



Integrated Cloud Applications & Platform Services

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

Student Guide - Volume II

D80170GC20

Edition 2.0 | April 2017 | D98645

Learn more from Oracle University at education.oracle.com

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



10

Creating Compound, DDL, and Event Database Triggers

ORACLE

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

Objectives

After completing this lesson, you should be able to:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers



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.

Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



What is a Compound Trigger?

- A single trigger on a table that allows you to specify actions for each of the following four timing points:
 - BEFORE
 - BEFORE EACH ROW
 - AFTER EACH ROW
 - AFTER
- Each timing instance has an executable part and an optional exception handling section.
- All the timing instances access a common PL/SQL state.

ORACLE

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

Working with Compound Triggers

- The compound trigger body supports a common PL/SQL state that the code for each timing point can access.
- The compound trigger common state is:
 - Established when the triggering statement starts
 - Destroyed when the triggering statement completes
- A compound trigger has a declaration section and a section for each of its timing points.
- The section of each timing point may have an optional exception handling section



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

Why Compound Triggers?

You can use compound triggers to:

- Program an approach where you want the actions you implement for the various timing points to share common data
- Accumulate rows destined for a second table so that you can periodically bulk-insert them
- Avoid the mutating-table error (ORA-04091)



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

The current PL/SQL state is stored before starting the execution of the trigger and destroyed after completing the execution of the trigger. All the trigger timing points can access the same PL/SQL state in such scenario.

If you are inserting values into a second table while executing the trigger body, compound triggers allow you to accumulate all the newly inserted rows and bulk insert them into the table at periodic intervals. Thus improving the performance of the PL/SQL block.

What is ORA-4091?

Consider a trigger, you have defined for event BEFORE INSERT on a table T. In the trigger body definition, you execute a DML statement on the same table T, such action would result in ORA-4091 : table T mutating, trigger/function may not see it. The trigger execution here is initiated to modify the table. When you try to modify it again in the trigger body, it might lead to an inconsistent state, thereby giving the ORA-4091 error.

Compound Trigger Structure

```
CREATE OR REPLACE TRIGGER schema.trigger
FOR dml_event_clause ON schema.table
COMPOUND TRIGGER
[ declare_section ]
timing_point_section [ timing_point_section ]... END [ trigger ] ;
```

The slide illustrates the structure of a compound trigger. The difference between a simple trigger and a compound trigger syntax is the `COMPOUND TRIGGER` clause.

The optional declarative part of a compound trigger declares variables and subprograms that all of its timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run, thus creating the PL/SQL state of the trigger. The variables and subprograms exist for the duration of the triggering statement.

The compound triggers further have the timing point sections for the timing points – `BEFORE`, `BEFORE FOR EACH ROW`, `AFTER`, `AFTER FOR EACH ROW`.

Compound Trigger Structure for Views

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS
BEGIN
statement;
END INSTEAD OF EACH ROW;
```

A compound trigger defined on a view has only an `INSTEAD OF` timing point section defined in the trigger body.

Compound Trigger Restrictions

- A compound trigger must be a DML trigger defined on either a table or a view.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- A timing-point section cannot handle exceptions raised in another timing-point section.
- `:OLD` and `:NEW` cannot appear in the declaration, BEFORE STATEMENT, or the AFTER STATEMENT sections.
- Only the BEFORE EACH ROW section can change the value of `:NEW`.



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

Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



Mutating Tables

- A mutating table is:
 - A table that is being modified by an UPDATE, DELETE, or INSERT statement or
 - A table that might be updated by the effects of a DELETE CASCADE constraint
- The session that issued the triggering statement cannot query or modify a mutating table.
- This restriction prevents a trigger from seeing inconsistent data.
- This restriction applies to all triggers that use the FOR EACH ROW clause.
- Views being modified in the INSTEAD OF triggers are not considered mutating.



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

Mutating Table: Example

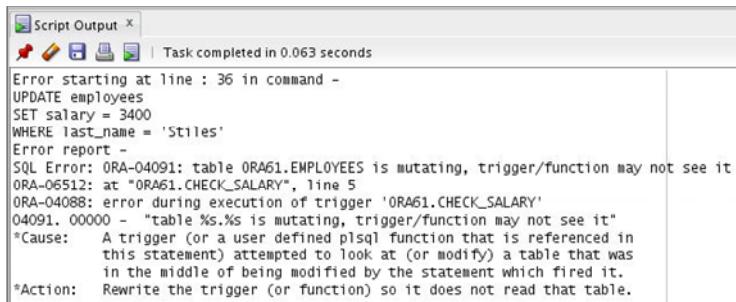
```
CREATE OR REPLACE TRIGGER check_salary
BEFORE INSERT OR UPDATE OF salary, job_id
ON employees
FOR EACH ROW
WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
v_minsalary employees.salary%TYPE;
v_maxsalary employees.salary%TYPE;
BEGIN
SELECT MIN(salary), MAX(salary)
INTO v_minsalary, v_maxsalary
FROM employees
WHERE job_id = :NEW.job_id;
IF :NEW.salary < v_minsalary OR :NEW.salary > v_maxsalary THEN
RAISE_APPLICATION_ERROR(-20505,'Out of range');
END IF;
END;
/
```



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

Mutating Table: Example

```
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles';
```



The screenshot shows a 'Script Output' window from Oracle SQL Developer. The command entered was:

```
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles';
```

The output shows an error message:

```
Error starting at line : 36 in command -  
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles'  
Error report -  
SQL Error: ORA-04091: table ORA61.EMPLOYEES is mutating, trigger/function may not see it  
ORA-06512: at "ORA61.CHECK_SALARY", line 5  
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY'  
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.

In the example in the slide, the trigger code tries to read or select data from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value, then you get a runtime error. The EMPLOYEES table is mutating or is in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

Possible Solutions

Possible solutions to this mutating table problem include the following:

- Use a compound trigger as described earlier in this lesson.
- Store the summary data (the minimum salaries and the maximum salaries) in another summary table, which is kept up-to-date with other DML triggers.
- Store the summary data in a PL/SQL package, and access the data from the package. This can be done in a BEFORE statement trigger.

Using a Compound Trigger to Resolve the Mutating Table Error

```
CREATE OR REPLACE TRIGGER check_salary
  FOR INSERT OR UPDATE OF salary, job_id
  ON employees
  WHEN (NEW.job_id <> 'AD_PRES')
  COMPOUND TRIGGER

    TYPE salaries_t          IS TABLE OF employees.salary%TYPE;
    min_salaries             salaries_t;
    max_salaries              salaries_t;

    TYPE department_ids_t    IS TABLE OF employees.department_id%TYPE;
    department_ids            department_ids_t;

    TYPE department_salaries_t IS TABLE OF employees.salary%TYPE
      INDEX BY VARCHAR2(80);
    department_min_salaries   department_salaries_t;
    department_max_salaries   department_salaries_t;

  -- example continues on next slide
```



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

Using a Compound Trigger to Resolve the Mutating Table Error

```
BEFORE STATEMENT IS
BEGIN
  SELECT MIN(salary), MAX(salary), NVL(department_id, -1)
  BULK COLLECT INTO min_Salaries, max_salaries, department_ids
  FROM employees
  GROUP BY department_id;
  FOR j IN 1..department_ids.COUNT() LOOP
    department_min_salaries(department_ids(j)) := min_salaries(j);
    department_max_salaries(department_ids(j)) := max_salaries(j);
  END LOOP;
END BEFORE STATEMENT;
AFTER EACH ROW IS
BEGIN
  IF :NEW.salary < department_min_salaries(:NEW.department_id)
    OR :NEW.salary > department_max_salaries(:NEW.department_id) THEN
    RAISE_APPLICATION_ERROR(-20505,'New Salary is out of acceptable
                                range');
  END IF;
END AFTER EACH ROW;
END check_salary;
```



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

department_ids is used to hold the department IDs. If the employee who earns the minimum or maximum salary does not have an assigned department, you use the NVL function to store –1 for the department id, instead of NULL.

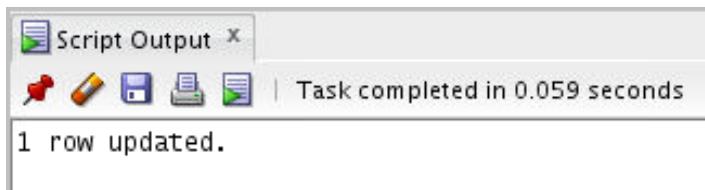
Next, you collect the minimum salary, maximum salary, and the department ID using a bulk insert into the min_salaries, max_salaries, and department_ids respectively grouped by department ID. The select statement returns 13 rows. The values of the department_ids are used as an index for the department_min_salaries and department_max_salaries tables.

Therefore, the index for those two tables (VARCHAR2) represents the actual department_ids.

After each row is added, if the new salary is less than the minimum salary for that department or greater than the department's maximum salary, then an error message is displayed.

To test the newly created compound trigger, issue the following statement:

```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```



To ensure that the salary for employee Stiles was updated, issue the following query by using the F9 key in SQL Developer:

```
SELECT employee_id, first_name,  
last_name, job_id, department_id,salary  
FROM employees  
WHERE last_name = 'Stiles';
```

The screenshot shows the 'Query Result' window in Oracle SQL Developer. The window title is 'Query Result'. Below the title, there are icons for Run, Stop, Refresh, and SQL, followed by the text 'All Rows Fetched: 1 in 0.001 seconds'. The result set is displayed in a table with the following columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, DEPARTMENT_ID, and SALARY. A single row is shown with values: 1, 138, Stephen, Stiles, ST_CLERK, and 50, 3400 respectively.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	DEPARTMENT_ID	SALARY
1	138	Stephen	Stiles	ST_CLERK	50, 3400

Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



Creating Triggers on DDL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- Timing
[ddl_event1 [OR ddl_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Sample DDL Events	Fires When
CREATE	Any database object is created using the CREATE command.
ALTER	Any database object is altered using the ALTER command.
DROP	Any database object is dropped using the DROP command.



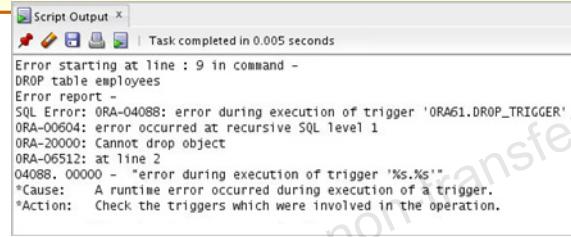
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Creating Triggers on DDL Statements -Example

```
CREATE OR REPLACE TRIGGER drop_trigger
BEFORE DROP ON ora61.SCHEMA
BEGIN
RAISE_APPLICATION_ERROR ( num => -20000, msg => 'Cannot drop object');
END;
/
```

To check the execution of this trigger

```
DROP TABLE employees;
```



ORACLE

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

Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



Creating Database Triggers

- Triggering user event:
 - Logging on or off
- Triggering a database event:
 - Shutting down or starting up the database
 - A specific error (or any error) being raised



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

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level. Triggers on DDL statements, or a user logging on or off, can also be defined at either the database level or the schema level. Triggers on DML statements are defined on a specific table or a view.

A trigger defined at the database level fires for all users, whereas a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- timing
[database_event1 [OR database_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Database Event	Triggers Fires When
AFTER SERVERERROR	An Oracle error is raised
AFTER LOGON	A user logs on to the database
BEFORE LOGOFF	A user logs off the database
AFTER STARTUP	The database is opened
BEFORE SHUTDOWN	The database is shut down normally



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

LOGON and LOGOFF Triggers: Example

```
-- Create the log_trig_table shown in the notes page
-- first
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
        VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
        VALUES (USER, SYSDATE, 'Logging off');
END;
/
```



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

You can create these triggers to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify ON SCHEMA, the trigger fires for the specific user. If you specify ON DATABASE, the trigger fires for all users.

The definition of the log_trig_table used in the slide examples is as follows:

```
CREATE TABLE log_trig_table(
    user_id  VARCHAR2(30),
    log_date TIMESTAMP,
    action   VARCHAR2(40))
/
```

To see the execution of these triggers you can disconnect from the database and connect with the database. Then execute

```
select * from log_trig_table;
```

You can see the output as:

USER_ID	LOG_DATE	ACTION
1 ORA61	06-SEP-16 11.57.39.000000000 PM	Logging off
2 ORA61	06-SEP-16 11.57.43.000000000 PM	Logging on

Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



Guidelines for Designing Triggers

- You can design triggers to:
 - ensure that whenever a specific event occurs, any necessary actions are done.
- Don't create triggers:
 - which duplicate the function of the database
 - which are recursive
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.



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

- Use triggers to ensure that whenever a specific event occurs, any necessary actions are done.
For example, use a trigger to ensure that whenever anyone updates a table, its log file is updated.
- Don't create triggers that duplicate database features.
For example, don't create a trigger to reject invalid data if you can do the same with constraints.
- Don't create recursive triggers.
For example, don't create an AFTER UPDATE trigger that issues an UPDATE statement on the table on which the trigger is defined. The trigger fires recursively until it runs out of memory.

Quiz



Which of the following statements are true for a trigger?

- a. A trigger is defined with a CREATE TRIGGER statement.
- b. A trigger's source code is contained in the USER_TRIGGERS data dictionary.
- c. A trigger is explicitly invoked.
- d. A trigger is implicitly invoked by DML.
- e. COMMIT, SAVEPOINT, and ROLLBACK are not allowed when working with a trigger.



ORACLE

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

Summary

In this lesson, you should have learned how to:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers



ORACLE

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

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

This practice covers the following topics:

- Creating advanced triggers to manage data integrity rules
- Creating triggers that cause a mutating table exception
- Creating triggers that use package state to solve the mutating table problem



ORACLE

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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

GANG LIU (gangl@baylorhealth.edu) has a non-transferable license
to use this Student Guide.



Objectives

After completing this lesson, you should be able to:

- Write standard PL/SQL code
- Grant and control runtime privileges of subprograms
- Create and use autonomous transactions
- Use NOCOPY, PARALLEL_ENABLE, and DETERMINISTIC clauses for optimization
- Use result caching for optimization
- Use bulk binding for optimization



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 11: Design Considerations for the PL/SQL Code

▶ Lesson 12: Tuning the PL/SQL Compiler

▶ Lesson 13: Managing Dependencies

ORACLE

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

Lesson Agenda

- Standardization of the code
- Managing security for PL/SQL packages and subprograms
- Design considerations for autonomous transactions
- Performance optimization in PL/SQL blocks
 - NO COPY clause
 - PARALLEL_ENABLE clause
 - RESULT_CACHE clause
 - DETERMINISTIC clause
 - RETURNING clause
- Bulk Binding



ORACLE

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

Standardizing Constants and Exceptions

- Standardizing helps to:
 - Develop programs that are consistent
 - Promote a higher degree of code reuse
 - Ease code maintenance
 - Implement company standards across entire applications
- Constants and exceptions are typically implemented by using a bodiless package (that is, a package specification).
- Start with standardization of:
 - Exception names
 - Constant definitions



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

When several developers are writing their own exception handlers in an application, there could be inconsistencies in the handling of error situations. Unless certain standards are adhered to, the situation can become confusing because of the different approaches followed in handling the same error or because of the display of conflicting error messages that confuse users. To overcome these, you can:

- Implement company standards that use a consistent approach to error handling across the entire application
- Create predefined, generic exception handlers that produce consistency in the application
- Write and call programs that produce consistent error messages

Good programming environments promote naming and coding standards. In PL/SQL, a good place to start implementing naming and coding standards is with commonly used constants and exceptions that occur in the application domain.

The PL/SQL package specification construct is an excellent component to support standardization because all identifiers declared in the package specification are public. They are visible to the subprograms that are developed by the owner of the package and all code with EXECUTE rights to the package specification.

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
    e_fk_err      EXCEPTION;
    e_seq_nbr_err EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
    PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
    ...
END error_pkg;
/
```



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

In the slide, the `error_pkg` package is a standardized exception package. It declares a set of programmer-defined exception identifiers. An internally defined exception does not have a name unless either PL/SQL or you give it one. The example shown in the slide uses the `PRAGMA EXCEPTION_INIT` directive to associate selected exception names with an Oracle database error number. This enables you to refer to any of the exceptions in a standard way in your applications, as in the following example:

```
BEGIN
    DELETE FROM departments
    WHERE department_id = deptno;
    ...
EXCEPTION
    WHEN error_pkg.e_fk_err THEN
        ...
    WHEN OTHERS THEN
        ...
END;
/
```

The PL/SQL block here refers to `e_fk_err` mapped onto error number ORA-2292, which is a constraint violation error.

Standardizing Exception Handling

Consider writing a subprogram with common exception handling. The exception-handling code should:

- Display errors based on SQLCODE and SQLERRM values for exceptions
- Track runtime errors by using parameters in your code to identify:
 - The procedure in which the error occurred
 - The location (line number) of the error
 - RAISE_APPLICATION_ERROR with appropriate error code and message

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```



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

Standardized exception handling can be implemented either as a standalone subprogram or a subprogram added to the package that defines the standard exceptions.

Consider creating a package for exception handling in your application with the following:

- Every named exception that is to be used in the application
- All unnamed, programmer-defined exceptions that are used in the application. These are error numbers –20000 through –20999.
- A program to call RAISE_APPLICATION_ERROR based on package exceptions
- A program to display an error based on the values of SQLCODE and SQLERRM
- Additional objects, such as error log tables, and programs to access the tables

A common practice is to use parameters that identify the name of the procedure and the location in which the error has occurred. This enables you to keep track of runtime errors more easily. An alternative is to use the RAISE_APPLICATION_ERROR built-in procedure to keep a stack trace of exceptions that can be used to track the call sequence leading to the error. To do this, set the third optional argument to TRUE. If you specify TRUE for the third parameter, PL/SQL puts error_code on top of the error stack. Otherwise, PL/SQL replaces the error stack with error_code.

For example:

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS  
    c_order_received CONSTANT VARCHAR(2) := 'OR';  
    c_order_shipped   CONSTANT VARCHAR(2) := 'OS';  
    c_min_sal         CONSTANT NUMBER(3)  := 900;  
END constant_pkg;
```



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

By definition, a variable's value changes, whereas a constant's value cannot be changed. If you have programs that use local variables whose values should not and do not change, then convert those variables to constants. This can help with the maintenance and debugging of your code.

Consider creating a single shared package with all your constants in it. This makes maintenance and change of the constants much easier. This procedure or package can be loaded on system startup for better performance.

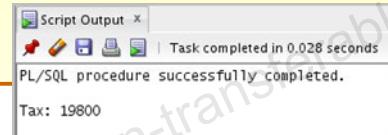
The example in the slide shows the `constant_pkg` package containing a few constants. Refer to any of the package constants in your application as required. Here is an example:

```
BEGIN  
    UPDATE employees  
        SET salary = salary + 200  
        WHERE salary <= constant_pkg.c_min_sal;  
END;  
/
```

Local Subprograms

A local subprogram is a PROCEDURE or FUNCTION defined at the end of the declarative section in a subprogram.

```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
  v_emp employees%ROWTYPE;
  FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN p_salary * 0.825;
  END tax;
BEGIN
  SELECT * INTO v_emp
  FROM EMPLOYEES WHERE employee_id = p_id;
  DBMS_OUTPUT.PUT_LINE('Tax: '|| tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```



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

Local subprograms reduce the size of a module by removing redundant code. This is one of the main reasons for creating a local subprogram. If a module needs the same routine several times and only this module needs the routine, then define it as a local subprogram.

You can define a named PL/SQL block in the declarative section of any PL/SQL program, procedure or function. Local subprograms have the following characteristics:

- They are accessible to only the block in which they are defined.
- They are compiled as part of their enclosing blocks.

The benefits of local subprograms are:

- Reduction of repetitive code
- Improved code readability and ease of maintenance
- Less administration because there is one program to maintain instead of two

The concept is simple. The example shown in the slide illustrates this with a basic example of an income tax calculation of an employee's salary.

Lesson Agenda

- Standardization of the code
- Managing security for PL/SQL packages and subprograms
- Design considerations for autonomous transactions
- Performance optimization in PL/SQL blocks
 - NO COPY clause
 - PARALLEL_ENABLE clause
 - RESULT_CACHE clause
 - DETERMINISTIC clause
 - RETURNING clause
- Bulk Binding



ORACLE

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

Definer's and Invoker's Rights

- You can control access to privileges required to execute a PL/SQL subprogram by using Definer's or Invoker's rights
- With Definer's rights, the subprogram executes with the access privileges of the owner of the procedure.
- With Invoker's rights, the subprogram executes with the privileges of the user who invokes the procedures



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

Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER

```
CREATE OR REPLACE PROCEDURE add_dept(
    p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
    INSERT INTO departments
    VALUES (p_id, p_name, NULL, NULL);
END;
```

When used with standalone functions, procedures, or packages:

- Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema
- Calls to other packages, functions, and procedures are resolved in the definer's schema



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

Specifying Invoker's Rights

You can set the invoker's rights for different PL/SQL subprogram constructs as follows:

```
CREATE FUNCTION name RETURN type AUTHID CURRENT_USER IS...
CREATE PROCEDURE name AUTHID CURRENT_USER IS...
CREATE PACKAGE name AUTHID CURRENT_USER IS...
CREATE TYPE name AUTHID CURRENT_USER IS OBJECT...
```

The default is AUTHID DEFINER, which specifies that the subprogram executes with the privileges of its owner. Most supplied PL/SQL packages such as DBMS_LOB, DBMS_ROWID, and so on, are invoker-rights packages.

Name Resolution

For a definer's rights procedure, all external references are resolved in the definer's schema. For an invoker's rights procedure, the resolution of external references depends on the kind of statement in which they appear:

- Names used in queries, DML statements, dynamic SQL, and DBMS_SQL are resolved in the invoker's schema.
- All other statements, such as calls to packages, functions, and procedures, are resolved in the definer's schema.

Granting Privileges to Invoker's Rights Unit

- When the invoker is a lower-privilege user, then the definer can grant privileges to execute a subprogram or a PL/SQL package to the invoking user.
- The SQL GRANT command grants roles to PL/SQL packages and standalone subprograms:
 - Create an IR unit.
 - Grant roles using the IR unit.
- The following command grants the read and execute roles to the `scott.func` function and the `sys.pkg` package:

```
GRANT read, execute TO FUNCTION scott.func, PACKAGE sys.pkg
```



ORACLE

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

Before Oracle Database 12c, a definer's rights unit always ran with the privileges of the definer and an invoker's rights unit always ran with the privileges of the invoker. If you wanted to create a PL/SQL unit that all users could invoke, even if their privileges were lower than yours, it had to be a definer's rights unit. The unit always ran with all the definer privileges, regardless of which user invoked it.

