- Debugging a subprogram
- Using the debugging Log Tab toolbar
- Debugging a procedure



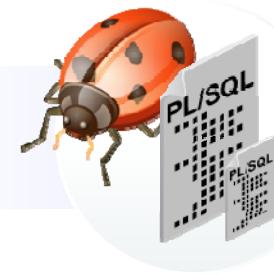
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Before Debugging PL/SQL Subprograms

Before you start debugging PL/SQL Subprograms, you should have the following privileges:

- DEBUG CONNECT SESSION privilege
- DEBUG ANY PROCEDURE privilege
- EXECUTE privilege on DBMS_DEBUG_JDWP
- EXECUTE privilege on the object you want to debug



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Debugging is the process of locating and removing any defects in the program code. To debug a PL/SQL Subprogram you (user of the schema through which you have logged in) should have certain privileges mentioned in the slide.

You can look at the privileges for the current user by executing the following query :

```
select * from user_sys_privs;
```

Before Debugging PL/SQL Subprograms

You can look at the privileges of the current user by executing the following statement:

```
SELECT * from user_sys_privs;
```

Query Result	
All Rows Fetched: 2 in 0.003 second	
USERNAME	PRIVILEGE
1 ORA61	CREATE JOB
2 ORA61	CREATE TABLE

ORA61 does not have DEBUG CONNECT SESSION and DEBUG ANY PROCEDURE privileges. You can grant them by executing the following statements as a SYSDBA role user.

```
GRANT DEBUG ANY PROCEDURE TO ora61;
GRANT DEBUG CONNECT SESSION TO ora61;
-- Login as ora61
select * from user_sys_privs;
```

Query Result	
All Rows Fetched: 4 in 0.003 second	
USERNAME	PRIVILEGE
1 ORA61	CREATE JOB
2 ORA61	DEBUG ANY PROCEDURE
3 ORA61	DEBUG CONNECT SESSION
4 ORA61	CREATE TABLE



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The process of granting privileges is demonstrated in the slide. You query the existing privileges using the query in the first code box.

Note: To grant privileges to a user, you have to be a SYSDBA role user.

To grant the required privileges to ora61, log in as a SYSDBA role user and run the GRANT statements in the second code box.

To see the current set of privileges, log in as ora61 and then run the query:

```
select * from user_sys_privs;
```

Before Debugging PL/SQL Subprograms

You might encounter the following error while running the SQL debugger in Oracle 12c.



```
Debugging: IdeConnections%23Myconnection.jpr - Log
Connecting to the database Myconnection.
Executing PL/SQL: ALTER SESSION SET PLSQL_DEBUG=TRUE
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.CONNECT_TCP( '127.0.0.1', '4000' )
ORA-24247: network access denied by access control list (ACL)
ORA-06512: at "SYS.DBMS_DEBUG_JDWP", line 68
ORA-06512: at line 1
Process exited.
Disconnecting from the database Myconnection.
```

The error implies that the current database session is trying to establish a connection with the debugger on the local host, but not able to do so.

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

While debugging in Oracle 12c, in spite of having the DEBUG CONNECT SESSION privilege, you might encounter the error mentioned in the slide. This error occurs when the database session tries to connect to the debugger on the localhost. Your current database session should have the jdwp ACL privilege to successfully connect to the debugger.

SQL developer uses Java Debug Wire Protocol (JDWP)-based debugger to debug an Oracle 12c PL/SQL subprogram. Debugger generally resides on the local host. Your database session should connect to the debugger on the local host to start the debugging process. You must be granted the jdwp ACL privilege to connect to the debugger on the local host.

Before Debugging PL/SQL Subprograms

You have to execute the following PL/SQL block to grant the jdwp ACL privilege as a SYSDBA role user.

```
BEGIN
  DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE
  (
    host => '127.0.0.1',
    lower_port => null,
    upper_port => null,
    ace => xs$ace_type(privilege_list => xs$name_list('jdwp'),
    principal_name => 'ora61',
    principal_type => xs_acl.ptype_db)
  );
END;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

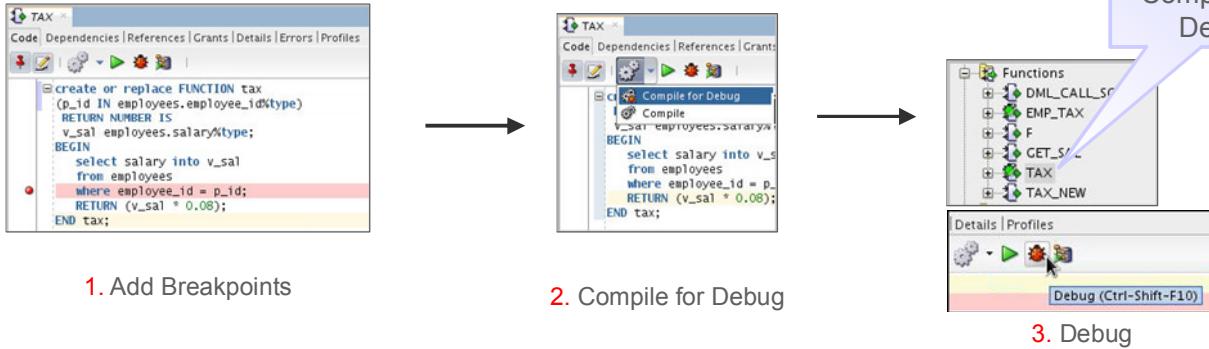
- Before debugging PL/SQL subprograms
- Debugging a subprogram
- Using the debugging Log Tab toolbar
- Debugging a procedure



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Debugging a Subprogram: Overview



ORACLE®

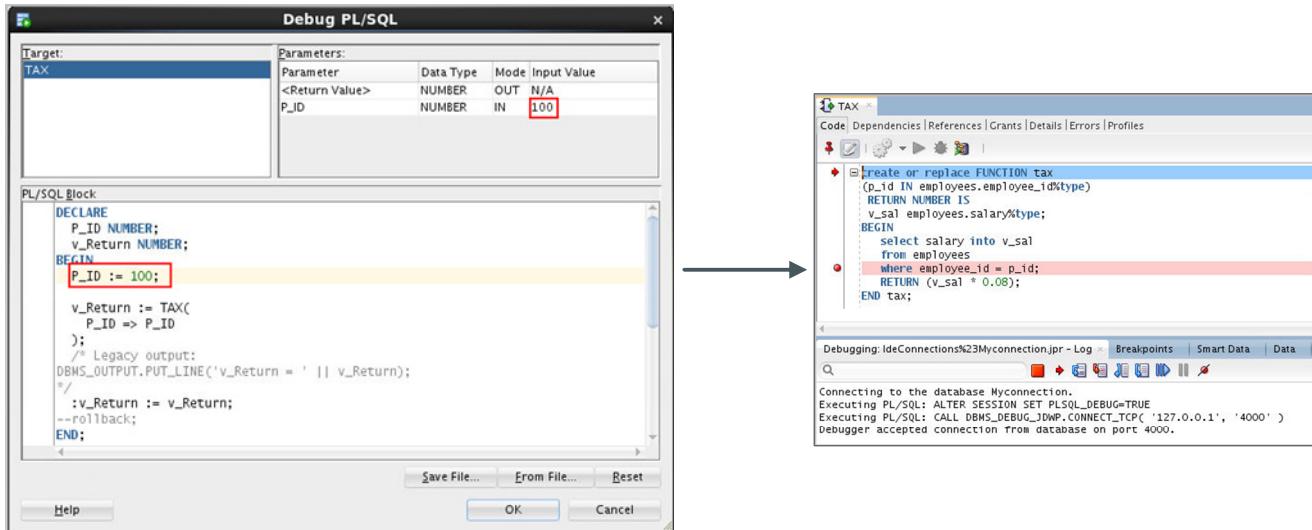
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The process of debugging starts by adding breakpoints. Breakpoint is added at a statement where the developer wants to inspect various parameters of the code or values of different variables in the code. Identify such statements in your code and add breakpoints.

When the debugging process reaches the breakpoint, you can use Data and Smart Data tabs in SQL Developer to inspect the state of various variables and data objects during the program execution.

After adding breakpoints, compile the code through the Compile for Debug option.

Debugging a Subprogram: Overview



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When you start debugging the subprogram, the debugger creates an anonymous PL/SQL block which invokes the function "tax" from an anonymous PL/SQL block.

As you see in the slide the wizard has a PL/SQL block which invokes the function "tax" from an anonymous PL/SQL block.

You can modify the value of the input variables accepted by the function in this wizard. In the slide the value of the variable `p_id` is set to 100.

The debug process starts once you click OK in the wizard.

Lesson Agenda

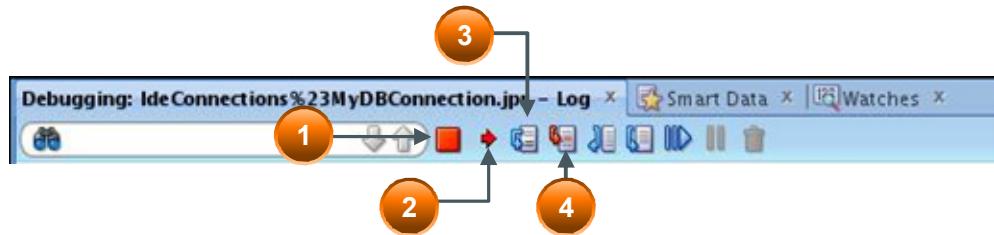
- Before debugging PL/SQL subprograms
- Debugging a subprogram
- Using the debugging Log Tab toolbar
- Debugging a procedure



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The Debugging – Log Tab Toolbar



Icon	Description
1. Terminate	Halts and exits the execution
2. Find Execution Point	Goes to the next execution point
3. Step Over	Bypasses the execution of statements of a subprogram invoked from the current subprogram and executes the statement after the subprogram call
4. Step Into	Executes a single program statement at a time. If the execution point is located on a call to a subprogram, it steps into the first statement in that subprogram.

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

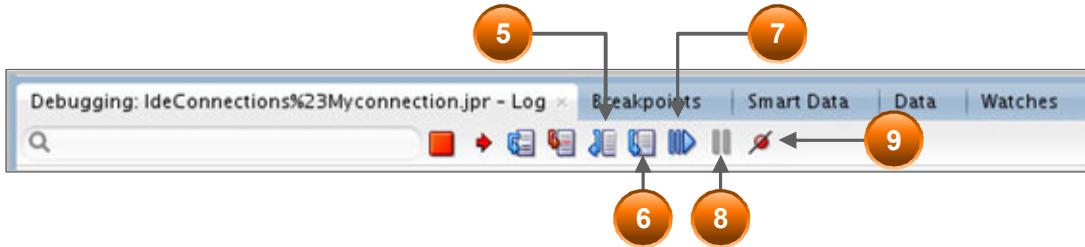
After the debugging process starts, the debug toolbar shown in the slide appears in the SQL developer.

The Debugging Log has all the informational messages and the following tabs:

1. **Terminate:** Halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the subprogram, click the **Run** or **Debug** icon in the Source tab toolbar.
2. **Find Execution Point:** Goes to the execution point (the next line of source code to be executed by the debugger)
3. **Step Over:** When there is a subprogram call, **Step Over** skips entering the subprogram block. The next statement to be executed will be the statement following the subprogram call. So the debugger steps over debugging the subprogram body which is invoked from the subprogram currently being executed.
4. **Step Into:** Executes a single program statement at a time. If the execution point is located on a call to a subprogram, **Step Into** steps into that subprogram and places the execution point on the first statement of the invoked subprogram.

Note: The functionality of **Step Over** and **Step Into** would effectively be the same when there are no subprogram calls in the current subprogram.

The Debugging – Log Tab Toolbar



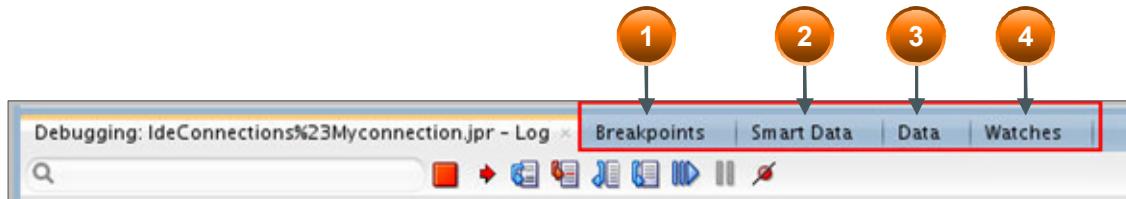
Icon	Description
5. Step Out	Leaves the current subprogram and goes to the next statement with a breakpoint
6. Step to End of Method	Goes to the last statement of the current subprogram
7. Resume	Continues execution
8. Pause	Halts execution but does not exit
9. Suspend all breakpoints	Turns off all the breakpoints in the current PL/SQL block

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

5. **Step Out:** Leaves the current subprogram and goes to the next executable statement
6. **Step to End of Method:** Goes to the last statement of the current subprogram
7. **Resume:** Continues execution
8. **Pause:** Halts execution but does not exit, thus allowing you to resume execution
9. **Suspend all breakpoints:** Renders all the breakpoints defined in the subprogram ineffective

Tracking Data and Execution



Tab	Description
1. Breakpoints	It displays breakpoints, both system defined and user defined.
2. Smart Data	Displays information about variables. You can specify these preferences by right-clicking in the Smart Data window and selecting Preferences.
3. Data	You can see the data in all the variables.
4. Watches	You can monitor all the watches set while debugging.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can monitor how the data is changing while executing the subprogram through the tabs: Breakpoints, Data, Smart Data, and Watches.

Breakpoints: Breakpoints are defined by the developer while debugging. The execution of the subprogram stops at the breakpoint and allows the developer to inspect data or modify certain data if required. After performing the data inspection, developer can start the debugging process by clicking on Step Into or Step Over.

Smart Data: Smart Data tab displays information about the variables in the subprogram. SQL Developer allows to modify the data here enabling the developer to try all the possibilities.

Data: The Data tab has all the values of the variables used in the subprogram being debugged.

Watches: A watch enables you to monitor the changing values of variables as the subprogram executes. You can create a Watch expression in the Watches tab based on the values of variables used in the subprogram.

Lesson Agenda

- Before debugging PL/SQL subprograms
- Debugging a subprogram
- Using the debugging Log Tab toolbar
- Debugging a procedure



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Debugging a Procedure Example: Creating a New emp_list Procedure

```

1 CREATE OR REPLACE PROCEDURE emp_list(pmaxrows IN NUMBER) AS
2 CURSOR emp_cursor IS
3 SELECT d.department_name,
4       e.employee_id,
5       e.last_name,
6       e.salary,
7       e.commission_pct
8 FROM departments d,
9      employees e
10 WHERE d.department_id = e.department_id;
11 emp_record emp_cursor % rowtype;
12 type emp_tab_type IS TABLE OF emp_cursor % rowtype INDEX BY binary_integer;
13 emp_tab emp_tab_type;
14 i NUMBER := 1;
15 v_city VARCHAR2(30);
16 BEGIN
17
18   OPEN emp_cursor;
19   FETCH emp_cursor
20   INTO emp_record;
21   emp_tab(i) := emp_record;
22   WHILE(emp_cursor % FOUND)
23     AND (i <= pmaxrows)
24   LOOP
25     i := i + 1;
26     FETCH emp_cursor
27     INTO emp_record;
28     emp_tab(i) := emp_record;
29     v_city := get_location(emp_record.department_name);
30     DBMS_OUTPUT.PUT_LINE('Employee ' || emp_record.last_name || ' works in ' || v_city);
31   END LOOP;
32
33   CLOSE emp_cursor;
34   FOR j IN REVERSE 1 .. i
35   LOOP
36     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
37   END LOOP;
38 END emp_list;

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Now let us look at the debugging process of the procedure `emp_list`. The procedure retrieves the employee details such as `employee_id`, `last_name`, `salary`, and `commission_pct` from the `EMPLOYEES` table along with the department name in which the employee works in the `DEPARTMENT` table. The procedure also has a function call to retrieve the location of the employee based on the department in which the employee works. All these details are retrieved into a record called `emp_record`, these records are later stored in a table `emp_tab`. And finally displayed in a for loop.

Let us see the process of debugging this procedure.

This procedure has a cursor, function call, and for loop. A developer would like to set a breakpoint at these statements because they are some likely locations of error in the subprogram.

Note: Make sure that you are displaying the procedure code in edit mode. To edit the procedure code, click the Edit icon on the procedure's toolbar.

Debugging a Procedure Example: Creating a New get_location Function

```
1 CREATE OR REPLACE FUNCTION get_location(p_deptname IN VARCHAR2) RETURN VARCHAR2 AS
2   v_loc_id NUMBER;
3   v_city VARCHAR2(30);
4   BEGIN
5     SELECT d.location_id,
6           l.city
7     INTO v_loc_id,
8          v_city
9    FROM departments d,
10         locations l
11   WHERE UPPER(department_name) = UPPER(p_deptname)
12     AND d.location_id = l.location_id;
13   RETURN v_city;
14 END get_location;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This function returns the city in which an employee works. It is called from the `emp_list` procedure.

Setting Breakpoints and Compiling emp_list for Debug Mode

```

1 create or replace
2 PROCEDURE emp_list
3 (p_maxrows IN NUMBER)
4 IS
5 CURSOR cur_emp IS
6   SELECT d.department_name, e.employee_id, e.last_name,
7         e.salary, e.commission_pct
8   FROM department d, employees e
9   WHERE d.department_id = e.department_id;
10  TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11  emp_tab emp_tab_type;
12  i NUMBER := 1;
13  v_city VARCHAR2(30);
14
15 BEGIN
16   OPEN cur_emp;
17   FETCH cur_emp INTO rec_emp;
18   emp_tab(i) := rec_emp;
19   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20     i := i + 1;
21     FETCH cur_emp INTO rec_emp;
22     emp_tab(i) := rec_emp;
23     v_city := get_location(rec_emp.department_name);
24     dbms_output.put_line('Employee ' || rec_emp.last_name ||
25                           ' works in ' || v_city);
26   END LOOP;
27   CLOSE cur_emp;
28   FOR j IN REVERSE 1..i LOOP
29     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30   END LOOP;
31 END emp_list;

```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows adding breakpoints for the cursor, function call, and the loop statement. You generally add a break point by clicking the appropriate line number.

Before debugging, it is essential to **Compile for Debug**. This will add additional overhead which enables debugging process.

The slide shows the emp_list procedure displayed in edit mode, and four breakpoints added to various locations in the code. To compile the procedure for debugging, right-click the code, and then select **Compile for Debug** from the shortcut menu. The **Messages – Log** tab displays the message that the procedure was compiled.

Compiling the get_location Function for Debug Mode

The screenshot shows the Oracle SQL Developer interface. A PL/SQL function named 'GET_LOCATION' is open in the editor. The code is as follows:

```
1 create or replace
2 FUNCTION get_location
3   ( p_deptname IN VARCHAR2 ) RETURN VARCHAR2;
4   AS
5     v_loc_id NUMBER;
6     v_city   VARCHAR2(30);
7   BEGIN
8     SELECT d.location_id, l.city INTO v_loc_id, v_city
9     FROM departments d, locations l
10    WHERE upper(department_name) = upper(p_deptname)
11      and d.location_id = l.location_id;
12    RETURN v_city;
13  END GET_LOCATION;
```

A right-click context menu is displayed over the code, with the 'Compile for Debug' option highlighted. The status bar at the bottom shows the message 'Compiled'.

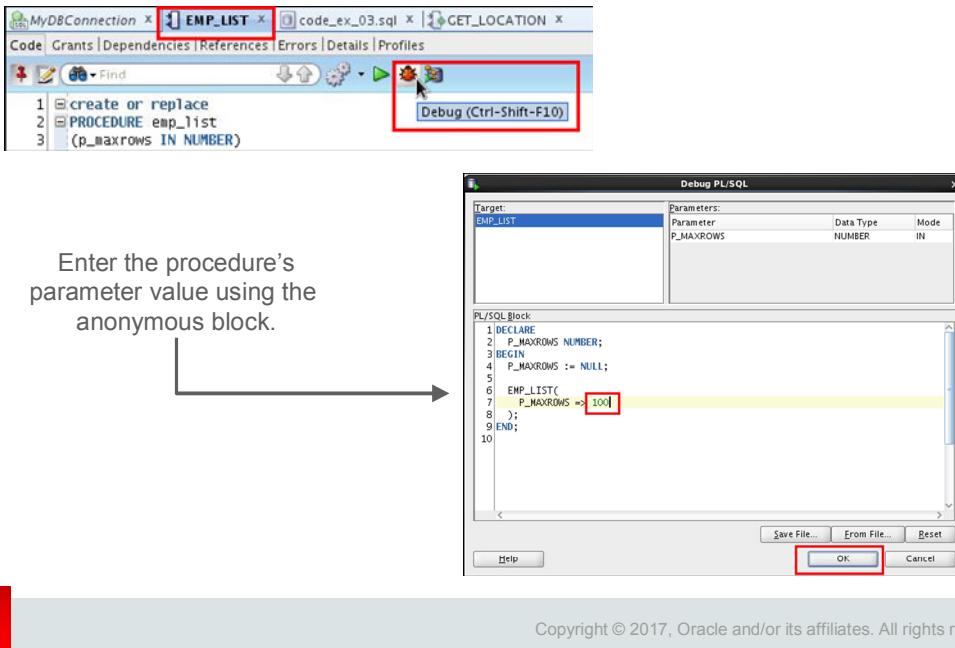
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You should also compile the user-defined functions invoked from the subprogram you intend to debug. Therefore, you should debug the `get_location` function also.

In the slide, the `get_location` function is displayed in edit mode. To compile the function for debugging, right-click the code, and then select **Compile for Debug** from the shortcut menu. The **Messages – Log** tab displays the message that the function was compiled.

Debugging emp_list and Entering Values for the PMAXROWS Parameter



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

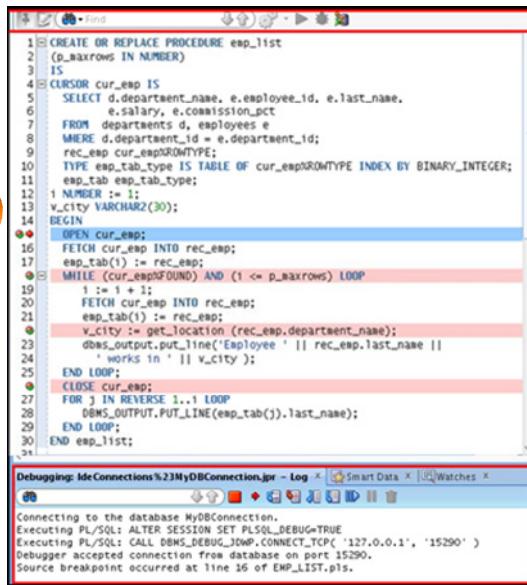
The next step in the debugging process is to debug the procedure using any of the several available methods mentioned earlier, such as clicking the **Debug** icon on the procedure's toolbar. When you start the debugging process an anonymous block is displayed, where you are prompted to enter the parameters for this procedure. `emp_list` has one parameter, `PMAXROWS`, which specifies the number of records to return. Replace the second `PMAXROWS` with a number such as 100, and then click **OK**.

After you start the debugging process, you can use **Step Into** or **Step Over** to debug the subprogram step by step.

Note: Use **Step Over** to avoid entering the called function subprogram from the currently executing subprogram.

Debugging emp_list: Step Into (F7) the Code

Program control stops at first breakpoint.



The screenshot shows the Oracle SQL Developer interface. In the PL/SQL Editor, a procedure named emp_list is displayed. A red circle with the number '1' is placed over the first line of code, indicating the current execution point. The code includes a cursor declaration, a loop that fetches records from the cursor, and a loop that prints the results. In the Log window below, connection details and the accepted debugger connection are shown.