As of Oracle Database 12c, you can grant roles to individual PL/SQL packages and standalone subprograms. Instead of a designer's rights unit, you can create an invoker's rights unit and then grant it roles. The invoker's rights unit runs with the privileges of both the invoker and the roles, but without any additional privileges that you have.

You grant roles to an invoker's rights unit so that users with lower privileges than yours can run the unit with only the privileges needed to do so. There is no reason to grant roles to a designer's rights unit, because its invokers run it with all your privileges.

Using the SQL GRANT command, you can grant roles to PL/SQL packages and standalone stored subprograms. Roles granted to a PL/SQL unit do not affect compilation. They affect the privilege checking of SQL statements that the unit issues at run time; the unit runs with the privileges of both its own roles and any other currently enabled roles.

Lesson Agenda

- Standardization of the code
- Managing security for PL/SQL packages and subprograms
- Design considerations for autonomous transactions
- Performance optimization in PL/SQL blocks
 - NO COPY clause
 - PARALLEL_ENABLE clause
 - RESULT_CACHE clause
 - DETERMINISTIC clause
 - RETURNING clause
- Bulk Binding

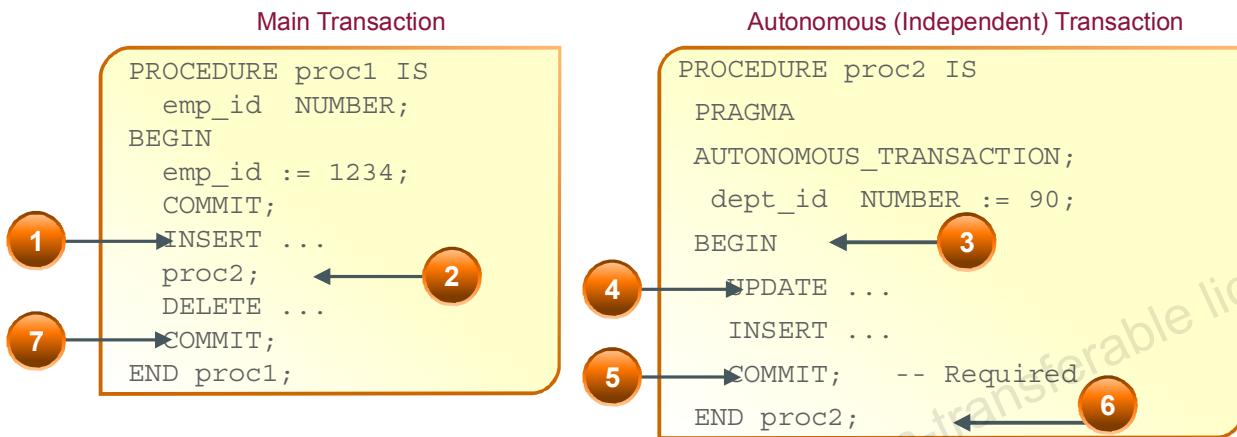


ORACLE

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

Autonomous Transactions

- Are independent transactions started by another main transaction
- Are defined with PRAGMA AUTONOMOUS_TRANSACTION



ORACLE

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

Features of Autonomous Transactions

- Are independent of the main transaction
- Suspend the calling transaction until the autonomous transactions are completed
- Are not nested transactions
- Don't roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are started and ended by individual subprograms and not by nested or anonymous PL/SQL blocks



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

Autonomous transactions exhibit the following features:

- Although called within a transaction, autonomous transactions are independent of that transaction; that is, they aren't nested transactions.
- If the main transaction rolls back, autonomous transactions don't.
- Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits.
- With their stack-like functionality, only the “top” transaction is accessible at any given time. After completion, the autonomous transaction is popped and the calling transaction is resumed.
- There are no limits other than resource limits on how many autonomous transactions can be recursively called.
- Autonomous transactions must be explicitly committed or rolled back; otherwise, an error is returned when attempting to return from the autonomous block.
- You can't use PRAGMA to mark all subprograms in a package as autonomous. Only individual routines can be marked autonomous.
- You can't mark a nested or anonymous PL/SQL block as autonomous.

Using Autonomous Transactions: Example

```
CREATE TABLE usage (card_id NUMBER, loc NUMBER)
/
CREATE TABLE txn (acc_id NUMBER, amount NUMBER)
/
CREATE OR REPLACE PROCEDURE log_usage (p_card_id NUMBER, p_loc NUMBER) IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO usage
    VALUES (p_card_id, p_loc);
    COMMIT;
END log_usage;
/
CREATE OR REPLACE PROCEDURE bank_trans(p_cardnbr NUMBER,p_loc NUMBER) IS
BEGIN
    INSERT INTO txn VALUES (9001, 1000);
    log_usage (p_cardnbr, p_loc);
END bank_trans;
/
EXECUTE bank_trans(50, 2000)
```



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

To define autonomous transactions, you use `PRAGMA AUTONOMOUS_TRANSACTION`. `PRAGMA` instructs the PL/SQL compiler to mark a routine as autonomous (independent). In this context, the term “routine” includes top-level (not nested) anonymous PL/SQL blocks; local, standalone, and packaged functions and procedures; methods of a SQL object type; and database triggers. You can code `PRAGMA` anywhere in the declarative section of a routine. However, for readability, it is best placed at the top of the Declaration section.

In the example in the slide, you track where the bankcard is used, regardless of whether the transaction is successful. The following are the benefits of autonomous transactions:

- After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit dependencies with the main transaction, so you can log events, increment retry counters, and so on even if the main transaction rolls back.
- Autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure’s autonomous operations and the procedure need not know about the application’s transaction context. That makes autonomous transactions less error-prone than regular transactions and easier to use.

The output of the previous slide examples, the TXN and USAGE tables, are as follows:

Script Output

Table USAGE created.

Table TXN created.

Procedure LOG_USAGE compiled

Procedure BANK_TRANS compiled

PL/SQL procedure successfully completed.

To run the code on the previous page, enter the following code:

```
EXECUTE bank_trans(50, 2000)
```

Use the Data tab in the Tables node of the Object Navigator tree to display the values in the TXN and USAGE tables as follows:

TXN

Columns Data Constraints Grants Statistics Triggers Flashback Dependencies Details Partitions Indexes SQL

ACC_ID AMOUNT

	ACC_ID	AMOUNT
1	9001	1000

USAGE

Columns Data Constraints Grants Statistics Triggers Flashback Dependencies Details Partitions Indexes SQL

CARD_ID LOC

	CARD_ID	LOC
1	50	2000

Lesson Agenda

- Standardization of the code
- Managing security for PL/SQL packages and subprograms
- Design considerations for autonomous transactions
- Performance optimization in PL/SQL blocks
 - NO COPY clause
 - PARALLEL_ENABLE clause
 - RESULT_CACHE clause
 - DETERMINISTIC clause
 - RETURNING clause
- Bulk Binding



ORACLE

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

Using the NOCOPY Hint

- Allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
    TYPE      rec_emp_type IS TABLE OF employees%ROWTYPE;
    rec_emp  rec_emp_type;
    PROCEDURE populate(p_tab IN OUT NOCOPY emptabtype) IS
        BEGIN
        . . .
        END;
BEGIN
    populate(rec_emp);
END;
/
```



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

Effects of the NOCOPY Hint

- If the subprogram exits with an exception that is not handled:
 - You cannot rely on the values of the actual parameters passed to a NOCOPY parameter
 - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.



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

When Does the PL/SQL Compiler Ignore the NOCOPY Hint?

The NOCOPY hint has no effect if:

- The actual parameter:
 - Is an element of associative arrays (index-by tables)
 - Is constrained (for example, by scale or NOT NULL)
 - And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ
 - Requires an implicit data type conversion
- The subprogram is involved in an external or RPC



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

Using the PARALLEL_ENABLE Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```



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

The PARALLEL_ENABLE keyword can be used in the syntax for declaring a function. It is an optimization hint that indicates that the function can be used in a parallelized query or parallelized DML statement. Oracle's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement that is run in parallel can have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process.

For DML statements, before Oracle 8*i*, the parallelization optimization looked to see whether a function was noted as having all four of RNDS, WNDS, RNPS, and WNPS specified in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine whether they were actually pure enough; parallelized execution might occur even though a PRAGMA cannot be specified on these functions.

The PARALLEL_ENABLE keyword is placed after the return value type in the declaration of the function, as shown in the example in the slide.

Note: The function should not use session state, such as package variables, because those variables may not be shared among the parallel execution servers.

Using the Cross-Session PL/SQL Function Result Cache

- Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in cache.
- The function result cache is stored in a shared global area (SGA), making it available to any session that runs your application.
- Subsequent calls to the same function with the same parameters uses the result from cache.
- Performance and scalability are improved.
- This feature is used with functions that are called frequently and dependent on information that changes infrequently.



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

This caching mechanism provides you with a language-supported and system-managed means for storing the results of PL/SQL functions in a SGA, which is available to every session that runs your application. The caching mechanism is both efficient and easy to use, and it relieves you of the burden of designing and developing your own caches and cache-management policies.

Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in the cache. Subsequently, when the same function is called with the same parameter values, the result is retrieved from the cache, instead of being recomputed. If a database object that was used to compute a cached result is updated, the cached result becomes invalid and must be recomputed.

Use the result-caching feature with functions that are called frequently and are dependent on information that never changes or changes infrequently.

Note: For additional information about Cross-Session PL/SQL Function Result Cache, refer to the *Oracle Database Advanced PL/SQL* course, the *Oracle Database SQL and PL/SQL New Features* course, or *Oracle Database PL/SQL Language Reference*.

Declaring and Defining a Result-Cached Function: Example

- Use the RESULT_CACHE clause to enable result-caching for a function.
- Example:

```
CREATE OR REPLACE FUNCTION get_hire_date (emp_id NUMBER)
  RETURN VARCHAR
RESULT_CACHE
AUTHID CURRENT_USER
IS
  date_hired DATE;
BEGIN
  SELECT hire_date INTO date_hired
    from employees
   WHERE employee_id = emp_id;
  RETURN TO_CHAR(date_hired);
END;
/
SELECT get_hire_date(206)  from DUAL;
```



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

Before Oracle Database 12c, an invoker's rights function could not be result-cached. As of 12c, the invoker's rights function can be result-cached.

To enable result-caching for a function, use the RESULT_CACHE clause. When a result-cached function is invoked, the system checks the cache. If the cache contains the result from a previous invocation of the function with the same parameter values, the system returns the cached result to the invoker and doesn't execute the function body again. If the cache does not contain the result, the system runs the function body and adds the result (for these parameter values) to the cache before returning control to the invoker.

Note: If function execution results in an unhandled exception, the exception result is not stored in the cache.

The code example in the slide depicts the creation of a function `get_hire_date` that takes the `employee_id` as a parameter and returns the `hire_date`.

To enable result caching in a package function, you have to include the RESULT_CACHE clause in both function declaration and function definition.

Using the DETERMINISTIC Clause with Functions

- Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its arguments.
- Add the DETETMINISTIC clause in function declaration and definition.
- This helps the optimizer avoid redundant function calls.
- If a function was called previously with the same arguments, the optimizer can elect to use the previous result.
- Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects.



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

Using the RETURNING Clause

- Improves performance by returning column values with `INSERT`, `UPDATE`, and `DELETE` statements
- Eliminates the need for a `SELECT` statement

```
CREATE OR REPLACE PROCEDURE update_salary(p_emp_id NUMBER) IS
    v_name      employees.last_name%TYPE;
    v_new_sal   employees.salary%TYPE;
BEGIN
    UPDATE employees
        SET salary = salary * 1.1
       WHERE employee_id = p_emp_id
    RETURNING last_name, salary INTO v_name, v_new_sal;
    DBMS_OUTPUT.PUT_LINE(v_name || ' new salary is ' ||
                          v_new_sal);
END update_salary;
/
```



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

Often, applications need information about the row affected by a SQL operation; for example, to generate a report or to take a subsequent action. The `INSERT`, `UPDATE`, and `DELETE` statements can include a `RETURNING` clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to `SELECT` the row after an `INSERT` or `UPDATE`, or before a `DELETE`. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

The example in the slide shows how to update the salary of an employee and, at the same time, retrieve the employee's last name and new salary into a local PL/SQL variable. To test the code example, issue the following commands that check the salary of `employee_id` 108 before updating it. Next, the procedure is invoked with the `employee_id` 108 as the parameter.

```
select last_name, salary from employees where employee_id=108;
/
EXECUTE update_salary(108)
```

The screenshot shows two panes from the Oracle Database 12c R2 interface. The left pane, titled "Query Result", displays a table with columns "LAST_NAME" and "SALARY". A single row is shown: "Greenberg" in the LAST_NAME column and "12008" in the SALARY column. The right pane, titled "Script Output", shows the message "PL/SQL procedure successfully completed." followed by the output of the DBMS_OUTPUT.PUT_LINE statement: "Greenberg new salary is 13208.8".

Lesson Agenda

- Standardization of code
- Managing security for PL/SQL packages and subprograms
- Design considerations for autonomous transactions
- Performance optimization in PL/SQL blocks
 - NO COPY clause
 - PARALLEL_ENABLE clause
 - RESULT_CACHE clause
 - DETERMINISTIC clause
 - RETURNING clause
- Bulk Binding

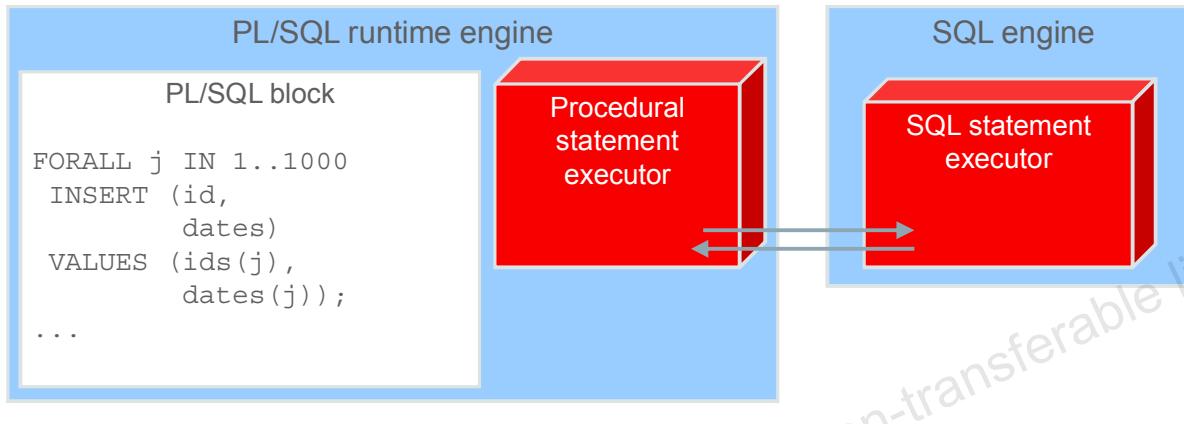


ORACLE

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

Using Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



ORACLE

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

The Oracle server uses two engines to run PL/SQL blocks and subprograms:

- The PL/SQL runtime engine, which runs procedural statements, but passes the SQL statements to the SQL engine
- The SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine

During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty for excessive amounts of SQL processing. This is typical of applications that have a SQL statement in a loop that uses values from indexed collections. Collections include nested tables, varying arrays, index-by tables, and host arrays.

Performance can be substantially improved by minimizing the number of context switches through the use of bulk binding. Bulk binding causes an entire collection to be bound in one call, a context switch, to the SQL engine. That is, a bulk bind process passes the entire collection of values back and forth between the two engines in a single context switch, compared with incurring a context switch for each collection element in an iteration of a loop. The more rows affected by a SQL statement, the greater the performance gain with bulk binding.

Bulk Binding: Syntax and Keywords

- The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound  
  [SAVE EXCEPTIONS]  
  sql_statement;
```

- The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO  
  collection_name[,collection_name] ...
```



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

Use bulk binds to improve the performance of:

- DML statements that reference collection elements
- SELECT statements that reference collection elements
- Cursor FOR loops that reference collections and the `RETURNING INTO` clause

The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine. Although the `FORALL` statement contains an iteration scheme, it is not a `FOR` loop.

The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections, before returning them to the PL/SQL engine. This enables you to bind locations into which SQL can return the retrieved values in bulk. Thus, you can use these keywords in the `SELECT INTO`, `FETCH INTO`, and `RETURNING INTO` clauses.

Bulk Binding FORALL: Example

```
CREATE TABLE parts1 ( pnum INTEGER, pname VARCHAR2(15) );
/
CREATE TABLE parts2 ( pnum INTEGER, pname VARCHAR2(15) );
/
DECLARE
TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
pnums NumTab;
pnames NameTab;
iterations CONSTANT PLS_INTEGER := 50000;
t1 INTEGER;
t2 INTEGER;
t3 INTEGER;
```



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

Bulk Binding FORALL: Example

```
BEGIN
    FOR j IN 1..iterations LOOP -- populate collections
        pnums(j) := j;
        pnames(j) := 'Part No. ' || TO_CHAR(j);
    END LOOP;
    t1 := DBMS_UTILITY.get_time;
    FOR i IN 1..iterations LOOP
        INSERT INTO parts1 (pnum, pname) VALUES (pnums(i), pnames(i));
    END LOOP;
    t2 := DBMS_UTILITY.get_time;
    FORALL i IN 1..iterations
        INSERT INTO parts2 (pnum, pname) VALUES (pnums(i), pnames(i));
    t3 := DBMS_UTILITY.get_time;
```



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

Bulk Binding FORALL: Example

```
DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
DBMS_OUTPUT.PUT_LINE('-----');
DBMS_OUTPUT.PUT_LINE('FOR LOOP: ' || TO_CHAR((t2 - t1)/100));
DBMS_OUTPUT.PUT_LINE('FORALL: ' || TO_CHAR((t3 - t2)/100));
COMMIT;
END;
/
```

```
PL/SQL procedure successfully completed.

Execution Time (secs)
-----
FOR LOOP: .62
FORALL: .02
```

Using BULK COLLECT INTO with Queries

The SELECT statement supports the BULK COLLECT INTO syntax.

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
  TYPE dept_tab_type IS
    TABLE OF departments%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  SELECT * BULK COLLECT INTO v_depts
  FROM departments
  WHERE location_id = p_loc;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
      || ' ' || v_depts(i).department_name);
  END LOOP;
END;
```



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

Using BULK COLLECT INTO with Cursors

The FETCH statement has been enhanced to support the BULK COLLECT INTO syntax.

```
CREATE OR REPLACE PROCEDURE get_departments(p_loc NUMBER) IS
  CURSOR cur_dept IS
    SELECT * FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tab_type IS TABLE OF cur_dept%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  OPEN cur_dept;
  FETCH cur_dept BULK COLLECT INTO v_depts;
  CLOSE cur_dept;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
      ||'-'|| v_depts(i).department_name);
  END LOOP;
END;
```



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

The example in the slide shows how BULK COLLECT INTO can be used with cursors.

You can also add a LIMIT clause to control the number of rows fetched in each operation. The code example in the slide could be modified as follows:

```
CREATE OR REPLACE PROCEDURE get_departments(p_loc NUMBER,
  p_nrows NUMBER) IS
  CURSOR dept_csr IS SELECT *
    FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts LIMIT p_nrows;
  CLOSE dept_csr;
  DBMS_OUTPUT.PUT_LINE(depts.COUNT|| ' rows read');
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

```
CREATE OR REPLACE PROCEDURE raise_salary(p_rate NUMBER) IS
    TYPE emplist_type IS TABLE OF NUMBER;
    TYPE numlist_type IS TABLE OF employees.salary%TYPE
        INDEX BY BINARY_INTEGER;
    v_emp_ids emplist_type := emplist_type(100,101,102,104);
    v_new_sals numlist_type;
BEGIN
    FORALL i IN v_emp_ids.FIRST .. v_emp_ids.LAST
        UPDATE employees
            SET commission_pct = p_rate * salary
            WHERE employee_id = v_emp_ids(i)
            RETURNING salary BULK COLLECT INTO v_new_sals;
    FOR i IN 1 .. v_new_sals.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(v_new_sals(i));
    END LOOP;
END;
```



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

Quiz



The `NOCOPY` hint allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference rather than by value. This enhances performance by reducing overhead when passing parameters.

- a. True
- b. False



ORACLE

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

Answer: a

PL/SQL subprograms support three parameter-passing modes: `IN`, `OUT`, and `IN OUT`.

By default:

- The `IN` parameter is passed by reference. A pointer to the `IN` actual parameter is passed to the corresponding formal parameter. So, both the parameters reference the same memory location, which holds the value of the actual parameter.
- The `OUT` and `IN OUT` parameters are passed by value. The value of the `OUT` or `IN OUT` actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the `OUT` and `IN OUT` formal parameters are copied into the corresponding actual parameters.

Copying parameters that represent large data structures (such as collections, records, and instances of object types) with `OUT` and `IN OUT` parameters slows down execution and uses up memory. To prevent this overhead, you can specify the `NOCOPY` hint, which enables the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference.

Summary

In this lesson, you should have learned how to:

- Create standard constants and exceptions
- Write and call local subprograms
- Control the runtime privileges of a subprogram
- Perform autonomous transactions
- Grant roles to PL/SQL packages and stored subprograms
- Pass parameters by reference using a `NOCOPY` hint
- Use the `PARALLEL_ENABLE` hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the `DETERMINISTIC` clause with functions
- Use the `RETURNING` clause and bulk binding with DML



ORACLE

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

Practice 11 Overview: Design Considerations for PL/SQL Code

This practice covers the following topics:

- Creating a package that uses bulk fetch operations
- Creating a local subprogram to perform an autonomous transaction to audit a business operation



ORACLE

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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

GANG LIU (gangl@baylorhealth.edu) has a non-transferable license
to use this Student Guide.



Tuning the PL/SQL Compiler

ORACLE®

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

Objectives

After completing this lesson, you should be able to:

- Use the PL/SQL compiler initialization parameters
- Use the PL/SQL compile-time warnings



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 11: Design Considerations for the PL/SQL Code

▶ Lesson 12: Tuning the PL/SQL Compiler

▶ Lesson 13: Managing Dependencies

ORACLE

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

Lesson Agenda

- Using PL/SQL initialization parameters
- Using the PL/SQL compile-time warnings:
 - Using the `PLSQL_WARNING` initialization parameter
 - Using the `DBMS_WARNING` package subprograms



ORACLE

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

Optimizing PL/SQL Compiler Performance

- The PL/SQL compiler implicitly rearranges the code for better performance.
- The compiler behavior is defined by its initialization parameters.
- You can use initialization parameters to tune the performance of the compiler according to the application requirement.



ORACLE

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

Initialization Parameters for PL/SQL Compilation

Compilation Parameter	Description
PLSCOPE_SETTINGS	Controls the compile-time collection, cross-reference, and storage of the PL/SQL source text identifier data
PLSQL_CCFLAGS	Lets you control conditional compilation of each PL/SQL unit independently
PLSQL_CODE_TYPE	Specifies the compilation mode for PL/SQL units - INTERPRETED (the default) or NATIVE
PLSQL_OPTIMIZE_LEVEL	Specifies the optimization level at which to compile PL/SQL units
PLSQL_WARNINGS	Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors
NLS_LENGTH_SEMANTICS	Lets you create CHAR and VARCHAR2 columns by using either byte-length or character-length semantics



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

- The `PLSCOPE_SETTINGS` parameter determines whether the compiler should collect data about the identifiers in the PL/SQL program unit and provide it to the static data dictionary views. The collected data includes information about identifier types, usages, and location of each usage in the source code.
The `PLSCOPE_SETTINGS` parameter can assume either '`IDENTIFIERS : ALL`' or '`IDENTIFIERS : NONE`'.
- The `PLSQL_CCFLAGS` lets you control the conditional compilation of each program unit. Conditional compilation lets you customize the functionality of a PL/SQL application without removing the source text. For example:

- Using new features with the latest database release and disabling them when running the application in an older database release.
- Activate debugging or tracing statements in the development environment and hide them when running the application at a production site.

- The `PLSQL_CODE_TYPE` parameter can have either `INTERPRETED` or `NATIVE` as its value. It determines the mode of compilation.

The PL/SQL statements in a PL/SQL unit are compiled into an intermediate form, system code, which is stored in the catalog and interpreted at run time. This is the default behavior when the `PLSQL_CODE_TYPE` parameter is set to `INTERPRETED` value.

When the `PLSQL_CODE_TYPE` value is set to `NATIVE`, the PL/SQL statements in a PL/SQL unit are compiled into native code and stored in the catalog. The native code need not be interpreted at run time, so it runs faster.

- The `PLSQL_OPTIMIZE_LEVEL` parameter can have a value in the 0-3 range. A higher value of the parameter implies a greater level of optimization.
- The `PLSQL_WARNINGS` parameter configures the response of the application to various PL/SQL warnings in the application.
- The `NLS_LENGTH_SEMANTICS` parameter determines whether the compiler should use byte-length or character-length semantics for the `VARCHAR2` and `CHAR` variables in the PL/SQL program unit.

The values at the time of compilation of `PLSQL_CCFLAGS`, `PLSQL_CODE_TYPE`, `PLSQL_OPTIMIZE_LEVEL`, `PLSQL_WARNINGS`, and `NLS_LENGTH_SEMANTICS` initialization parameters are stored with the unit's metadata. You can view information about the settings of these parameters with the `ALL_PLSQL_OBJECT_SETTINGS` view.

Using the Initialization Parameters for PL/SQL Compilation

- `PLSQL_CODE_TYPE`: Specifies the compilation mode for PL/SQL library units

```
PLSQL_CODE_TYPE = { INTERPRETED | NATIVE }
```

- `PLSQL_OPTIMIZE_LEVEL`: Specifies the optimization level to be used to compile PL/SQL library units

```
PLSQL_OPTIMIZE_LEVEL = { 0 | 1 | 2 | 3 }
```



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

`PLSQL_CODE_TYPE` determines the compilation mode for the PL/SQL program unit. When the value of this parameter is changed, it has no effect on PL/SQL library units that have already been compiled. The value of this parameter is stored persistently with each library unit. If a PL/SQL library unit is compiled natively, all subsequent automatic recompilations of that library unit will use native compilation.

`PLSQL_OPTIMIZE_LEVEL` defines the extent of optimization applicable to the compilation of the program unit.

`PLSQL_OPTIMIZE_LEVEL`, when set to 0: the compiler would not reorder the code for better performance.

`PLSQL_OPTIMIZE_LEVEL`, when set to 1: applies a wide range of optimizations to PL/SQL programs, including the elimination of unnecessary computations and exceptions, but generally does not move the source code out of its original source order

`PLSQL_OPTIMIZE_LEVEL`, when set to 2: applies a wide range of modern optimization techniques beyond those of level 1, including changes that may move the source code relatively far from its original location.

`PLSQL_OPTIMIZE_LEVEL`, when set to 3: applies a wide range of optimization techniques beyond those of level 2, automatically including techniques not specifically requested

Displaying the PL/SQL Initialization Parameters

Use the `USER | ALL | DBA_PLSQL_OBJECT_SETTINGS` data dictionary views to display the settings for a PL/SQL object:

```
DESCRIBE USER_PLSQL_OBJECT_SETTINGS;
```

Name	Null	Type
NAME	NOT NULL	VARCHAR2(128)
TYPE		VARCHAR2(12)
PLSQL_OPTIMIZE_LEVEL		NUMBER
PLSQL_CODE_TYPE		VARCHAR2(4000)
PLSQL_DEBUG		VARCHAR2(4000)
PLSQL_WARNINGS		VARCHAR2(4000)
NLS_LENGTH_SEMANTICS		VARCHAR2(4000)
PLSQL_CCFLAGS		VARCHAR2(4000)
PLSCOPE_SETTINGS		VARCHAR2(4000)
ORIGIN_CON_ID		NUMBER



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

The columns of the `USER_PLSQL_OBJECTS_SETTINGS` data dictionary view are:

- **Owner:** The owner of the object. This column is not displayed in the `USER_PLSQL_OBJECTS_SETTINGS` view.
- **Name:** The name of the object
- **Type:** The available choices are PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, TYPE, TYPE BODY, or any other user-defined objects.
- **PLSQL_OPTIMIZE_LEVEL:** The optimization level that was used to compile the object
- **PLSQL_CODE_TYPE:** The compilation mode for the object
- **PLSQL_DEBUG:** Specifies whether or not the object was compiled for debugging
- **PLSQL_WARNINGS:** The compiler warning settings used to compile the object
- **NLS_LENGTH_SEMANTICS:** The NLS length semantics used to compile the object
- **PLSQL_CCFLAGS:** The conditional compilation flag used to compile the object
- **PLSCOPE_SETTINGS:** Controls the compile-time collection, cross reference, and storage of PL/SQL source code identifier data
- **ORIGIN_CON_ID:** The identifier of the container where the data originates. This value is 0 for non-container databases.

Displaying and Setting PL/SQL Initialization Parameters

```
SELECT name, type, plsql_code_type,
       plsql_optimize_level
  FROM user_plsql_object_settings;
```

NAME	TYPE	PLSQL_OPTIMIZE_LEVEL
1 ADD_COL	PROCEDURE	INTERPRETED
2 ADD_DEPARTMENT	PROCEDURE	INTERPRETED
3 ADD_DEPARTMENT_NOEX	PROCEDURE	INTERPRETED
4 ADD_DEPT	PROCEDURE	INTERPRETED
5 ADD_JOB_HISTORY	PROCEDURE	INTERPRETED
6 ADD_ROW	PROCEDURE	INTERPRETED
7 ANNUAL_SAL	FUNCTION	INTERPRETED
8 AUDIT_EMP_VALUES	TRIGGER	INTERPRETED
9 BANK_TRANS	PROCEDURE	INTERPRETED
10 CHECK_SALARY	TRIGGER	INTERPRETED

- Set the compiler initialization parameter's value by using the ALTER SYSTEM or ALTER SESSION statements.
- The parameters' values are accessed when the CREATE OR REPLACE statement is executed.



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

You can set the value of the compilation parameter through ALTER SESSION as follows:

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL=1;
```

Changing PL/SQL Initialization Parameters: Example

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;
ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE';
```

Session altered.

Session altered.

```
-- code displayed in the notes page
CREATE OR REPLACE PROCEDURE add_job_history
. . .
```

@code_11_09_sa.sql

The screenshot shows a table titled 'Script Output' with 50 rows fetched in 0.011 seconds. The columns are NAME, TYPE, CODE_TYPE, and OPT_LVL. The last row, '6 ADD_JOB_HISTORY', has its entire row highlighted with a red border. The 'CODE_TYPE' column for this row contains 'NATIVE' and the 'OPT_LVL' column contains '1'. The other five rows have 'INTERPRETED' in the 'CODE_TYPE' column and '0' in the 'OPT_LVL' column.

NAME	TYPE	CODE_TYPE	OPT_LVL
1 ADD_COL	PROCEDURE	INTERPRETED	0
2 ADD_DEPARTMENT	PROCEDURE	INTERPRETED	0
3 ADD_DEPARTMENT_NOEX	PROCEDURE	INTERPRETED	0
4 ADD_DEPT	PROCEDURE	INTERPRETED	0
5 ADD_EMPLOYEE	PROCEDURE	INTERPRETED	0
6 ADD_JOB_HISTORY	PROCEDURE	NATIVE	1

ORACLE

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

Lesson Agenda

- Using PL/SQL initialization parameters
- Using the PL/SQL compile-time warnings:
 - Using the `PLSQL_WARNING` initialization parameter
 - Using the `DBMS_WARNING` package subprograms



PL/SQL Compile-Time Warnings

- Messages from compiler that help in improving code maintainability
- Developers can use them to reduce the chances of bugs creeping in to the application.
- A PL/SQL unit flagged with warnings may execute successfully, but might exhibit unexpected behavior or inefficient performance.
- All PL/SQL warning messages use the prefix `PLW`.



ORACLE

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

To make your programs more robust and avoid problems at run time, you can turn on checking for certain warning conditions. These conditions are not serious enough to produce an error and keep you from compiling a subprogram. They may point out something in the subprogram that produces an undefined result or might create a performance problem.

With the PL/SQL compiler-warning feature, compiling a PL/SQL program could have the following two possible outcomes:

- Success with compilation warnings
- Failure with compilation errors and compilation warnings

Note that the compiler may issue warning messages even on a successful compile. A compilation error must be corrected to be able to use the stored procedure, whereas a warning is for informational purposes.

Examples of warning messages

SP2-0804: Procedure created with compilation warnings

PLW-07203: Parameter “`IO_TBL`” may benefit from use of the `NOCOPY` compiler hint.

Benefits of Compiler Warnings

- Make programs more robust and avoid problems at run time
- Identify potential performance problems
- Identify factors that produce undefined results



ORACLE

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

Using compiler warnings can help you to:

- Make your programs more robust and avoid problems at run time
- Identify potential performance problems
- Identify factors that produce undefined results

All PL/SQL warning messages use the prefix PLW.

Categories of PL/SQL Compile-Time Warning Messages

SEVERE	PERFORMANCE	INFORMATIONAL
Condition might cause unexpected action or wrong results.	Condition might cause performance problems.	Condition does not affect performance or correctness, but you might want to change it for better maintainability.

ORACLE

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

Enabling Warning Messages

You can enable displaying warning messages by using one of the following methods:

- The `PLSQL_WARNINGS` initialization parameter
- The `DBMS_WARNING` package



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

You can enable the display of compiler warning messages by using one of the following methods:

The `PLSQL_WARNINGS` Initialization Parameter

The `PLSQL_WARNINGS` parameter enables or disables warning messages being reported by the PL/SQL compiler and specifies which warning messages to show as errors. The settings for the `PLSQL_WARNINGS` parameter are stored along with each compiled subprogram. You can use the `PLSQL_WARNINGS` initialization parameter to do the following:

- Enable or disable the reporting of all warnings, warnings of a selected category, or specific warning messages.
- Treat all warnings, a selected category of warning, or specific warning messages as errors.
- Following is the syntax to do so:

```
PLSQL_WARNINGS = '<DISABLE | ENABLE | ERROR>:<ALL | INFORMATIONAL |  
PERFORMANCE | SEVERE>'  
PLSQL_WARNINGS = 'DISABLE:<warning#>'ENABLE:<warning#>',  
'ERROR:<warning#>'
```

You can disable or enable certain category of warnings. You can also configure the compiler to report certain category of warnings as errors.

Through the `PLSQL_WARNINGS` parameter you can enable, disable, or report specific warnings as errors.

The `DBMS_WARNING` Package

The `DBMS_WARNING` package provides a way to manipulate the behavior of PL/SQL warning messages, in particular by reading and changing the setting of the `PLSQL_WARNINGS` initialization parameter, to control what kinds of warnings are suppressed, displayed, or treated as errors. This package provides the interface to query, modify, and delete current system or session settings.

Setting Compiler Warning Levels: Using PLSQL_WARNINGS, Examples

```
ALTER SESSION  
SET plsql_warnings = 'enable:severe',  
    'enable:performance',  
    'disable:informational';
```

Session altered.

```
ALTER SESSION  
SET plsql_warnings = 'enable:severe';
```

Session altered.

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE',  
    'DISABLE:PERFORMANCE' , 'ERROR:05003';
```

Session altered.

ORACLE

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

You can use the ALTER SESSION or ALTER SYSTEM command to change the PLSQL_WARNINGS initialization parameter. The graphic in the slide shows the various examples of enabling and disabling compiler warnings.

Example 1

In this example, you are enabling SEVERE and PERFORMANCE warnings and disabling INFORMATIONAL warnings.

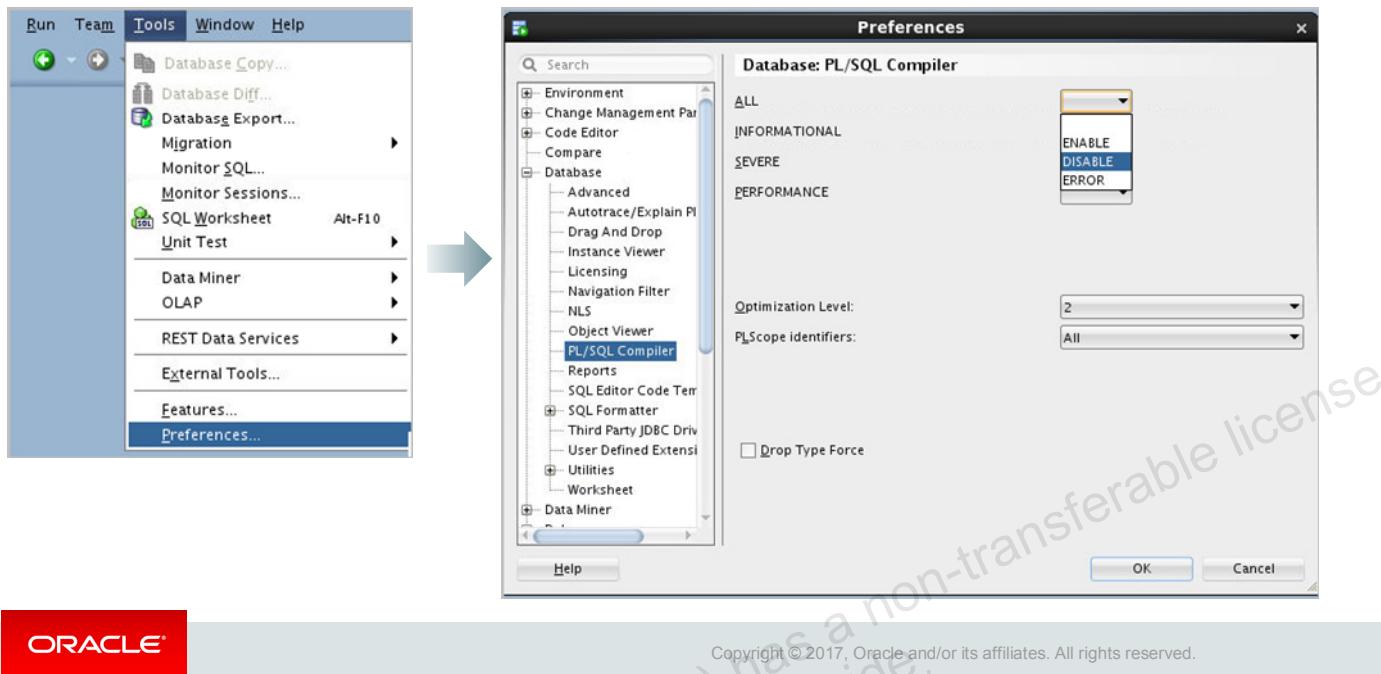
Example 2

In the second example, you are enabling only SEVERE warnings.

Example 3

You can treat particular messages as errors, instead of warnings. In this example, if you know that the warning message PLW-05003 represents a serious problem in your code; ERROR:05003 in the PLSQL_WARNINGS setting makes that condition trigger an error message (PLS_05003) instead of a warning message. An error message causes the compilation to fail. In this example, you are also disabling PERFORMANCE warnings.

Enabling Compiler Warnings: Using PLSQL_WARNINGS in SQL Developer



The PL/SQL Compiler pane specifies options for compilation of PL/SQL subprograms.

Setting and Viewing the PL/SQL Compile-Time Warning Messages Categories in SQL Developer

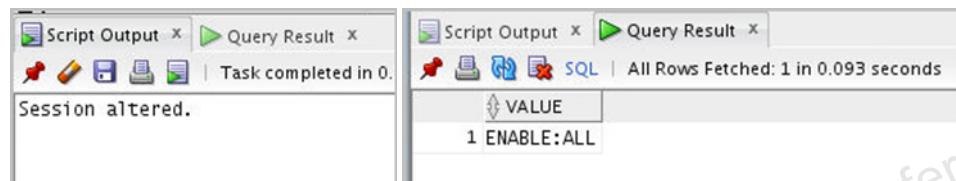
You can control the display of informational, severe, and performance-related messages. The **ALL** type overrides any individual specifications for the other types of messages. For each type of message, you can specify any of the following:

- **No entry (blank)**: Use any value specified for **ALL**; and if no value is specified, use the Oracle default.
- **Enable**: Enable the display of all messages of this category.
- **Disable**: Disable the display of all messages of this category.
- **Error**: Enable the display of only error messages of this category.

Viewing the Current Setting of PLSQL_WARNINGS

You can examine the current setting for the PLSQL_WARNINGS parameter by issuing a SELECT statement on the V\$PARAMETER view.

```
ALTER SESSION SET plsql_warnings = 'enable:severe',
'enable:performance', 'enable:informational';
/
SELECT value FROM v$parameter WHERE name='plsql_warnings';
/
```

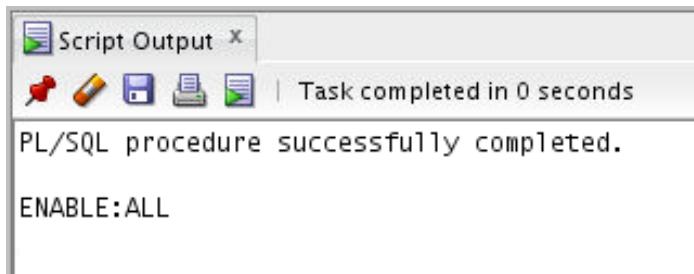


ORACLE

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

Alternatively, you can use the DBMS_WARNING.GET_WARNING_SETTING_STRING package and procedure to retrieve the current settings for the PLSQL_WARNINGS parameter:

```
DECLARE s VARCHAR2(1000);
BEGIN
  s := dbms_warning.get_warning_setting_string();
  dbms_output.put_line (s);
END;
/
```



Viewing Compiler Warnings

- You can view compiler warnings through:
 - SQL*Plus with SHOW ERRORS command
 - Data Dictionary views



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

You can use SQL*Plus to see any warnings raised as a result of the compilation of a PL/SQL block. SQL*Plus indicates that a compilation warning has occurred. The “**SP2-08xx: <object> created with compilation warnings.**” message is displayed for objects compiled .You must enable the compiler warnings before compiling the program. You can display the compiler warning messages by using any of the following methods:

Using the SQL*Plus SHOW ERRORS Command

This command displays any compiler errors including the new compiler warnings and informational messages. This command is invoked immediately after a CREATE [PROCEDURE | FUNCTION | PACKAGE] command is used. The SHOW ERRORS command displays warnings and compiler errors. New compiler warnings and informational messages are “interleaved” with compiler errors when SHOW ERRORS is invoked.

Using Data Dictionary Views

You can select from the USER_ | ALL_ | DBA_ERRORS data dictionary views to display PL/SQL compiler warnings. The ATTRIBUTES column of these views has a new attribute called WARNING and the warning message displays in the TEXT column.

SQL*Plus Warning Messages: Example

```
CREATE OR REPLACE PROCEDURE bad_proc(p_out ...) IS
BEGIN
. . .
END;
/
SP2-0804: Procedure created with compilation warnings.
```

```
Procedure BAD_PROC compiled
Errors: check compiler log
Errors for PROCEDURE ORA61.BAD_PROC:
LINE/COL ERROR
-----
1/26    PLS-00103: Encountered the symbol "?" when expecting one of the following:
          in out <an identifier> <a double-quoted delimited-identifier>
          table ... columns long double ref char time timestamp
          interval date binary national character nchar
```



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

Use the `SHOW ERRORS` command in SQL*Plus to display the compilation errors of a stored procedure. When you specify this option with no arguments, SQL*Plus displays the compilation errors for the most recently created or altered stored procedure. If SQL*Plus displays a compilation warnings message after you create or alter a stored procedure, you can use `SHOW ERRORS` commands to obtain more information.

With the introduction of support for PL/SQL warnings, the range of feedback messages is expanded to include a third message as follows:

`SP2-08xx: <object> created with compilation warnings.`

This enables you to differentiate between the occurrence of a compilation warning and a compilation error. You must correct an error if you want to use the stored procedure, whereas a warning is for informational purposes only.

The `SP2` prefix is included with the warning message because this provides you with the ability to look up the corresponding message number in the *SQL*Plus User's Guide and Reference* to determine the cause and action for the particular message.

Note: You can also view the compiler errors and warnings by using the `USER_ | ALL_ | DBA_ERRORS` data dictionary views.