```

1 CREATE OR REPLACE PROCEDURE emp_list
2   (p_maxrows IN NUMBER)
3 IS
4   CURSOR cur_dept IS
5     SELECT d.department_name, e.employee_id, e.last_name,
6           e.salary, e.commission_pct
7     FROM departments d, employees e
8    WHERE d.department_id = e.department_id;
9   rec_emp cur_dept%TYPE;
10  TYPE emp_tab%TYPE IS TABLE OF cur_dept%ROWTYPE INDEX BY BINARY_INTEGER;
11  emp_tab emp_tab%TYPE;
12  i NUMBER := 1;
13  v_city VARCHAR2(30);
14  BEGIN
15    OPEN cur_dept;
16    FETCH cur_dept INTO rec_emp;
17    emp_tab(1) := rec_emp;
18    WHILE (cur_dept%FOUND) AND (1 <= p_maxrows) LOOP
19      i := i + 1;
20      FETCH cur_dept INTO rec_emp;
21      emp_tab(i) := rec_emp;
22      v_city := get_location(rec_emp.department_name);
23      dbms_output.put_line('Employee ' || rec_emp.last_name ||
24                           ' works in ' || v_city );
25    END LOOP;
26    CLOSE cur_dept;
27    FOR j IN REVERSE 1..i LOOP
28      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
29    END LOOP;
30  END emp_list;
31

```

Debugging: IdeConnections%23MyDBConnection.jpr - Log X Smart Data X Watches X

Connecting to the database MyDBConnection.
Executing PL/SQL: ALTER SESSION SET PLSQL_DEBUG=TRUE
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.CONNECT_TCP('127.0.0.1', '15290')
Debugger accepted connection from database on port 15290.
Source breakpoint occurred at line 16 of EMP_LIST.pls.

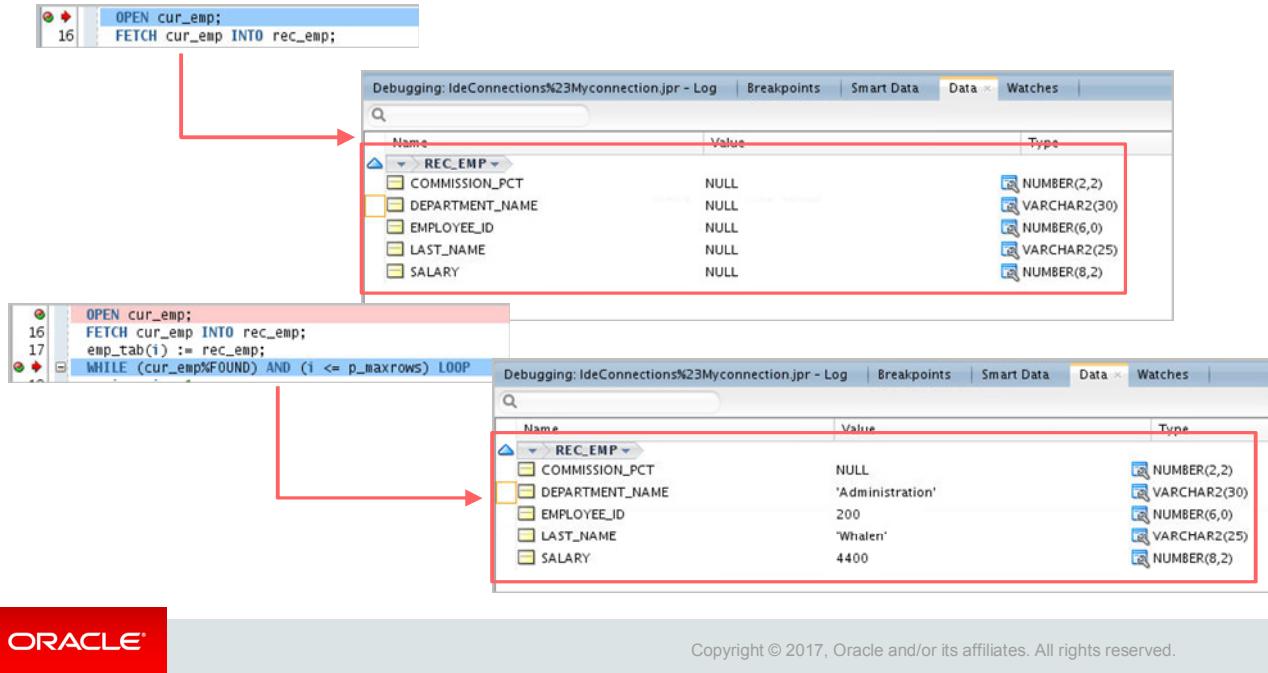


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide demonstrates the debugging process. The slide shows that the program control is stopped at the first breakpoint in the code. The arrow next to the breakpoint indicated that this is the line of code that will be executed next. Note the various tabs at the bottom of the window.

At this breakpoint, you can inspect different values of the data and monitor the expression watches set by navigating through the **Smart Data**, **Data**, and **Watches** tabs.

Viewing the Data

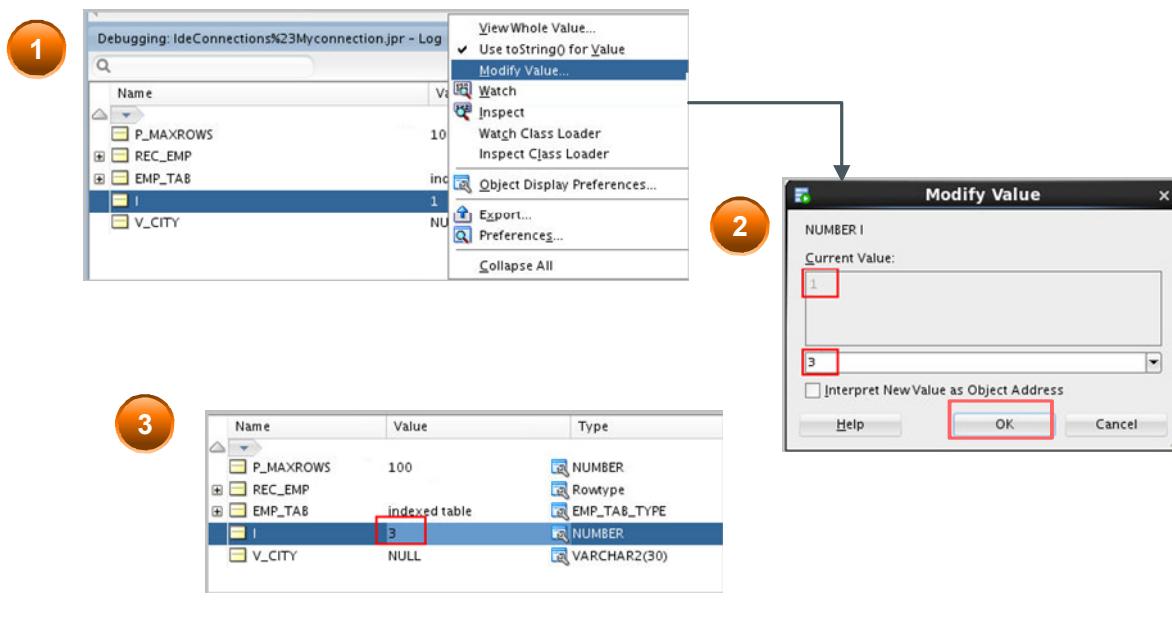


ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

While you are debugging your code, you can use the **Data** tab to display and modify the variables. You can also set watches to monitor a subset of the variables displayed in the **Data** tab. To display or hide the **Data**, **Smart Data**, and **Watch** tabs, select **View > Debugger**, and then select the tabs that you want to display or hide.

Modifying the Variables While Debugging the Code



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To modify the value of a variable in the **Data** tab, right-click the variable name, and then select **Modify Value** from the shortcut menu. The **Modify Value** window is displayed with the current value for the variable. You can enter a new value in the second text box, and then click **OK**.

You can see in the slide that the value of variable `i` is modified from 1 to 3.

Debugging emp_list: Step Over Versus Step Into

The diagram illustrates the difference between the **Step Into** and **Step Over** processes in a debugger. On the left, a code editor shows a PL/SQL block with several statements highlighted in pink. A red arrow points from the first highlighted statement (line 16) to a separate window titled "Step Into" containing the definition of the `get_location` function. Below this, another red arrow points from the same highlighted statement to a second code editor window titled "Step Over", which shows the code with the `get_location` call still highlighted.

```

10  TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11  emp_tab emp_tab_type;
12  i NUMBER := 1;
13  v_city VARCHAR2(30);
14  BEGIN
15      OPEN cur_emp;
16      FETCH cur_emp INTO rec_emp;
17      emp_tab(i) := rec_emp;
18      WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19          i := i + 1;
20          FETCH cur_emp INTO rec_emp;
21          emp_tab(i) := rec_emp;
22          v_city := get_location (rec_emp.department_name);
23          dbms_output.put_line('Employee ' || rec_emp.last_name ||
24              ' works in ' || v_city );

```

```

1  create or replace FUNCTION get_location
2  ( p_deptname IN VARCHAR2 ) RETURN VARCHAR2
3  AS
4      v_loc_id NUMBER;
5      v_city    VARCHAR2(30);
6  BEGIN
7      SELECT d.location_id, l.city INTO v_loc_id, v_city
8      FROM departments d, locations l
9      WHERE upper(department_name) = upper(p_deptname)
10         and d.location_id = l.location_id;
11      RETURN v_city;
12  END GET_LOCATION;

```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can appreciate the difference between **Step Into** and **Step Over** process of debugger at code line number 21.

We have a function call `get_location` at line 21. At this point when you use Step Into, the debugging control shifts to the first line of the function definition of the `get_location` function. Instead of Step Into if you use Step Over, the debug control will remain in the current sub program. The control will be transferred to next statement (line 22) in the `emp_list` subprogram.

Debugging emp_list: Step Out of the Code (Shift + F7)

The diagram illustrates the 'Step Out' operation. On the left, a code editor window shows the `get_location` function. A red arrow labeled 'Step Out' points from the bottom of the `get_location` window to the right. On the right, another code editor window shows the `emp_list` procedure. The cursor is positioned at line 21, which contains the `get_location` function call. The line is highlighted in pink, and the number 21 is in a red box.

```

1 ◆ create or replace FUNCTION get_location
2   ( p_deptname IN VARCHAR2) RETURN VARCHAR2
3   AS
4     v_loc_id NUMBER;
5     v_city  VARCHAR2(30);
6   BEGIN
7     SELECT d.location_id, l.city INTO v_loc_id, v_city
8     FROM departments d, locations l
9     WHERE upper(department_name) = upper(p_deptname)
10    and d.location_id = l.location_id;
11
12   RETURN v_city;
13 END GET_LOCATION;

```

```

19 WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20   i := i + 1;
21   FETCH cur_emp INTO rec_emp;
22   emp_tab(i) := rec_emp;
23   v_city := get_location (rec_emp.department_name);
24   dbms_output.put_line('Employee ' || rec_emp.last_name ||
                           ' works in ' || v_city );

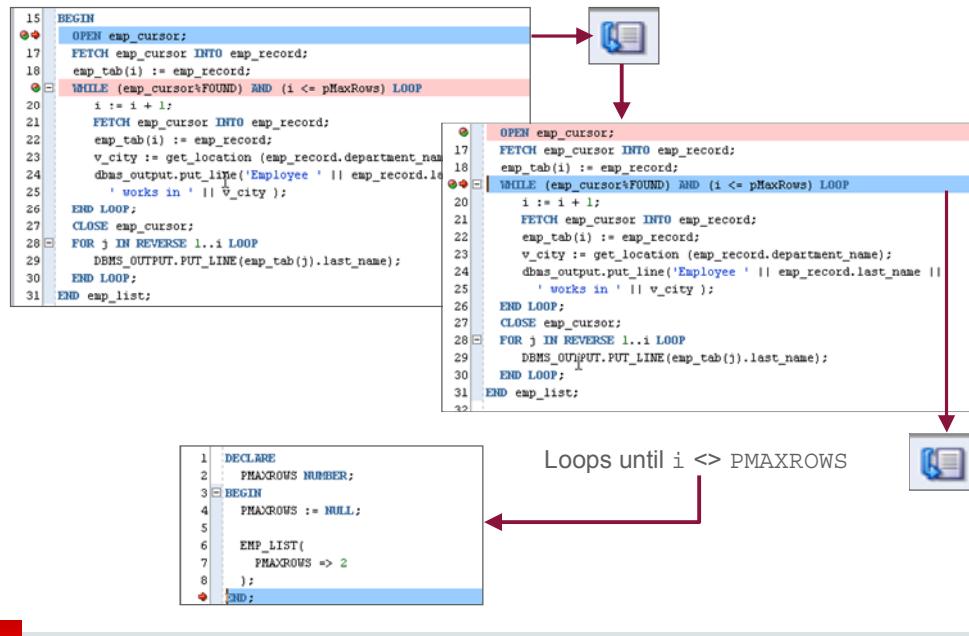
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To look at how **Step Out** works, let us consider that currently we are debugging the first line of the `get_location` function. When we **Step Out** control here, debugging of the entire `get_location` function is skipped and the control moves to the statement after the function call in the `emp_list` procedure.

Debugging emp_list: Step to End of Method



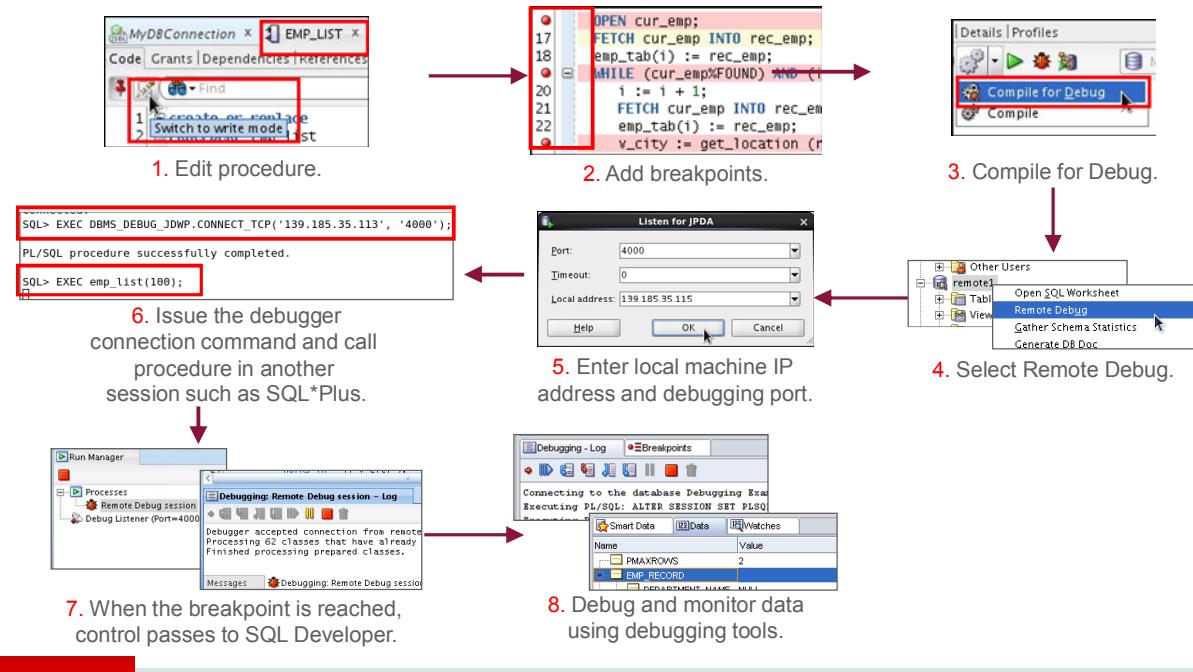
ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Step to End of Method goes to the last statement in the current subprogram or to the next breakpoint if there are any in the current subprogram.

In the example in the slide, the **Step to End** of Method of debugging tool is used. Because there is a second breakpoint, selecting **Step to End** of Method transfers control to that breakpoint. Selecting **Step to End** of Method again goes through the iterations of the while loop first, and then transfers the program control to the next executable statement in the anonymous block.

Debugging a Subprogram Remotely: Overview



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use remote debugging to connect to any PL/SQL code in a database and debug it, regardless of where the database is located as long as you have access to the database and have created a connection to it. Remote debugging is when something else, other than you as a developer, kicks off a procedure that you can then debug. Remote debugging and local debugging have many steps in common. To debug remotely, perform the following steps:

1. Edit the subprogram that you would like to debug.
2. Add the breakpoints in the subprogram.
3. Click the **Compile for Debug** icon in the toolbar.
4. Right-click the connection for the remote database, and select **Remote Debug**.
5. In the **Debugger - Attach to JPDA** dialog box, enter the local machine IP address and port number, such as 4000, and then click **OK**.
6. Issue the debugger connection command using a different session such as SQL*Plus, and then call the procedure in that session.
7. When a breakpoint is reached, control is passed back to the original SQL Developer session and the debugger toolbar is displayed.
8. Debug the subprogram and monitor the data using the debugging tools discussed earlier.

Summary

In this lesson, you should have learned how to:

- Describe the basic functionality of the SQL Developer debugger
- Run the debugging process
- Use various features that can be used while debugging



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

A function can be created or dropped.

A function is invoked as a part of an expression.

Practice 4 Overview: Introduction to the SQL Developer Debugger

This practice covers the following topics:

- Creating a procedure and a function
- Inserting breakpoints in the procedure
- Compiling the procedure and function for debug mode
- Debugging the procedure and stepping into the code
- Displaying and modifying the subprograms' variables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You must use SQL Developer for this practice. This practice introduces you to the basic functionality of the SQL Developer debugger.

5

Creating Packages

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

- ▶ Lesson 2: Creating Procedures
- ▶ Lesson 3: Creating Functions
- ▶ Lesson 4: Debugging Subprograms
- ▶ Lesson 5: Creating Packages**
- ▶ Lesson 6: Working with Packages
- ▶ Lesson 7: Using Oracle-Supplied Packages in Application Development
- ▶ Lesson 8: Using Dynamic SQL

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Working with Subprograms, Working with Triggers, and Working with PL/SQL Code.

In the first unit, we have six lessons: Creating Stored Procedures, Creating Functions and Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms, packages and use them in Application Development.

Objectives

After completing this lesson, you should be able to:

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Lesson Agenda

- Identifying the benefits and the components of packages
- Working with packages:
 - Creating the package specification and body
 - Invoking the package subprograms
 - Displaying the package information
 - Removing a package



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

DBMS_OUTPUT.PUT_LINE

What is DBMS_OUTPUT?

- DBMS_OUTPUT is an Oracle-supplied package.

What is PUT_LINE?

- A procedure in the DBMS_OUTPUT package that enables displaying messages to output buffer.

DBMS_OUTPUT also have other procedures such as PUT_LINE.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You've been using DBMS_OUTPUT.PUT_LINE in your code in functions and procedures.

What does DBMS_OUTPUT.PUT_LINE mean?

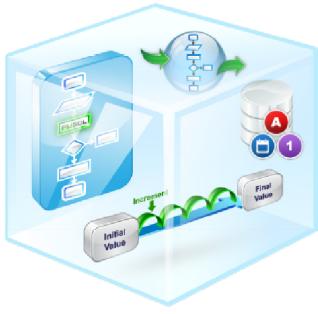
Oracle supplies many PL/SQL packages with the database to extend its functionality. You can use these packages when creating your applications. DBMS_OUTPUT is one such Oracle-supplied package and PUT_LINE is a procedure in that package. By using this procedure, you avoid writing code for displaying information onto the output buffer. Oracle provides several such packages which can be used in applications by developers.

Apart from the PUT_LINE procedure there are other procedures also defined in the DBMS_OUTPUT package. All the procedures in the DBMS_OUTPUT package are collectively used to display messages or read data from the end user or perform some other related functionality. Developers can use these procedures in their applications.

Oracle allows developers create their own packages with various procedures in it.

From the design perspective, you create a package to create a group of related procedures, functions and other program units. For instance, DBMS_OUTPUT has procedures that deal with displaying output. In your application, you can group multiple procedures performing a logical function into a package.

What Is a Package?



A package is a schema object that groups logically related PL/SQL types, variables, sequences, and subprograms.

A package is compiled and stored in the database, where many applications can share its contents.

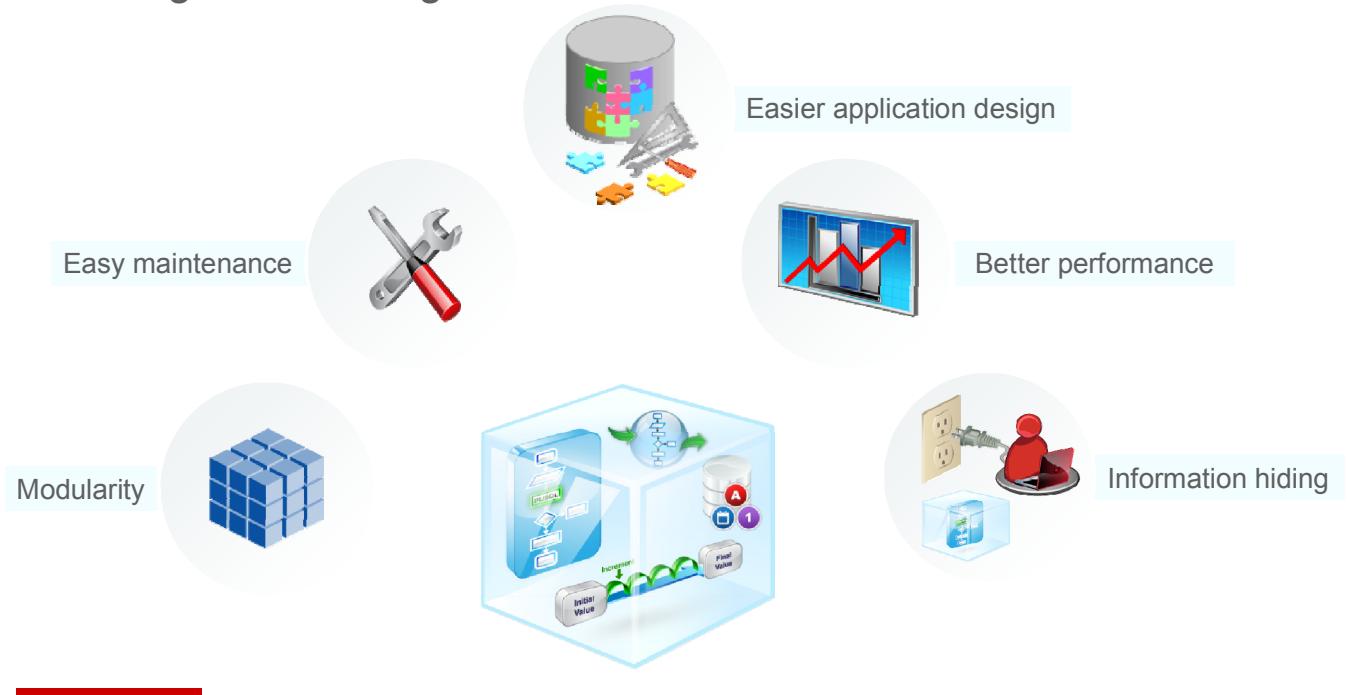
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A package is a schema object that groups all logically related objects in an application. For instance, the DBMS_OUTPUT package, which is an Oracle Supplied Package, has all the procedures and functions related to handling output in PL/SQL.

A package may contain various members such as variables, constants, cursors, exceptions, sequences, and so on.

Advantages of Packages



ORACLE®

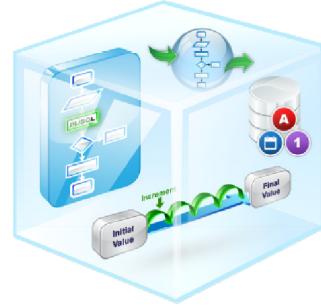
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The following are the advantages of packages:

- **Modularity:** Packages enable modularity by allowing the developer to logically group various variables, sequences, procedures in the application. You encapsulate logically related programming structures in a named module.
- **Easier maintenance:** The specification and body of the package are independently maintained. You can modify the body of the package without modifying the specification of the package. This flexibility enables better maintenance of the application.
- **Easier application design:** When designing an application, all you need is the interface information in the package specifications. You can code and compile specifications without their bodies. Next, you can compile stand-alone subprograms that reference the packages. You need not fully define the package bodies until you are ready to complete the application.
- **Information hiding:** Packages let you share your interface information in the package specification, and hide the implementation details in the package body. This enables you to change the implementation details without affecting the application interface.
- **Better performance:** The first time you invoke a package subprogram, Oracle database loads the entire package into memory. Subsequent invocations of other subprograms don't result in additional I/O.

How Do You Create PL/SQL Packages?

- Packages usually have two parts:
 - A specification (spec)
 - A body
- The specification is the interface to the package.
- The body has the implementation.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

If you are a developer creating a `HR` package, then the package may have a sequence for generating `employee_id` (which has a requirement of being unique in the company) for a new employee, a variable which provides the head count of employees in the company, a procedure to calculate the tax applicable to an employee and so on.

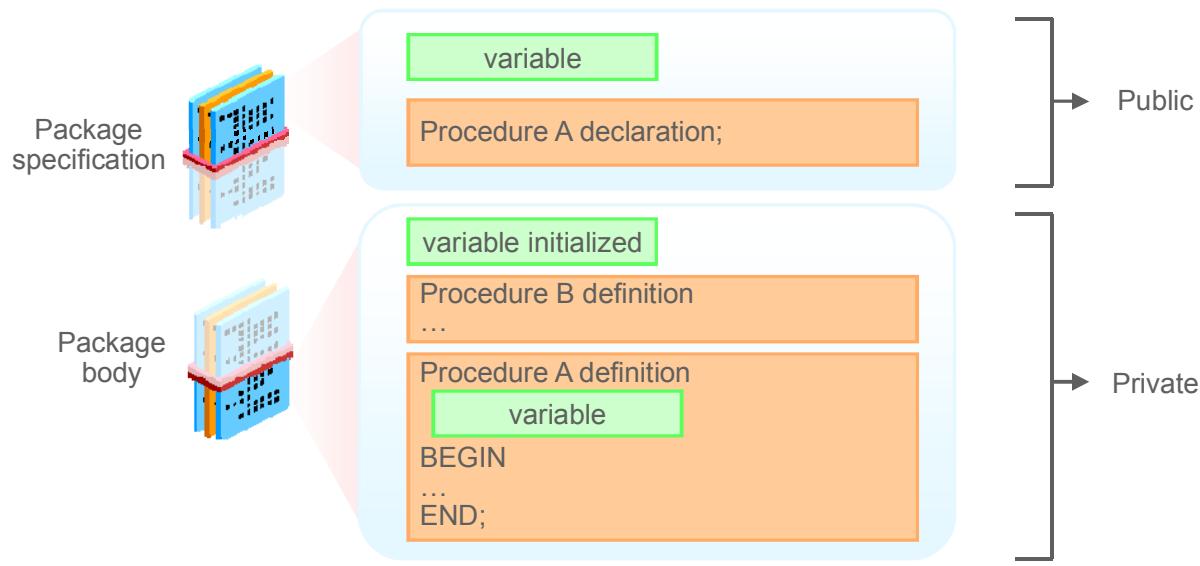
A package usually consists of two parts stored separately in the database:

- A specification
- A body

Package specification is declarative, where you declare the types, variables, constants, exceptions, declared cursors, and subprograms that can be referenced from outside the package. A package specification is an **application program interface (API)**: it has all the information that client programs need to invoke its subprograms. The package specification is about what is in the package for the client.

The package body is about implementation of the declarations in the package specification. You can also define local subprograms in the package body which aren't declared in the package specification. These local subprograms are hidden from client programs.