Defining PLSQL_WARNINGS for Program Units

- The settings for the `PLSQL_WARNINGS` parameter are stored along with each compiled subprogram.
- If you recompile the subprogram using one of the following statements, the current settings for that session are used:
 - `CREATE OR REPLACE`
 - `ALTER ... COMPILE`
- If you recompile the subprogram using the `ALTER ... COMPILE` statement with the `REUSE SETTINGS` clause, the original setting stored with the program is used.



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

Lesson Agenda

- Using the PL/SQL initialization parameters
- Using the PL/SQL compile-time warnings:
 - Using the `PLSQL_WARNING` initialization parameter
 - Using the `DBMS_WARNING` package subprograms



Using the DBMS_WARNINGS Package

You can use the `DBMS_WARNINGS` package to:

- Manipulate the behavior of PL/SQL warning messages
- Change the definition of `PLSQL_WARNINGS` parameter
- Query, modify, and delete current system or session settings



ORACLE

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

Using the DBMS_WARNING Package Subprograms

Scenario	Subprograms to Use
Set warnings	ADD_WARNING_SETTING_CAT (procedure) ADD_WARNING_SETTING_NUM (procedure)
Query warnings	GET_WARNING_SETTING_CAT (function) GET_WARNING_SETTING_NUM (function) GET_WARNING_SETTING_STRING (function)
Replace warnings	SET_WARNING_SETTING_STRING (procedure)
Get the warnings' categories names	GET_CATEGORY (function)



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

The DBMS_WARNING Procedures: Syntax, Parameters, and Allowed Values

```
EXECUTE DBMS_WARNING.ADD_WARNING_SETTING_CAT (-  
    warning_category      IN      VARCHAR2,  
    warning_value         IN      VARCHAR2,  
    scope                 IN      VARCAHR2) ;
```

```
EXECUTE DBMS_WARNING.ADD_WARNING_SETTING_NUM (-  
    warning_number        IN      NUMBER,  
    warning_value         IN      VARCHAR2,  
    scope                 IN      VARCAHR2) ;
```

```
EXECUTE DBMS_WARNING.SET_WARNING_SETTING_STRING (-  
    warning_value        IN      VARCHAR2,  
    scope                 IN      VARCHAR2) ;
```



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

The `warning_category` is the name of the category. The allowed values are:

- ALL
- INFORMATIONAL
- SEVERE
- PERFORMANCE.

The `warning_value` is the value for the category. The allowed values are:

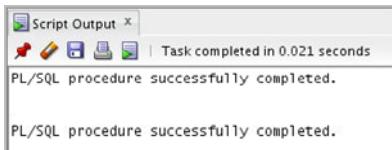
- ENABLE
- DISABLE
- ERROR

The `warning_number` is the warning message number. The allowed values are all valid warning numbers.

The `scope` specifies whether the changes are being performed in the session context or the system context. The allowed values are SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.

The DBMS_WARNING Procedures: Example

```
-- Establish the following warning setting string in the  
-- current session:  
-- ENABLE:INFORMATIONAL,  
-- DISABLE:PERFORMANCE,  
-- ENABLE:SEVERE  
  
EXECUTE DBMS_WARNING.SET_WARNING_SETTING_STRING(-  
    'ENABLE:ALL', 'SESSION');  
  
EXECUTE DBMS_WARNING.ADD_WARNING_SETTING_CAT(-  
    'PERFORMANCE', 'DISABLE', 'SESSION');
```



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

By using the SET_WARNING_SETTING_STRING procedure, you can set one warning setting. If you have multiple warning settings, you should perform the following steps:

1. Call SET_WARNING_SETTING_STRING to set the initial warning setting string.
2. Call ADD_WARNING_SETTING_CAT (or ADD_WARNING_SETTING_NUM) repeatedly to add more settings to the initial string.

The example in the slide establishes the following warning setting string in the current session:

ENABLE : INFORMATIONAL, DISABLE : PERFORMANCE, ENABLE : SEVERE

The DBMS_WARNING Functions: Syntax, Parameters, and Allowed Values

```
DBMS_WARNING.GET_WARNING_SETTING_CAT (-  
    warning_category IN VARCHAR2) RETURN warning_value;
```

```
DBMS_WARNING.GET_WARNING_SETTING_NUM (-  
    warning_number IN NUMBER) RETURN warning_value;
```

```
DBMS_WARNING.GET_WARNING_SETTING_STRING  
    RETURN pls_integer;
```

```
DBMS_WARNING.GET_CATEGORY (-  
    warning_number IN pls_integer) RETURN VARCHAR2;
```



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

The `warning_category` is the name of the category. The allowed values are:

- ALL
- INFORMATIONAL
- SEVERE
- PERFORMANCE

The `warning_number` is the warning message number. The allowed values are all valid warning numbers.

The `scope` specifies whether the changes are being performed in the session context or the system context. The allowed values are SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.

Note: Use the `GET_WARNING_SETTING_STRING` function when you do not have the SELECT privilege on the v\$parameter or v\$parameter2 fixed tables, or if you want to parse the warning string yourself, and then modify and set the new value by using `SET_WARNING_SETTING_STRING`.

The DBMS_WARNING Functions: Example

```
-- Determine the current session warning settings  
SET SERVEROUTPUT ON  
EXECUTE DBMS_OUTPUT.PUT_LINE( -  
DBMS_WARNING.GET_WARNING_SETTING_STRING);
```

PL/SQL procedure successfully completed.
ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE

```
-- Determine the category for warning message number  
-- PLW-07203  
SET SERVEROUTPUT ON  
EXECUTE DBMS_OUTPUT.PUT_LINE( -  
DBMS_WARNING.GET_CATEGORY(7203));
```

PL/SQL procedure successfully completed.
PERFORMANCE



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

Using DBMS_WARNING: Example

```
CREATE OR REPLACE PROCEDURE compile_code(p_pkg_name VARCHAR2) IS
    v_warn_value    VARCHAR2(200);
    v_compile_stmt VARCHAR2(200) := 
        'ALTER PACKAGE '|| p_pkg_name ||' COMPILE';

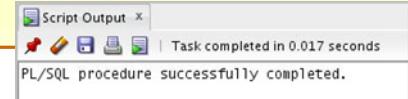
    BEGIN
        v_warn_value := DBMS_WARNING.GET_WARNING_SETTING_STRING;
        DBMS_OUTPUT.PUT_LINE('Current warning settings: ' ||
            v_warn_value);
        DBMS_WARNING.ADD_WARNING_SETTING_CAT(
            'PERFORMANCE', 'DISABLE', 'SESSION');
        DBMS_OUTPUT.PUT_LINE('Modified warning settings: ' ||
            DBMS_WARNING.GET_WARNING_SETTING_STRING);
        EXECUTE IMMEDIATE v_compile_stmt;
        DBMS_WARNING.SET_WARNING_SETTING_STRING(v_warn_value,
            'SESSION');
        DBMS_OUTPUT.PUT_LINE('Restored warning settings: ' ||
            DBMS_WARNING.GET_WARNING_SETTING_STRING);
    END;
/
```



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

Using DBMS_WARNING: Example

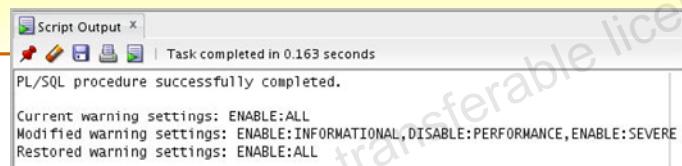
```
EXECUTE DBMS_WARNING.SET_WARNING_SETTING_STRING(-  
  'ENABLE:ALL', 'SESSION');
```



```
@/home/oracle/labs/plpu/code_ex/code_ex_scripts/code_12_33_s.sql
```



```
EXECUTE compile_code('EMP_PKG');
```



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

Quiz



The categories of PL/SQL compile-time warning messages are:

- a. SEVERE
- b. PERFORMANCE
- c. INFORMATIONAL
- d. ALL
- e. CRITICAL



ORACLE

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

Summary

In this lesson, you should have learned how to:

- Use the PL/SQL compiler initialization parameters
- Use the PL/SQL compile-time warnings



ORACLE®

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

Practice 12 Overview: Tuning PL/SQL Compiler

This practice covers the following topics:

- Displaying the compiler initialization parameters
- Enabling native compilation for your session and compiling a procedure
- Disabling the compiler warnings, and then restoring the original session-warning settings
- Identifying the categories for some compiler-warning message numbers



ORACLE

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



13

Managing Dependencies

ORACLE

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

Objectives

After completing this lesson, you should be able to:

- Track procedural dependencies
- Predict the effect of changing a database object on procedures and functions
- Manage procedural dependencies
- Manage local and remote dependencies



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 the with PL/SQL Code

▶ Lesson 11: Design Considerations for the PL/SQL Code

▶ Lesson 12: Tuning the PL/SQL Compiler

▶ Lesson 13: Managing Dependencies

ORACLE

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

Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



What are Dependencies in a Schema?

- Dependency refers to a situation where one schema object depends on another object for its definition.

Examples

- There is a dependency between a view and the set of tables on which it is defined.
- There is a dependency between the procedure and the tables on which the procedure is defined.

How Dependencies Work?

Let us create the following views in the HR schema:

```
CREATE OR REPLACE VIEW commissioned AS SELECT first_name, last_name,  
commission_pct FROM employees WHERE commission_pct > 0.00;  
  
CREATE OR REPLACE VIEW emp_mails AS SELECT first_name, last_name, email FROM  
employees;
```

```
SELECT object_name, status  
FROM user_objects  
WHERE object_type = 'VIEW'  
ORDER BY object_name;
```

Query Result	
OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	VALID

ORACLE

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

How Dependencies Work?

Let us modify the email column of the employees table:

```
ALTER TABLE employees MODIFY email VARCHAR2(100);
```

And check the state of the views:

```
SELECT object_name, status  
FROM user_objects  
WHERE object_type = 'VIEW'  
ORDER BY object_name;
```

OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	INVALID

You can see that after the email column in the employees table is modified, the status emp_mails view has become invalid.



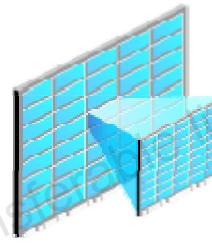
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Dependent and Referenced Objects

Some types of schema objects can reference other objects in their definitions.

For example:

- A view is defined by a query that references tables.
 - View is a dependent object.
 - Tables are referenced objects.



ORACLE

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

Some types of schema objects can reference other objects in their definitions. For example, a view is defined by a query that references tables or other views, and the body of a subprogram can include SQL statements that reference other objects. If the definition of object A references object B, then A is a dependent object (with respect to B) and B is a referenced object (with respect to A).

Dependency Issues

- If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, the procedure may or may not continue to work without error.
- The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary and you can view the status in the `USER_OBJECTS` data dictionary view.
- If the status of a schema object is `VALID`, then the object has been compiled and can be immediately used when referenced.
- If the status of a schema object is `INVALID`, then the schema object must be compiled before it can be used.

The following table shows schema objects can be dependent, referenced, or both

Object Type	Can Be Dependent or Referenced
Package body	Dependent only
Package specification	Both
Sequence	Referenced only
Subprogram	Both
Synonym	Both
Table	Both
Trigger	Both
User-defined object	Both
User-defined collection	Both
View	Both

Querying Object Dependencies: Using the USER_DEPENDENCIES View

```
desc user_dependencies;
```

Name	Null	Type
NAME	NOT NULL	VARCHAR2(128)
TYPE		VARCHAR2(19)
REFERENCED_OWNER		VARCHAR2(128)
REFERENCED_NAME		VARCHAR2(128)
REFERENCED_TYPE		VARCHAR2(19)
REFERENCED_LINK_NAME		VARCHAR2(128)
SCHEMAD		NUMBER
DEPENDENCY_TYPE		VARCHAR2(4)

```
SELECT name, type, referenced_name, referenced_type  
FROM user_dependencies  
WHERE referenced_name IN ('EMPLOYEES', 'EMP_VW' );
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
1 SECURE_EMPLOYEES	TRIGGER	EMPLOYEES	TABLE
2 DML_CALL_SQL	FUNCTION	EMPLOYEES	TABLE
3 GET_EMP	FUNCTION	EMPLOYEES	TABLE
4 SECURE_EMP	TRIGGER	EMPLOYEES	TABLE
5 EMPLOYEE_SAL	PROCEDURE	EMPLOYEES	TABLE
6 EMP_MAILS	VIEW	EMPLOYEES	TABLE
7 UPDATE_JOB_HISTORY	TRIGGER	EMPLOYEES	TABLE
8 RAISE_SAL	PROCEDURE	EMPLOYEES	TABLE

ORACLE

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

Querying an Object's Status

Every database object has one of the following status values:

Status	Description
VALID	The object was successfully compiled by using the current definition in the data dictionary.
COMPILED WITH ERRORS	The most recent attempt to compile the object produced errors.
INVALID	The object is marked invalid because an object that it references has changed. (Only a dependent object can be invalid.)
UNAUTHORIZED	An access privilege on a referenced object was revoked. (Only a dependent object can be unauthorized.)



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

Every database object has one of the status values shown in the table in the slide.

Note: The `USER_OBJECTS`, `ALL_OBJECTS`, and `DBA_OBJECTS` static data dictionary views do not distinguish between Compiled with errors, Invalid, and Unauthorized; instead, they describe all these as `INVALID`.

Categorizing Dependencies

- Local dependencies – Both the referenced and dependent objects are on the same database. Oracle Database manages them implicitly. There are two types of dependencies in local dependencies using the internal tables.
 - Direct dependencies
 - Indirect dependencies
- Remote dependencies – The referenced and dependent objects are on different nodes across the network. Oracle Database uses different mechanisms to manage remote dependencies, depending on the objects involved.

ORACLE

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

Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



Direct Dependencies

- When a view V is defined on a table T, there is a direct dependency of the view V on the table T.
- Direct dependents are invalidated only by changes to the referenced objects that affect them.

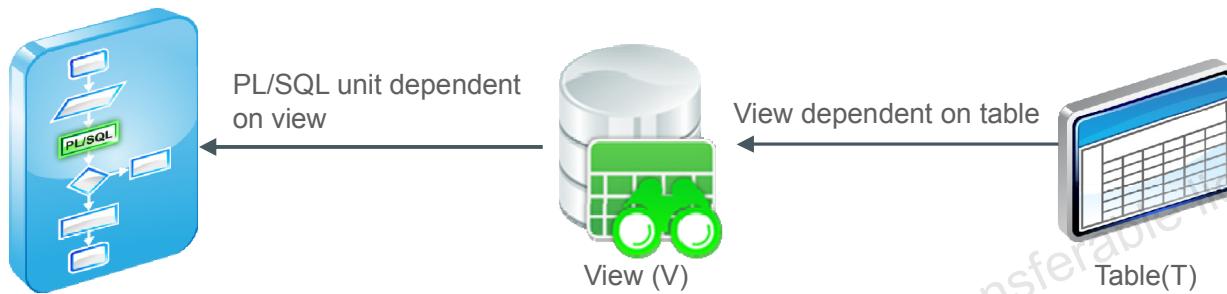


ORACLE

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

Indirect Dependencies

- Consider a PL/SQL program unit dependent on the view V, which in turn is dependent on the table T. The PL/SQL program unit is indirectly dependent on the table T.
- Indirect dependents can be invalidated by changes to the referenced object that do not affect them.
- The PL/SQL unit may be invalidated if there is a modification to the table.



ORACLE

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

The slide shows an indirect dependency between the PL/SQL unit and the table. So, the PL/SQL unit is an indirect dependent on the table.

Indirect dependents can be invalidated by changes to the reference object that do not affect them. A modification to the table may invalidate the PL/SQL unit. This is called cascading invalidation.

Displaying Direct and Indirect Dependencies

- Run the `utldtree.sql` script that creates the objects that enable you to display the direct and indirect dependencies.

```
@/home/oracle/labs/plpu/labs/utldtree.sql
```

- This script creates four objects, as given below:
 - A table `deptree temptab` to hold dependency data
 - A procedure `deptree_fill` to populate the table
 - Two views – `deptree` and `ideptree` – to select and format dependency data from the populated table



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

Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle

You can display direct and indirect dependencies from additional user views, called `DEPTREE` and `IDEPTREE`; these views are provided by Oracle.

Example

- Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/labs/plpu/labs` folder. You can run the script as follows:

```
@?/labs/plpu/labs/utldtree.sql
```

Note: In this class, this script is supplied in the `labs` folder of your class files. The code example above uses the student account `ORA61`. (This applies to a Linux environment. If the file is not found, locate the file in your `labs` subdirectory.)

- Populate the `DEPTREE_TEMP TAB` table with information for a particular referenced object by invoking the `DEPTREE_FILL` procedure. There are three parameters for this procedure:

<code>object_type</code>	Type of the referenced object
<code>object_owner</code>	Schema of the referenced object
<code>object_name</code>	Name of the referenced object

Displaying Direct and Indirect Dependencies

- Execute the DEPTREE_FILL procedure.

```
EXECUTE deptree_fill('TABLE', 'ORA61', 'EMPLOYEES')
```

- Display the dependencies

```
SELECT nested_level, type, name  
FROM deptree  
ORDER BY seq#;
```

NESTED_LEVEL	TYPE	NAME
1	0 TABLE	EMPLOYEES
2	1 TRIGGER	SECURE_EMPLOYEES
3	1 TRIGGER	UPDATE_JOB_HISTORY
4	1 PROCEDURE	RAISE_SAL
5	1 PROCEDURE	DISPLAY_NEW_SAL
6	1 FUNCTION	GET_SAL
7	1 FUNCTION	TAX
8	1 FUNCTION	EMP_TAX

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



You can display a tabular representation of all dependent objects by querying the DEPTREE view. You can display an indented representation of the same information by querying the IDEPTREE view, which consists of a single column named DEPENDENCIES as follows:

```
SELECT *  
FROM ideptree;
```

DEPENDENCIES
1 PROCEDURE ORA61.RAISE_SAL
2 FUNCTION ORA61.DML_CALL_SQL
3 PROCEDURE ORA61.SAL_STATUS
4 PROCEDURE ORA61.RAISE_SALARY
5 VIEW ORA61.COMMISSIONED
6 FUNCTION ORA61.TAX
7 FUNCTION ORA61.EMP_TAX
8 VIEW ORA61.EMP_MAILS

...

Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



Fine-Grained Dependency Management

```
CREATE OR REPLACE VIEW commissioned AS SELECT first_name, last_name,
commission_pct FROM employees WHERE commission_pct > 0.00;
CREATE OR REPLACE VIEW emp_mails AS SELECT first_name, last_name, email FROM
employees;
```

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
ORDER BY object_name;
```

Query Result	
OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	VALID

```
ALTER TABLE employees MODIFY email VARCHAR2(100);
```

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
ORDER BY object_name;
```

Query Result	
OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	INVALID



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

Fine-Grained Dependency Management

- Starting with Oracle Database 11g, dependencies are now tracked at the level of element within unit.
- Element-based dependency tracking covers the following:
 - Dependency of a single-table view on its base table
 - Dependency of a PL/SQL program unit (package specification, package body, or subprogram) on the following:
 - Other PL/SQL program units
 - Tables
 - Views

ORACLE

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

Fine-Grained Dependency Management: Example 1

```
CREATE TABLE t2 (col_a NUMBER, col_b NUMBER, col_c NUMBER);
CREATE VIEW v AS SELECT col_a, col_b FROM t2;
```

```
SELECT ud.name, ud.type, ud.referenced_name,
       ud.referenced_type, uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:					
#	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1	V	VIEW	T2	TABLE	VALID

```
ALTER TABLE t2 ADD (col_d VARCHAR2(20));
```

```
SELECT ud.name, ud.type, ud.referenced_name,
       ud.referenced_type, uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:					
#	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1	V	VIEW	T2	TABLE	VALID



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

Example of Dependency of a Single-Table View on Its Base Table

In the first example in the slide, table T2 is created with three columns: COL_A, COL_B, and COL_C. A view named V is created based on columns COL_A and COL_B of table T2. The dictionary views are queried and view V is dependent on table T and its status is valid.

In the third example, table T2 is altered. A new column named COL_D is added. The dictionary views still report that view V is dependent because element-based dependency tracking grasps that columns COL_A and COL_B are not modified and, therefore, the view does not need to be invalidated.

Fine-Grained Dependency Management: Example 1

```
ALTER TABLE t2 MODIFY (col_a VARCHAR2(20));
SELECT ud.name, ud.referenced_name, ud.referenced_type, uo.status
FROM user_dependencies ud, user_objects uo
WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:

NAME	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1 V	T2	TABLE	INVALID

ORACLE

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

Fine-Grained Dependency Management: Example 2

```
CREATE OR REPLACE PACKAGE pkg IS
  PROCEDURE proc_1;
END pkg;
/
CREATE OR REPLACE PROCEDURE p IS
BEGIN
  pkg.proc_1();
END p;
/
CREATE OR REPLACE PACKAGE pkg
IS
  PROCEDURE proc_1;
  PROCEDURE unheard_of;
END pkg;
/
```

```
PACKAGE PKG compiled
PROCEDURE P compiled
PACKAGE PKG compiled
```



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

In the example in the slide, you create a package named `PKG` that has procedure `PROC_1` declared. A procedure named `P` invokes `PKG.PROC_1`.

The definition of the `PKG` package is modified and another subroutine is added to the package declaration.

When you query the `USER_OBJECTS` dictionary view for the status of the `P` procedure, it is still valid as shown because the element you added to the definition of `PKG` is not referenced through procedure `P`.

```
SELECT status FROM user_objects
WHERE object_name = 'P';
```

Results:	
	STATUS
	1 VALID

Guidelines for Reducing Invalidation

To reduce invalidation of dependent objects:

- Add new items to the end of the package
- Reference each table through a view



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

Add New Items to the End of the Package

When adding new items to a package, add them to the end of the package. This preserves the slot numbers and entry-point numbers of existing top-level package items, thereby preventing their invalidation. For example, consider the following package:

```
CREATE OR REPLACE PACKAGE pkg1 IS
FUNCTION get_var RETURN VARCHAR2;
PROCEDURE set_var (v VARCHAR2);
END;
```

Adding an item to the end of `pkg1` does not invalidate dependents that reference `get_var`. Inserting an item between the `get_var` function and the `set_var` procedure invalidates dependents that reference the `set_var` function.

Reference Each Table Through a View

Reference tables indirectly by using views. This allows you to do the following:

- Add columns to the table without invalidating dependent views or dependent PL/SQL objects
- Modify or delete columns not referenced by the view without invalidating dependent objects

Object Revalidation

- An object that is not valid when it is referenced must be validated before it can be used.
- Validation occurs automatically when an object is referenced; it does not require explicit user action.
- Revalidation may not happen automatically if:
 - The object is compiled with errors
 - The object is an unauthorized object
 - The object is an invalid object



ORACLE

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

The compiler cannot automatically revalidate an object that compiled with errors. The compiler recompiles the object, and if it recompiles without errors, it is revalidated; otherwise, it remains invalid.

The compiler checks whether the unauthorized object has access privileges to all of its referenced objects. If so, the compiler revalidates the unauthorized object without recompiling it. If not, the compiler issues appropriate error messages.

The SQL compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid.

For an invalid PL/SQL program unit (procedure, function, or package), the PL/SQL compiler checks whether any referenced object is an invalid object.

- If so, the compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid.
- If not, the compiler revalidates the invalid object without recompiling it. Fast revalidation is usually performed on objects that were invalidated due to cascading invalidation.

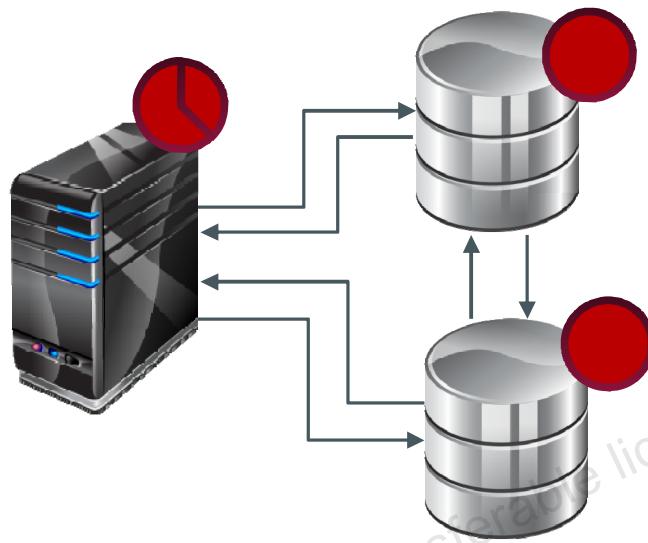
Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



Remote Dependencies

- The referenced and dependent objects are on different nodes across the network.
- Oracle Database does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.



ORACLE

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

In the case of remote dependencies, the objects are on separate nodes. The local stored procedure and all its dependent objects are invalidated, but are automatically recompile when called for the first time. Oracle Database does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.

For example, assume that a local view is created and defined by a query that references a remote table. Also assume that a local procedure includes a SQL statement that references the same remote table. Later, the definition of the table is altered.

As a result, the local view and procedure are never invalidated, even if the view or procedure is used after the table is altered and even if the view or procedure now returns errors when used. In this case, the view or procedure must be altered manually so that errors are not returned. In such cases, lack of dependency management is preferable to unnecessary recompilations of dependent objects.

Recompilation of Dependent Objects: Local and Remote

- Verify successful explicit recompilation of the dependent remote procedures and implicit recompilation of the dependent local procedures by checking the status of these procedures within the `USER_OBJECTS` view.
- If an automatic implicit recompilation of the dependent local procedures fails, the status remains invalid and the Oracle server issues a runtime error. Therefore, to avoid disrupting production, it is strongly recommended that you recompile local dependent objects manually, rather than rely on an automatic mechanism.

Managing Remote Procedure Dependencies

Mechanisms used for managing remote procedure dependencies are:

- Timestamp checking



- Signature checking



ORACLE

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

TIMESTAMP Checking

Each PL/SQL program unit carries a time stamp that is set when it is created or recompiled. Whenever you alter a PL/SQL program unit or a relevant schema object, all its dependent program units are marked as invalid and must be recompiled before they can execute. The actual time stamp comparison occurs when a statement in the body of a local procedure calls a remote procedure.

SIGNATURE Checking

For each PL/SQL program unit, both the time stamp and the signature are recorded. The signature of a PL/SQL construct contains information about the following:

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

The recorded time stamp in the calling program unit is compared with the current time stamp in the called remote program unit. If the time stamps match, the call proceeds. If they do not match, the remote procedure call layer performs a simple comparison of the signature to determine whether the call is safe or not. If the signature has not been changed in an incompatible manner, execution continues; otherwise, an error is returned.

Setting the REMOTE_DEPENDENCIES_MODE Parameter

- You can define whether a remote procedure call should undergo Timestamp checking or Signature checking by setting the REMOTE_DEPENDENCIES_MODE parameter.
 - At the system level:
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE = TIMESTAMP | SIGNATURE
 - At the session level:
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE = TIMESTAMP | SIGNATURE
 - As an init.ora parameter:
REMOTE_DEPENDENCIES_MODE = TIMESTAMP /
SIGNATURE
- If the REMOTE_DEPENDENCIES_MODE parameter is not specified, either in the init.ora parameter file or by using the ALTER SESSION or ALTER SYSTEM DDL statements, TIMESTAMP is the default value.

ORACLE

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

Timestamp Checking



Remote procedure B:
compiles and is VALID
at 8:00 AM

ORACLE

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

Local Procedures Referencing Remote Procedures

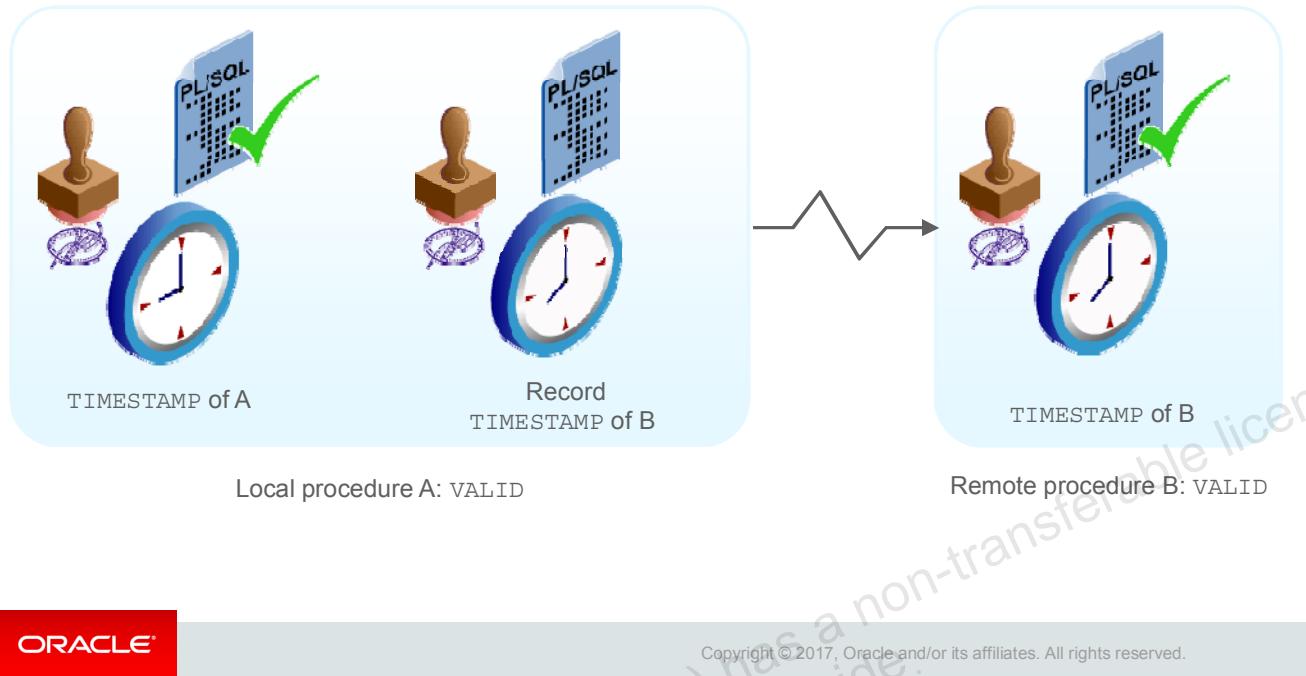
A local procedure that references a remote procedure is invalidated by the Oracle server if the remote procedure is recompiled after the local procedure is compiled.

Automatic Remote Dependency Mechanism

When a procedure compiles, the Oracle server records the time stamp of that compilation within the `P` code of the procedure.

In the slide, when the remote procedure B is successfully compiled at 8:00 AM, this time is recorded as its time stamp.

Timestamp Checking



ORACLE

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

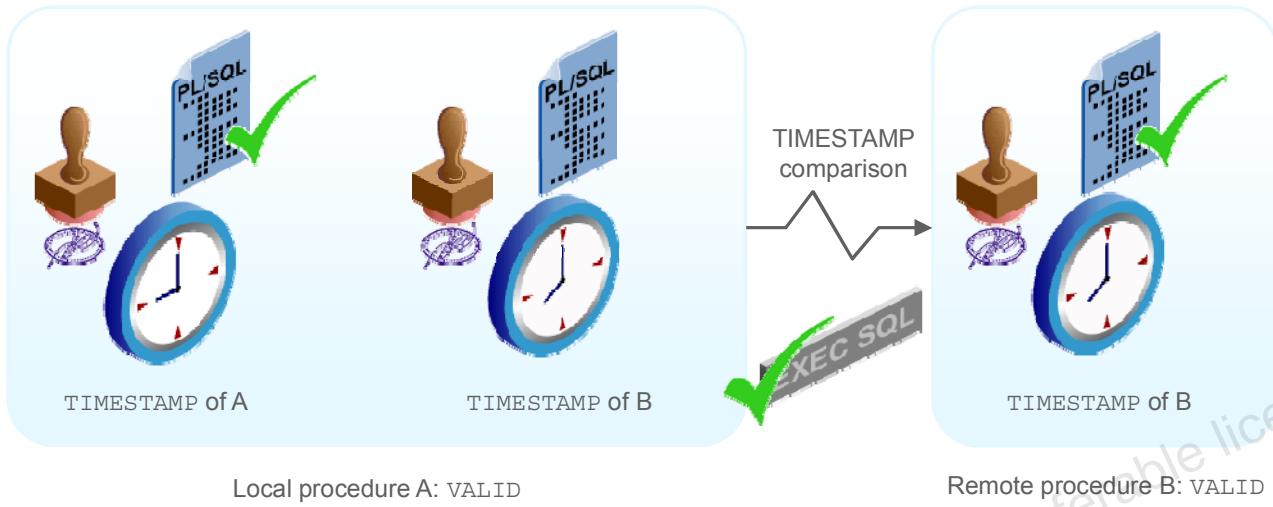
Local Procedures Referencing Remote Procedures

Automatic Remote Dependency Mechanism

When a local procedure referencing a remote procedure compiles, the Oracle server also records the time stamp of the remote procedure in the P code of the local procedure.

In the slide, local procedure A (which is dependent on remote procedure B) is compiled at 9:00 AM. The time stamps of both procedure A and remote procedure B are recorded in the P code of procedure A.

Timestamp Checking



ORACLE

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

Timestamp Checking



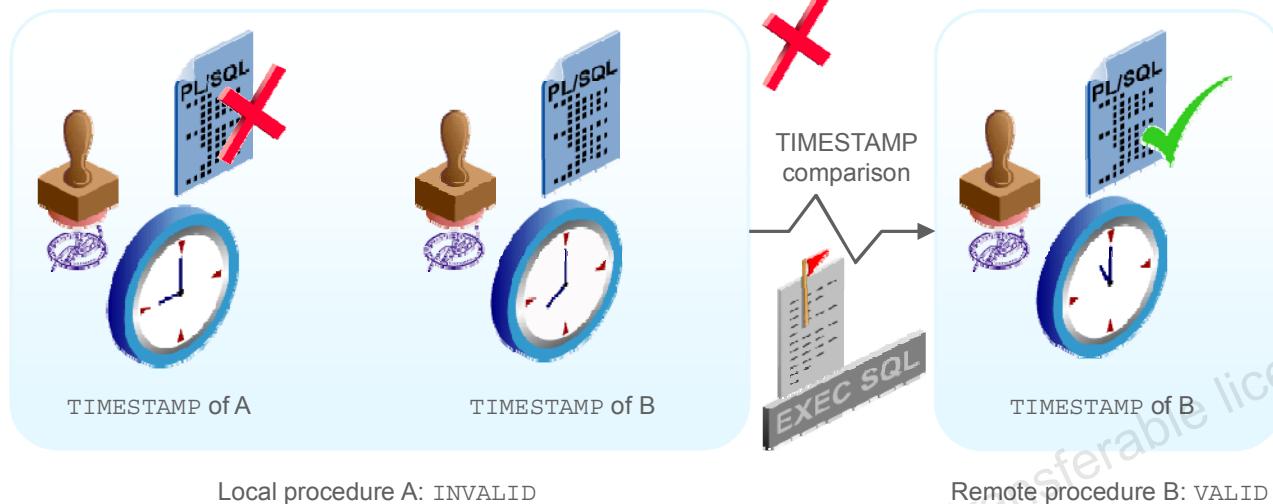
Remote procedure B:
Recompiles and is VALID
at 11:00 AM

ORACLE

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

Timestamp Checking

Saved TIMESTAMP of B != COMPILE TIME of B



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

Automatic Remote Dependency

If the time stamps are not equal (indicating that the remote procedure has recompiled), then the Oracle server invalidates the local procedure and returns a runtime error. If the local procedure (which is now tagged as invalid) is invoked a second time, then the Oracle server recompiles it before executing, in accordance with the automatic local dependency mechanism.

Note: If a local procedure returns a runtime error the first time it is invoked (indicating that the remote procedure's time stamp has changed), then you should develop a strategy to reinvoke the local procedure.

In the example in the slide, the remote procedure is recompiled at 11:00 AM and this time is recorded as its time stamp in the P code. The P code of local procedure A still has 8:00 AM as the time stamp for remote procedure B. Because the time stamp recorded with the P code of local procedure A is different from that recorded with the remote procedure B, the local procedure is marked invalid. When the local procedure is invoked for the second time, it can be successfully compiled and marked valid.

A disadvantage of the time stamp mode is that it is unnecessarily restrictive. Recompilation of dependent objects across the network is often performed when not strictly necessary, leading to performance degradation.

Signature Checking

- The signature of a procedure is:
 - The name of the procedure
 - The data types of the parameters
 - The modes of the parameters
- The signature of the remote procedure is saved in the local procedure.
- When executing a dependent procedure, the signature of the referenced remote procedure is compared.



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

Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



Revalidating PL/SQL Program Units

You revalidate a PL/SQL program unit by recompiling it. There are two ways of recompiling:

- Implicit runtime recompilation
 - Local dependencies are resolved implicitly
- Explicit recompilation with the `ALTER` statement
 - Here is the syntax for using the `ALTER` statement.

```
ALTER PROCEDURE | FUNCTION | TRIGGER | PACKAGE  
[SCHEMA.] object_name COMPILE;
```



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

Unsuccessful Recompilation

Recompiling-dependent procedures and functions are unsuccessful when:

- The referenced objects are compiled with errors
- The referenced objects are dropped or renamed
- The data type of the referenced column is changed
- The referenced column is dropped
- A referenced view is replaced by a view with different columns
- The parameter list of a referenced procedure is modified



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

Successful Recompilation

Recompiling-dependent procedures and functions are successful if:

- The referenced table has new columns
- The data type of referenced columns has not changed
- A private table is dropped, but a public table that has the same name and structure exists
- The PL/SQL body of a referenced procedure has been modified and recompiled successfully



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

The recompilation of dependent objects is successful if:

- New columns are added to a referenced table
- All `INSERT` statements include a column list
- No new column is defined as `NOT NULL`

When a private table is referenced by a dependent procedure and the private table is dropped, the status of the dependent procedure would become invalid. When the procedure is recompiled (either explicitly or implicitly) and a public table exists, the procedure can recompile successfully but is now dependent on the public table. The recompilation is successful only if the public table contains the columns that the procedure requires; otherwise, the status of the procedure remains invalid.

Recompiling Procedures

Minimize dependency failures by:

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



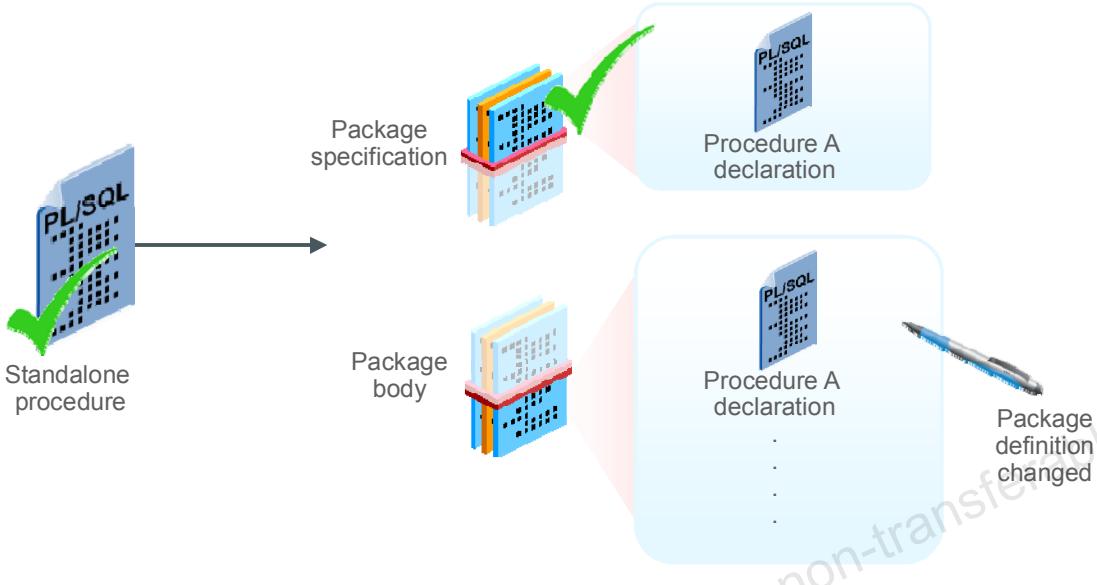
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



Packages and Dependencies: Subprogram References the Package



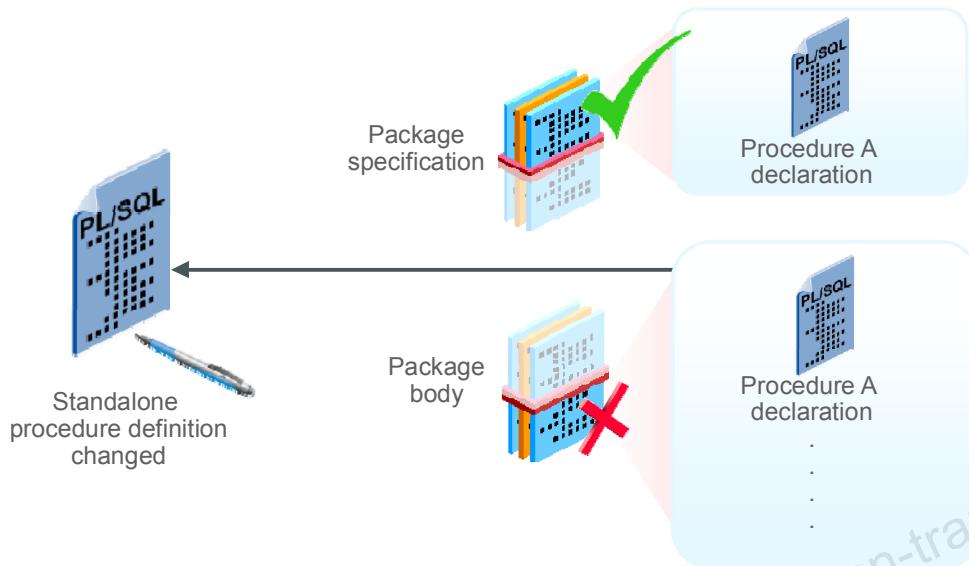
ORACLE

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

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

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

Packages and Dependencies: Package Subprogram References Procedure



ORACLE

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

Quiz



You can display direct and indirect dependencies by running the `utldtree.sql` script, populating the `DEPTREE_TEMP TAB` table with information for a particular referenced object, and querying the `DEPTREE` or `IDEPTREE` views.

- a. True
- b. False



ORACLE

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

Summary

In this lesson, you should have learned how to:

- Track procedural dependencies
- Predict the effect of changing a database object on procedures and functions
- Manage procedural dependencies
- Manage local and remote dependencies



ORACLE

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

Practice 13 Overview: Managing Dependencies in Your Schema

This practice covers the following topics:

- Using `DEPTREE_FILL` and `IDEPTREE` to view dependencies
- Recompiling procedures, functions, and packages



ORACLE

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



Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

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

Oracle Cloud Overview

Lesson Objectives

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

- Describe the salient features of Oracle Cloud
- Discuss the features of Oracle Database Exadata Express Cloud Service



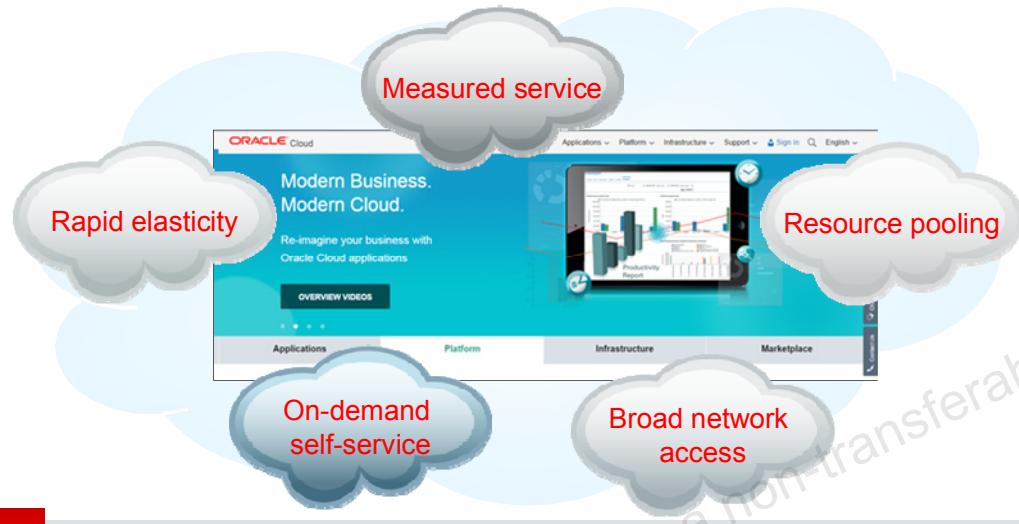
Lesson Agenda

- Overview of Oracle Cloud
- Working with Oracle Database Exadata Express Cloud Service



Introduction to Oracle Cloud

- Any business can now use the enterprise cloud provided by Oracle.
- You can access the Oracle Cloud from cloud.oracle.com.