Components of a PL/SQL Package



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You create a package in two parts:

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

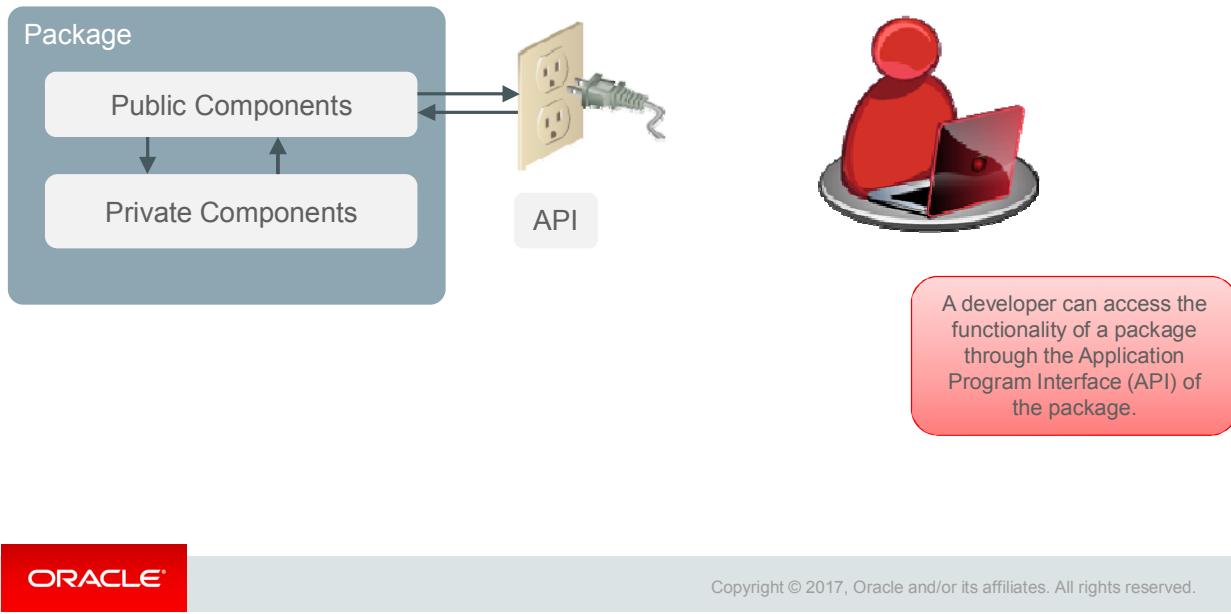
When you populate the package with variables, procedures, and other components, you can define whether the component can be accessed by the client or other external components by specifying them to be **public** or **private**.

When certain package member is declared as public, then those package members can be accessed by entities outside the package. All the members declared in package specification are public.

When certain package members are declared as private, then those package members can only be accessed by other package members. Private package members can't be accessed by components external to the package.

In the slide, you can see that the package specification has two members, a variable and procedure A. However in the package body, you see that there is definition of procedure A and procedure B. Procedure A is public and, therefore, can be accessed by external components, whereas procedure B is private and, therefore, can only be accessed by procedure A or other components within the package.

Application Program Interface



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The **Application Programming Interface (API)** comprises of all the public components of the application.

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

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

Note: If a package specification doesn't contain subprogram declarations, there is no requirement for a package body.

A client can access the package functionality by invoking procedures or functions in the package. The implementation of the package procedures and functions is hidden from the client.

Lesson Agenda

- Identifying the benefits and the components of packages
- Working with packages:
 - Creating the package specification and body
 - Invoking the package subprograms
 - Displaying the package information
 - Removing a package



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Creating the Package Specification: Using the CREATE PACKAGE Statement

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

The following are definitions of items in the package syntax:

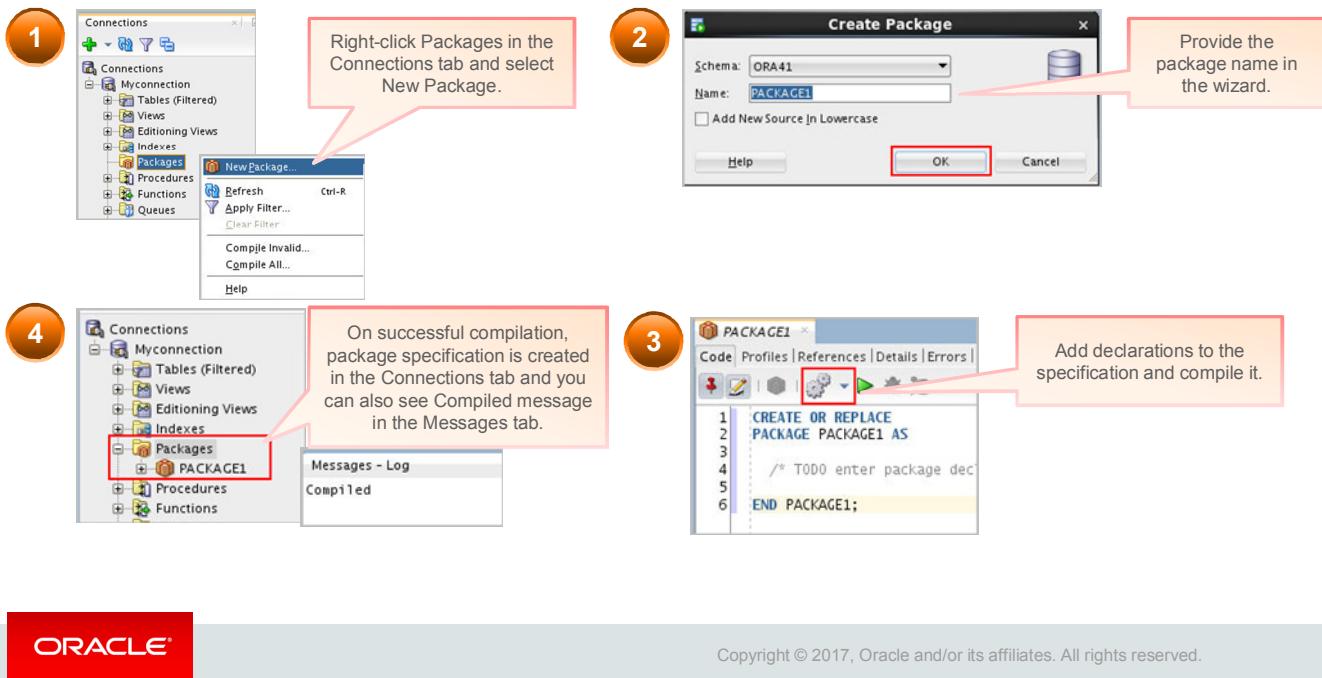
- **package_name** specifies a name for the package that must be unique among objects within the owning schema. Including the package name after the END keyword is optional.
- **public type and variable declarations** declares public variables, constants, cursors, exceptions, user-defined types, and subtypes.
- **subprogram specification** specifies the public procedure or function declarations.

The package specification should contain procedure and function headings terminated by a semicolon, without the IS (or AS) keyword and its PL/SQL block. The implementation of a procedure or function that is declared in a package specification is done in the package body.

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

All the constructs declared in a package specification are visible to users who are granted privileges on the package.

Creating Package Specification: Using SQL Developer



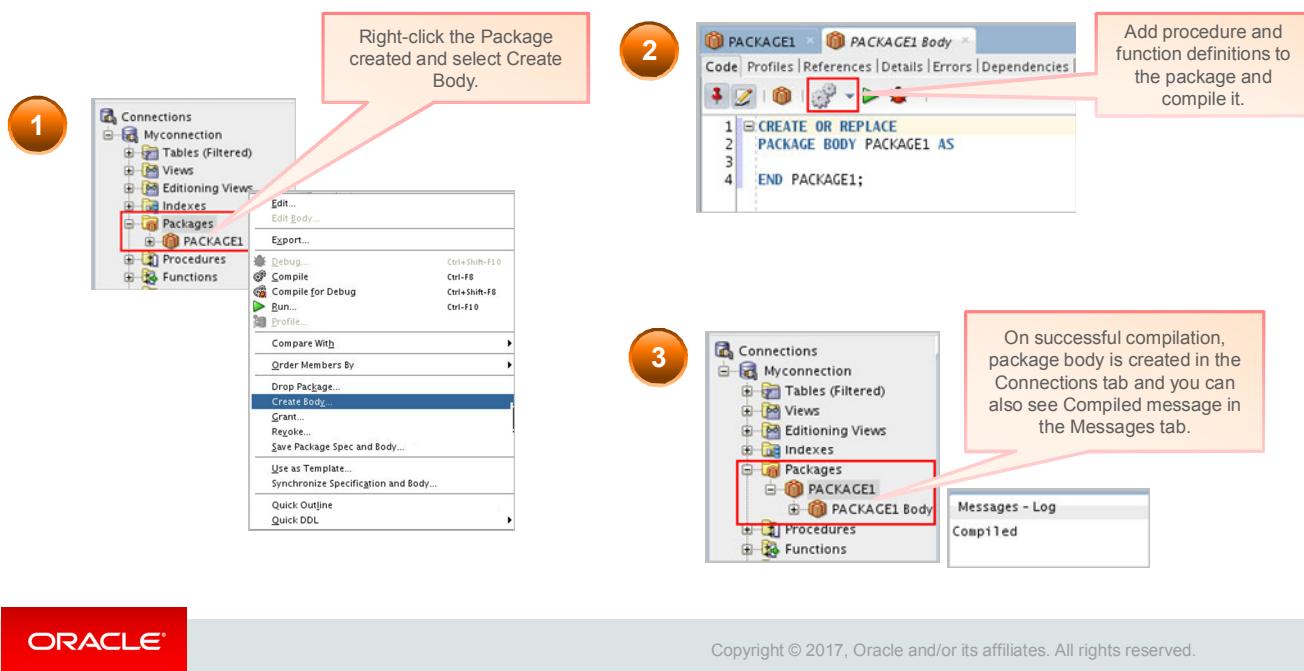
ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use SQL Developer to create the package specification as follows:

1. Right-click the **Packages** node in the Connections navigation tree and select **New Package** from the shortcut menu.
2. In the **Create PL/SQL Package** window, select the schema name, enter the name for the new package, and then click OK. A tab for the new package is displayed along with the shell for the new package.
3. Add code for the declarations in the package specification for the new package. Compile or save the new package. If there are errors in the package specification, the **Messages – Log** tab will display Compiled (with errors). You've to rectify the errors and recompile the package specification.
4. If there are no errors in the specification, then the **Messages – Log** tab will display Compiled as shown in the slide. The newly created package is displayed under the **Packages** node in the **Connections** navigation tree.

Creating the Package Body: Using SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use SQL Developer to create the package body by performing the following steps:

1. Right-click the package name for which you are creating a body in the **Packages** node in the **Connections** navigation tree. Select **Create Body** from the shortcut menu. A tab for the new package body is displayed along with the shell for the new package body.
2. Enter the code for the new package body. Compile or save the package body.
3. The **Messages – Log** tab displays whether compilation was successful or not.
4. The newly created package body is displayed under the **Packages** node in the **Connections** navigation tree.

Example of a Package Specification: comm_pkg

```
-- The package spec with a public variable and a  
-- public procedure that are accessible from  
-- outside the package.  
  
CREATE OR REPLACE PACKAGE comm_pkg IS  
    v_std_comm NUMBER := 0.10;  --initialized to 0.10  
    PROCEDURE reset_comm(p_new_comm NUMBER);  
END comm_pkg;  
/
```

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

Creating the Package Body

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Create a package body to define and implement all public subprograms and supporting private constructs. While creating a package body, perform the following steps:

- Specify the OR REPLACE option to overwrite an existing package body.
- Define the subprograms in an appropriate order. The basic principle is that you must declare a variable or subprogram before it can be referenced by other components in the same package body. It is common to see all private variables and subprograms defined first and the public subprograms defined last in the package body.
- Complete the implementation for all procedures or functions declared in the package specification within the package body.

The following are definitions of items in the package body syntax:

- ***package_name*** specifies a name for the package that must be the same as its package specification. Using the package name after the END keyword is optional.
- ***private type and variable declarations*** declares private variables, constants, cursors, exceptions, user-defined types, and subtypes.
- ***subprogram specification*** specifies the full implementation of any private and/or public procedures or functions.
- ***[BEGIN initialization statements]*** is an optional block of initialization code that executes when the package is first referenced.

Example of a Package Body: comm_pkg

```

CREATE OR REPLACE PACKAGE BODY comm_pkg IS
    FUNCTION validate(p_comm NUMBER) RETURN BOOLEAN IS
        v_max_comm employees.commission_pct%type;
    BEGIN
        SELECT MAX(commission_pct) INTO v_max_comm
        FROM employees;
        RETURN (p_comm BETWEEN 0.0 AND v_max_comm);
    END validate;

    PROCEDURE reset_comm(p_new_comm NUMBER) IS
    BEGIN
        IF validate(p_new_comm) THEN
            v_std_comm := p_new_comm; -- reset public var
        ELSE
            RAISE_APPLICATION_ERROR(
                -20210, 'Bad Commission');
        END IF;
    END reset_comm;
END comm_pkg;

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

Invoking the Package Subprograms: Examples

```
-- Invoke a function within the same package:
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...
  PROCEDURE reset_comm(p_new_comm NUMBER) IS
  BEGIN
    IF validate(p_new_comm) THEN
      ...
    END IF;
  END reset_comm;
END comm_pkg;
```

```
-- Invoke a package procedure from SQL*Plus:
EXECUTE comm_pkg.reset_comm(0.15)
```

```
-- Invoke a package procedure in a different schema:
EXECUTE scott.comm_pkg.reset_comm(0.15)
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

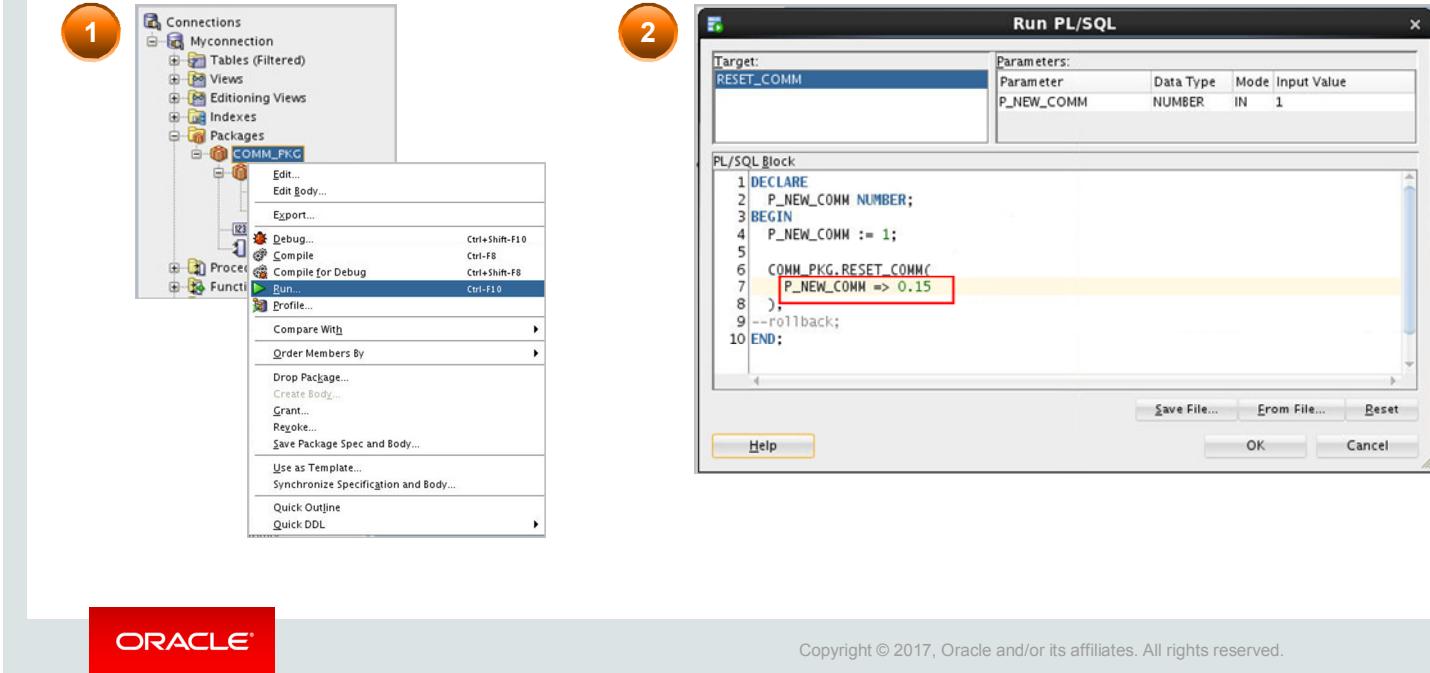
Example 2: Calls the `reset_comm` procedure from SQL*Plus (an environment external to the package) to reset the prevailing commission to 0.15 for the user session

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

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

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

Invoking Package Subprograms: Using SQL Developer



You can use SQL Developer to invoke a package's subprogram by performing the following steps:

1. Right-click the package's name in the **Packages** node in the Navigation tree and Select **Run** from the floating menu. The **Run PL/SQL** window is displayed. You can use the **Run PL/SQL** window to specify parameter values for running a PL/SQL function or procedure. (If you specify a package, select a function or procedure in the package.) Specify the following:
 - a. **Target:** Select the name of the function or procedure to run.
 - b. **Parameters:** This section lists each parameter for the specified target. The mode of each parameter can be **IN** (the value is passed in), **OUT** (the value is returned), or **IN/OUT** (the value is passed in, and the result of the function or procedure's action is stored in the parameter).
2. In the **PL/SQL Block** section, change the formal **IN** and **IN/OUT** parameter specifications in this block to actual values that you want to use for running the function or procedure. For example, to specify **0 . 15** as the value for an input parameter named **P_NEW_COMM**, change **P_NEW_COMM => P_NEW_COMM** to **P_NEW_COMM => 0 . 15**.
3. Click **OK**. SQL Developer runs the function or procedure.

Creating and Using Bodiless Packages

```
CREATE OR REPLACE PACKAGE global_consts IS
    c_mile_2_kilo CONSTANT NUMBER := 1.6093;
    c_kilo_2_mile CONSTANT NUMBER := 0.6214;
    c_yard_2_meter CONSTANT NUMBER := 0.9144;
    c_meter_2_yard CONSTANT NUMBER := 1.0936;
END global_consts;
```

```
SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
        20 * global_consts.c_mile_2_kilo || ' km');
END;
```

```
SET SERVEROUTPUT ON
CREATE FUNCTION mtr2yrd(p_m NUMBER) RETURN NUMBER IS
BEGIN
    RETURN (p_m * global_consts.c_meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The variables and constants declared within stand-alone subprograms exist only for the duration that the subprogram executes. To provide data that would exist throughout the user session, create a package specification containing public (global) variables and constant declarations. In this case, a package body isn't required, such packages are known as a *bodiless packages*. As discussed earlier in this lesson, if a specification declares only types, constants, variables, and exceptions, then the package body is unnecessary.

Examples:

The first code box in the slide creates a bodiless package specification with several constants to be used for conversion rates. A package body isn't required to support this package specification. It is assumed that the `SET SERVEROUTPUT ON` statement was issued before executing the code examples in the slide.

The second code box references the `c_mile_2_kilo` constant in the `global_consts` package by prefixing the package name to the identifier of the constant.

The third example creates a stand-alone function `c_mtr2yrd` to convert meters to yards, and uses the constant conversion rate `c_meter_2_yard` declared in the `global_consts` package. The function is invoked in a `DBMS_OUTPUT.PUT_LINE` parameter.

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

Viewing Packages by Using the Data Dictionary

```
-- View the package specification.
SELECT text
FROM user_source
WHERE name = 'COMM_PKG' AND type = 'PACKAGE'
ORDER BY LINE;
```

TEXT
1 PACKAGE comm_pkg IS 2 v_std_comm NUMBER := 0.10; --initialized to 0.10 3 PROCEDURE reset_comm(p_new_comm NUMBER); 4 END comm_pkg;

```
-- View the package body.
SELECT text
FROM user_source
WHERE name = 'COMM_PKG' AND type = 'PACKAGE BODY'
ORDER BY LINE;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The source code for PL/SQL packages is also stored in the `USER_SOURCE` and `ALL_SOURCE` data dictionary views. The `USER_SOURCE` table is used to display PL/SQL code that you own. The `ALL_SOURCE` table is used to display PL/SQL code to which you've been granted the `EXECUTE` right by the owner of that subprogram code and provides an `OWNER` column in addition to the preceding columns.

When querying the package, use a condition in which the `TYPE` column is:

- Equal to '`PACKAGE`' to display the source code for the package specification
- Equal to '`PACKAGE BODY`' to display the source code for the package body

You can also view the package specification and body in SQL Developer using the package name in the Packages node.

Note: You can't display the source code for Oracle PL/SQL built-in packages, or PL/SQL whose source code has been wrapped by using a WRAP utility or obfuscation. Obfuscating and wrapping PL/SQL source code is covered in the lesson titled "Working with Packages." Clicking the **Execute Statement** icon (or pressing F9) instead of the **Run Script** icon in the SQL Worksheet toolbar, sometimes displays a better formatted output in the Results tab as shown in the slide examples.

Viewing Packages by Using SQL Developer

- 1** To view the package spec, click the package name.

The screenshot shows the SQL Developer interface. On the left, the Connections tree shows a connection named 'Myconnection' with nodes for Tables, Views, Editioning Views, Indexes, Packages, Procedures, and Functions. Under 'Packages', the node 'COMM_PKG' is selected. On the right, the main window has a title bar 'COMM_PKG'. Below it is a toolbar with icons for Save, Run, Stop, and others. The 'Code' tab is selected, showing the package specification code:

```

1 create or replace PACKAGE comm_pkg IS
2   v_std_comm NUMBER := 0.10; --initialized to 0.10
3   PROCEDURE reset_comm(p_new_comm NUMBER);
4 END comm_pkg;

```

- 2** To view the package body, click the package body.

The screenshot shows the SQL Developer interface. On the left, the Connections tree shows the same 'Myconnection' connection. Under 'Packages', the node 'COMM_PKG' is expanded, and its child node 'COMM_PKG Body' is selected. On the right, the main window has a title bar 'COMM_PKG' and 'COMM_PKG Body'. Below it is a toolbar with icons for Save, Run, Stop, and others. The 'Code' tab is selected, showing the package body code:

```

1 create or replace PACKAGE BODY comm_pkg IS
2   FUNCTION validate(p_com NUMBER) RETURN BOOLEAN IS
3     v_max_comm employees.commission_pcttype;
4   BEGIN
5     SELECT MAX(commission_pct) INTO v_max_comm
6       FROM employees;
7     RETURN (p_com BETWEEN 0.0 AND v_max_comm);
8   END validate;
9
10  PROCEDURE reset_comm (p_new_comm NUMBER) IS BEGIN
11    IF validate(p_new_comm) THEN
12      v_std_comm := p_new_comm; -- reset public var
13    ELSE
14      RAISE_APPLICATION_ERROR(-20210, 'Bad Commission');
15    END IF;
16    END reset_comm;
17  END comm_pkg;

```

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To view a package's spec in SQL Developer, perform the following steps:

1. Click the **Packages** node in the **Connections** tab.
2. Click the package's name.
3. The package's spec code is displayed in the **Code** tab as shown in the slide.

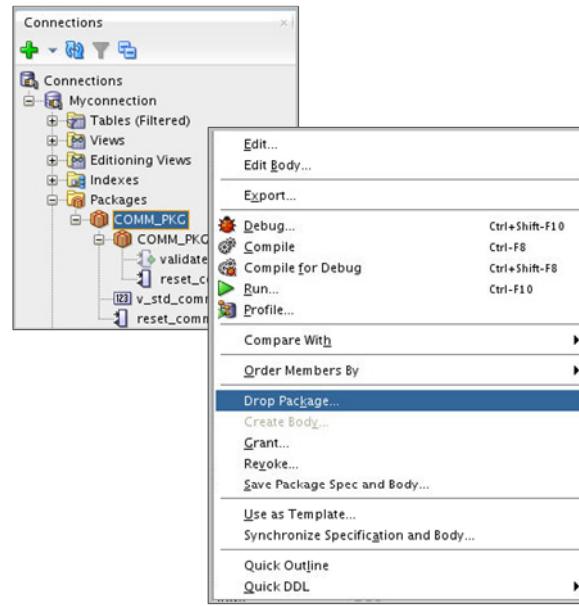
To view a package's body in SQL Developer, perform the following steps:

1. Click the **Packages** node in the **Connections** tab.
2. Click the package's body.
3. The package's body code is displayed in the **Code** tab as shown in the slide.

Removing Packages

Drop the package specification and body.

```
DROP PACKAGE package_name;
```



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

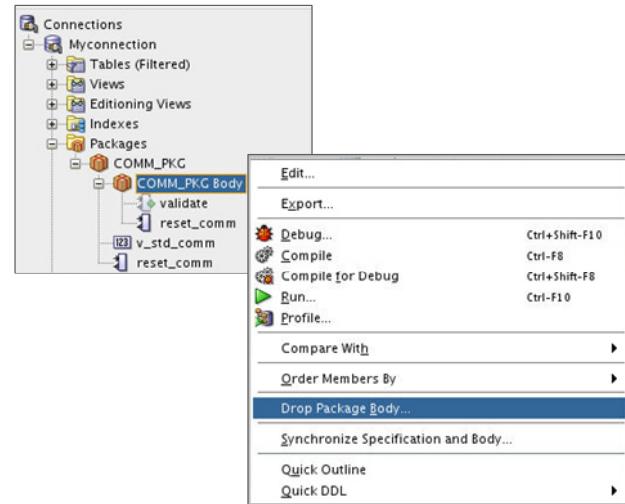
The slide shows removing both the package specification and package body. Through the command-line interface you can remove a package by executing `DROP PACKAGE package_name`.

In SQL Developer, you can drop a package by right-clicking the package name and selecting **Drop Package** from the menu.

Removing Package Bodies

Drop only the package body.

```
DROP PACKAGE BODY package_name;
```



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide demonstrates how you can drop only the package body, instead of dropping the entire package.

You can drop the package body by executing the `DROP PACKAGE BODY package_name` command.

In SQL Developer, you drop package body by right-clicking the package body and selecting **Drop Package Body** from the menu.