The Oracle Cloud is an enterprise cloud for business. Oracle Cloud services are built on Oracle Exalogic Elastic Cloud and Oracle Exadata Database Machine, together offering a platform that delivers extreme performance and scalability.

The top two benefits of cloud computing are speed and cost.

As a result, the applications and databases deployed in the Oracle Cloud are portable and you can easily move them to or from a private cloud or on-premise environment.

- You can request and get the cloud services provisioned through a self-service interface.
- You can either use an integrated development and deployment platform to rapidly extend and create new services.

Using Oracle Cloud services, you can benefit from the following five essential characteristics:

- **On-demand self-service:** You can provision, monitor, and manage cloud on your own.
- **Resource pooling:** You can share resources and maintain a level of abstraction between consumers and services.
- **Rapid elasticity:** You can quickly scale up or down as needed.
- **Measured service:** You pay for what you use with either internal chargeback (private cloud) or external billing (public cloud).
- **Broad network access:** You can access the cloud services through a browser on any networked device.

Oracle Cloud Services

Oracle Cloud provides three types of services:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)



ORACLE

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

SaaS generally refers to applications that are delivered to end users over the Internet. Oracle CRM On Demand is an example of a SaaS offering that provides both multitenant as well as single-tenant options, depending on the customer's preferences.

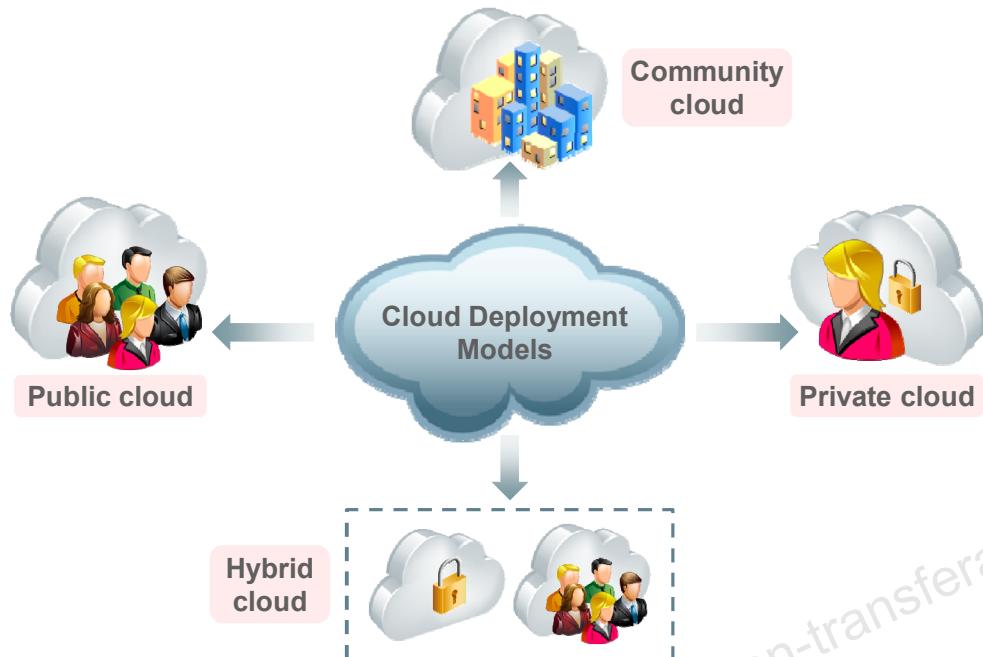
PaaS generally refers to an application development and deployment platform that is delivered as a service to developers, enabling them to quickly build and deploy a SaaS application to end users. The platform typically includes databases, middleware, and development tools, all delivered as a service via the Internet.

IaaS refers to computing hardware (servers, storage, and network) delivered as a service. This service typically includes the associated software as well as operating systems, virtualization, clustering, and so on. Examples of IaaS in the public cloud include Amazon's Elastic Compute Cloud (EC2) and Simple Storage Service (S3).

The Oracle Cloud Database is built as a PaaS model. It provides on-demand access to database services in a self-service, scalable, and metered manner. You can deploy a database within a virtual machine in an IaaS platform.

You can rapidly deploy Oracle Cloud Database on Oracle Exadata, which is a pre-integrated and optimized hardware platform that supports both online transaction processing (OLTP) and Data Warehouse workloads.

Cloud Deployment Models



ORACLE

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

- **Private cloud:** A single organization uses a private cloud, which it typically controls, manages, and hosts in private data centers. However, the organization can also outsource hosting and operation to a third-party service provider. Amazon's Virtual Private Cloud is an example of a private cloud in an external provider setting.
- **Public cloud:** Multiple organizations (tenants) use a private cloud on a shared basis. This private cloud is hosted and managed by a third-party service provider. For example: Amazon's Elastic Compute Cloud (EC2), IBM's Blue Cloud, Sun Cloud, and Google AppEngine
- **Community cloud:** A group of related organizations, who want to make use of a common cloud computing environment, uses the community cloud. It is managed by the participating organizations or by a third-party managed service provider. It is hosted internally or externally. For example, a community might consist of the different branches of the military, all the universities in a given region, or all the suppliers to a large manufacturer.
- **Hybrid cloud:** A single organization that wants to adopt both private and public clouds for a single application uses the hybrid cloud. A third model, the hybrid cloud, is maintained by both internal and external providers. For example, an organization might use a public cloud service, such as Amazon Simple Storage Service (Amazon S3), for archived data but continue to maintain in-house (private cloud) storage for operational customer data.

Lesson Agenda

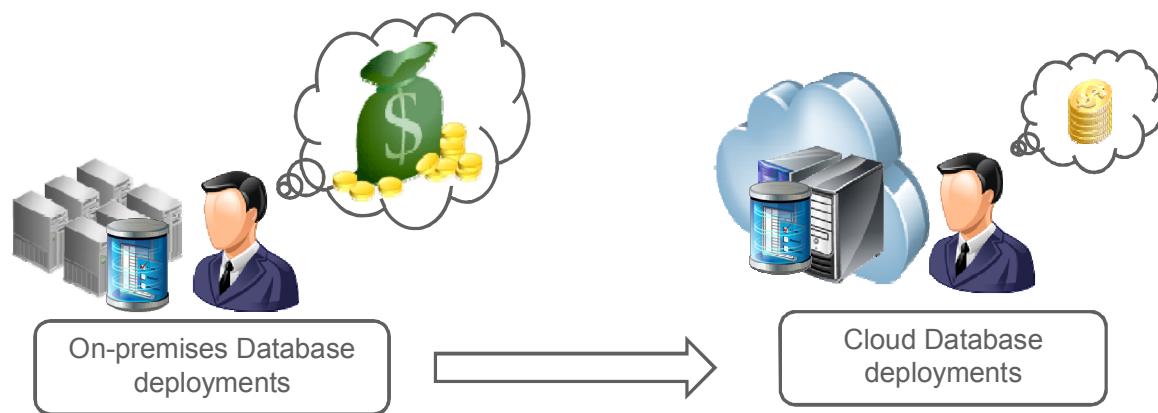
- Overview of Oracle Cloud
- Working with Oracle Database Exadata Express Cloud Service



ORACLE

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

Evolving from On-premises to Exadata Express



ORACLE

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

Cloud deployments provide end users and enterprises with different capabilities to store and process data. They enable users to have high performance and huge computing resources at a lower price as compared to traditional on-premises deployments.

Exadata Express is a powerful database machine, extended as a cloud service. End users can use it for Oracle 12c database deployments. It delivers a complete database experience for developers and enterprises.

Exadata express being a cloud deployment provides high scalability, performance and availability to its users.

It is fully managed database, therefore you need not worry about patching, upgrading or other DBA tasks.

What is in Exadata Express?

- A fully managed database service
- Provides powerful yet elastic database cloud service for developers
- Provides on-demand access to a shared pool of database resources
- Comes with built-in tools for rapid application development
 - APEX for web application development
 - Compatibility with clients such as SQL Developer, SQLcl



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

Exadata Express is a fully managed database service where end users need not worry about upgrades to the database and other components of the service. All enhancements are automatically managed by the cloud service.

Being a cloud deployment, Exadata Express, allows end users to scale their data virtually to unlimited size.

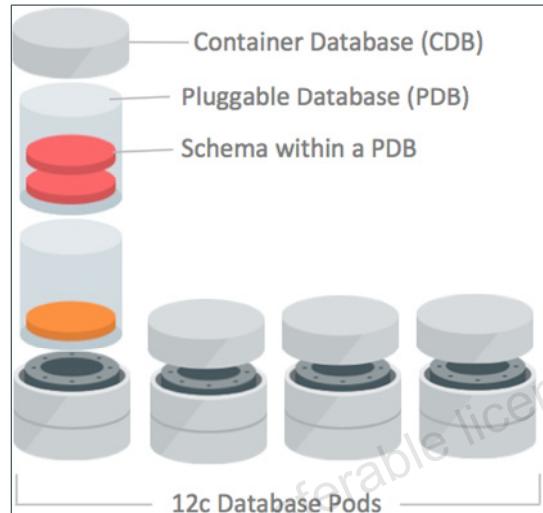
Dynamic provisioning of resources allows users to access huge amount of compute and storage resources in no time.

For developers, it provides built-in application development tool – APEX. APEX(Application Express) is a rapid web application development tool for Oracle database. Developers with minimal development experience can develop and deploy professional applications through web browser using APEX.

Oracle makes a variety of database client drivers and tools available for use with Oracle Database Exadata Express Cloud Service. You can use Exadata Express with Oracle SQL Developer, an IDE used for SQL, PL/SQL development and Oracle SQLcl, an enhanced command line interface.

Exadata Express for Users

- Oracle manages the service as multiple Container databases(CDBs), also known as database pods
- Each CDB can accommodate upto 1000 Pluggable databases(PDBs).
- Each user is provisioned with a PDB on subscribing to the service, where the user can create several schemas.



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

Exadata Express is ideal for production applications that consist of small to medium sized data as well as developers, testers, evaluators and other users who are looking for a full Oracle Database experience at an affordable entry-level price. It is a fully managed database service, is organized into Container databases(CDBs). These container databases are also known as database pods.

Each container database in turn can contain several Pluggable databases(PDBs). When a user subscribes to the Exadata Service, a pluggable database is provisioned. Within the PDB, the user can create several schemas. However, PDB Services are constrained by CPU, storage and memory.

Exadata Express for Developers

- Developers can connect with a wide range of data sources for their applications
 - JSON Document Storage
 - Document Style data access
 - Oracle Rest Data Services



ORACLE

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

Getting Started with Exadata Express

1. Purchase a subscription.
2. Activate and verify the service.
3. Verify activation.
4. Learn about users and roles.
5. Create accounts for your users and assign them appropriate privileges and roles.
6. Set the password for the database user authorized to perform administrative tasks for your service (PDB_ADMIN).

Note: You can refer to Using Oracle Database Exadata Express Cloud Service (<https://docs.oracle.com/cloud/latest/exadataexpress-cloud/CSDBP/toc.htm>) for details on the subscription process.



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

The steps in the slide provide you a brief overview of how to get started with Exadata Express. In the following slides, you will learn in detail how to perform the aforementioned steps.

Oracle Exadata Express Cloud Service

You can refer to Working with Oracle Database Exadata Express Cloud Service (<http://oukc.oracle.com/public/redir.html?type=player&offid=1984115860>) to gain an introduction to the service and its features.



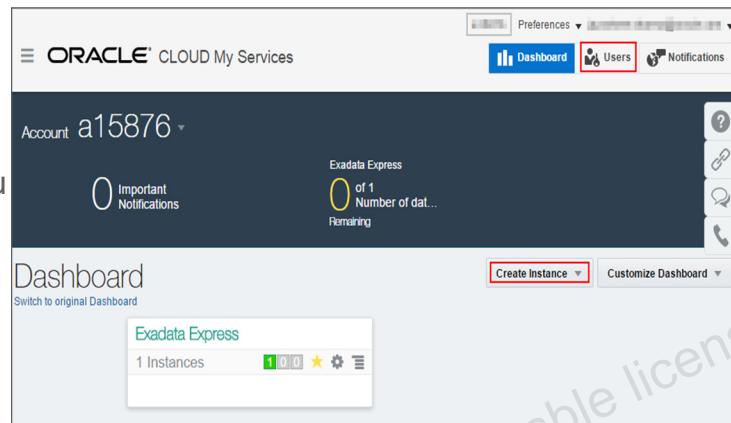
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this video, you gain an introduction to Oracle Exadata Express Cloud Service and its features. You will take a tour of its service console and also learn about the different database clients such as Oracle SQL Developer, SQL CL, SQL Workshop and SQL * Plus that can be used to connect to Oracle Exadata Express Cloud Service.

Note: This demonstration has audio, which cannot be played inside OU Classroom. However participants can access this link on open internet and review at their convenience.

Getting Started with Exadata Express

- On signing into the service, you get access to the dashboard.
- Dashboard allows you to create database instances and users.
- The number of instances you create is limited by the amount of resources you have access to.



ORACLE

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

After successful subscription to the service, you can login to your account and access the dashboard. Based on the type of subscription you can create instances.

The instances would appear on the dashboard. You can see an instance created in the image . To manage the instance, click on the instance.

Managing Exadata

The screenshot shows two main panels. On the left is the 'Service Instances' dashboard for the 'exa4' instance. It displays basic information: Service Type: Exadata Express, Instance Id: 500033811, Status: Active, Size: BASIC, Service SFTP User Name: us148271, and Service SFTP Host & Port: den00rcus.oracle.com. A red box highlights the 'Open Service Console' button. An arrow points from this button to the 'Service Console: exa4' interface on the right. The 'Service Console' interface is titled 'Service Console: exa4'. It contains several sections: 'Web Access' (Develop database and web apps using Oracle Application Express (APEX)), 'Define REST Data Services' (Create and manage RESTful web service interfaces to your database), 'Develop with App Builder' (Declaratively develop and deploy data-driven apps), 'Install Productivity Apps' (Browse and install productivity apps), 'Download Client Credentials' (Download a zip file containing your security credentials and network configuration files), 'Disable Client Access' (Disable SQL*Net access and invalidate all existing client credential files), 'Download Drivers' (Get database drivers for Java, .NET, Node.js, Python, PHP, Ruby, C, C++, Instant Client and more), 'Download Tools' (Get SQL*Plus command-line and developer tools including SQL Developer and JDeveloper), 'Administration' (Manage your cloud database), 'Create Database Schema' (Create a new schema for database objects), 'Set Administrator Password' (Set or reset your database's privileged user (PDB_ADMIN) account password), and 'Create Document Store' (Enable or disable a schema-less document-style interface, with JSON storage and access). Each section includes 'Learn More' and 'Watch Video' links.

ORACLE

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

On clicking the instance on the dashboard, you see various details about the dashboard. You can access the services by clicking on 'Open Service Console'.

The service console provides you access to tools for Web Access, Client Access and Administration.

Service Console

- Service Console is the interface to use and manage the Exadata service
- It provides three different perspectives of the instance
 - Web Access
 - Client Access
 - Administration

The screenshot shows the Service Console interface for an Exadata system. It is divided into three main sections:

- Web Access:** Develop database and web apps using Oracle Application Express (APEX). Includes links to "Go to SQL Workshop", "Define REST Data Services", and "Install Productivity Apps".
- Client Access:** Enable database client access, then connect using drivers and tools. Includes links to "Download Client Credentials", "Disable Client Access", and "Download Drivers".
- Administration:** Manage your cloud database. Includes links to "Create Database Schema", "Set Administrator Password", and "Create Document Store".

ORACLE

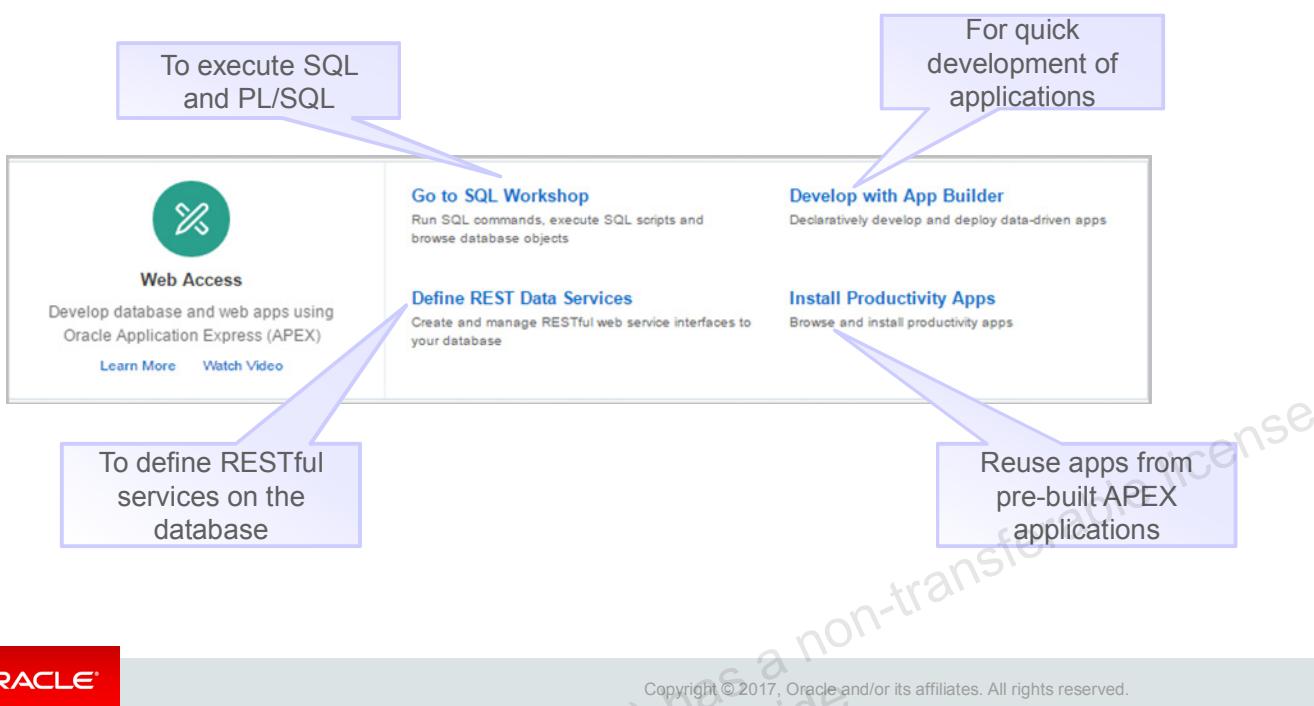
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Web access provides utilities which enable you to develop database and web applications using Oracle Application Express(APEX).

Database clients can connect to Exadata Express service using SQL *Net Access. Some examples of supported database clients are SQLcl, SQL Developer, SQL *Plus, JDBC Thin client,ODP.NET, OCI and Instant Client. Client Access in the Service Console allows you to configure with the client you use.

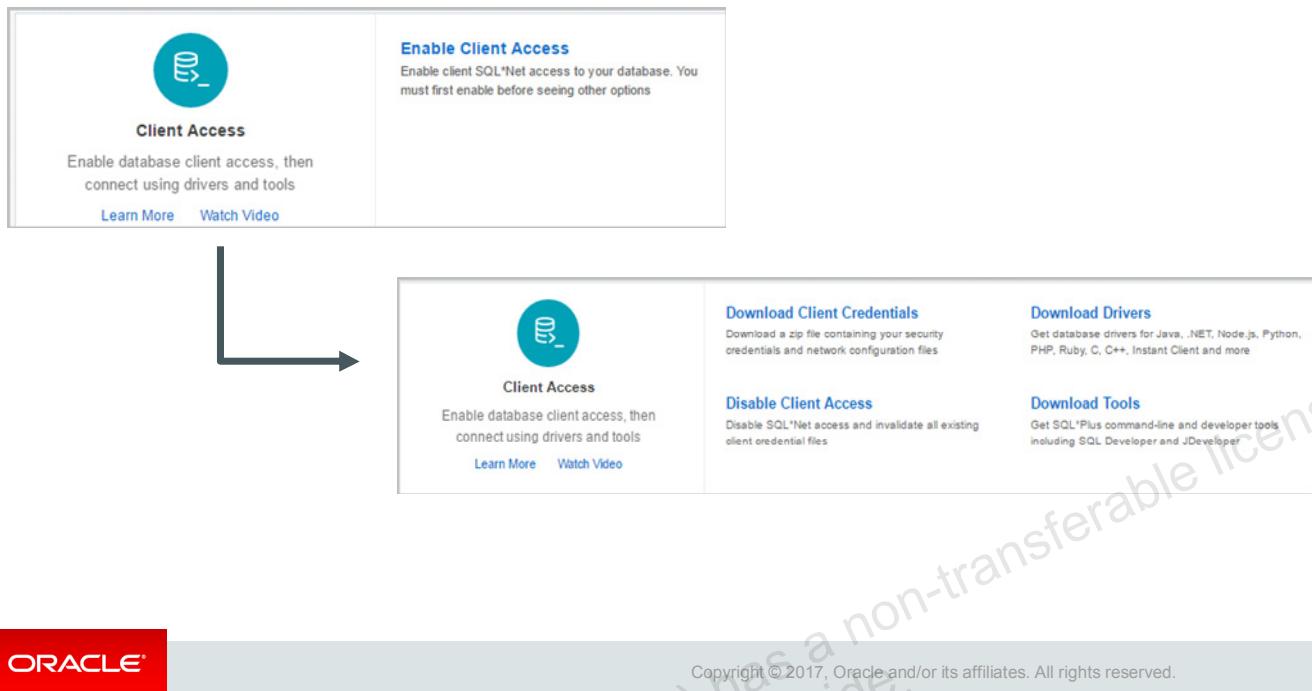
Administration in the Service Console provides for performing administration tasks such as create new database schemas for database objects, set or reset administration password, create a schema-less documents and collections interface, and use administrative options to manage Oracle Application Express.