Guidelines for Writing Packages

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

Quiz



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

- a. True
- b. False



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a

Summary

In this lesson, you should have learned how to:

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

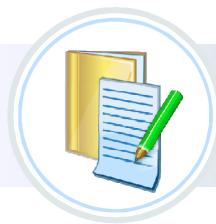
A package consists of a package specification and a package body. You can change a package body without affecting its package specification.

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

Practice 5 Overview: Creating and Using Packages

This practice covers the following topics:

- Creating packages
- Invoking package program units

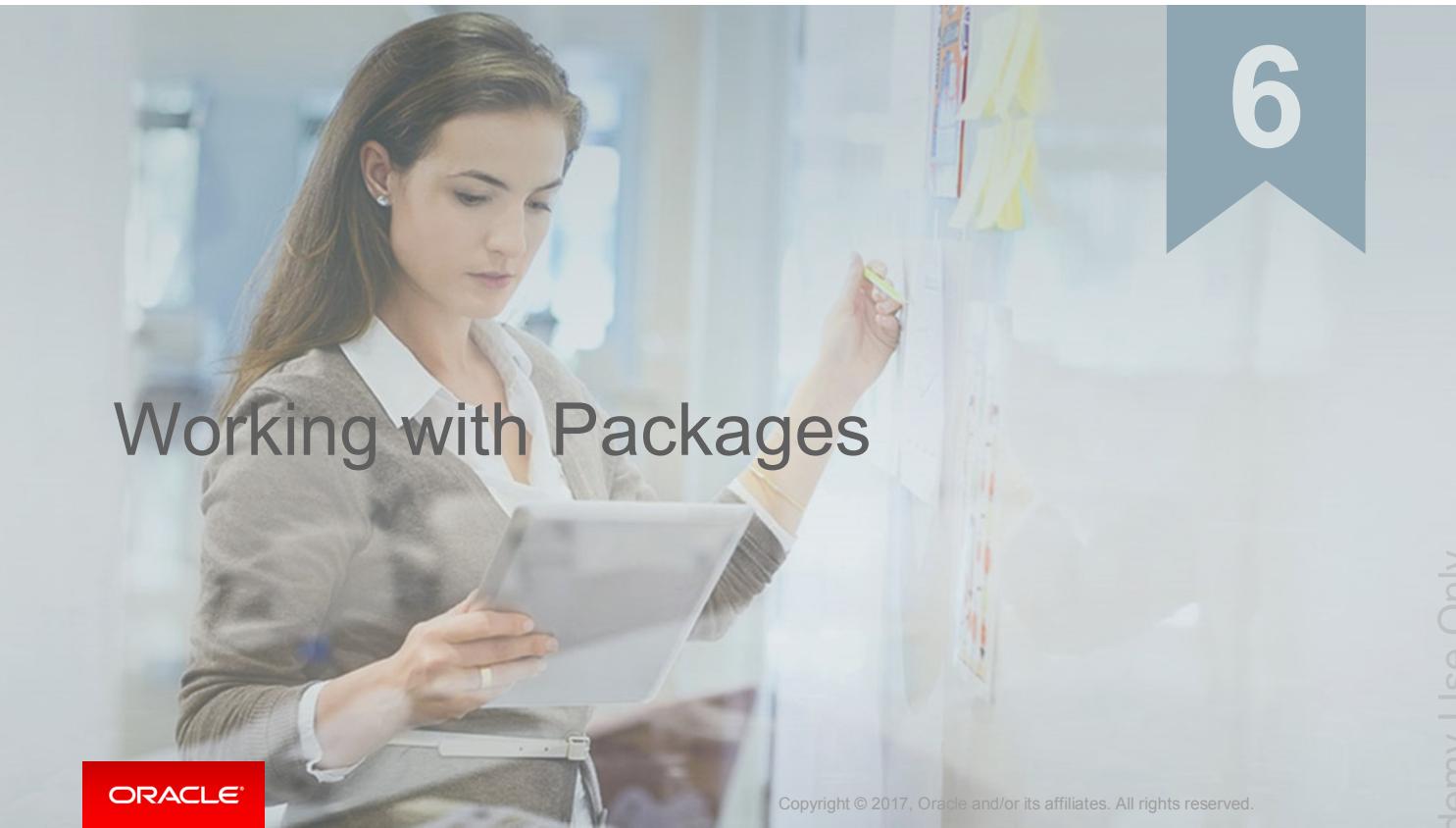


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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



6

Working with Packages

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

- ▶ Lesson 2: Creating Procedures
- ▶ Lesson 3: Creating Functions
- ▶ Lesson 4: Debugging Subprograms
- ▶ Lesson 5: Creating Packages
- ▶ **Lesson 6: Working with Packages**
- ▶ Lesson 7: Using Oracle-Supplied Packages in Application Development
- ▶ Lesson 8: Using Dynamic SQL

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Working with Subprograms, Working with Triggers, and Working with PL/SQL code.

In the first unit, we have seven lessons: Creating Stored Procedures, Creating Functions, Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms, packages and use them in Application Development.

Objectives

After completing this lesson, you should be able to:

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Lesson Agenda

- Overloading package subprograms
- Initializing packages
- Managing persistent package state in a session



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Why Overloading of Subprogram?



Hey Alice, Can you help me with this task at hand.
I want to calculate the tax of all employees in the
company

.....Wait not all employees, but only employees of
a single department

.....or may be a single employee Or I think I
need all of those options. Can you help... ?

Happy to help 😊



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Here is a situation, where the HR manager has to calculate the tax of the employees. It is not just calculating the tax applicable to all the employees always. Sometimes he might want to calculate tax for all the employees in a particular department . Sometimes the HR manager may also want to calculate tax for a single employee. All these are variants of the same operation tax calculation with different inputs.

Let's look at this problem from Alice's perspective. Alice may have to write a procedure to calculate tax for the employees. However the procedure may have different set of input parameters each time. In order to accommodate this varying set of inputs, Alice can write an overloaded subprogram.

An overloaded subprogram is one with the same name, but different set of input parameters.

Overloading Subprograms in PL/SQL

- Enables you to create two or more subprograms with the same name
- Requires that the subprogram's formal parameters differ in number, order, or data type family
- Enables you to build flexible ways for invoking subprograms with different data
- Provides a way to extend functionality without loss of existing code; that is, adding new parameters to existing subprograms
- Provides a way to overload local subprograms, package subprograms, and type methods, but not stand-alone subprograms



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

Consider using overloading when you are:

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

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

Restrictions

You cannot overload:

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

Resolving Calls

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

Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
    PROCEDURE add_department
        (p_deptno departments.department_id%TYPE,
         p_name departments.department_name%TYPE := 'unknown',
         p_loc   departments.location_id%TYPE := 1700);

    PROCEDURE add_department
        (p_name departments.department_name%TYPE := 'unknown',
         p_loc   departments.location_id%TYPE := 1700);
END dept_pkg;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

It is better to specify data types using the `%TYPE` attribute for variables that are used to populate columns in database tables, as shown in the example in the slide; however, you can also specify the data types as follows:

```
CREATE OR REPLACE PACKAGE dept_pkg_method2 IS
    PROCEDURE add_department(p_deptno NUMBER,
                             p_name VARCHAR2 := 'unknown', p_loc NUMBER := 1700);
    ...

```

Overloading Procedures Example: Creating the Package Body

```
-- Package body of package defined on previous slide.
CREATE OR REPLACE PACKAGE BODY dept_pkg IS
PROCEDURE add_department -- First procedure's definition
(p_deptno departments.department_id%TYPE,
 p_name   departments.department_name%TYPE := 'unknown',
 p_loc    departments.location_id%TYPE := 1700) IS
BEGIN
  INSERT INTO departments(department_id,
    department_name, location_id)
  VALUES (p_deptno, p_name, p_loc);
END add_department;
PROCEDURE add_department -- Second procedure's definition
(p_name   departments.department_name%TYPE := 'unknown',
 p_loc   departments.location_id%TYPE := 1700) IS
BEGIN
  INSERT INTO departments (department_id,
    department_name, location_id)
  VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_department;
END dept_pkg; /
```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	980 Education	(null)	2500

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

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

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	340 Training	(null)	2400

Restrictions on Overloading

- You cannot overload:
 - Stand-alone subprograms
 - Subprograms whose parameters differ only in the mode (IN, OUT)
 - Subprograms whose parameters belong to the same family respectively
 - Functions which return only in the return type but not the formal parameters



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can't overload the subprograms in the scenarios mentioned in the slide.

You can't overload stand-alone subprograms because they are independent units and it will result in a naming conflict.

In a package, you can't overload subprograms which differ only in the parameter mode type.
Consider

```
procedure A(p1 IN NUMBER) and procedure A(p2 OUT NUMBER)
```

These subprograms cannot be overloaded and will result in naming conflict.

Two subprograms can't be overloaded , if their parameters correspond to same family of data types.

For instance,

```
PROCEDURE s (p INTEGER) and PROCEDURE s (p REAL)
```

can't be overloaded, the parameters belong to the same family of data types.

Similarly two functions which only differ in the return type can't be overloaded.

For instance,

```
FUNCTION f (p INTEGER) RETURN BOOLEAN and
```

```
FUNCTION f (p INTEGER) RETURN INTEGER
```

can't be overloaded.

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

Note: The preceding restrictions apply if the names of the parameters are also the same.

If you use different names for the parameters, you can invoke the subprograms by using named notation for the parameters.

STANDARD package

- Is an Oracle-supplied package
- Defines the PL/SQL environment
- Declares types, exceptions, and subprograms, which are available automatically to every PL/SQL program



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The PL/SQL environment is defined by the STANDARD package which is an Oracle-supplied package.

The package specification declares public types, variables, exceptions, subprograms, which are available automatically to PL/SQL programs. For example, the STANDARD package declares function ABS, which returns the absolute value of its argument.

The contents of the STANDARD package are directly visible to applications. You need not qualify references to its contents by prefixing the package name.

For instance to invoke the ABS function of the STANDARD package, you need not write STANDARD . ABS; writing ABS will automatically refer to the STANDARD package.

Overloading and the STANDARD Package

- Most built-in functions in the STANDARD package are overloaded.
- An example is the TO_CHAR function:

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Most of the built-in functions that are found in the STANDARD package are overloaded. For example, the TO_CHAR function has four different declarations, as shown in the slide. The TO_CHAR function can take either the DATE or the NUMBER data type and convert it to the character data type. The format to which the date or number has to be converted can also be specified in the function call.

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

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

In the example in the slide, PL/SQL resolves a call to TO_CHAR by matching the number and data types of the formal and actual parameters.

Lesson Agenda

- Overloading package subprograms
- Initializing packages
- Managing persistent package state in a session



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Package Instantiation and Initialization

- Packages are stored as schema objects.
- Packages are instantiated when referenced.
- Each application which references the package has its own instance.
- The package is initialized whenever instantiated.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When you create and compile packages, they are stored as schema objects in the database.

Applications can access the members of the package through the package name as `package_name.procedure_name` like `DBMS_OUTPUT.PUT_LINE` where `DBMS_OUTPUT` is the package name and `PUT_LINE` is the procedure name.

Whenever an application references a package to use its functionality, an instance of the package is created for the application.

Various package members are initialized on instantiation. Following steps are executed as part of initialization:

- Assign initial values to public constants.
- Assign initial values to public variables whose declarations specify them.
- Execute the initialization part of the package body.

Initializing Packages in Package Body

In package body, you can initialize package members in the initialization block at the end of the package body.

```
CREATE OR REPLACE PACKAGE taxes IS
    v_tax    NUMBER;
    ... -- declare all public procedures/functions
END taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes IS
    ... -- declare all private variables
    ... -- define public/private procedures/functions
BEGIN
    SELECT    rate_value INTO v_tax
    FROM      tax_rates
    WHERE     rate_name = 'TAX';
END taxes;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

Using User-Defined Package Functions in SQL

- Functions in user-defined packages can be accessed like those in Oracle-supplied packages.
- The user-defined package function usage is similar to that of functions such as MAX, MIN and other predefined functions.
- The package specification and body should be compiled before using them in the SQL statements in the current schema.
- You have to identify the functions with the package name.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

After compiling both the package specification and package body, user-defined packages are as good as Oracle-supplied packages. You can access the package functions in SQL statements by referencing the function through the package name.

The function execution should be free of any side effects, which we discussed in the lesson titled “Creating Functions.”

User-Defined Package Function in SQL: Example

```

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

```

```

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

```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The first code example in the slide shows how to create the package specification and the body encapsulating the tax function in the `taxes_pkg` package. The second code example shows how to call the packaged tax function in the `SELECT` statement. The results are as follows:

	TAXES_PKG.TAX(SALARY)	SALARY	LAST_NAME
1	1920	24000 King	
2	1360	17000 Kochhar	
3	1360	17000 De Haan	
4	720	9000 Hunold	
5	480	6000 Ernst	
6	384	4800 Austin	
7	384	4800 Pataballa	
8	336	4200 Lorentz	
9	1056.704	13208.8 Greenberg	
10	720	9000 Faviet	

Lesson Agenda

- Overloading package subprograms
- Initializing packages
- Managing persistent package state in a session



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Package State

- Package State refers to the values of the variables, constants, and cursors which are part of the package.
- When a package is instantiated, its state is initialized.
- The state of the package may change during application execution.
- A package without any members that hold a value is a stateless package.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Serially Reusable Packages

- SERIALLY_REUSABLE packages let you design applications that manage memory better for scalability.



How does the memory management actually work in an Oracle database for packages?

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Serially Reusable Packages are no different from packages in usage perspective. They are efficient in managing memory in scalable applications.

To understand how they are different from packages, we first need to understand how packages use memory.

Memory Architecture

There are three important data structures defined according to the Memory Architecture of the database instance:

System Global Area

- Group of shared memory structures, which has information of one database instance. All server and background processes use it.

User Global Area

- Memory area associated with a user session connecting with the database instance

Program Global Area

- It is the unshared memory region that contains data and control information exclusively for use by an Oracle process.
- One PGA exists for each server process and background process.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When an instance is started, Oracle Database allocates a memory area and starts background processes.

The memory area stores information such as the following:

- Program code
- Information about each connected session, even if it is not currently active
- Information needed during program execution, for example, the current state of a query from which rows are being fetched
- Information such as lock data that is shared and communicated among processes
- Cached data, such as data blocks and redo records, that also exists on disk

There are three primary memory areas defined according to the memory architecture of the database instance:

1. System global area
2. Program global area
3. User global area

System global area has all shared memory structures associated with a database instance. All server processes and background processes of the database instance are in the System global area.

Program Global Area is created by the database whenever an Oracle process is initiated. It is used only by the process and not shared with other processes.

User Global Area is the memory area which has all the data pertaining to the communication between an application and database.

When a user connects with the database, all the data used by the user is stored in the User Global Area.

If a session loads a **PL/SQL package** into memory, then the UGA contains the package state, which is the set of values stored in all the package variables at a specific time. The package state changes when a package subprogram changes the variables. By default, the package variables are unique to and persist for the life of the session.

Seriously Reusable Packages

- Package state of a package is stored in the user global area (UGA) for each user.
- Package state of a SERIALLY REUSABLE PACKAGE is stored in a work area in a small pool in the system global area (SGA).
- Storing the package state in the UGA limits the scalability of the application.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When the package state is stored in the UGA, the amount of UGA memory needed increases linearly with the number of users. The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days. In such scenarios, the UGA is occupied for an extended period of time limiting the scalability.

When a package is declared to be `SERIALLY_REUSABLE`, its package state is stored in a work area in a small pool in the system global area (SGA). The package state persists only for the life of a server call. After the server call, the work area returns to the pool. Because the memory is returned to the pool, the scalability of the application is better.

However, If a subsequent server call references the package, then Oracle Database reuses an instantiation from the pool. Reusing an instantiation reinitializes it; therefore, changes made to the package state in previous server calls are invisible. We have to persist the package state after returning the server call, so that the user can reference to the earlier package state.

Persistent State of Packages

The collection of package variables and the values define the package state. The package state is:

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Package variables are initialized the first time a package is loaded into memory for a user session. The package variables are, by default, unique to each session and hold their values until the user session is terminated. In other words, the variables are stored in the User Global Area (UGA) memory allocated by the database for each user session. The package state changes when a package subprogram is invoked and its logic modifies the variable state. Public package state can be directly modified by operations appropriate to its type.

`PRAGMA` signifies that the statement is a compiler directive. `PRAGMAS` are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler. If you add `PRAGMA SERIALLY_RESUABLE` to the package specification, then the database stores package variables in the System Global Area (SGA) shared across user sessions. In this case, the package state is maintained for the life of a subprogram call or a single reference to a package construct. The `SERIALLY_REUSABLE` directive is useful if you want to conserve memory and if the package state does not need to persist for each user session.

Persistent State of Package Variables: Example

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

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

Persistent State of a Package Cursor: Example

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

CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  -- Package body
  CURSOR cur_c IS
    SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT cur_c%ISOPEN THEN
      OPEN cur_c;
    END IF;
  END open;
  . . . -- code continued on next slide
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows the CURS_PKG package specification and body. The body declaration is continued in the next slide.

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

1. Call the open procedure to open the cursor.

Persistent State of a Package Cursor: Example

```
 . . .
FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN IS
    v_emp_id employees.employee_id%TYPE;
BEGIN
    FOR count IN 1 .. p_n LOOP
        FETCH cur_c INTO v_emp_id;
        EXIT WHEN cur_c%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Id: ' || (v_emp_id));
    END LOOP;
    RETURN cur_c%FOUND;
END next;
PROCEDURE close IS
BEGIN
    IF cur_c%ISOPEN THEN
        CLOSE cur_c;
    END IF;
END close;
END curs_pkg;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

Executing the CURS_PKG Package

```

1 :SET SERVEROUTPUT ON
2 :
3 :EXECUTE curs_pkg.open
4 DECLARE
5   v_more BOOLEAN := curs_pkg.next(3);
6 BEGIN
7   IF NOT v_more THEN
8     curs_pkg.close;
9   END IF;
10 END;
11

```

PL/SQL procedure successfully completed.
 PL/SQL procedure successfully completed.
 Id: 100
 Id: 101
 Id: 102
 PL/SQL procedure successfully completed.
 PL/SQL procedure successfully completed.
 Id: 103
 Id: 104
 Id: 105



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

```

Id :100
Id :101
Id :102

```

If you click the Run Script icon (or press F5) again, the next three rows are displayed:

```

Id :103
Id :104
Id :105

```

To close the cursor, you can issue the following command to close the cursor in the package, or exit the session:

```
EXECUTE curs_pkg.close
```

Quiz



Overloading subprograms in PL/SQL:

- a. Enables you to create two or more subprograms with the same name
- b. Requires that the subprogram's formal parameters differ in number, order, or data type family
- c. Enables you to build flexible ways for invoking subprograms with different data
- d. Provides a way to extend functionality without loss of existing code; that is, adding new parameters to existing subprograms



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, c, d

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

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

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

Consider using overloading when:

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

Summary

In this lesson, you should have learned how to:

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

Practice 6 Overview: Working with Packages

This practice covers the following topics:

- Using overloaded subprograms
- Creating a package initialization block
- Using a forward declaration



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this practice, you modify an existing package to contain overloaded subprograms and you use forward declarations. You also create a package initialization block within a package body to populate a PL/SQL table.



7

Using Oracle-Supplied Packages in Application Development

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with PL/SQL Code

- ▶ Lesson 2: Creating Procedures
- ▶ Lesson 3: Creating Functions
- ▶ Lesson 4: Debugging Subprograms
- ▶ Lesson 5: Creating Packages
- ▶ Lesson 6: Working with Packages
- ▶ Lesson 7: Using Oracle-Supplied Packages in Application Development**
- ▶ Lesson 8: Using Dynamic SQL

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Working with Subprograms, Working with Triggers, and Working with PL/SQL Code.

In the first unit, we have seven lessons: Creating Stored Procedures, Creating Functions, Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms, packages, and use them in Application Development.

Objectives

After completing this lesson, you should be able to:

- Describe how the DBMS_OUTPUT package works
- Use UTL_FILE to direct output to operating system files
- Describe the main features of UTL_MAIL



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

Lesson Agenda

- Identifying the benefits of using the Oracle-supplied packages and listing some of those packages
- Using the following Oracle-supplied packages:
 - DBMS_OUTPUT
 - UTL_FILE
 - UTL_MAIL



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using Oracle-Supplied Packages

- The Oracle-supplied packages:
 - Are provided with the Oracle server
 - Extend the functionality of the database
 - Provide PL/SQL access to SQL statements
- Most of the Oracle-supplied packages are installed during database creation.



Modifying Oracle-supplied packages can cause internal errors and database security violations. Do not modify supplied packages.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

The PL/SQL functions defined in the Oracle-supplied packages can be invoked in SQL statements, thus SQL statements can access PL/SQL functions. You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

Most of the standard packages are created while creating the database. Certain packages are not installed automatically; you have to install them if required by referring to appropriate Oracle documentation.

You can use Oracle-supplied packages while creating your own packages or you can refer to them for ideas in creating your own stored procedures.

Examples of Some Oracle-Supplied Packages

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. This lesson covers the first three packages in the slide. For more information, refer to the *Oracle Database PL/SQL Packages and Types Reference*. The following is a brief description about the remaining listed packages in the slide:

- DBMS_OUTPUT provides debugging and buffering of text data.
- UTL_FILE enables your PL/SQL programs to read and write operating system text files and provides a restricted version of standard operating system stream file I/O.
- UTL_MAIL is a utility for managing email which includes commonly used email features, such as attachments, CC, BCC, and return receipt.
- DBMS_ALERT supports asynchronous notification of database events.
- DBMS_LOCK is used to request, convert, and release locks through Oracle Lock Management services.
- DBMS_SESSION provides access to SQL ALTER SESSION statements, and other session information, from stored procedures.
- DBMS_APPLICATION_INFO lets you register an application name with the database for auditing or performance-tracking purposes.
- HTP package has hypertext procedures to generate HTML tags.
- DBMS_SCHEDULER provides a collection of scheduling functions that are callable from any PL/SQL program.

Lesson Agenda

- Identifying the benefits of using the Oracle-supplied packages and listing some of those packages
- Using the following Oracle-supplied packages:
 - DBMS_OUTPUT
 - UTL_FILE
 - UTL_MAIL



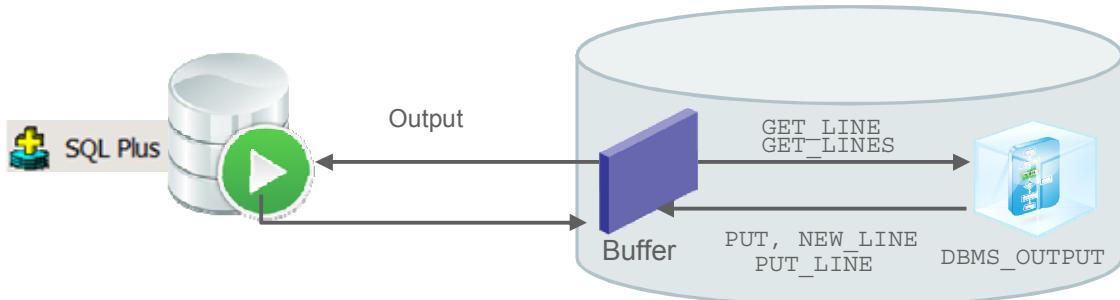
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

How the DBMS_OUTPUT Package Works

The DBMS_OUTPUT package enables you to send messages from stored subprograms and triggers to other stored subprograms or buffer.

- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Messages are not sent until the sending subprogram or trigger completes.



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The DBMS_OUTPUT package sends textual messages from any PL/SQL block into a buffer in the database. You can output results to the window for debugging purposes. You can trace a code execution path for a function or procedure. You can send messages between subprograms and triggers.

Procedures provided by the package include the following:

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

The buffer size can be set by using:

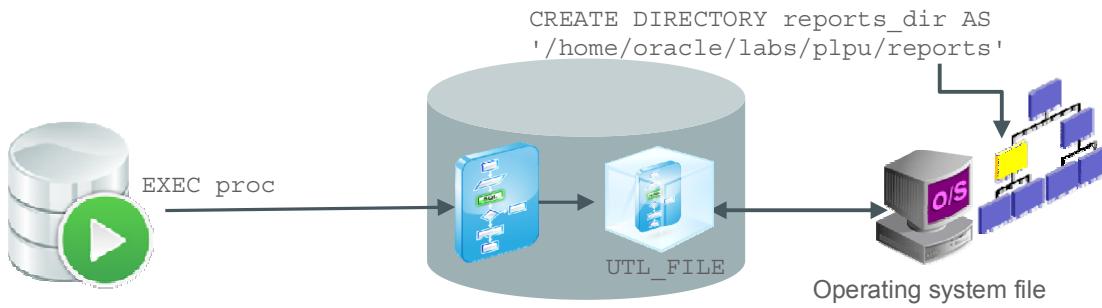
- The SIZE n option appended to the SET SERVEROUTPUT ON command where n is the number of characters. The minimum is 2,000 and the maximum is unlimited. The default is 20,000.
- An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

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

Using the UTL_FILE Package

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

- Provides a restricted version of operating system stream file I/O for text files
- Enables creating files in operating system directories defined by a CREATE DIRECTORY statement



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

With the UTL_FILE package, you can write PL/SQL programs that can read and write operating system text files. The set of files and directories that are accessible through UTL_FILE is defined by the database parameters and other factors.

If certain database user say ‘Scott’ has read and write access to an operating system directory dir1, then the user can only access the files in that directory on the disk. The database user cannot access the files in its subdirectories or parent directories.

In the past, accessible directories for the UTL_FILE functions were specified in the initialization file using the UTL_FILE_DIR parameter. However this is no longer recommended. Oracle recommends you use directory object feature. This feature provides more granularity and flexibility with respect to security.

You can use the directory objects by using the CREATE DIRECTORY instead of UTL_FILE_DIR parameter. A CREATE DIRECTORY statement associates an alias with an operating system directory. The database directory alias can be granted the READ and WRITE privileges to control the type of access to files in the operating system.