Web Access through Service Console



Option	Description
Go to SQL Workshop	Allows you go directly to browser-based SQL Workshop, where you can run SQL statements, execute scripts and explore database objects.
Develop with App Builder	Quickly declaratively develop database and websheet applications. You can import files such as database applications and plug-ins. There is a dashboard showing metrics about your applications and workspace utilities to manage defaults, themes, metadata, exports, and more.
Define REST Data Services	Directly access the page to define and manage RESTful web services that view and manipulate data objects within your database.
Install Productivity Apps	Install from a gallery of pre-built Oracle Application Express Productivity Apps.

Client Access Configuration through Service Console



ORACLE

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

You have to enable client access to allow SQL*Net Access to your service. Using SQL *Net Access software you can connect the Exadata instance to different clients. This option is only available when client access has not yet been enabled. Once you enable the client access, four options appear in the console:

Option	Description
Download Client Credentials	Download client credentials needed for clients to access your service.
Download Drivers	Go directly to the Oracle Technology Network page to download and install database drivers including for Java, Instant Client, C, C++, Microsoft .NET, Node.js, Python, PHP, Ruby, and more.
Disable Client Access	Use this option to disallow SQL*Net access to your service. This option is only available when client access has been enabled.
Download tools	Go directly to the Oracle Technology Network page to download and install tools such as SQL*Plus, SQLcl, command-line and integrated development environments such as Oracle SQL Developer, JDeveloper, Oracle JET, and more

Database Administration through Service Console

The screenshot shows the Oracle Database Service Console. On the left, there's a navigation bar with a gear icon labeled "Administration" and a sub-section "Manage your cloud database". Below it are "Learn More" and "Watch Video" links. The main content area has four sections:

- Create Database Schema**: Create a new schema for database objects.
- Set Administrator Password**: Set or reset your database's privileged user (PDB_ADMIN) account password.
- Create Document Store**: Enable or disable a schema-less document-style interface, with JSON storage and access.
- Manage Application Express**: Use Application Express (APEX) administrative options.

Callout boxes provide additional context:

- A box points to "Create Database Schema" with the text "Create a new schema for database objects".
- A box points to "Create Document Store" with the text "To create a document store using a schema".
- A box points to "Set Administrator Password" with the text "Set or reset password for admin".
- A box points to "Manage Application Express" with the text "To manage tasks such as archiving APEX schemas and association among APEX schema".

ORACLE

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

You can perform various administration tasks through 'Administration' in the service console.

Option	Description
Create Database Schema	Create a new schema for database objects. Schema is the set of database objects, such as tables and views that belong to that user account.
Create Document Store	This option enables you to create a document store, using either an existing schema or new schema, and to enable SODA for REST, which enables REST-based operations on the schema using Oracle's SODA for REST API. It also enables SODA for Java, which is Oracle's SODA for Java API for use with Java programs.
Set Administrator Password	Use this option to set the password for the PDB_ADMIN database user that is authorized to perform administrative tasks.
Manage Application Express	Options here allow you to enable application archiving to archive your Oracle Application Express applications to database tables, manage the association between schemas and Oracle Application Express, and manage messages and set preferences for the workspace.

SQL Workshop

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

Web Access

Go to SQL Workshop

Run SQL commands, execute SQL scripts and browse database objects

Define REST Data Services

Create and manage RESTful web service interfaces to your database

Install Productivity Apps

Browse and install productivity apps

Develop with App Builder

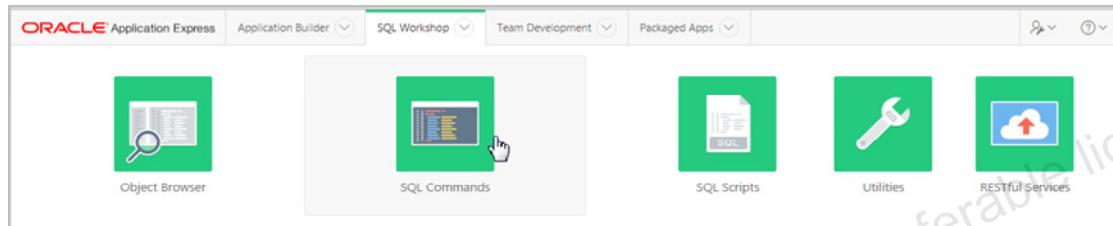
Declaratively develop and deploy data-driven apps

Learn More Watch Video



1

Clicking on SQL workshop will lead you to APEX interface



2

To run SQL or PL/SQL you can use the SQL commands utility

ORACLE

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

SQL Workshop

You can run SQL statements in the editor.

The screenshot shows the Oracle Application Express SQL Workshop interface. At the top, there are tabs for Application Builder, SQL Workshop (which is selected), Team Development, and Packaged Apps. Below the tabs, a toolbar includes a magnifying glass icon, a help icon, and a user icon. The main area is titled "SQL Commands" and shows a schema dropdown set to "BZJNMKSSA". A "Rows" dropdown is set to 10, and buttons for "Clear Command" and "Find Tables" are visible. A "Save" button and a "Run" button are also present. The SQL command entered is "SELECT * FROM emp;". The results tab is selected, displaying a table with the following data:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	-	11/17/1981	5000	-	10
7698	BLAKE	MANAGER	7839	05/01/1981	2850	-	30
7782	CLARK	MANAGER	7839	06/09/1981	2450	-	10
7566	JONES	MANAGER	7839	04/02/1981	2975	-	20
7788	SCOTT	ANALYST	7566	12/09/1982	3000	-	20
7902	FORD	ANALYST	7566	12/03/1981	3000	-	20

Below the results table, there are tabs for Explain, Describe, Saved SQL, and History. The Oracle logo is in the bottom left corner, and a copyright notice "Copyright © 2017, Oracle and/or its affiliates. All rights reserved." is in the bottom right corner.

Connecting through Database Clients

You can connect to Exadata Express through various database clients.

Some of the database clients include:

- SQL*Plus
- SQLcl
- SQL Developer
- .Net and Visual Studio
- JDBC Thin Client



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

You must first configure database clients and Oracle Database Exadata Express Cloud Service to communicate with each other.

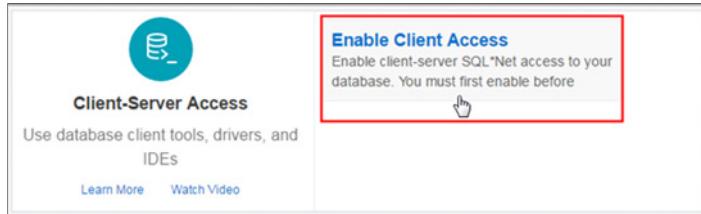
Prerequisite tasks for database client connectivity require you to:

- Enable SQL*Net access to your service.
- Download client credentials.
- Follow set-up instructions for the specific database client you want to connect with.

In the following topics, you will learn how to enable SQL*Net access, download client credentials and make connections using Oracle SQL Developer and SQLcl.

Enabling SQL*Net Access for Client Applications

Enable SQL*Net Access in the Service Console to obtain the various Database Client options.



ORACLE

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

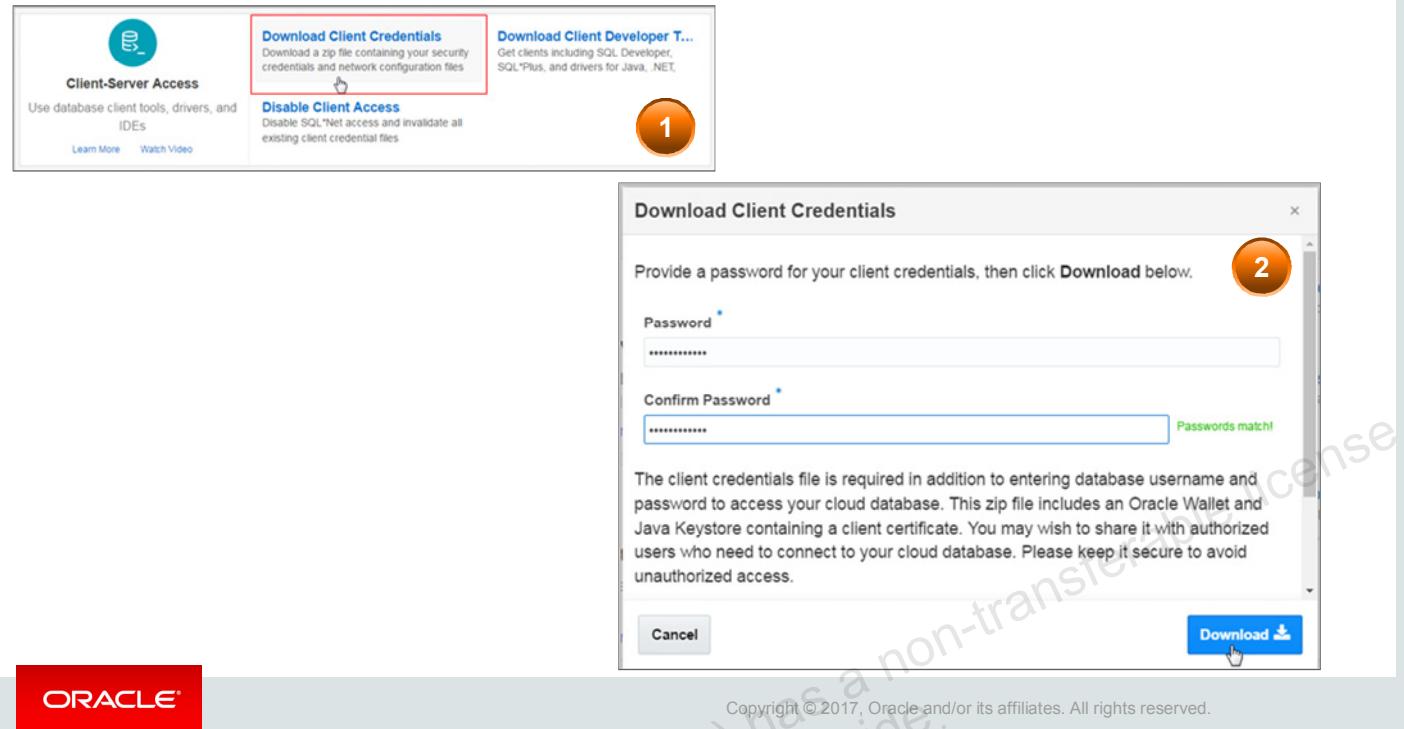
You can connect to your Exadata Express cloud service from diverse database clients over SQL*Net also called as Oracle Database Net Services. Some examples of supported clients include SQL*Plus, SQLcl, SQL Developer, JDBC Thin, ODP.NET, OCI, and Instant Client.

Database drivers for all popular programming and scripting languages such as Python, PHP, Node.js, C/C++, Ruby and Perl are supported. SQL*Net access has to be enabled as a prerequisite for all clients and drivers connecting over SQL*Net.

The Service Administrator must do the following to enable SQL*Net:

- Navigate to the Service Console for **Exadata Express** and open the service console.
- Click **Enable Client Access**.
- Download the client credentials.
- Now, depending on the client-side application and driver being used, you need to configure the application connection string for that application.

Downloading Client Credentials



You can easily download the zip files for client credentials from service console. Its contents include Oracle Wallet and Java Keystore as well as essential client configuration files. While downloading the zip file, you are prompted to enter a password.

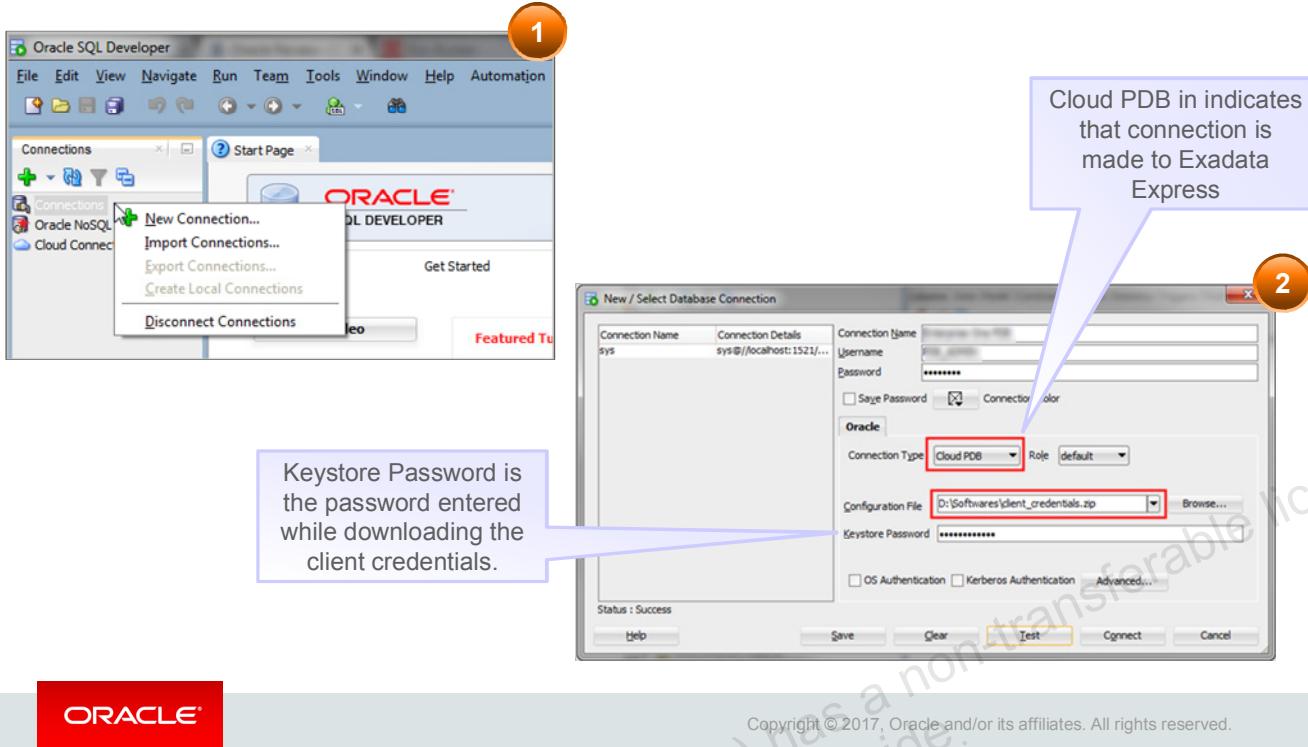
Note that client credential zip files should be carefully managed. Please remember to keep the file secure to avoid unauthorized database access. If you believe the security of this file has been compromised, then immediately disable client access using the cloud service console.

The steps to download the client credentials are as follows.

1. Navigate to Exadata Express and open the service console.
2. Click Download Client Credentials to download a zip file containing your security credentials and network configuration files.
3. Enter a password to create a password-protected Oracle Wallet and Java Keystore files for the service.
4. Click Download and save the downloaded zip file to a secure location that is accessible by your database client(s).

In the following topics, you will learn how to use these credentials to connect to the cloud database.

Connecting Oracle SQL Developer



In order to connect to Oracle Database Exadata Express cloud service, you need to download and install Oracle SQL Developer Release 4.1.5 or later. You should have also downloaded the Client Credentials from Oracle Exadata Express service console. You should then configure an Oracle Cloud connection in the Oracle SQL Developer.

The connection can be created as follows:

1. Run Oracle SQL Developer locally.
2. Under Connections, right click Connections and select **New Connection**.
3. Enter the following details:
 - Connection Name – Enter a name for this cloud connection.
 - Username – Enter username required to sign into Exadata Express.
 - Password – Enter password required to sign into Exadata Express.
 - Connection type – Select **Cloud PDB**.
 - Configuration File - Click Browse and select the **Client Credentials** zip file that you previously downloaded from the Exadata Express service console.
 - Keystore Password – Enter the password provided while downloading the Client Credentials from the Exadata Express service console.
4. Click Test. If the status is Success, click Connect.

If you have connected successfully, the tables and other objects from Exadata Express display under the new connection.

Connecting Oracle SQLcl

```
D:\PDB Service\SQL CL\sqlcl-no-jre-latest\sqlcl\bin>sql /nolog  
  
SQLcl: Release 4.2.0.16.160.2007 RC on Thu Sep 08 12:18:07 2016  
  
Copyright (c) 1982, 2016, Oracle. All rights reserved.  
SQL>
```

1

```
SQL> set cloudconfig client_credentials.zip  
Wallet Password: *****  
Using temp directory:C:\Users\APOTHU~1.ORA\AppData\Local\Temp\  
oracle_cloud_config6707346342028726502
```

2

```
SQL> conn pdb_admin/welcome1@dbaccess  
Connected.  
SQL>
```

3



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

Summary

In this lesson, you should have learned about:

- The salient features of Oracle Cloud
- Oracle Database Exadata Express Cloud Service



ORACLE

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

Relational database management systems are composed of objects or relations. They are managed by operations and governed by data integrity constraints.

Oracle Corporation produces products and services to meet your RDBMS needs. The main products are the following:

- Oracle Database, which you use to store and manage information by using SQL
- Oracle Fusion Middleware, which you use to develop, deploy, and manage modular business services that can be integrated and reused
- Oracle Enterprise Manager Grid Control, which you use to manage and automate administrative tasks across sets of systems in a grid environment

SQL

The Oracle server supports ANSI-standard SQL and contains extensions. SQL is the language that is used to communicate with the server to access, manipulate, and control data.

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

GANG LIU (gangl@baylorhealth.edu) has a non-transferable license
to use this Student Guide.



A

Commonly Used SQL Commands

ORACLE®

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

Objectives

After completing this appendix, you should be able to:

- Execute a basic SELECT statement
- Create, alter, and drop a table using the data definition language (DDL) statements
- Insert, update, and delete rows from one or more tables using data manipulation language (DML) statements
- Commit, roll back, and create save points using the transaction control statements
- Perform join operations on one or more tables



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

Basic SELECT Statement

- Use the `SELECT` statement to:
 - Identify the columns to be displayed
 - Retrieve data from one or more tables, object tables, views, object views, or materialized views
- A `SELECT` statement is also known as a query because it queries a database.
- Syntax:

```
SELECT { * | [DISTINCT] column|expression [alias] , ... }  
FROM      table;
```



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

In its simplest form, a `SELECT` statement must include the following:

- A `SELECT` clause, which specifies the columns to be displayed
- A `FROM` clause, which identifies the table containing the columns that are listed in the `SELECT` clause

In the syntax:

<code>SELECT</code>	Is a list of one or more columns
<code>*</code>	Selects all columns
<code>DISTINCT</code>	Suppresses duplicates
<code>column expression</code>	Selects the named column or the expression
<code>alias</code>	Gives different headings to the selected columns
<code>FROM table</code>	Specifies the table containing the columns

Note: Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- A *keyword* refers to an individual SQL element (for example, `SELECT` and `FROM` are keywords).
- A *clause* is a part of a SQL statement (for example, `SELECT employee_id, last_name`).
- A *statement* is a combination of two or more clauses (for example, `SELECT * FROM employees`).

SELECT Statement

- Select all columns:

```
SELECT *  
FROM job_history;
```

	EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
1	102	13-JAN-01	24-JUL-06	IT_PROG	60
2	101	21-SEP-97	27-OCT-01	AC_ACCOUNT	110
3	101	28-OCT-01	15-MAR-05	AC_MGR	110
4	201	17-FEB-04	19-DEC-07	MK_REP	20
5	114	24-MAR-06	31-DEC-07	ST_CLERK	50
6	122	01-JAN-07	31-DEC-07	ST_CLERK	50
7	200	17-SEP-95	17-JUN-01	AD_ASST	90
8	176	24-MAR-06	31-DEC-06	SA REP	80
9	176	01-JAN-07	31-DEC-07	SA_MAN	80
10	200	01-JUL-02	31-DEC-06	AC_ACCOUNT	90

- Select specific columns:

```
SELECT manager_id, job_id  
FROM employees;
```

	MANAGER_ID	JOB_ID
1	(null)	AD_PRES
2	100	AD_VP
3	100	AD_VP
4	102	IT_PROG
5	103	IT_PROG
6	103	IT_PROG
7	100	ST_MAN
8	124	ST_CLERK
9	124	ST_CLERK
10	124	ST_CLERK



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

You can display all columns of data in a table by following the `SELECT` keyword with an asterisk (*) or by listing all the column names after the `SELECT` keyword. The first example in the slide displays all the rows from the `job_history` table. Specific columns of the table can be displayed by specifying the column names, separated by commas. The second example in the slide displays the `manager_id` and `job_id` columns from the `employees` table.

In the `SELECT` clause, specify the columns in the order in which you want them to appear in the output. For example, the following SQL statement displays the `location_id` column before displaying the `department_id` column:

```
SELECT location_id, department_id FROM departments;
```

Note: You can enter your SQL statement in a SQL Worksheet and click the Run Statement icon or press F9 to execute a statement in SQL Developer. The output displayed on the Results tabbed page appears as shown in the slide.

WHERE Clause

- Use the optional WHERE clause to:
 - Filter rows in a query
 - Produce a subset of rows
- Syntax:

```
SELECT * FROM table  
[WHERE condition];
```

- Example:

```
SELECT location_id from departments  
WHERE department_name = 'Marketing';
```



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

The WHERE clause specifies a condition to filter rows, producing a subset of the rows in the table. A condition specifies a combination of one or more expressions and logical (Boolean) operators. It returns a value of TRUE, FALSE, or NULL. The example in the slide retrieves the location_id of the marketing department.

The WHERE clause can also be used to update or delete data from the database.

For example:

```
UPDATE departments  
SET department_name = 'Administration'  
WHERE department_id = 20;  
and  
DELETE from departments  
WHERE department_id =20;
```

ORDER BY Clause

- Use the optional ORDER BY clause to specify the row order.
- Syntax:

```
SELECT * FROM table  
[WHERE condition]  
[ORDER BY {<column>}|<position> } [ASC|DESC] [, ...] ;
```

- Example:

```
SELECT last_name, department_id, salary  
FROM employees  
ORDER BY department_id ASC, salary DESC;
```



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

GROUP BY Clause

- Use the optional GROUP BY clause to group columns that have matching values into subsets.
- No group has two rows having the same value for the grouping column or columns.
- Syntax:

```
SELECT <column1, column2, ... column_n>
  FROM table
 [WHERE condition]
 [GROUP BY <column> [, ...] ]
 [ORDER BY <column> [, ...] ] ;
```

- Example:

```
SELECT department_id, MIN(salary), MAX (salary)
  FROM employees
 GROUP BY department_id ;
```



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

The GROUP BY clause is used to group selected rows based on the value of `expr(s)` for each row. The clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

Any SELECT list elements that are not included in aggregation functions must be included in the GROUP BY list of elements. This includes both columns and expressions. The database returns a single row of summary information for each group.