For example:

```
CREATE DIRECTORY my_dir AS '/temp/my_files';
GRANT READ, WRITE ON DIRECTORY my_dir TO Scott;
```

The database user must have the CREATE ANY DIRECTORY privilege granted to create a directory object in the operating system file system. The CREATE DIRECTORY privilege is granted only to SYS and SYSTEM by default.

Some of the UTL_FILE Procedures and Functions

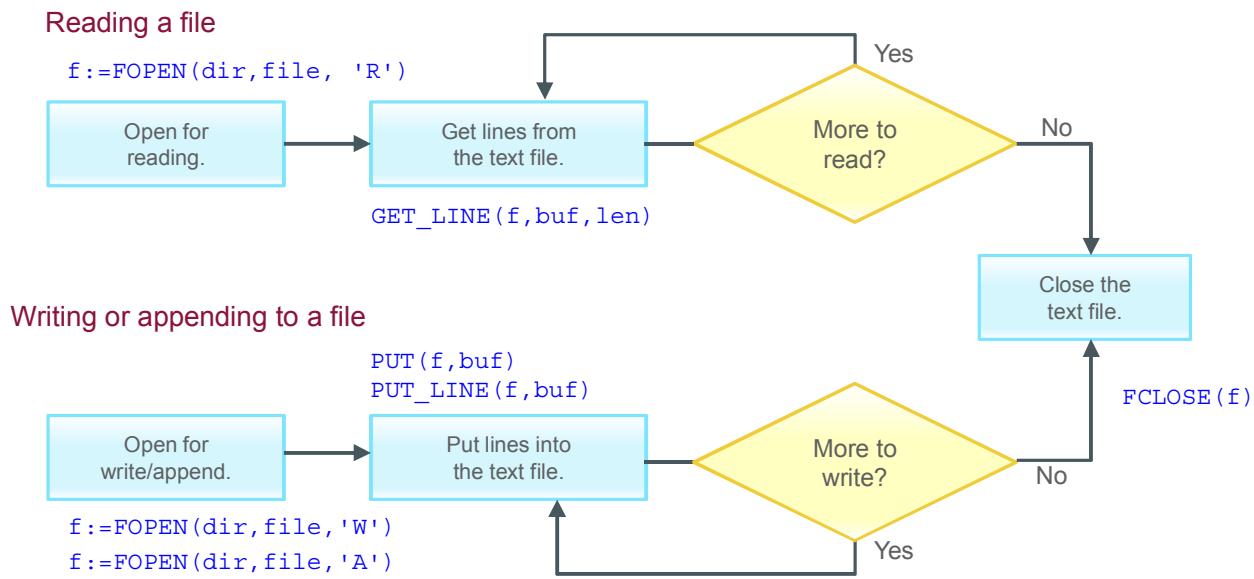
Subprogram	Description
ISOPEN function	Determines if a file handle refers to an open file
FOPEN function	Opens a file for input or output
FCLOSE function	Closes all open file handles
FCOPY procedure	Copies a contiguous portion of a file to a newly created file
FGETATTR procedure	Reads and returns the attributes of a disk file
GET_LINE procedure	Reads text from an open file
FREMOVE procedure	Deletes a disk file, if you have sufficient privileges
FRENAME procedure	Renames an existing file to a new name
PUT procedure	Writes a string to a file
PUT_LINE procedure	Writes a line to a file, and so appends an operating system-specific line terminator



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The table in the slide lists some of the UTL_FILE Package subprograms. For a complete list of the package's subprograms, see *Oracle Database PL/SQL Packages and Types Reference*.

File Processing Using the UTL_FILE Package: Overview



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use the procedures and functions in the UTL_FILE package to open files with the FOPEN function. You can then either read from or write or append to the file until processing is done. After you finish processing the file, close the file by using the FCLOSE procedure.

The slide demonstrates the process of reading from or writing to a file using the subprograms of UTL_FILE package.

To read from a file:

1. You open the file using the FOPEN procedure with the required parameters.
2. After the file is open, you read data from it using GET_LINE, GET_LINE_NCHAR, or GET_RAW procedures. You can check whether a file is open or not by using the ISOPEN procedure.
3. You invoke appropriate subprograms until you complete reading data from the file and close the file.

The process is similar for writing to a file using the UTL_FILE package. To write to a file:

1. You open the file using the FOPEN procedure, the difference being the mode in which you open the file. While reading the file you open the file in the read mode '`R`', whereas while writing to the file you open the file in write '`W`' or append '`A`' mode.
2. Once the file is open, you can write to the file using PUT, PUT_LINE, PUT_LINE_NCHAR, PUT_NCHAR, PUTF, PUTF_NCHAR, PUT_RAW subprograms of the UTL_FILE package.
3. After you complete writing to the file, you close the file.

Using the Available Declared Exceptions in the UTL_FILE Package

Exception Name	Description
INVALID_PATH	File location invalid
INVALID_MODE	The open_mode parameter in FOPEN invalid
INVALID_FILEHANDLE	File handle invalid
INVALID_OPERATION	File could not be opened or operated on as requested
READ_ERROR	Destination buffer too small, or operating system error occurred during the read operation
WRITE_ERROR	Operating system error occurred during the write operation
INTERNAL_ERROR	Unspecified PL/SQL error



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The UTL_FILE package declares 15 exceptions that indicate an error condition in the operating system file processing. You may have to handle one of these exceptions when using UTL_FILE subprograms.

A subset of the exceptions are displayed in the slide. For additional information about the remaining exceptions, refer to *Oracle Database PL/SQL Packages and Types Reference*.

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

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

FOPEN and IS_OPEN Functions: Example

- This FOPEN function opens a file for input or output.

```
FUNCTION FOPEN (p_location  IN VARCHAR2,
               p_filename   IN VARCHAR2,
               p_open_mode  IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

- The IS_OPEN function determines whether a file handle refers to an open file.

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The parameters include the following:

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

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

The IS_OPEN function parameter is the file handle. The IS_OPEN function tests a file handle to see whether it identifies an opened file. It returns a Boolean value of TRUE if the file has been opened; otherwise it returns a value of FALSE indicating that the file has not been opened. The example in the slide shows how to combine the use of the two subprograms. For the full syntax, refer to *Oracle Database PL/SQL Packages and Types Reference*.

Ensure that the external file and the database are on the same PC.

```
CREATE OR REPLACE PROCEDURE read_file(p_dir VARCHAR2,
  p_filename VARCHAR2) IS
  f_file UTL_FILE.FILE_TYPE;
  buffer VARCHAR2(200);
  lines  PLS_INTEGER := 0;
BEGIN
  DBMS_OUTPUT.PUT_LINE(' Start ');
  IF NOT UTL_FILE.IS_OPEN(f_file) THEN
    DBMS_OUTPUT.PUT_LINE(' Open ');
    f_file := UTL_FILE.FOPEN (p_dir, p_filename, 'R');
    DBMS_OUTPUT.PUT_LINE(' Opened ');
    BEGIN
      LOOP
        UTL_FILE.GET_LINE(f_file, buffer);
        lines := lines + 1;
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(lines, '099') || |
' |||buffer);
      END LOOP;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE(' ** End of File **');
      END;
      DBMS_OUTPUT.PUT_LINE(lines||' lines read from file');
      UTL_FILE.FCLOSE(f_file);
    END IF;
  END read_file;
/
SHOW ERRORS
SET SERVEROUTPUT ON
EXECUTE read_file('REPORTS_DIR', 'instructor.txt')
```

The partial output of the above code is as follows:

```
Script Output X | Task completed in 0.017 seconds
PL/SQL procedure successfully completed.

Start
Open
Opened
001 SALARY REPORT: GENERATED ON
002                               08-MAR-01
003
004 DEPARTMENT: 10
005   EMPLOYEE: Whalen earns: 4400
006 DEPARTMENT: 20
007   EMPLOYEE: Hartstein earns: 13000
008   EMPLOYEE: Fay earns: 6000
009 DEPARTMENT: 30
010  EMPLOYEE: Raphaely earns: 11000
011  EMPLOYEE: Khoo earns: 3100
012  EMPLOYEE: Baida earns: 2900
013  EMPLOYEE: Colmenares earns: 2500
014  EMPLOYEE: Himuro earns: 2600
015  EMPLOYEE: Tobias earns: 2800
016 DEPARTMENT: 40
017  EMPLOYEE: Mavris earns: 6500
018 DEPARTMENT: 50
```

...

```
106  EMPLOYEE: Tucker earns: 10000
107  EMPLOYEE: Cambrault earns: 11000
108  EMPLOYEE: Partners earns: 13500
109 DEPARTMENT: 90
110  EMPLOYEE: King earns: 24000
111  EMPLOYEE: Kochhar earns: 17000
112  EMPLOYEE: De Haan earns: 17000
113 DEPARTMENT: 100
114  EMPLOYEE: Greenberg earns: 12000
115  EMPLOYEE: Faviet earns: 9000
116  EMPLOYEE: Chen earns: 8200
117  EMPLOYEE: Sciarra earns: 7700
118  EMPLOYEE: Urman earns: 7800
119  EMPLOYEE: Popp earns: 6900
120 DEPARTMENT: 110
121  EMPLOYEE: Higgins earns: 12000
122  EMPLOYEE: Gietz earns: 8300
123  EMPLOYEE: Grant earns: 7000
124 *** END OF REPORT ***
** End of File **
124 lines read from file
```

Using UTL_FILE: Example

```

CREATE OR REPLACE PROCEDURE sal_status(
    p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
    f_file UTL_FILE.FILE_TYPE;
    CURSOR cur_emp IS
        SELECT last_name, salary, department_id
        FROM employees ORDER BY department_id;
    v_newdeptno employees.department_id%TYPE;
    v_olddeptno employees.department_id%TYPE := 0;
BEGIN
    f_file:= UTL_FILE.FOPEN (p_dir, p_filename, 'W');
    UTL_FILE.PUT_LINE(f_file,
        'REPORT: GENERATED ON ' || SYSDATE);
    UTL_FILE.NEW_LINE (f_file);
    . .

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide example, the `sal_status` procedure creates a report of employees for each department, along with their salaries. The data is written to a text file by using the `UTL_FILE` package.

In the code example, the `file` variable is declared as `UTL_FILE.FILE_TYPE`, is a record type which is private to the package `UTL_FILE`. This record has three fields, `id`, `data_type` and `byte_mode`. A variable of `FILE_TYPE` serves a file handle; you can perform various operations on the file with the help of this handle.

The `sal_status` procedure accepts two parameters:

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

For example, to call the procedure, use the following after ensuring that the external file and the database are on the same PC:

```
EXECUTE sal_status ('REPORTS_DIR', 'salreport2.txt')
```

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

Using UTL_FILE: Example

```

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

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Execute the `sal_status` procedure:

```
EXECUTE sal_status ('REPORTS_DIR', 'salreport2.txt')
```

The following is a sample of the `salreport2.txt` output file:

```

code_ex_06.sql * salreport2.txt *
Source History
Find |
```

Line Number	Content
1	REPORT: GENERATED ON 29-AUG-16
2	
3	DEPARTMENT: 10
4	
5	EMPLOYEE: Whalen earns: 4400
6	
7	DEPARTMENT: 20
8	
9	EMPLOYEE: Hartstein earns: 13000
10	
11	EMPLOYEE: Fay earns: 6000
12	
13	DEPARTMENT: 30
14	
15	EMPLOYEE: Raphaely earns: 11000
16	
17	EMPLOYEE: Khoo earns: 3100
18	
19	EMPLOYEE: Baida earns: 2900
20	

What Is the UTL_MAIL Package?

- Is a utility for managing email
- Requires the setting of the `SMTP_OUT_SERVER` database initialization parameter
- Provides the following procedures:
 - `SEND` for messages without attachments
 - `SEND_ATTACH_RAW` for messages with binary attachments
 - `SEND_ATTACH_VARCHAR2` for messages with text attachments



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

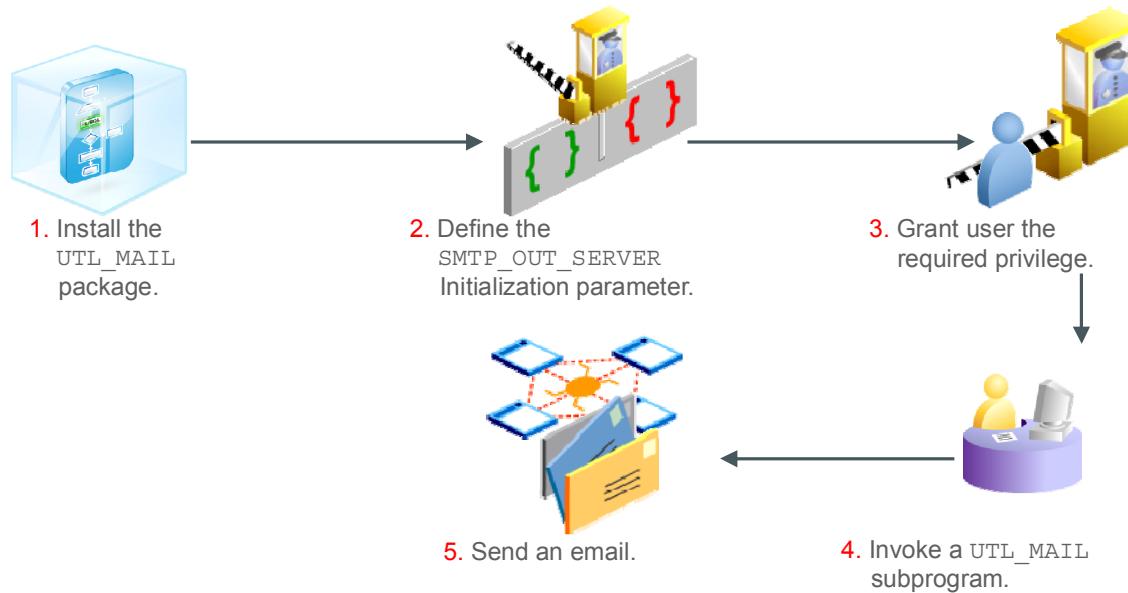
The `UTL_MAIL` package is not installed by default because of the `SMTP_OUT_SERVER` configuration requirement and the security exposure this involves. When installing `UTL_MAIL`, you should take steps to prevent the port defined by `SMTP_OUT_SERVER` being swamped by data transmissions. To install `UTL_MAIL`,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

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

You should define the `SMTP_OUT_SERVER` parameter in the `init.ora` initialization file. However, if `SMTP_OUT_SERVER` is not defined, this invokes a default of `DB_DOMAIN`, which is guaranteed to be defined to perform appropriately.

The `SMTP_OUT_SERVER` parameter specifies the SMTP host and port to which `UTL_MAIL` delivers outbound email. Multiple servers can be specified, separated by commas. If the first server in the list is unavailable, then `UTL_MAIL` tries the second server, and so on.

Setting Up and Using the UTL_MAIL: Overview



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Oracle Database 11g, the UTL_MAIL package is now an invoker's rights package and the invoking user will need the CONNECT privilege granted in the access control list assigned to the remote network host to which he wants to connect. The Security Administrator performs this task. You also require an EXECUTE privilege on UTL_MAIL to send a mail from the PL/SQL program.

Note

- For information about how a user with SYSDBA capabilities grants a user the required fine-grained privileges required for using this package, refer to the “Managing Fine-Grained Access to External Network Services” topic in *Oracle Database Security Guide* and the *Oracle Database Advanced PL/SQL* instructor-led training course.
- Because of firewall restrictions, the UTL_MAIL examples in this lesson cannot be demonstrated; therefore, no labs were designed to use UTL_MAIL.

Summary of UTL_MAIL Subprograms

Subprogram	Description
SEND procedure	Packages an email message, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients
SEND_ATTACH_RAW Procedure	Represents the SEND procedure overloaded for RAW attachments
SEND_ATTACH_VARCHAR2 Procedure	Represents the SEND procedure overloaded for VARCHAR2 attachments



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Installing and Using UTL_MAIL

- As SYSDBA, using SQL Developer or SQL*Plus:
 - Install the UTL_MAIL package

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

- Set the SMTP_OUT_SERVER

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

- As a developer, invoke a UTL_MAIL procedure:

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

The last example in the slide shows the simplest way to send a text message by using the UTL_MAIL.SEND procedure with at least a subject and a message. The first two required parameters are the following :

- The sender email address (in this case, otn@oracle.com)
- The recipients email address (for example, user@oracle.com). The value can be a comma-separated list of addresses.

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

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

The SEND Procedure Syntax

Packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients

```
UTL_MAIL.SEND (
    sender      IN      VARCHAR2 CHARACTER SET ANY_CS,
    recipients   IN      VARCHAR2 CHARACTER SET ANY_CS,
    cc          IN      VARCHAR2 CHARACTER SET ANY_CS
                        DEFAULT NULL,
    bcc         IN      VARCHAR2 CHARACTER SET ANY_CS
                        DEFAULT NULL,
    subject     IN      VARCHAR2 CHARACTER SET ANY_CS
                        DEFAULT NULL,
    message     IN      VARCHAR2 CHARACTER SET ANY_CS,
    mime_type   IN      VARCHAR2
                        DEFAULT 'text/plain; charset=us-ascii',
    priority    IN      PLS_INTEGER DEFAULT NULL
    replyto    IN      VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This procedure packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients. It hides the SMTP API and exposes a one-line email facility for ease of use.

The SEND Procedure Parameters

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipients, separated by commas
- **cc:** The email addresses of the CC recipients, separated by commas. The default is **NULL**.
- **bcc:** The email addresses of the BCC recipients, separated by commas. The default is **NULL**.
- **subject:** A string to be included as email subject string. The default is **NULL**.
- **message:** A text message body
- **mime_type:** The mime type of the message, default is '**text/plain; charset=us-ascii**'
- **priority:** The message priority. The default is **NULL**.
- **reply_to:** Defines to whom the reply email is to be sent

The SEND_ATTACH_RAW Procedure

This procedure is the SEND procedure overloaded for RAW attachments.

```
UTL_MAIL.SEND_ATTACH_RAW (
    sender          IN  VARCHAR2 CHARACTER SET ANY_CS,
    recipients     IN  VARCHAR2 CHARACTER SET ANY_CS,
    cc              IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    bcc             IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    subject         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    message         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    mime_type       IN  VARCHAR2 DEFAULT CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    priority        IN  PLS_INTEGER DEFAULT 3,
    attachment      IN  RAW,
    att_inline      IN  BOOLEAN DEFAULT TRUE,
    att_mime_type   IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    att_filename    IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL
    replyto         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL) ;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipients, separated by commas
- **cc:** The email addresses of the CC recipients, separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipients, separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **message:** A text message body
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'
- **priority:** The message priority. The default is NULL.
- **attachment:** A RAW attachment
- **att_inline:** Specifies whether the attachment is viewable inline with the message body. The default is TRUE.
- **reply_to:** Defines to whom the reply email is to be sent.

Sending Email with a Binary Attachment: Example

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

The `get_image` function in the example uses a `BFILE` to read the image data. Using a `BFILE` requires creating a logical directory name in the database by using the `CREATE DIRECTORY` statement. The code for `get_image` is shown on the following page.

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

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

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

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

Note

- You need the `CREATE ANY DIRECTORY` system privilege to execute this statement.
- Because of firewall restrictions at the Oracle Education Center, the examples on this page and the previous page are not available for demonstration.

The SEND_ATTACH_VARCHAR2 Procedure

This procedure is the SEND procedure overloaded for VARCHAR2 attachments.

```
UTL_MAIL.SEND_ATTACH_VARCHAR2 (
    sender          IN  VARCHAR2 CHARACTER SET ANY_CS,
    recipients     IN  VARCHAR2 CHARACTER SET ANY_CS,
    cc              IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    bcc             IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    subject         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    message         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    mime_type       IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    priority        IN  PLS_INTEGER DEFAULT 3,
    attachment      IN  VARCHAR2 CHARACTER SET ANY_CS,
    att_inline      IN  BOOLEAN DEFAULT TRUE,
    att_mime_type   IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    att_filename    IN  VARCHAR2CHARACTER SET ANY_CS DEFAULT NULL
    replyto         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipients, separated by commas
- **cc:** The email addresses of the CC recipients, separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipients, separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **Message:** A text message body
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'
- **priority:** The message priority. The default is NULL.
- **attachment:** A text attachment
- **att_inline:** Specifies whether the attachment is inline. The default is TRUE.
- **att_mime_type:** The mime type of the attachment, default is 'text/plain; charset=us-ascii'
- **att_filename:** The string specifying a file name containing the attachment. The default is NULL.

Sending Email with a Text Attachment: Example

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

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

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

```
CREATE DIRECTORY temp AS '/temp';
```

Note

- The `CREATE ANY DIRECTORY` system privilege is required to execute this statement.
- Because of firewall restrictions at the Oracle Education Center, the examples on this page and the previous page are not available for demonstration.

Quiz



The Oracle-supplied UTL_FILE package is used to access text files in the operating system of the database server.

The database provides functionality through directory objects to allow access to specific operating system directories.

- a. True
- b. False



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a

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

- A CREATE DIRECTORY statement that associates an alias with an operating system directory. The database directory alias can be granted the READ and WRITE privileges to control the type of access to files in the operating system.
- The paths specified in the utl_file_dir database initialization parameter

Summary

In this lesson, you should have learned:

- How the DBMS_OUTPUT package works
- How to use UTL_FILE to direct output to operating system files
- About the main features of UTL_MAIL



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

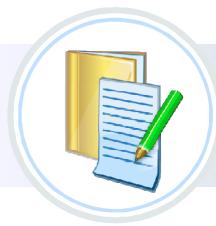
This lesson covers a small subset of packages provided with the Oracle database. You have extensively used DBMS_OUTPUT for debugging purposes and displaying procedurally generated information on the screen in SQL*Plus.

In this lesson, you should have learned how to use the power features provided by the database to create text files in the operating system by using UTL_FILE. You also learned how to send email with or without binary or text attachments by using the UTL_MAIL package.

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

Practice 7 Overview: Using Oracle-Supplied Packages in Application Development

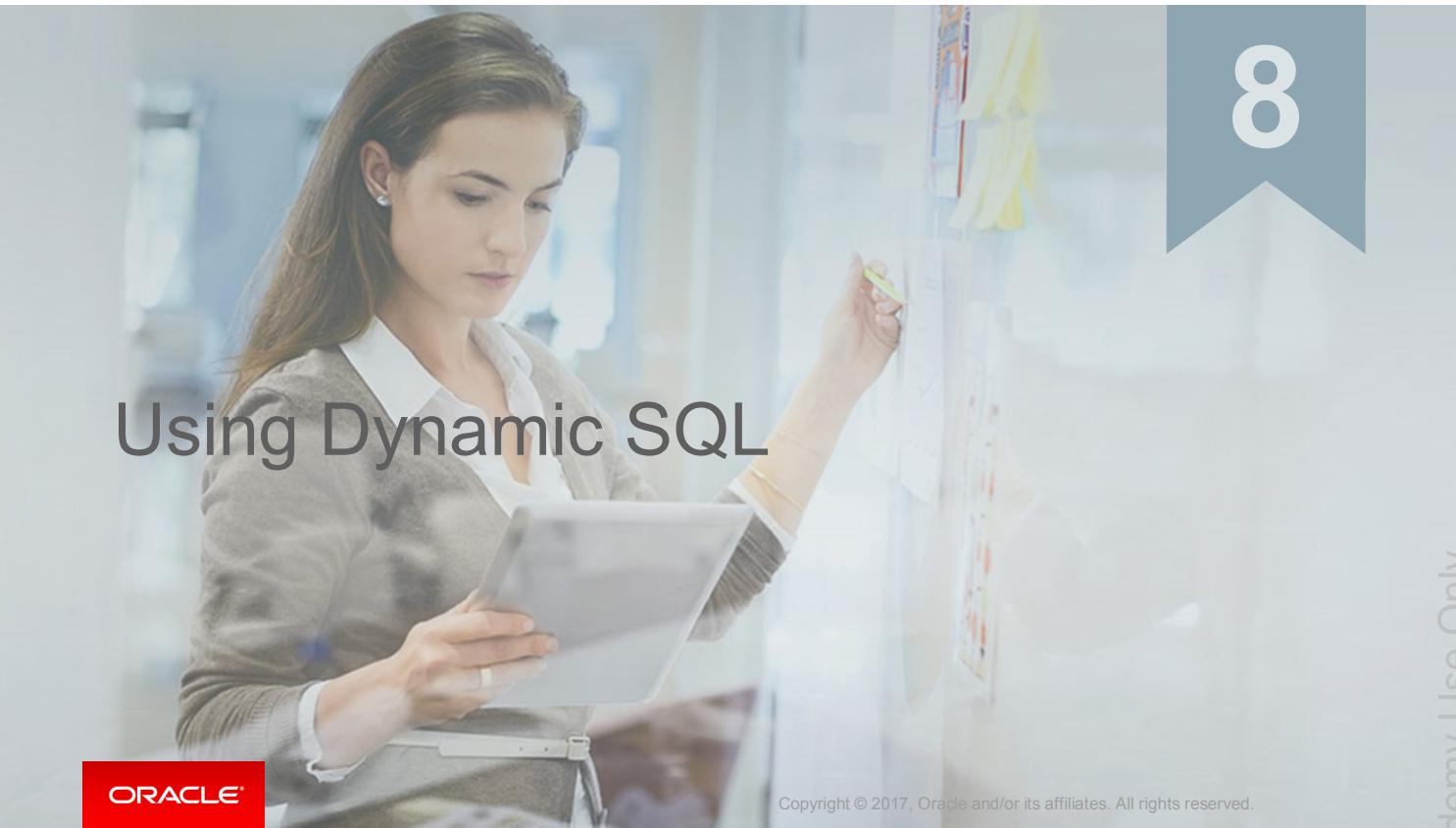
This practice covers how to use `UTL_FILE` to generate a text report.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this practice, you use `UTL_FILE` to generate a text file report of employees in each department.



8

Using Dynamic SQL

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically by using Native Dynamic SQL (NDS)
- Identify situations when you must use the DBMS_SQL package instead of NDS to build and execute SQL statements dynamically



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn to construct and execute SQL statements dynamically, that is, at run time by using NDS statements in PL/SQL.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with the PL/SQL Code

- ▶ Lesson 2: Creating Procedures
- ▶ Lesson 3: Creating Functions
- ▶ Lesson 4: Debugging Subprograms
- ▶ Lesson 5: Creating Packages
- ▶ Lesson 6: Working with Packages
- ▶ Lesson 7: Using Oracle-Supplied Packages in Application Development
- ▶ **Lesson 8: Using Dynamic SQL**

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units – Working with Subprograms, Working with Triggers, and Working with the PL/SQL code.

In the first unit, you have seven lessons – Creating Stored Procedures, Creating Functions, Debugging Subprograms, Creating Packages, Working with Packages, Using Oracle Supplied Packages in Application Development, and Using Dynamic SQL. You will learn to write PL/SQL subprograms and packages, and use them in Application Development.

Lesson Agenda

- Dynamic SQL and its features
- Using NDS
- Using the DBMS_SQL package



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

What is Dynamic SQL?

- Dynamic SQL enables you to generate and run SQL statements at run time.
- You may write a dynamic SQL statement when you :
 - Don't know the full text of the SQL statement at compile time
 - Don't know the number of variables
 - Don't know data types of input variables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When do you use Dynamic SQL?

Hey Alice, I'm not quite good at writing codes to get my jobs done. Can you make it simpler, where I click some buttons and enter some values to get the job done? Some thing which will look like this:



Employee ID:

New Salary:

SUBMIT

I could just enter the values and click submit, and the data would be updated in the database.

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider the requirement of the human resource (HR) manager in this scenario. The HR manager wants a simple user interface in which he/she could enter the values and the values would be updated in the database. As a PL/SQL developer, the user interface is not your focus here. You can create such a user interface by using frontend technologies like HTML.

As a PL/SQL developer, whose job is to reflect the changes onto the database, you have a challenge of handling unknown data. The data is known only at the run time, but not at compile time. When the HR manager enters the values from the front end, the data has to reach the PL/SQL block that updates the database. The values of the `employee_id` and `new salary` are passed onto the PL/SQL block that in turn runs the update.

You can use dynamic SQL in this case, i.e. where the developer has to handle unknown data. Apart from unknown data, you can use dynamic SQL when you don't know about the data types of the variables and so on. In the following slides, we will look at how you can use dynamic SQL for changing the requirements of SQL statements.

If the HR manager enters 100 for `employee_id` and 4000 for salary, then the query should be:

```
UPDATE employees
SET salary = 4000
WHERE employee_id = 100;
```

If the HR manager enters some other values, like 150 for `employee_id` and 3000 for salary, then the corresponding SQL statement will change. The actual query is known only at run time. It is ideal to use dynamic SQL in such situations.

Using Dynamic SQL

- Use dynamic SQL when the full text of the dynamic SQL statement is unknown until run time. Therefore, its syntax is checked at *run time* rather than at *compile time*.
- Use dynamic SQL when one of the following items is unknown at precompile time:
 - Text of the SQL statement such as commands, clauses, and so on
 - The number and data types of host variables
 - References to database objects such as tables, columns, indexes, sequences, usernames, and views
- Use dynamic SQL to make your PL/SQL programs more general and flexible.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In PL/SQL, you need dynamic SQL to execute the following SQL statements where the full text is unknown at compile time, such as:

- A SELECT statement that includes an identifier that is unknown at compile time (such as a table name)
- A WHERE clause in which the column name is unknown at compile time

Note

For additional information about dynamic SQL, see the following resources:

- *Pro*C/C++ Programmer's Guide*
 - *Lesson 13, Oracle Dynamic SQL*, covers the four available methods that you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method and then offers guidelines for choosing the right method. Later sections in the guide show you how to use the methods and include example programs that you can study.
 - *Lesson 15, Oracle Dynamic SQL: Method 4*, contains detailed information about Method 4 (query with unknown number of select-list items or input host variables, more on slide 21) when defining dynamic SQL statements.
- *Oracle PL/SQL Programming* by Steven Feuerstein and Bill Pribyl
- *Lesson 16, Dynamic SQL and Dynamic PL/SQL*, contains additional information about dynamic SQL.

Execution Flow of SQL Statements

- All SQL statements go through some or all of the following stages:



- Some stages may not be relevant for all statements:
 - The fetch phase is applicable to queries.
 - For embedded SQL statements such as SELECT, DML, MERGE, COMMIT, SAVEPOINT, and ROLLBACK, the parse and bind phases are performed at compile time.
 - For dynamic SQL statements, all phases are performed at run time.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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

Dynamic SQL implementation

- PL/SQL provides the following two methods to use Dynamic SQL in PL/SQL blocks:
 - NDS
 - DBMS_SQL package



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

There are two methods of implementing dynamic SQL in PL/SQL blocks:

1. **NDS** – It is a PL/SQL language feature for building and running dynamic SQL statements.
2. **DBMS_SQL package** – It is an API that enables you to build, run, and describe dynamic SQL statements.

To write an NDS code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement. If you do not have this information at compile time, you must use the `DBMS_SQL` package.

Lesson Agenda

- Dynamic SQL and its features
- NDS
- Using the DBMS_SQL package



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Native Dynamic SQL (NDS)

- Provides native support for dynamic SQL directly in the PL/SQL language
- Processes most dynamic SQL statements with the `EXECUTE IMMEDIATE` statement
- Gives the following choices if the dynamic SQL statement is a `SELECT` statement that returns multiple rows:
 - Use the `EXECUTE IMMEDIATE` statement with the `BULK COLLECT INTO` clause
 - Use the `OPEN-FOR`, `FETCH`, and `CLOSE` statements



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

NDS provides the ability to dynamically execute SQL statements whose structure is constructed at execution time. NDS supports it through the following statements:

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

Using the EXECUTE IMMEDIATE Statement

Use the EXECUTE IMMEDIATE statement for NDS or PL/SQL anonymous blocks:

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

- **INTO** is used for single-row queries and specifies the variables or records into which column values are retrieved.
- **USING** is used to hold all bind arguments. The default parameter mode is **IN**.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

- **dynamic_string** is a string expression that represents a dynamic SQL statement (without a terminator) or a PL/SQL block (with a terminator).
- The **INTO** clause specifies the variables or record into which column values are retrieved. It is used only for single-row queries. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the **INTO** clause.
- **define_variable** is a PL/SQL variable that stores the selected column value.
- **record** is a user-defined or **%ROWTYPE** record that stores a selected row.
- **bind_argument** is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.
- The **USING** clause holds all bind arguments, which are input to the dynamic SQL statement to be executed. The default parameter mode is **IN**. You cannot pass a **NULL** literal value; instead you can pass an expression that might evaluate to **NULL**.

NDS supports all SQL data types. The input to the dynamic SQL statement through the bind arguments can be strings, numbers, dates, schema-level collections, LOBs, associative arrays, and so on. The **INTO** clause can have PL/SQL variables or records that correspond to the expected output of the dynamic SQL statement.

Note: Use **OPEN-FOR**, **FETCH**, and **CLOSE** for a multi row query.

Dynamic SQL with a DDL Statement: Examples

```
-- Create a table using dynamic SQL

CREATE OR REPLACE PROCEDURE create_table(
    p_table_name VARCHAR2, p_col_specs VARCHAR2) IS
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE ' || p_table_name ||
                      ' (' || p_col_specs || ')';
END;
/
```

```
-- Call the procedure

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The code examples show the creation of a `create_table` procedure that accepts the table name and column definitions (specifications) as parameters.

The procedure call shows the creation of a table called `EMPLOYEE_NAMES` with two columns:

- An ID column with a `NUMBER` data type used as a primary key
- A name column of up to 40 characters for the employee name

Any data definition language (DDL) statement can be executed by using the syntax shown in the slide, whether the statement is dynamically constructed or specified as a literal string. You can create and execute a statement that is stored in a PL/SQL string variable, as in the following example:

```
CREATE OR REPLACE PROCEDURE add_col(p_table_name VARCHAR2,
                                    p_col_spec    VARCHAR2) IS
    v_stmt VARCHAR2(100) := 'ALTER TABLE ' || p_table_name ||
                           ' ADD ' || p_col_spec;
BEGIN
    EXECUTE IMMEDIATE v_stmt;
END;
/
```

To add a new column to a table, enter the following:

```
EXECUTE add_col('employee_names', 'salary number(8,2)')
```

Dynamic SQL with DML Statements

```
-- Insert a row into a table with two columns:
CREATE PROCEDURE add_row(p_table_name VARCHAR2,
    p_id NUMBER, p_name VARCHAR2) IS
BEGIN
    EXECUTE IMMEDIATE 'INSERT INTO '|| p_table_name ||
        ' VALUES (:1, :2)' USING p_id, p_name;
END;
```

```
-- Delete rows from any table:
CREATE FUNCTION del_rows(p_table_name VARCHAR2)
RETURN NUMBER IS
BEGIN
    EXECUTE IMMEDIATE 'DELETE FROM '|| p_table_name;
    RETURN SQL%ROWCOUNT;
END;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The first code example in the slide defines a dynamic SQL statement, which isn't a query and is without host variables. The examples in the slide demonstrate the following:

- The `add_row` procedure shows how to provide input values to a dynamic SQL statement with the `USING` clause. The bind variable names `:1` and `:2` are not important; however, the order of the parameter names (`p_id` and `p_name`) in the `USING` clause is associated with the bind variables by position, in the order of their respective appearance. Therefore, the PL/SQL parameter `p_id` is assigned to the `:1` placeholder, and the `p_name` parameter is assigned to the `:2` placeholder. Placeholder or bind variable names can be alphanumeric but must be preceded with a colon.
- The `del_rows` function deletes rows from a specified table and returns the number of rows deleted by using the implicit SQL cursor `%ROWCOUNT` attribute. Executing the function is shown below the example for creating a function.

Execute the following code to see the execution of both the procedures:

```
BEGIN
    add_row('EMPLOYEE_NAMES',100,'KING');
    DBMS_OUTPUT.PUT_LINE(
        del_rows('EMPLOYEE_NAMES') || ' rows deleted.');
END;
/
```

Note: The `EXECUTE IMMEDIATE` statement prepares (parses) and immediately executes the dynamic SQL statement. Dynamic SQL statements are always parsed.

Also, note that a `COMMIT` operation is not performed in either example. Therefore, the operations can be undone with a `ROLLBACK` statement.

Dynamic SQL with a Single-Row Query: Example

```

CREATE FUNCTION get_emp( p_emp_id NUMBER )
RETURN employees%ROWTYPE IS
    v_stmt VARCHAR2(200);
    v_emprec employees%ROWTYPE;
BEGIN
    v_stmt := 'SELECT * FROM employees ' ||
              'WHERE employee_id = :p_emp_id';
    EXECUTE IMMEDIATE v_stmt INTO v_emprec USING p_emp_id ;
    RETURN v_emprec;
END;
/
DECLARE
    v_emprec employees%ROWTYPE := get_emp(100);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Emp: ' || v_emprec.last_name);
END;
/

```

Function GET_EMP compiled
 PL/SQL procedure successfully completed.
 Emp: King



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The code example in the slide is an example of defining a dynamic SQL statement with a single row queried—that is, query with a known number of select-list items and input host variables.

The single-row query example demonstrates the `get_emp` function that retrieves an `EMPLOYEES` record into a variable specified in the `INTO` clause. It also shows how to provide input values for the `WHERE` clause.

The anonymous block is used to execute the `get_emp` function and return the result into a local `EMPLOYEES` record variable.

The example could be enhanced to provide alternative `WHERE` clauses depending on input parameter values, making it more suitable for dynamic SQL processing.

Note

- For an example of “Dynamic SQL with a Multirow Query: Example” using `REF CURSORS`, see the `demo_07_13_a` in the `/home/oracle/labs/plpu/demo` folder.
- For an example of using `REF CURSORS`, see the `demo_07_13_b` in the `/home/oracle/labs/plpu/demo` folder.
- `REF CURSORS` are covered in the *Oracle Database: Advanced PL/SQL* course.

Executing a PL/SQL Anonymous Block Dynamically

```

CREATE FUNCTION annual_sal( p_emp_id NUMBER)
RETURN NUMBER IS
  v_plsql varchar2(200) := 
    'DECLARE ' ||
    '  rec_emp employees%ROWTYPE; ' ||
    'BEGIN ' ||
    '  rec_emp := get_emp(:empid); ' ||
    '  :res := rec_emp.salary * 12; ' ||
    'END;';
  v_result NUMBER;
BEGIN
  EXECUTE IMMEDIATE v_plsql
    USING IN p_emp_id, OUT v_result;
  RETURN v_result;
END;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))

```

Anonymous
PL/SQL block
initialized to
variable v_plsql



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The `annual_sal` function dynamically constructs an anonymous PL/SQL block. The PL/SQL block contains bind variables for:

- The input of the employee ID using the `:empid` placeholder
- The output result computing the annual employees' salary using the placeholder called `:res`

Note: This example demonstrates how to use the `OUT` result syntax (in the `USING` clause of the `EXECUTE IMMEDIATE` statement) to obtain the result calculated by the PL/SQL block. The procedure output variables and function return values that can be obtained in a similar way from a dynamically executed anonymous PL/SQL block.

The output of the slide example is as follows:

```

Function ANNUAL_SAL compiled

PL/SQL procedure successfully completed.

288000

```

BULK COLLECT INTO clause

- It is used instead of `INTO` in SQL statements retrieving data into PL/SQL block.
- The variable followed by `BULK COLLECT INTO` should be capable of holding multiple rows such as a collection or a cursor.
- Improves the performance of a PL/SQL block with SQL statements retrieving huge volumes of data
- The presence of `BULK COLLECT INTO` in the `SELECT` statements returns the results from SQL to PL/SQL engine in batches instead of one at a time.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You use the `BULK COLLECT INTO` clause in `SELECT` statements or any other SQL statements that retrieve data in huge volumes.

While executing a PL/SQL block with SQL statements, the SQL engine and PL/SQL engine communicate with each other. The SQL statement execution in the PL/SQL block is handed over to the SQL engine. The SQL engine returns the result to PL/SQL statement as it processes the statement. This exchange of information between the SQL engine and PL/SQL engine results in context switches that further add to the performance overhead.

Executing a `SELECT` statement with `BULK COLLECT INTO` clause results in SQL returning results to PL/SQL in batches instead of one at a time, thus improving the performance of the PL/SQL block.

OPEN FOR clause

- The OPEN FOR clause is used to associate a cursor variable with a query.

```
OPEN { cursor_variable | :host_cursor_variable }
FOR select_statement [ using_clause ] ;
```

- It allocates database resources to process the query.
- It identifies the result set based on the query.
- If the query has a FOR UPDATE clause, then the OPEN FOR statement locks the rows of the result set.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using BULK COLLECT and OPEN FOR clause

```
DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    TYPE NumList IS TABLE OF NUMBER;
    TYPE NameList IS TABLE OF VARCHAR2(25);
    emp_cv EmpCurTyp;
    empids NumList;
    enames NameList;
    sals   NumList;
BEGIN
    OPEN emp_cv FOR 'SELECT employee_id, last_name FROM employees';
    FETCH emp_cv BULK COLLECT INTO empids, enames;
    CLOSE emp_cv;
    DBMS_OUTPUT.PUT_LINE(empids.count);
END;
/
```

PL/SQL procedure successfully completed.
107



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The code in the slide demonstrates the usage of BULK COLLECT and OPEN FOR statements.

The OPEN FOR statement in the code opens a REF CURSOR for the SELECT statement. To learn more about REF CURSORS, you can refer to appendix REF CURSORS.

Summarizing Methods for Using Dynamic SQL

Method #	SQL Statement Type	NDS SQL Statements Used
Method 1	Non-query without host variables	EXECUTE IMMEDIATE without the USING and INTO clauses
Method 2	Non-query with known number of input host variables	EXECUTE IMMEDIATE with a USING clause
Method 3 (returning a single row)	Query with known number of select-list items and input host variables returning a single row as result set	EXECUTE IMMEDIATE with the USING and INTO clauses
Method 3 (returning multiple rows)	Query with known number of select-list items and input host variables returning multiple rows as result set	EXECUTE IMMEDIATE with cursors using BULK COLLECT and OPEN FOR clauses
Method 4	Query with unknown number of select-list items or input host variables	Use the DBMS_SQL package



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide we discuss the four available methods for NDS. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and Method 4 encompasses Methods 1, 2, and 3. However, each method is most useful for handling a certain kind of SQL statement, as follows:

Method 1 applies to those SQL statements that don't accept host variables as input and those that don't return any results. They are also known as non-query statements.

You execute them directly with EXECUTE IMMEDIATE as in this example

```
EXECUTE IMMEDIATE 'DELETE FROM EMPLOYEES WHERE DEPTNO = 20'
```

With Method 1, the SQL statement is parsed every time it is executed. Examples of non-queries include DDL statements, UPDATES, INSERTS, or DELETES.

Method 2 applies to those SQL statements that accept from the host environment in the form of host variables.

You execute such statements by using EXECUTE IMMEDIATE with the USING clause as shown below:

```
EXECUTE IMMEDIATE 'INSERT INTO EMPLOYEES (FIRST_NAME, LAST_NAME, JOB_ID)
VALUES (:emp_first_name, :emp_last_name, :job_id)' USING 'Stephen',
'King', 'AD_VP'
```

The number of placeholders for input host variables and the data types of the input host variables must be known at precompile time.

Method 3 applies to those state SQL statements that accept a known number of variables and return a single row or multiple rows as the result set of the query.

For example, the following host strings qualify:

```
EXECUTE IMMEDIATE SELECT DEPARTMENT_ID, SALARY  
  FROM EMPLOYEES  
 WHERE DEPARTMENT_ID = :dep_num  
   INTO DEP_ID, SAL;
```

You may also have queries that accept a known number of input variables and return multiple rows. In such a case, you use OPEN FOR, FETCH, and CLOSE statements to retrieve the data into a cursor and perform operations on it.

Method 4

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables. With this method, you use the DBMS_SQL package, which is covered in the following slides.

Note

For additional information about the four dynamic SQL methods, see the following lessons in the *Pro*C/C++ Programmer's Guide*.

- *Lesson 13, Oracle Dynamic SQL*
- *Lesson 15, Oracle Dynamic SQL: Method 4*

Lesson Agenda

- Dynamic SQL and its features
- NDS
- Using the DBMS_SQL package



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.



Hey Alice, can you create a simple interface where I can just enter the table name, column names, and output I want to see in the user interface, and I get the job done on submission to the server?

Yes...I'll be able to do that.



For the audience that is wondering how we write the query, the DBMS_SQL package is the answer



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.



In a situation of uncertainty, where you aren't sure about the table on which the query would execute or about the number of columns you would retrieve from the table or about the output, you can use the procedures provided by the DBMS_SQL package.

You first construct the query based on user input with respect to the tables, columns, and so on. Then, you execute the query. The DBMS_SQL package provides various procedures for the purpose.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using the DBMS_SQL Package

- The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements.
- You must use the DBMS_SQL package to execute a dynamic SQL statement that has an unknown number of input or output variables, also known as Method 4.
- In most cases, NDS is easier to use and performs better than DBMS_SQL, except when dealing with Method 4.
- For example, you must use the DBMS_SQL package in the following situations:
 - You do not know the SELECT list at compile time.
 - You do not know how many columns a SELECT statement will return or what their data types will be.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

By using DBMS_SQL, you can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL, such as executing DDL statements in PL/SQL. An example is executing a DROP TABLE statement. The operations provided by this package are performed under the current user, and not under the package owner SYS.

Method 4: Method 4 refers to situations where, in a dynamic SQL statement, the number of columns selected for a query or the number of bind variables set is not known until run time. In this case, you should use the DBMS_SQL package.

Using the DBMS_SQL Package Subprograms

The procedures used through the execution flow of dynamic SQL statements using DBMS_SQL package:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE or BIND_ARRAY
- DEFINE_COLUMN, DEFINE_COLUMN_LONG or DEFINE_ARRAY
- EXECUTE
- FETCH_ROWS or EXECUTE_AND_FETCH
- VARIABLE_VALUE, COLUMN_VALUE or COLUMN_VALUE_LONG
- CLOSE_CURSOR



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The DBMS_SQL package provides the following subprograms to execute dynamic SQL:

- **OPEN_CURSOR:** To process a SQL statement, you must have an open cursor. The DBMS_SQL package provides OPEN_CURSOR functions, which when called return a cursor ID number for a valid cursor. These cursors are used only by the DBMS_SQL package.
- **PARSE** procedures: Every SQL statement must be parsed by calling the PARSE procedures. Parsing the statement checks the statement's syntax and associates it with the cursor in your program. DDL statements are immediately executed when parsed.
- **BIND_VARIABLE OR BIND_ARRAY:** You have to define bind variables when the SQL statement you execute need input data to be supplied at runtime. The SQL statement should have placeholders to mark where the data has to be supplied. You must invoke a BIND_ARRAY or BIND_VARIABLE procedure for each placeholder.
- **DEFINE_COLUMN OR DEFINE_ARRAY:** If the result of the query has to be received in the PL/SQL block, then you must call one of the define procedures of the package. You use the DEFINE_COLUMN procedure when you have to receive a column of values and you use the DEFINE_ARRAY procedure when you have to retrieve a collection of values.
- **EXECUTE:** The EXECUTE function is invoked to execute the SQL statement and return the number of rows processed.
- **FETCH:** The FETCH_ROWS function is invoked to retrieve the rows from the result set of a query. You execute FETCH_ROWS multiple times to retrieve consequent set of rows.
- **COLUMN_VALUE:** The COLUMN_VALUE procedure is invoked to retrieve the value of the column fetched through FETCH_ROWS.
- **CLOSE_CURSOR:** The CLOSE_CURSOR procedure is used to close the specified cursor.

Using DBMS_SQL with a DML Statement: Deleting Rows

```
CREATE OR REPLACE FUNCTION delete_all_rows
(p_table_name  VARCHAR2) RETURN NUMBER IS
  v_cur_id      INTEGER;
  v_rows_del    NUMBER;
BEGIN
  v_cur_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQLPARSE(v_cur_id,
    'DELETE FROM '|| p_table_name, DBMS_SQL.NATIVE);
  v_rows_del := DBMS_SQL.EXECUTE (v_cur_id);
  DBMS_SQLCLOSE_CURSOR(v_cur_id);
  RETURN v_rows_del;
END;
/
CREATE TABLE temp_emp AS SELECT * FROM employees;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' || delete_all_rows('temp_emp'));
END;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide, the table name is passed into the `delete_all_rows` function. The function uses dynamic SQL to delete rows from the specified table and returns a count representing the number of rows that are deleted after successful execution of the statement.

To process a DML statement dynamically, perform the following steps:

1. Use `OPEN_CURSOR` to establish an area in memory to process a SQL statement.
2. Use `PARSE` to establish the validity of the SQL statement. `DBMS_SQL.NATIVE` is a constant that is the value for `language_flag` parameter of `PARSE` procedure. It implies normal behavior for the connected database.
3. Use the `EXECUTE` function to run the SQL statement. This function returns the number of rows processed.
4. Use `CLOSE_CURSOR` to close the cursor.

The steps to execute a DDL statement are similar, but step 3 is optional because a DDL statement is immediately executed when the PARSE is successfully done—that is, the statement syntax and semantics are correct. If you use the EXECUTE function with a DDL statement, then it does not do anything and returns a value of 0 for the number of rows processed because DDL statements do not process rows.

```
Function DELETE_ALL_ROWS compiled  
Table TEMP_EMP created.  
PL/SQL procedure successfully completed.  
Rows Deleted: 107  
Table TEMP_EMP dropped.
```

Using DBMS_SQL with a Parameterized DML Statement

```

CREATE PROCEDURE insert_row (p_table_name VARCHAR2,
    p_id VARCHAR2, p_name VARCHAR2, p_region NUMBER) IS
    v_cur_id      INTEGER;
    v_stmt        VARCHAR2(200);
    v_rows_added NUMBER;
BEGIN
    v_stmt := 'INSERT INTO '|| p_table_name |||
              ' VALUES (:cid, :cname, :rid)';
    v_cur_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQLPARSE(v_cur_id, v_stmt, DBMS_SQL.NATIVE);
    DBMS_SQL.BIND_VARIABLE(v_cur_id, ':cid', p_id);
    DBMS_SQL.BIND_VARIABLE(v_cur_id, ':cname', p_name);
    DBMS_SQL.BIND_VARIABLE(v_cur_id, ':rid', p_region);
    v_rows_added := DBMS_SQL.EXECUTE(v_cur_id);
    DBMS_SQL.CLOSE_CURSOR(v_cur_id);
    DBMS_OUTPUT.PUT_LINE(v_rows_added||' row added');
END;
/

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide performs the DML operation to insert a row into a specified table. The example demonstrates the extra step required to associate values to bind variables that exist in the SQL statement. For example, a call to the procedure shown in the slide is:

```
EXECUTE insert_row('countries', 'LB', 'Lebanon', 4)
```

After the statement is parsed, you must call the DBMS_SQL.BIND_VARIABLE procedure to assign values for each bind variable that exists in the statement. The binding of values must be done before executing the code. To process a SELECT statement dynamically, perform the following steps after opening and before closing the cursor:

1. Execute DBMS_SQL.DEFINE_COLUMN for each column selected.
2. Execute DBMS_SQL.BIND_VARIABLE for each bind variable in the query.
3. For each row, perform the following steps:
 - a. Execute DBMS_SQL.FETCH_ROWS to retrieve a row and return the number of rows fetched. Stop additional processing when a zero value is returned.
 - b. Execute DBMS_SQL.COLUMN_VALUE to retrieve each selected column value into each PL/SQL variable for processing.

Although this coding process is not complex, it is more time consuming to write and is prone to error compared to using the Native Dynamic SQL approach.

Quiz



The full text of the dynamic SQL statement might be unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.

- a. True
- b. False



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a

Summary

In this lesson, you should have learned how to:

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically by using NDS
- Identify situations when you must use the `DBMS_SQL` package instead of NDS to build and execute SQL statements dynamically



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learned how to dynamically create a SQL statement and execute it by using NDS statements. Dynamically executing SQL and PL/SQL code extends the capabilities of PL/SQL beyond query and transactional operations. For earlier releases of the database, you could achieve similar results with the `DBMS_SQL` package.

Practice 8 Overview: Using Dynamic SQL

This practice covers the following topics:

- Creating a package that uses Native Dynamic SQL to create or drop a table and to populate, modify, and delete rows from a table
- Creating a package that compiles the PL/SQL code in your schema

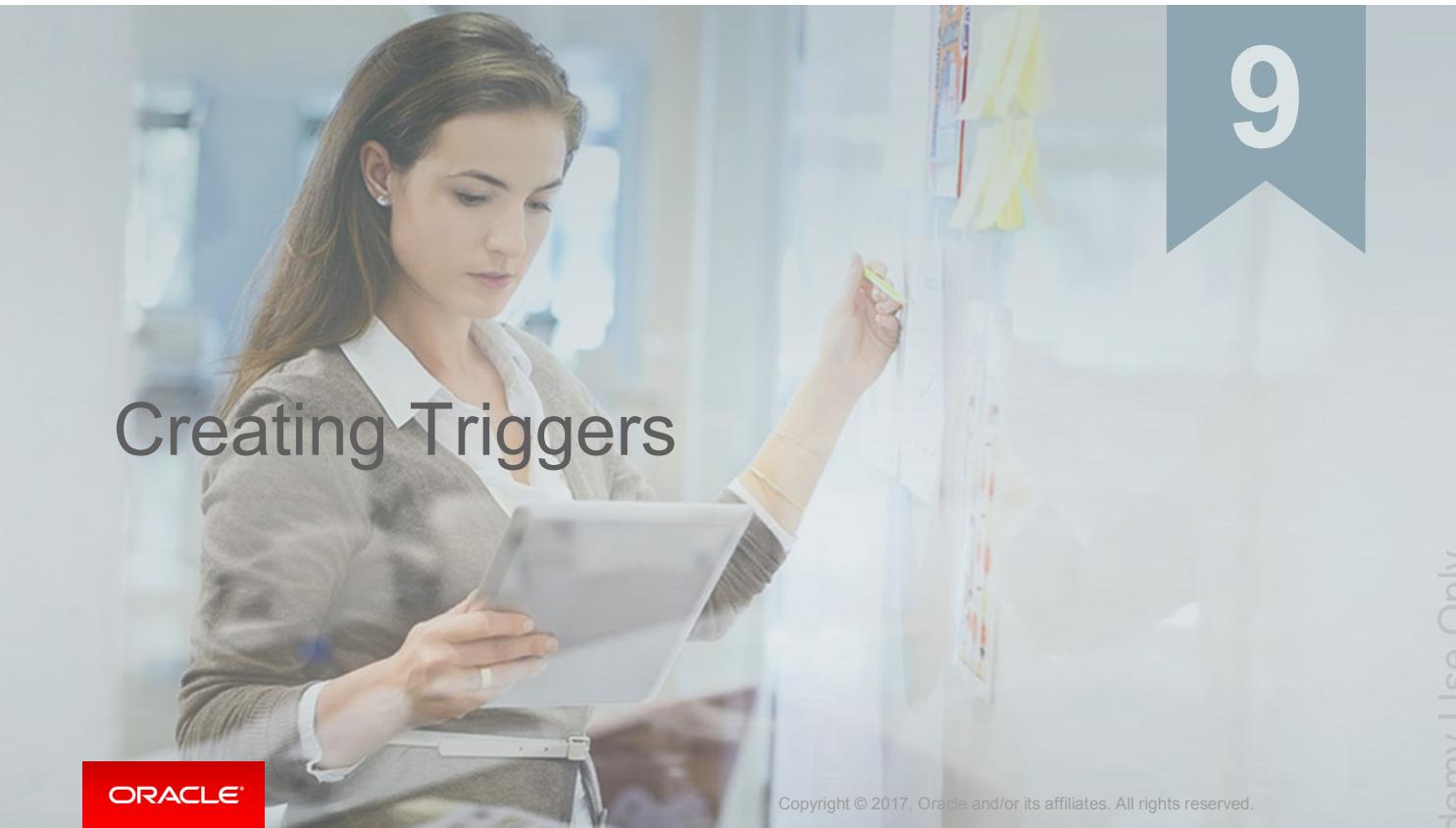


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this practice, you write code to perform the following tasks:

- Create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.
- Create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an `INVALID` status in the `USER_OBJECTS` table.



9

Creating Triggers

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe database triggers and their uses
- Describe the different types of triggers
- Create database triggers
- Describe database trigger-firing rules
- Remove database triggers
- Display trigger information



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

Unit 3: Working with the PL/SQL Code

▶ Lesson 9: Creating Triggers

▶ Lesson 10: Creating Compound, DDL, and Event Database Triggers

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 2, you learn to create and use triggers in database applications.

Lesson Agenda

- Understanding the usage of triggers
- Creating triggers
- Creating row-level triggers
- Creating INSTEAD OF triggers
- Managing Triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.



Hey Alice, I have a situation where I have to keep track of the amount of salary increase given to an employee whenever it is given. I can request the managers to update a certain table whenever they give a hike, but usually they forget about this update. Can you automate this?

I can definitely create something for this requirement. So, friends, let's get to work!

We have triggers in PL/SQL that can help us in creating something that the HR manager has asked for. Let's look at the concept first.

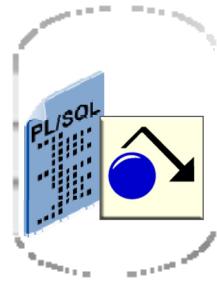


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

What are Triggers?

- A trigger is a PL/SQL block that is stored in the database and fired (executed) in response to a specified event.
- The Oracle database automatically executes a trigger when specified conditions occur.



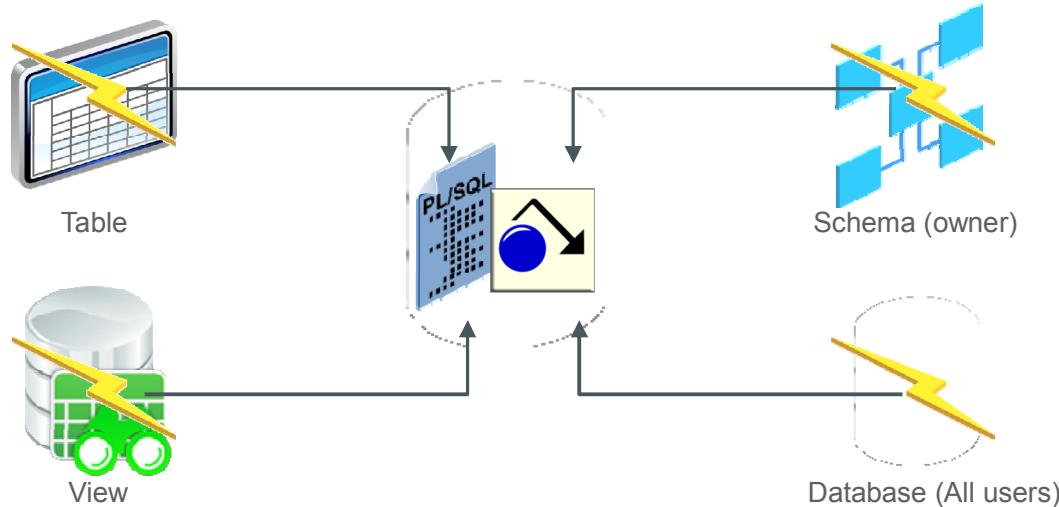
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Triggers are similar to stored procedures. A trigger stored in the database contains PL/SQL in the form of an anonymous block, a call statement, or a compound trigger block. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger. Triggers are implicitly fired by the Oracle database when a triggering event occurs, regardless of which user is connected and which application is being used.

Defining Triggers

A trigger can be defined on the table, view, schema (schema owner), or database (all users).



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can create a trigger on a table, view, schema, or database. You can also specify the time when the trigger is expected to be invoked. You generally define a trigger for an event and define it to be invoked before or after the event.

Why do you use Triggers?

You might use a trigger for one of the following purposes:

- To log events
- To gather statistics on table access
- To modify table data when DML statements are issued against views
- To enforce referential integrity constraints when child and parent tables are on different nodes
- To prevent DML operations on a table based on application requirements
- To prevent invalid transactions
- To implement complex business rules

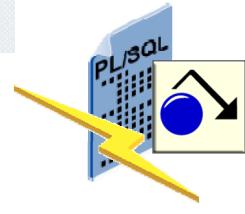


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Trigger Event Types

You can write triggers that fire whenever one of the following operations occurs in the database:

DML Statements	DDL Statements	Database Events
<ul style="list-style-type: none">• DELETE• INSERT• UPDATE	<ul style="list-style-type: none">• CREATE• ALTER• DROP	<ul style="list-style-type: none">• SERVERERROR• LOGON• LOGOFF• STARTUP• SHUTDOWN



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:

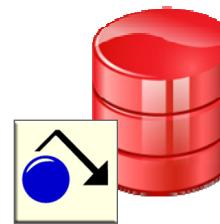
- An **INSERT**, **UPDATE**, or **DELETE** statement on a specific table (or view, in some cases)
- A **CREATE**, **ALTER**, or **DROP** statement on any schema object
- A database startup or instance shutdown
- A specific error message or any error message
- A user logon or logoff

Available Trigger Types

- Simple DML triggers
 - BEFORE
 - AFTER
 - INSTEAD OF
- Compound triggers
- System triggers
 - DDL event triggers
 - Database event triggers



Application triggers



Database triggers



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Triggers are classified based on the time when they are defined to be executed, how they are organized, and for what type of statements they execute.

Simple DML triggers are defined for DML statements. These triggers can execute BEFORE, AFTER, or INSTEAD OF the triggering statement.

Compound triggers are those that have a single trigger defined to execute over multiple events.

System triggers are also known as Schema triggers, which are defined to execute on events pertaining to the database or schema. The triggering event might be either a DDL statement or a database operation.

Note: In this lesson, you will learn about the BEFORE, AFTER, and INSTEAD OF triggers. The other trigger types are discussed in the lesson titled “Creating Compound, DDL, and Event Database Triggers.”

Trigger Event Types and Body

- A trigger event type determines which DML statement causes the trigger to execute. The possible events are:
 - INSERT
 - UPDATE [OF column]
 - DELETE
- A trigger body determines what action is performed and is a PL/SQL block or a CALL to a procedure.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The triggering event or statement can be an INSERT, UPDATE, or DELETE statement on a table.

- When the triggering event is an UPDATE statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement because it always affects entire rows.
 . . . UPDATE OF salary . . .
- The triggering event can contain one, two, or all three of these DML operations.
 . . . INSERT or UPDATE or DELETE
 . . . INSERT or UPDATE OF job_id . . .

The trigger body defines the action—that is, what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Note: The size of a trigger cannot be greater than 32 KB.

Lesson Agenda

- Understanding the usage of triggers
- Creating triggers
- Creating row-level triggers
- Creating INSTEAD OF triggers
- Managing Triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Creating DML Triggers by Using the CREATE TRIGGER Statement

```

CREATE [OR REPLACE] TRIGGER trigger_name
timing -- when to fire the trigger
event1 [OR event2 OR event3]
ON object_name
[REFERENCING OLD AS old / NEW AS new]
FOR EACH ROW -- default is statement level trigger
WHEN (condition) ]
DECLARE]
BEGIN
... trigger_body -- executable statements
[EXCEPTION . . .]
END [trigger_name];

```

timing = BEFORE | AFTER | INSTEAD OF

event = INSERT | DELETE | UPDATE | UPDATE OF column list

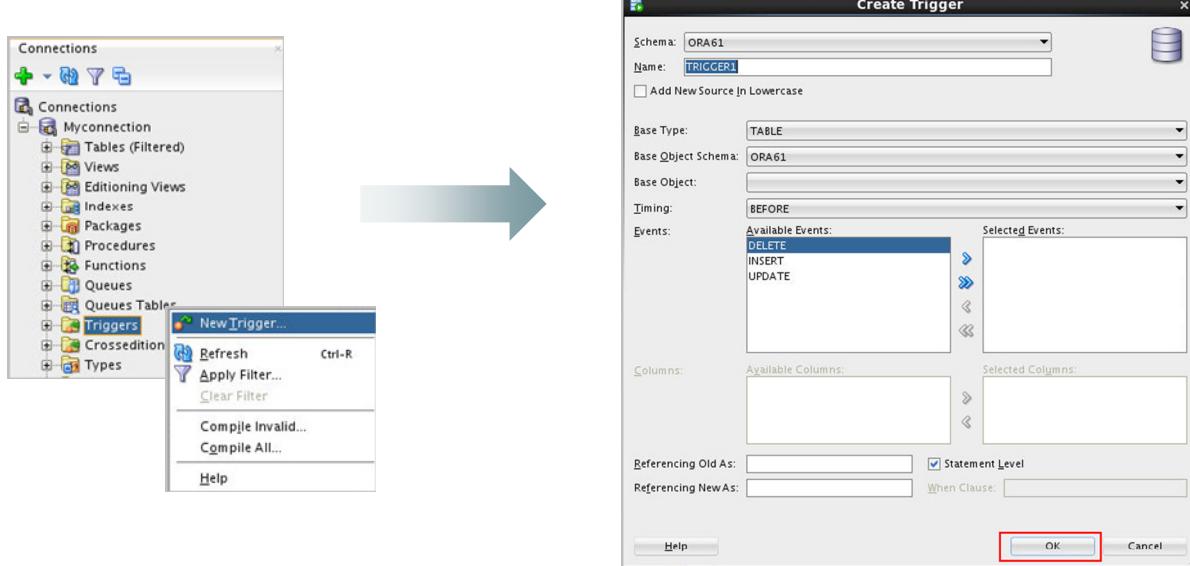
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The components of the trigger syntax are:

- **trigger_name**: uniquely identifies the trigger
- **timing**: indicates when the trigger fires in relation to the triggering event. Values are BEFORE, AFTER, and INSTEAD OF.
- **event** : identifies the DML operation causing the trigger to fire
- **Values are** INSERT, UPDATE [OF column], and DELETE
- **object_name** : indicates the table or view associated with the trigger
- For row triggers, you can specify:
 - A REFERENCING clause to choose correlation names for referencing the old and new values of the current row (default values are OLD and NEW)
 - FOR EACH ROW to designate that the trigger is a row trigger
 - A WHEN clause to apply a conditional predicate, in parentheses, which is evaluated for each row to determine whether or not to execute the trigger body
- **trigger_body** is the action performed by the trigger, implemented as either of the following:
 - An anonymous block with a DECLARE or BEGIN, and an END
 - A CALL clause to invoke a standalone or packaged stored procedure, such as:
CALL my_procedure;

Creating DML Triggers by Using SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide demonstrates how to create a trigger in SQL Developer.

To create DML triggers in SQL Developer, perform the following steps:

1. Right-click **Triggers** under your connection name.
2. Click **New Trigger**. This opens a trigger-creating wizard as shown in the slide.
3. In the wizard, provide the trigger name, type, the event on which the trigger has to fire, and other details.
4. Click **OK**.
5. A code editor with the skeletal trigger code opens; define the body of the trigger.

Specifying the Trigger Execution Time

You can specify the trigger timing as to whether to run the trigger's action before or after the triggering statement:

- BEFORE: Execute the trigger body before the triggering DML event on a table.
- AFTER: Execute the trigger body after the triggering DML event on a table.
- INSTEAD OF: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The BEFORE trigger timing is used in the following situations:

- To determine whether the triggering statement should execute or not. The triggering statement would not execute if an exception raises in the trigger execution.
- To derive column values before completing an `INSERT` or `UPDATE` statement
- To initialize global variables or flags, and to validate complex business rules

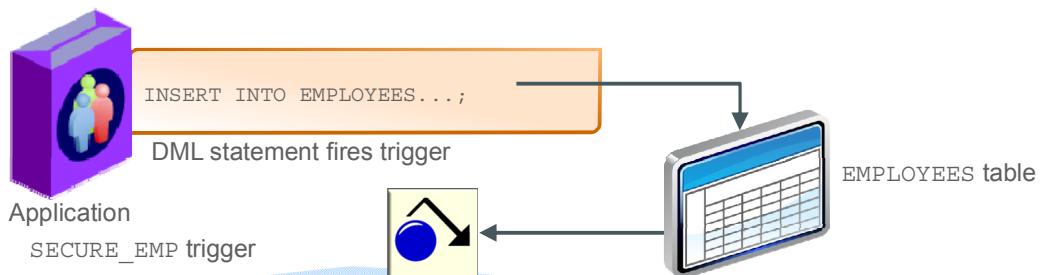
The AFTER triggers are used in the following situations:

- To complete the triggering statement before executing the triggering action
- To perform different actions on the same triggering statement if a BEFORE trigger is already present

The INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because a view is not always modifiable. You can write appropriate DML statements inside the body of an INSTEAD OF trigger to perform actions directly on the underlying tables of views.

If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order that you intend. If two or more triggers are defined with the same timing point, and the order in which they fire is important, you can control the firing order by using the `FOLLOWS` and `PRECEDES` clauses.

Creating a DML Statement Trigger Example: SECURE_EMP



```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON employees
BEGIN
    IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
       (TO_CHAR(SYSDATE,'HH24:MI')
        '08:00' AND '18:00') THEN
        RAISE_APPLICATION_ERROR(-20500, 'You may insert'
                                || ' into EMPLOYEES table only during '
                                || ' normal business hours.');
    END IF;
END;
```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

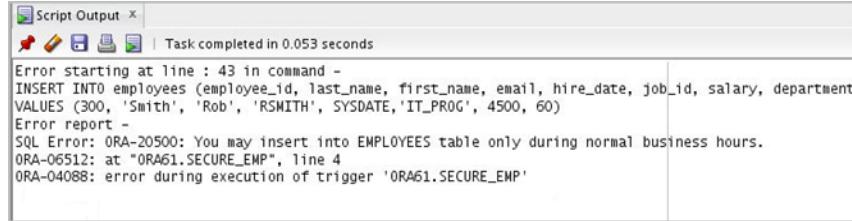
In the example in the slide, the SECURE_EMP database trigger is a BEFORE statement trigger that prevents the INSERT operation from succeeding if the business condition is violated. In this case, the trigger restricts inserts into the EMPLOYEES table during certain business hours, Monday through Friday.

If you attempt to insert a row into the EMPLOYEES table on Saturday, then you'll see an error message, the trigger fails, and the triggering statement is rolled back. Remember that the RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server.

Testing Trigger SECURE_EMP

```
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date,
job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60);
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The output pane displays the following text:

```
Script Output X
Task completed in 0.053 seconds
Error starting at line : 43 in command -
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date, job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60)
Error report -
SQL Error: ORA-20500: You may insert into EMPLOYEES table only during normal business hours.
ORA-06512: at "ORA61.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'ORA61.SECURE_EMP'
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To test the trigger, insert a row into the EMPLOYEES table during nonbusiness hours. When the date and time are out of the business hours specified in the trigger, you receive the error message shown in the slide.

Using Conditional Predicates

```

CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees
BEGIN
    IF (TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN')) OR
       (TO_CHAR(SYSDATE, 'HH24')
        NOT BETWEEN '08' AND '18') THEN
        IF DELETING THEN RAISE_APPLICATION_ERROR(
            -20502, 'You may delete from EMPLOYEES table'|||
            'only during normal business hours.');
        ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
            -20500, 'You may insert into EMPLOYEES table'|||
            'only during normal business hours.');
        ELSIF UPDATING ('SALARY') THEN
            RAISE_APPLICATION_ERROR(-20503, 'You may '|||
            'update SALARY only normal during business hours.');
        ELSE RAISE_APPLICATION_ERROR(-20504, 'You may' |||
            ' update EMPLOYEES table only during'|||
            ' normal business hours.');
        END IF;
    END IF;
END;

```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Detecting the DML Operation that Fired a Trigger

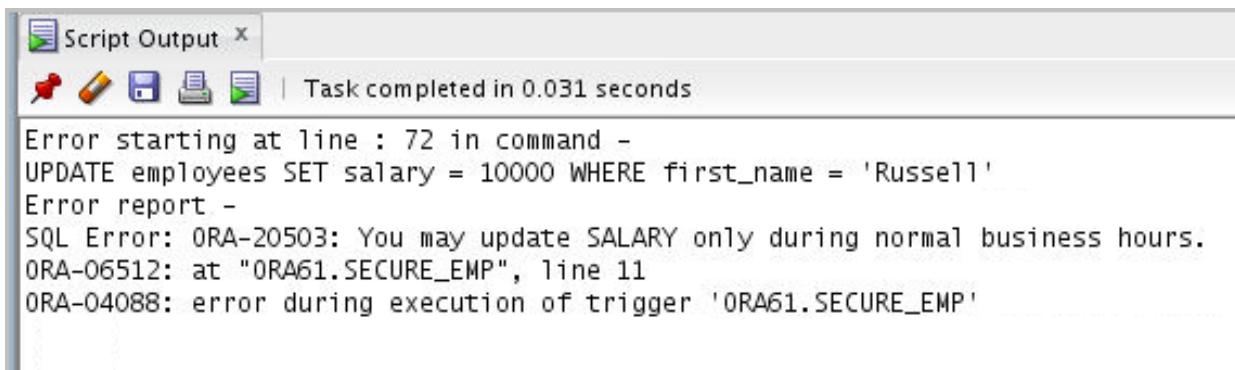
If more than one type of DML operation can fire a trigger (for example, ON INSERT OR DELETE OR UPDATE OF Emp_tab), the trigger body can use the conditional predicates **INSERTING**, **DELETING**, and **UPDATING** to check which type of statement fired the trigger.

You can combine several triggering events into one by taking advantage of the special conditional predicates **INSERTING**, **UPDATING**, and **DELETING** within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the EMPLOYEES table to certain business hours, 8 AM to 6 PM, Monday through Friday.

The following is the response from the trigger when you perform an insertion on a Saturday evening.



The screenshot shows the 'Script Output' window in Oracle SQL Developer. The window title is 'Script Output'. It contains a toolbar with icons for redo, undo, save, and others. Below the toolbar, it says 'Task completed in 0.031 seconds'. The main area displays an error message:

```

Error starting at line : 72 in command -
UPDATE employees SET salary = 10000 WHERE first_name = 'Russell'
Error report -
SQL Error: ORA-20503: You may update SALARY only during normal business hours.
ORA-06512: at "ORA61.SECURE_EMP", line 11
ORA-04088: error during execution of trigger 'ORA61.SECURE_EMP'

```

Multiple Triggers of the Same Type

- You can have multiple triggers defined for a timing instance.
- To resolve the sequence of execution, you can use:
 - PRECEDES
 - FOLLOWS
- To execute trigger2 after trigger1 for a table employees BEFORE UPDATE, you may write it syntactically as:

```
CREATE OR REPLACE TRIGGER2 ON EMPLOYEES BEFORE UPDATE  
FOLLOWS TRIGGER1...
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

CALL Statements in Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
CALL procedure_name
/
```

```
CREATE OR REPLACE PROCEDURE log_execution IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('log_execution: Employee Inserted');
END;
/
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
CALL log_execution -- no semicolon needed
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A CALL statement enables you to call a stored procedure, rather than code the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.

The call can reference the trigger attributes :NEW and :OLD as parameters, as in the following example:

```
CREATE OR REPLACE TRIGGER salary_check
BEFORE UPDATE OF salary, job_id ON employees
FOR EACH ROW
WHEN (NEW.job_id <> 'AD_PRES')
CALL sal_status(:NEW.job_id, :NEW.salary)
```

Note: There is no semicolon at the end of the CALL statement.

In the preceding example, the trigger calls the sal_status procedure. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

Lesson Agenda

- Understanding the usage of triggers
- Creating triggers
- Creating row-level triggers
- Creating INSTEAD OF triggers
- Managing Triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Statement-Level Triggers Versus Row-Level Triggers

Statement-Level Triggers	Row-Level Triggers
Is the default when creating a trigger	Use the FOR EACH ROW clause when creating a trigger
Fires once for the triggering event	Fires once for each row affected by the triggering event
Fires once even if no rows are affected	Does not fire if the triggering event does not affect any rows



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Types of DML Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple-row UPDATE) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself (for example, a trigger that performs a complex security check on the current user).

Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed. Row triggers are useful if the trigger action depends on data of the rows that are affected or on data provided by the triggering event itself.

Note: Row triggers use correlation names to access the old and new column values of the row being processed by the trigger.



In a situation where I have to log insertion of every row while inserting a bunch of 100 rows, I will use a row-level trigger.
While creating the trigger, I add FOR EACH ROW in the syntax.

In a situation where I have to check the date and time of insertion and then allow it if it is in a valid range, I will use a statement-level trigger.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a scenario where you have to log every row for backup while inserting a bunch of 100 rows. You may write a trigger with options `ON INSERT FOR EACH ROW` and define the trigger to perform insertions into the backup table.

In a situation where you have to check the day and time of insertion before performing an insertion and allow the insertion only if it is on a valid day and in a valid time range, you may write a trigger with the option `BEFORE INSERT` and define the trigger body to perform the range checks.

Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
    AND :NEW.salary > 15000 THEN
    RAISE_APPLICATION_ERROR (-20202,
      'Employee cannot earn more than $15,000.');
  END IF;
END;
```



```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

```
Error starting at line : 105 in command -
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell'
Error report -
SQL Error: ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "ORA61.RESTRICT_SALARY", line 4
ORA-04088: error during execution of trigger 'ORA61.RESTRICT_SALARY'
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

In the first example in the slide, a trigger is created to allow only employees whose job IDs are either AD_PRES or AD_VP to earn a salary of more than 15,000. If you try to update the salary of employee Russell whose employee ID is SA_MAN, the trigger raises the exception displayed in the slide.

Note: Before executing the first code example in the slide, make sure you disable the secure_emp and secure_employees triggers.

Correlation names and Pseudorecords

- Correlation names are used to refer to values of a column before and after the trigger firing event.
- There are three different correlation names – OLD, NEW, and PARENT
- Pseudorecords refer to the values of the rows before and after the trigger-firing event when they are manipulated in the trigger body as a record.
- Pseudorecords are also referred to with OLD, NEW, and PARENT qualifiers.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Correlation Names and Pseudorecords

Let us consider the logging of update operation by a logging utility in the database.



- While logging the update operation, the old value has to be written into the log table
- When you perform the log operation through a trigger, you have to refer to the value before **UPDATE**.
- You can refer to the value before the update with the **OLD** qualifier.
- The value after the update is referred with the **NEW** qualifier.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using OLD and NEW Qualifiers

- When a row-level trigger fires, the PL/SQL runtime engine creates and populates two data structures:
 - OLD: Stores the original values of the record processed by the trigger
 - NEW: Contains the new values
- NEW and OLD have the same structure as a record declared by using the %ROWTYPE on the table to which the trigger is attached.

Data Operations	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Within a ROW trigger, you can reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifiers.

Note

- The OLD and NEW qualifiers are available only in ROW triggers.
- Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.
- There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.
- Row triggers can decrease the performance if you perform many updates on larger tables.

Using OLD and NEW Qualifiers: Example

```
CREATE TABLE audit_emp (
    user_name      VARCHAR2(30),
    time_stamp     date,
    id             NUMBER(6),
    old_last_name VARCHAR2(25),
    new_last_name VARCHAR2(25),
    old_title      VARCHAR2(10),
    new_title      VARCHAR2(10),
    old_salary     NUMBER(8,2),
    new_salary     NUMBER(8,2) )
/
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp(user_name, time_stamp, id,
        old_last_name, new_last_name, old_title,
        new_title, old_salary, new_salary)
    VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary);
END;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the AUDIT_EMP_VALUES trigger is created on the EMPLOYEES table. The trigger adds rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

Using OLD and NEW Qualifiers: Example

```
INSERT INTO employees (employee_id, last_name, job_id, salary, email,
hire_date)
VALUES (999, 'Temp emp', 'SA_REP', 6000, 'TEMPEMP', TRUNC(SYSDATE))
/
UPDATE employees
SET salary = 7000, last_name = 'Smith'
WHERE employee_id = 999
/
SELECT *
FROM audit_emp;
```



USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
1 ORA61	03-SEP-16	(null)	Temp emp	(null)	SA_REP	(null)	6000	
2 ORA61	03-SEP-16	999 Temp emp	Smith	Smith	SA_REP	SA_REP	6000	7000



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide demonstrates two instances where the trigger AUDIT_EMP_VALUES is fired.

1. When you insert a row into the EMPLOYEES table, the trigger is fired and the first row is inserted into the AUDIT_EMP table.
2. When you update the salary of employee Smith, then the second row is inserted into the AUDIT_EMP table.

Using the WHEN Clause to Fire a Row Trigger Based on a Condition

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF INSERTING THEN
    :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL THEN
    :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct+0.05;
  END IF;
END;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Optionally, you can include a trigger restriction in the definition of a row trigger by specifying a Boolean SQL expression in a `WHEN` clause. If you include a `WHEN` clause in the trigger, then the expression in the `WHEN` clause is evaluated for each row that the trigger affects.

If the expression evaluates to `TRUE` for a row, then the trigger body executes on behalf of that row. However, if the expression evaluates to `FALSE` or `NOT TRUE` for a row (unknown, as with nulls), then the trigger body does not execute for that row. The evaluation of the `WHEN` clause does not have an effect on the execution of the triggering SQL statement (in other words, the triggering statement is not rolled back if the expression in a `WHEN` clause evaluates to `FALSE`).

Note: A `WHEN` clause cannot be included in the definition of a statement trigger.

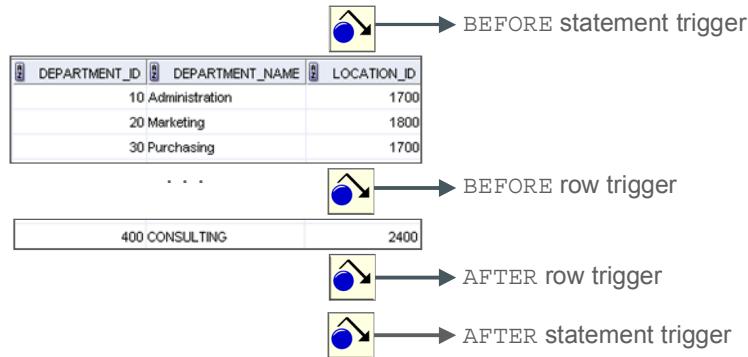
In the example in the slide, a trigger is created on the `EMPLOYEES` table to calculate an employee's commission when a row is added to the `EMPLOYEES` table, or when an employee's salary is modified.

The `NEW` qualifier cannot be prefixed with a colon in the `WHEN` clause because the `WHEN` clause is outside the PL/SQL blocks.

Trigger-Firing Sequence: Single-Row Manipulation

Use the following firing sequence for a trigger on a table when a single row is manipulated:

```
INSERT INTO departments
(department_id, department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You create a statement trigger when the statements in the trigger body have to execute only once for the event. You create a row-level trigger when the statements in the trigger body have to execute once for every row updated due to the trigger event.

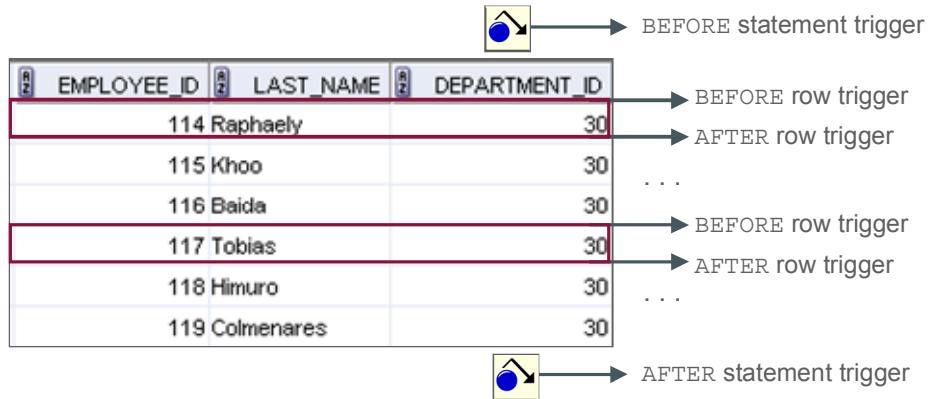
The slide shows the sequence of execution of multiple triggers when the triggering event effects only one row in a table. The `INSERT` statement in the slide inserts a new row into the table `departments`. Assuming that there are triggers defined on the `INSERT` event on the `departments` table for all the four instances(BEFORE , BEFORE with FOR EACH ROW, AFTER with FOR EACH ROW and AFTER). The sequence of execution of these triggers is:

1. Trigger defined to execute before the event
2. Trigger defined to execute before the modification of the effecting row
3. Trigger defined to execute after the modification of the effecting row
4. Trigger defined to execute after the event

Trigger-Firing Sequence: Multirow Manipulation

Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees
  SET salary = salary * 1.1
 WHERE department_id = 30;
```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When the triggering DML statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Consider the SQL statement in the slide causes a row-level trigger to fire a number of times equal to the number of rows that satisfy the WHERE clause (that is, the number of employees reporting to department 30).

Assuming that there are triggers defined to execute on employees table for all the four instances (BEFORE, BEFORE...FOR EACH ROW, AFTER with FOR EACH ROW, AFTER). Then following is the order in which the triggers will execute:

1. The BEFORE trigger
2. The BEFORE...FOR EACH ROW trigger before the update of each row
3. The AFTER...FOR EACH ROW trigger after the update of each row
4. Steps 2 and 3 will repeat for the number of times equal to the number of rows being updated (In case of the query shown in the slide, equal to the number of employees in department number 30.)
5. The AFTER trigger

Summary of the Trigger Execution Model

1. Execute all BEFORE STATEMENT triggers.
2. Loop for each row affected by the SQL statement:
 - a. Execute all BEFORE ROW triggers *for that row*.
 - b. Execute the DML statement and perform integrity constraint checking *for that row*.
 - c. Execute all AFTER ROW triggers *for that row*.
3. Execute all AFTER STATEMENT triggers.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A single DML statement can potentially fire up to four types of triggers:

- BEFORE and AFTER statement triggers
- BEFORE and AFTER row triggers

A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. However, you can defer constraint checking until a COMMIT operation is performed.

Triggers can also cause other triggers, known as cascading triggers, to fire.

All actions and checks performed as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, then 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 undergo changes by other users' transactions. In all cases, a read-consistent image is guaranteed for the modified values that the trigger needs to read (query) or write (update).

Note: Integrity checking can be deferred until the COMMIT operation is performed.

Lesson Agenda

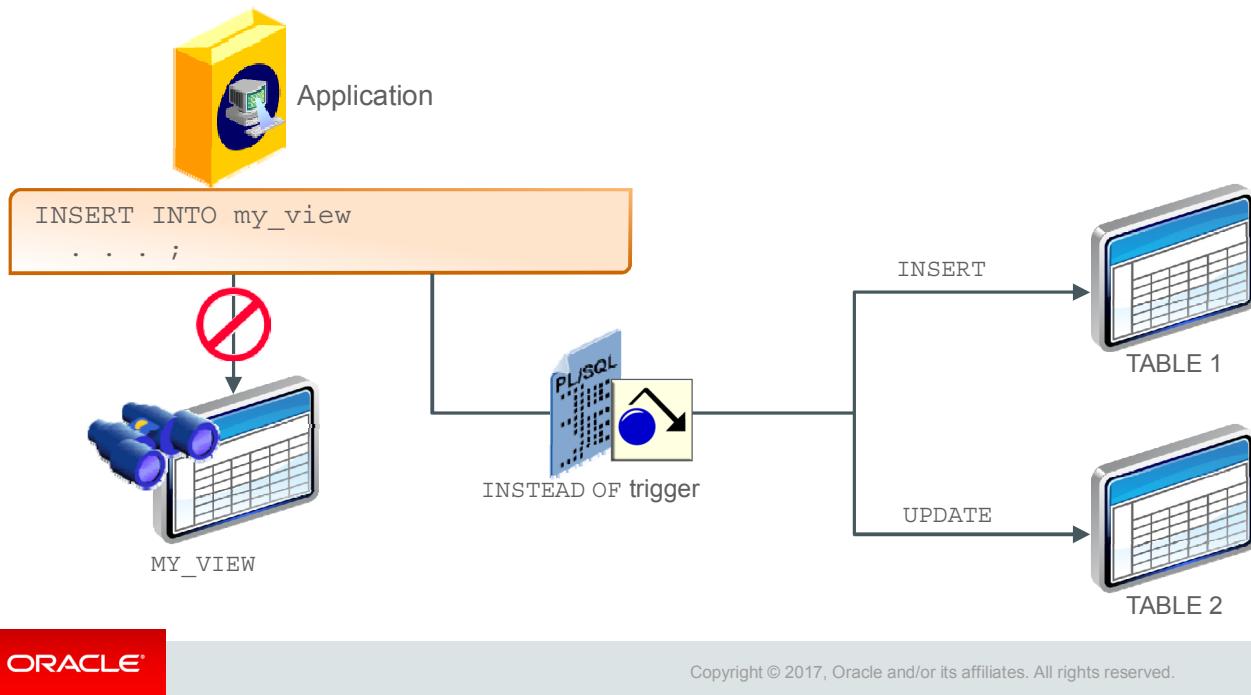
- Understanding the usage of triggers
- Creating triggers
- Creating row-level triggers
- Creating INSTEAD OF triggers
- Managing Triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

INSTEAD OF Triggers



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Use `INSTEAD OF` triggers to modify data in which the DML statement has been issued against an inherently unupdatable view. These triggers are called `INSTEAD OF` triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. These triggers are used to perform `INSERT`, `UPDATE`, and `DELETE` operations directly on the underlying tables.

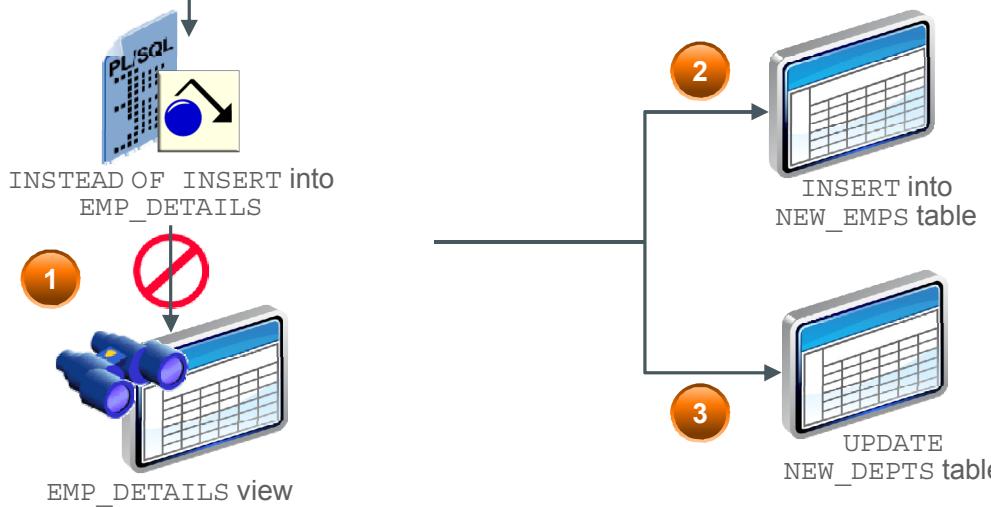
You can write `INSERT`, `UPDATE`, and `DELETE` statements against a view, and the `INSTEAD OF` trigger works invisibly in the background to make the right actions take place. A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as `GROUP BY`, `CONNECT BY`, `START`, the `DISTINCT` operator, or joins.

For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So you write an `INSTEAD OF` trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatable and has `INSTEAD OF` triggers, then the triggers take precedence. `INSTEAD OF` triggers are row triggers. The `CHECK` option for views is not enforced when insertions or updates to the view are performed by using `INSTEAD OF` triggers. The `INSTEAD OF` trigger body must enforce the check.

Creating an INSTEAD OF Trigger: Example

```
INSERT INTO emp_details  
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based.

The example in the slide illustrates an employee's details being inserted into the view `EMP_DETAILS`, whose query is based on the `EMPLOYEES` and `DEPARTMENTS` tables. The `NEW_EMP_DEPT` (INSTEAD OF) trigger executes in place of the `INSERT` operation that causes the trigger to fire. The INSTEAD OF trigger then issues the appropriate `INSERT` and `UPDATE` to the base tables used by the `EMP_DETAILS` view. Therefore, instead of inserting the new employee record into the `EMPLOYEES` table, the following actions take place:

1. The `NEW_EMP_DEPT` INSTEAD OF trigger fires.
2. A row is inserted into the `NEW_EMPS` table.
3. The `DEPT_SAL` column of the `NEW_DEPTS` table is updated. The salary value supplied for the new employee is added to the existing total salary of the department to which the new employee has been assigned.

Note: Before you run the example in the slide, you must create the required structures shown in the next two slides.

Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```

CREATE TABLE new_emps AS
SELECT employee_id, last_name, salary, department_id
  FROM employees;

CREATE TABLE new_depts AS
SELECT d.department_id, d.department_name,
       sum(e.salary) dept_sal
  FROM employees e, departments d
 WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
SELECT e.employee_id, e.last_name, e.salary,
       e.department_id, d.department_name
  FROM employees e, departments d
 WHERE e.department_id = d.department_id
 GROUP BY d.department_id, d.department_name;

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide creates two new tables, NEW_EMPS and NEW_DEPTS, that are based on the EMPLOYEES and DEPARTMENTS tables, respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables.

If a view has a complex query structure, it is not always possible to perform DML directly on the view to affect the underlying tables. The example requires creation of an INSTEAD OF trigger, called NEW_EMP_DEPT, shown on the next page. The NEW_DEPT_EMP trigger handles DML in the following way:

- When a row is inserted into the EMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, by using the data values supplied with the INSERT statement.
- When a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

Note: INSTEAD OF triggers can be written only for views, and the BEFORE and AFTER timing options are not valid.

```

CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO new_emps
    VALUES (:NEW.employee_id, :NEW.last_name,
            :NEW.salary, :NEW.department_id);
    UPDATE new_depts
    SET dept_sal = dept_sal + :NEW.salary
    WHERE department_id = :NEW.department_id;
  ELSIF DELETING THEN
    DELETE FROM new_emps
    WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
    SET dept_sal = dept_sal - :OLD.salary
    WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('salary') THEN
    UPDATE new_emps
    SET salary = :NEW.salary
    WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
    SET dept_sal = dept_sal +
      (:NEW.salary - :OLD.salary)
    WHERE department_id = :OLD.department_id;
  ELSIF UPDATING ('department_id') THEN
    UPDATE new_emps
    SET department_id = :NEW.department_id
    WHERE employee_id = :OLD.employee_id;
    UPDATE new_depts
    SET dept_sal = dept_sal - :OLD.salary
    WHERE department_id = :OLD.department_id;
    UPDATE new_depts
    SET dept_sal = dept_sal + :NEW.salary
    WHERE department_id = :NEW.department_id;
  END IF;
END;
/

```

You can check the execution of the trigger by executing the following statements:

```

insert into emp_details
values(999,'Himuro',3000,10,'Administration');
Select * from new_emps;
Select * from new_depts;

```

Lesson Agenda

- Understanding the usage of triggers
- Creating triggers
- Creating row-level triggers
- Creating INSTEAD OF triggers
- Managing Triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The Status of a Trigger

A trigger can be in either of the following distinct modes:

- **Enabled:** The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).
- **Disabled:** The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A trigger is in either of two distinct modes:

- **Enabled:** The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).
- **Disabled:** The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.

System Privileges Required to Manage Triggers

The following system privileges are required to manage triggers:

- The privileges that enable you to create, alter, and drop triggers in any schema:
 - GRANT CREATE TRIGGER TO ora61
 - GRANT ALTER ANY TRIGGER TO ora61
 - GRANT DROP ANY TRIGGER TO ora61
- The privilege that enables you to create a trigger on the database:
 - GRANT ADMINISTER DATABASE TRIGGER TO ora61
- The EXECUTE privilege (if your trigger refers to any objects that are not in your schema)



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To create a trigger in your schema, you need the CREATE TRIGGER system privilege, and you must own the table specified in the triggering statement, have the ALTER privilege for the table in the triggering statement, or have the ALTER ANY TABLE system privilege. You can alter or drop your triggers without any further privileges being required.

If the ANY keyword is used, you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.

You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the EXECUTE privilege on the referenced procedures, functions, or packages, and not through roles.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, you can drop the trigger but you cannot alter it.

Note: Similar to stored procedures, statements in the trigger body use the privileges of the trigger owner, and not the privileges of the user executing the operation that fires the trigger.

Managing Triggers by Using the ALTER and DROP SQL Statements

```
-- Disable or reenable a database trigger:
```

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

```
-- Disable or reenable all triggers for a table:
```

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

```
-- Recompile a trigger for a table:
```

```
ALTER TRIGGER trigger_name COMPILE;
```

```
-- Remove a trigger from the database:
```

```
DROP TRIGGER trigger_name;
```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Managing Triggers

A trigger has two modes or states: **ENABLED** and **DISABLED**. When a trigger is first created, it is enabled by default. You can create a disabled trigger on creation by adding **DISABLE**, as shown below:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE INSERT ON table_name FOR EACH ROW DISABLE
```

A disabled trigger wouldn't fire even if the triggering event occurs.

The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them.

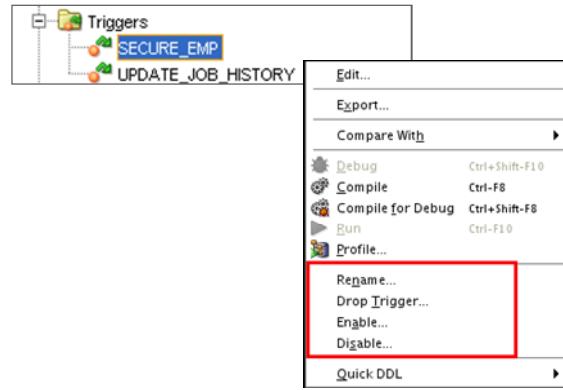
Disabling a Trigger

Use the **ALTER TRIGGER** command to disable a trigger. You can also disable all triggers on a table by using the **ALTER TABLE** command. You can disable triggers to improve performance or to avoid data integrity checks when loading massive amounts of data with utilities such as **SQL*Loader**.

Removing Triggers

When a trigger is no longer required, use **DROP TRIGGER** SQL statement in **SQL Developer** or **SQL*Plus** to remove it. When you remove a table, all triggers on that table are also removed.

Managing Triggers by Using SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use the Triggers node in the Connections navigation tree to manage triggers. Right-click a trigger name, and then select one of the following options:

- Edit
- Compile
- Compile for Debug
- Rename
- Drop Trigger
- Enable
- Disable

Viewing Trigger Information

You can view the following trigger information:

Data Dictionary View	Description
USER_OBJECTS	Displays object information
USER/ALL/DBA_TRIGGERS	Displays trigger information
USER_ERRORS	Displays PL/SQL syntax errors for a trigger



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows the data dictionary views that you can access to get information regarding the triggers.

The `USER_OBJECTS` view contains the name and status of the trigger and the date and time when the trigger was created.

The `USER_ERRORS` view contains the details about the compilation errors that occurred while a trigger was compiling. The contents of these views are similar to those for subprograms.

The `USER_TRIGGERS` view contains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

Using USER_TRIGGERS

DESCRIBE user_triggers

Name	Null	Type
TRIGGER_NAME		VARCHAR2(128)
TRIGGER_TYPE		VARCHAR2(16)
TRIGGERING_EVENT		VARCHAR2(246)
TABLE_OWNER		VARCHAR2(128)
BASE_OBJECT_TYPE		VARCHAR2(18)
TABLE_NAME		VARCHAR2(128)
COLUMN_NAME		VARCHAR2(4000)
REFERENCING_NAMES		VARCHAR2(422)
WHEN_CLAUSE		VARCHAR2(4000)
STATUS		VARCHAR2(8)
DESCRIPTION		VARCHAR2(4000)
ACTION_TYPE		VARCHAR2(11)
TRIGGER_BODY		LONG
CROSSEDITION		VARCHAR2(7)
BEFORE_STATEMENT		VARCHAR2(3)
BEFORE_ROW		VARCHAR2(3)
AFTER_ROW		VARCHAR2(3)
AFTER_STATEMENT		VARCHAR2(3)
INSTEAD_OF_ROW		VARCHAR2(3)
FIRE_ONCE		VARCHAR2(3)
APPLY_SERVER_ONLY		VARCHAR2(3)

```
SELECT trigger_type, trigger_body
FROM user_triggers
WHERE trigger_name = 'SECURE_EMP';
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

If the source file is unavailable, you can use the SQL Worksheet in SQL Developer or SQL*Plus to regenerate it from USER_TRIGGERS. You can also examine the ALL_TRIGGERS and DBA_TRIGGERS views, each of which contains the additional column OWNER, for the owner of the object. The result for the second example in the slide is as follows:

Query Result	
SQL All Rows Fetched: 1 in 0.388 seconds	
TRIGGER_TYPE	TRIGGER_BODY
1 BEFORE STATEMENT	BEGIN IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR (TO_CHAR(SYSDATE,'HH24') < 10) THEN

The TRIGGER_BODY column has the code of the trigger.

Testing Triggers

- Test each triggering and non-triggering data operation.
- Test each case of the `WHEN` clause.
- Make the trigger fire directly from a basic data operation as well as indirectly from a procedure.
- Test the effect of the trigger on other triggers.
- Test the effect of other triggers on the trigger.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Testing code can be a time-consuming process. Perform the following when you are testing triggers:

- Ensure that the trigger works properly by testing a number of cases separately:
 - Test the most common success scenarios first.
 - Test the most common failure conditions to see that they are properly managed.
- The more complex the trigger, the more detailed your testing is likely to be. For example, if you have a row trigger with a `WHEN` clause specified, then you should ensure that the trigger fires when the conditions are satisfied. Or, if you have cascading triggers, you need to test the effect of one trigger on the other and ensure that you end up with the desired results.
- Use the `DBMS_OUTPUT` package to debug triggers.

Quiz



A triggering event can be one or more of the following:

- a. An INSERT, UPDATE, or DELETE statement on a specific table (or view, in some cases)
- b. A CREATE, ALTER, or DROP statement on any schema object
- c. A database startup or instance shutdown
- d. A specific error message or any error message
- e. A user logon or logoff



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, c, d, e

Summary

In this lesson, you should have learned how to:

- Create database triggers that are invoked by DML operations
- Create statement and row trigger types
- Use database trigger-firing rules
- Enable, disable, and manage database triggers
- Develop a strategy for testing triggers
- Remove database triggers



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This lesson covered creating database triggers that execute before, after, or instead of a specified DML operation. Triggers are associated with database tables or views. The BEFORE and AFTER timings apply to DML operations on tables. The INSTEAD OF trigger is used as a way to replace DML operations on a view with appropriate DML statements against other tables in the database.

Triggers are enabled by default but can be disabled to suppress their operation until enabled again. If business rules change, triggers can be removed or altered as required.

Practice 9 Overview: Creating Statement and Row Triggers

This practice covers the following topics:

- Creating row triggers
- Creating a statement trigger
- Calling procedures from a trigger



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