The example in the slide returns the minimum and maximum salaries for each department in the employees table.

Data Definition Language

- DDL statements are used to define, structurally change, and drop schema objects.
- The commonly used DDL statements are:
 - CREATE TABLE, ALTER TABLE, and DROP TABLE
 - GRANT, REVOKE
 - TRUNCATE



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

Data definition language (DDL) statements enable you to alter the attributes of an object without altering the applications that access the object. You can also use DDL statements to alter the structure of objects while database users are performing work in the database. These statements are most frequently used to:

- Create, alter, and drop schema objects and other database structures, including the database itself and database users
- Delete all the data in schema objects without removing the structure of these objects
- Grant and revoke privileges and roles

Oracle Database implicitly commits the current transaction before and after every DDL statement.

CREATE TABLE Statement

- Use the CREATE TABLE statement to create a table in the database.
- Syntax:

```
CREATE TABLE tablename (
  {column-definition | Table-level constraint}
  [ , {column-definition | Table-level constraint} ] * )
```

- Example:

```
CREATE TABLE teach_dept (
  department_id NUMBER(3) PRIMARY KEY,
  department_name VARCHAR2(10));
```



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

Use the CREATE TABLE statement to create a table in the database. To create a table, you must have the CREATE TABLE privilege and a storage area in which to create objects.

The table owner and the database owner automatically gain the following privileges on the table after it is created:

- INSERT
- SELECT
- REFERENCES
- ALTER
- UPDATE

The table owner and the database owner can grant the preceding privileges to other users.

ALTER TABLE Statement

- Use the ALTER TABLE statement to modify the definition of an existing table in the database.
- Example 1:

```
ALTER TABLE teach_dept  
ADD location_id NUMBER NOT NULL;
```

- Example 2:

```
ALTER TABLE teach_dept  
MODIFY department_name VARCHAR2(30) NOT NULL;
```



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

The ALTER TABLE statement allows you to make changes to an existing table.

You can:

- Add a column to a table
- Add a constraint to a table
- Modify an existing column definition
- Drop a column from a table
- Drop an existing constraint from a table
- Increase the width of the VARCHAR and CHAR columns
- Change a table to have read-only status

Example 1 in the slide adds a new column called location_id to the teach_dept table.

Example 2 updates the existing department_name column from VARCHAR2 (10) to VARCHAR2 (30), and adds a NOT NULL constraint to it.

DROP TABLE Statement

- The DROP TABLE statement removes the table and all its data from the database.
- Example:

```
DROP TABLE teach_dept;
```

- DROP TABLE with the PURGE clause drops the table and releases the space that is associated with it.

```
DROP TABLE teach_dept PURGE;
```

- The CASCADE CONSTRAINTS clause drops all referential integrity constraints from the table.

```
DROP TABLE teach_dept CASCADE CONSTRAINTS;
```



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

The `DROP TABLE` statement allows you to remove a table and its contents from the database, and pushes it to the recycle bin. Dropping a table invalidates dependent objects and removes object privileges on the table.

Use the `PURGE` clause along with the `DROP TABLE` statement to release back to the tablespace the space allocated for the table. You cannot roll back a `DROP TABLE` statement with the `PURGE` clause, nor can you recover the table if you have dropped it with the `PURGE` clause.

The `CASCADE CONSTRAINTS` clause allows you to drop the reference to the primary key and unique keys in the dropped table.

GRANT Statement

- The GRANT statement assigns privileges to perform the following operations:
 - Insert or delete data
 - Create a foreign key reference to the named table or to a subset of columns from a table
 - Select data, a view, or a subset of columns from a table
 - Create a trigger on a table
 - Execute a specified function or procedure
- Example:

```
GRANT SELECT any table to PUBLIC;
```



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

You can use the GRANT statement to:

- Assign privileges to a specific user or role, or to all users, to perform actions on database objects
- Grant a role to a user, to PUBLIC, or to another role

Before you issue a GRANT statement, check that the `derby.database.sqlAuthorization` property is set to True. This property enables the SQL Authorization mode. You can grant privileges on an object if you are the owner of the database.

You can grant privileges to all users by using the PUBLIC keyword. When PUBLIC is specified, the privileges or roles affect all current and future users.

Privilege Types

Assign the following privileges by using the GRANT statement:

- ALL PRIVILEGES
- DELETE
- INSERT
- REFERENCES
- SELECT
- UPDATE



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

REVOKE Statement

- Use the REVOKE statement to remove privileges from a user to perform actions on database objects.
- Revoke a *system privilege* from a user:

```
REVOKE DROP ANY TABLE  
  FROM hr;
```

- Revoke a *role* from a user:

```
REVOKE dw_manager  
  FROM sh;
```



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

The REVOKE statement removes privileges from a specific user (or users) or role to perform actions on database objects. It performs the following operations:

- Revokes a role from a user, from PUBLIC, or from another role
- Revokes privileges for an object if you are the owner of the object or the database owner

Note: To revoke a role or system privilege, you must have been granted the privilege with the ADMIN OPTION.

TRUNCATE TABLE Statement

- Use the TRUNCATE TABLE statement to remove all the rows from a table.
- Example:

```
TRUNCATE TABLE employees_demo;
```

- By default, Oracle Database performs the following tasks:
 - Deallocates space used by the removed rows
 - Sets the NEXT storage parameter to the size of the last extent removed from the segment by the truncation process



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

The TRUNCATE TABLE statement deletes all the rows from a specific table. Removing rows with the TRUNCATE TABLE statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table:

- Invalidates the dependent objects of the table
- Requires you to:
 - Regrant object privileges
 - Re-create indexes, integrity constraints, and triggers.
 - Respecify its storage parameters

The TRUNCATE TABLE statement spares you from these efforts.

Note: You cannot roll back a TRUNCATE TABLE statement.

Data Manipulation Language

- DML statements query or manipulate data in existing schema objects.
- A DML statement is executed when:
 - New rows are added to a table by using the `INSERT` statement
 - Existing rows in a table are modified using the `UPDATE` statement
 - Existing rows are deleted from a table by using the `DELETE` statement
- A *transaction* consists of a collection of DML statements that form a logical unit of work.



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

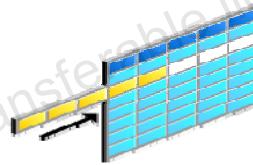
INSERT Statement

- Use the `INSERT` statement to add new rows to a table.
- Syntax:

```
INSERT INTO table [(column [, column...])]  
VALUES (value [, value...]);
```

- Example:

```
INSERT INTO departments  
VALUES (200, 'Development', 104, 1400);  
1 rows inserted.
```



ORACLE

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

The `INSERT` statement adds rows to a table. Be sure to insert a new row containing values for each column and to list the values in the default order of the columns in the table. Optionally, you can also list the columns in the `INSERT` statement.

For example:

```
INSERT INTO job_history (employee_id, start_date, end_date,  
job_id)  
VALUES (120, '25-JUL-06', '12-FEB-08', 'AC_ACCOUNT');
```

The syntax discussed in the slide allows you to insert a single row at a time. The `VALUES` keyword assigns the values of expressions to the corresponding columns in the column list.

UPDATE Statement Syntax

- Use the UPDATE statement to modify the existing rows in a table.
- Update more than one row at a time (if required).

```
UPDATE      table  
SET         column = value [, column = value, ...]  
[WHERE      condition];
```

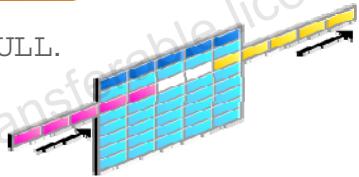
- Example:

```
UPDATE      copy_emp  
SET  
[22 rows updated]
```

- Specify SET *column_name*= NULL to update a column value to NULL.



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



The UPDATE statement modifies the existing values in a table. Confirm the update operation by querying the table to display the updated rows. You can modify a specific row or rows by specifying the WHERE clause.

For example:

```
UPDATE employees  
SET     salary = 17500  
WHERE   employee_id = 102;
```

In general, use the primary key column in the WHERE clause to identify the row to update. For example, to update a specific row in the employees table, use employee_id instead of employee_name, to identify the row, because more than one employee may have the same name.

Note: Typically, the condition keyword is composed of column names, expressions, constants, subqueries, and comparison operators.

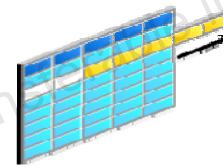
DELETE Statement

- Use the DELETE statement to delete the existing rows from a table.
- Syntax:

```
DELETE [FROM] table  
[WHERE condition];
```

- Write the DELETE statement using the WHERE clause to delete specific rows from a table.

```
DELETE FROM departments  
WHERE department_name = 'Finance';  
1 rows deleted
```



ORACLE

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

The DELETE statement removes existing rows from a table. You must use the WHERE clause to delete a specific row or rows from a table based on a condition. The condition identifies the rows to be deleted. It may contain column names, expressions, constants, subqueries, and comparison operators.

The first example in the slide deletes the finance department from the departments table. You can confirm the delete operation by using the SELECT statement to query the table.

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

If you omit the WHERE clause, all rows in the table are deleted. For example:

```
DELETE FROM copy_emp;
```

The preceding example deletes all the rows from the copy_emp table.

Transaction Control Statements

- Transaction control statements are used to manage the changes made by DML statements.
- The DML statements are grouped into transactions.
- Transaction control statements include:
 - COMMIT
 - ROLLBACK
 - SAVEPOINT

ORACLE

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

COMMIT Statement

- Use the COMMIT statement to:
 - Permanently save the changes made to the database during the current transaction
 - Erase all savepoints in the transaction
 - Release transaction locks
- Example:

```
INSERT INTO departments
VALUES      (201, 'Engineering', 106, 1400);
COMMIT;

1 rows inserted.
committed.
```



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

The COMMIT statement ends the current transaction by making all the pending data changes permanent. It releases all row and table locks, and erases any savepoints that you may have marked since the last commit or rollback. The changes made using the COMMIT statement are visible to all users.

Oracle recommends that you explicitly end every transaction in your application programs with a COMMIT or ROLLBACK statement, including the last transaction, before disconnecting from Oracle Database. If you do not explicitly commit the transaction and the program terminates abnormally, the last uncommitted transaction is automatically rolled back.

Note: Oracle Database issues an implicit COMMIT before and after any data definition language (DDL) statement.

ROLLBACK Statement

- Use the ROLLBACK statement to undo changes made to the database during the current transaction.
- Use the TO SAVEPOINT clause to undo a part of the transaction after the savepoint.
- Example:

```
UPDATE      employees
SET         salary = 7000
WHERE       last_name = 'Ernst';
SAVEPOINT   Ernst_sal;

UPDATE      employees
SET         salary = 12000
WHERE       last_name = 'Mourgos';

ROLLBACK TO SAVEPOINT Ernst_sal;
```



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

The ROLLBACK statement undoes work done in the current transaction. To roll back the current transaction, no privileges are necessary.

Using ROLLBACK with the TO SAVEPOINT clause performs the following operations:

- Rolls back only the portion of the transaction after the savepoint
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times.

Using ROLLBACK without the TO SAVEPOINT clause performs the following operations:

- Ends the transaction
- Undoes all the changes in the current transaction
- Erases all savepoints in the transaction

SAVEPOINT Statement

- Use the SAVEPOINT statement to name and mark the current point in the processing of a transaction.
- Specify a name to each savepoint.
- Use distinct savepoint names within a transaction to avoid overriding.
- Syntax:

```
SAVEPOINT savepoint;
```



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

The SAVEPOINT statement identifies a point in a transaction to which you can later roll back. You must specify a distinct name for each savepoint. If you create a second savepoint with the same identifier as an earlier savepoint, the earlier savepoint is erased.

After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

A simple rollback or commit erases all savepoints. When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you have rolled back is retained.

When savepoint names are reused within a transaction, the Oracle Database moves (overrides) the save point from its old position to the current point in the transaction.

Joins

Use a join to query data from more than one table:

```
SELECT    table1.column, table2.column  
FROM      table1, table2  
WHERE     table1.column1 = table2.column2;
```

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.



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

When data from more than one table in the database is required, a *join* condition is used. Rows in one table can be joined to rows in another table according to common values that exist in the corresponding columns (usually primary and foreign key columns).

To display data from two or more related tables, write a simple join condition in the WHERE clause.

In the syntax:

table1.column

Denotes the table and column from which data is retrieved

table1.column1 =

Is the condition that joins (or relates) the tables together

table2.column2

Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join n tables together, you need a minimum of $n-1$ join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

Types of Joins

- Natural join
- Equijoin
- Nonequijoin
- Outer join
- Self-join
- Cross join



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

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Use table aliases, instead of full table name prefixes.
- Table aliases give a table a shorter name.
 - This keeps SQL code smaller and uses less memory.
- Use column aliases to distinguish columns that have identical names, but reside in different tables.



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

Natural Join

- The NATURAL JOIN clause is based on all the columns in the two tables that have the same name.
- It selects rows from tables that have the same names and data values of columns.
- Example:

```
SELECT country_id, location_id, country_name, city  
FROM countries NATURAL JOIN locations;
```

	COUNTRY_ID	LOCATION_ID	COUNTRY_NAME	CITY
1	US	1400	United States of America	Southlake
2	US	1500	United States of America	San Francisco
3	US	1700	United States of America	Seattle
4	CA	1800	Canada	Toronto
5	UK	2500	United Kingdom	Oxford

ORACLE

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

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the NATURAL JOIN keywords.

Note: The join can happen only on those columns that have the same names and data types in both tables. If the columns have the same name but different data types, the NATURAL JOIN syntax causes an error.

In the example in the slide, the COUNTRIES table is joined to the LOCATIONS table by the COUNTRY_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

Equijoin

EMPLOYEES

	EMPLOYEE_ID	DEPARTMENT_ID
1	200	10
2	201	20
3	202	20
4	205	110
5	206	110
6	100	90
7	101	90
8	102	90
9	103	60
10	104	60

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

Foreign key

Primary key



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

Retrieving Records by Using Equijoins

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
  FROM employees e JOIN departments d
 WHERE e.department_id = d.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	Whalen	10	10	1700
2	Hartstein	20	20	1800
3	Fay	20	20	1800
4	Vargas	50	50	1500
5	Matos	50	50	1500
6	Davies	50	50	1500
7	Rajs	50	50	1500
8	Mourgos	50	50	1500
9	Hunold	60	60	1400
10	Ernst	60	60	1400
11	Lorentz	60	60	1400
...				



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

In the example in the slide:

- **The SELECT clause specifies the column names to retrieve:**
 - Employee last name, employee ID, and department ID, which are columns in the EMPLOYEES table
 - Department ID and location ID, which are columns in the DEPARTMENTS table
- **The FROM clause specifies the two tables that the database must access:**
 - EMPLOYEES table
 - DEPARTMENTS table
- **The WHERE clause specifies how the tables are to be joined:**

e.department_id = d.department_id

Because the DEPARTMENT_ID column is common to both tables, it must be prefixed with the table alias to avoid ambiguity. Other columns that are not present in both the tables need not be qualified by a table alias, but it is recommended for better performance.

Note: When you use the Execute Statement icon to run the query, SQL Developer suffixes a “_1” to differentiate between the two DEPARTMENT_IDS.

Adding Search Conditions by Using the AND and WHERE Operators

```
SELECT d.department_id, d.department_name, l.city
FROM departments d JOIN locations l
ON d.location_id = l.location_id
AND d.department_id IN (20, 50);
```

	DEPARTMENT_ID	DEPARTMENT_NAME	CITY
1	20	Marketing	Toronto
2	50	Shipping	South San Francisco

```
SELECT d.department_id, d.department_name, l.city
FROM departments d JOIN locations l
ON d.location_id = l.location_id
WHERE d.department_id IN (20, 50);
```



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

In addition to the join, you may have criteria for your WHERE clause to restrict the rows in consideration for one or more tables in the join. The example in the slide performs a join on the DEPARTMENTS and LOCATIONS tables and, in addition, displays only those departments with ID equal to 20 or 50. To add more conditions to the ON clause, you can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions.

Both queries produce the same output.

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Matos	50	Shipping

Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level  
FROM employees e JOIN job_grades j  
ON e.salary  
    BETWEEN j.lowest_sal AND j.highest_sal;
```

#	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Fay	6000	C

...



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

The example in the slide creates a nonequijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the job grade table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits that are provided by the job grade table. That is, no employee earns less than the lowest value contained in the `LOWEST_SAL` column or more than the highest value contained in the `HIGHEST_SAL` column.

Note: Other conditions (such as `<=` and `>=`) can be used, but `BETWEEN` is the simplest. Remember to specify the low value first and the high value last when using the `BETWEEN` condition. The Oracle server translates the `BETWEEN` condition to a pair of `AND` conditions. Therefore, using `BETWEEN` has no performance benefits, but should be used only for logical simplicity.

Table aliases have been specified in the example in the slide for performance reasons, not because of possible ambiguity.

Retrieving Records by Using the USING Clause

- You can use the USING clause to match only one column when more than one column matches.
- You cannot specify this clause with a NATURAL join.
- Do not qualify the column name with a table name or table alias.
- Example:

```
SELECT country_id, country_name, location_id, city
  FROM countries JOIN locations
 USING (country_id) ;
```

#	COUNTRY_ID	COUNTRY_NAME	#	LOCATION_ID	CITY
1	US	United States of America		1400	Southlake
2	US	United States of America		1500	South San Francisco
3	US	United States of America		1700	Seattle
4	CA	Canada		1800	Toronto
5	UK	United Kingdom		2500	Oxford



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

Retrieving Records by Using the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the `ON` clause to specify arbitrary conditions or specify columns to join.
- The `ON` clause makes code easy to understand.

```
SELECT e.employee_id, e.last_name, j.department_id,  
      FROM employees e JOIN job history j  
     ON      (e.employee_id = j.employee_id);
```

#	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	101	Kochhar	110
2	101	Kochhar	110
3	102	De Haan	60
4	176	Taylor	80
5	176	Taylor	80
6	200	Whalen	90
7	200	Whalen	90
8	201	Hartstein	20

ORACLE®

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

Use the `ON` clause to specify a join condition. With this, you can specify join conditions separate from any search or filter conditions in the `WHERE` clause.

In this example, the `EMPLOYEE_ID` columns in the `EMPLOYEES` and `JOB_HISTORY` tables are joined using the `ON` clause. Wherever an employee ID in the `EMPLOYEES` table equals an employee ID in the `JOB_HISTORY` table, the row is returned. The table alias is necessary to qualify the matching column names.

You can also use the `ON` clause to join columns that have different names. The parentheses around the joined columns, as in the example in the slide, `(e.employee_id = j.employee_id)`, is optional. So, even `ON e.employee_id = j.employee_id` will work.

Note: When you use the Execute Statement icon to run the query, SQL Developer suffixes a '`_1`' to differentiate between the two `employee_ids`.

Left Outer Join

- A join between two tables that returns all matched rows, as well as the unmatched rows from the left table is called a LEFT OUTER JOIN.
- Example:

```
SELECT c.country_id, c.country_name, l.location_id, l.city
FROM   countries c LEFT OUTER JOIN locations l
ON    (c.country_id = l.country_id) ;
```

COUNTRY_ID	COUNTRY_NAME	LOCATION_ID	CITY
1 CA	Canada	1800	Toronto
2 DE	Germany	(null)	(null)
3 UK	United Kingdom	2500	Oxford
4 US	United States of America	1400	Southlake
5 US	United States of America	1500	South San Francisco
6 US	United States of America	1700	Seattle



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

Right Outer Join

- A join between two tables that returns all matched rows, as well as the unmatched rows from the right table is called a **RIGHT OUTER JOIN**.
- Example:

```
SELECT e.last_name, d.department_id, d.department_name
  FROM employees e RIGHT OUTER JOIN departments d
  ON (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Davies	50	Shipping
...			
18	Higgins	110	Accounting
19	Gietz	110	Accounting
20	(null)	190	Contracting



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

Full Outer Join

- A join between two tables that returns all matched rows, as well as the unmatched rows from both tables is called a FULL OUTER JOIN.
- Example:

```
SELECT e.last_name, d.department_id, d.manager_id,
       d.department_name
  FROM employees e FULL OUTER JOIN departments d
 WHERE (e.manager_id = d.manager_id) ;
```

#	LAST_NAME	DEPARTMENT_ID	MANAGER_ID	DEPARTMENT_NAME
1	King	(null)	(null)	(null)
2	Kochhar	90	100	Executive
3	De Haan	90	100	Executive
4	Hunold	(null)	(null)	(null)

...

19	Higgins	(null)	(null)	(null)
20	Gietz	110	205	Accounting
21	(null)	190	(null)	Contracting
22	(null)	10	200	Administration



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

Self-Join: Example

```
SELECT worker.last_name || ' works for '
    || manager.last_name
FROM   employees worker JOIN employees manager
ON   worker.manager_id = manager.employee_id
ORDER BY worker.last_name;
```

WORKER.LAST_NAME 'WORKSFOR' MANAGER.LAST_NAME
1 Abel works for Zlotkey
2 Davies works for Mourgos
3 De Haan works for King
4 Ernst works for Hunold
5 Fay works for Hartstein
6 Gietz works for Higgins
7 Grant works for Zlotkey
8 Hartstein works for King
9 Higgins works for Kochhar

...

ORACLE

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

Cross Join

- A CROSS JOIN is a JOIN operation that produces the Cartesian product of two tables.
- Example:

```
SELECT department_name, city
FROM department CROSS JOIN location;
```

	DEPARTMENT_NAME	CITY
1	Administration	Oxford
2	Administration	Seattle
3	Administration	South San Francisco
4	Administration	Southlake
5	Administration	Toronto
6	Marketing	Oxford
7	Marketing	Seattle
8	Marketing	South San Francisco
9	Marketing	Southlake
10	Marketing	Toronto

ORACLE

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

Summary

In this appendix, you should have learned how to use:

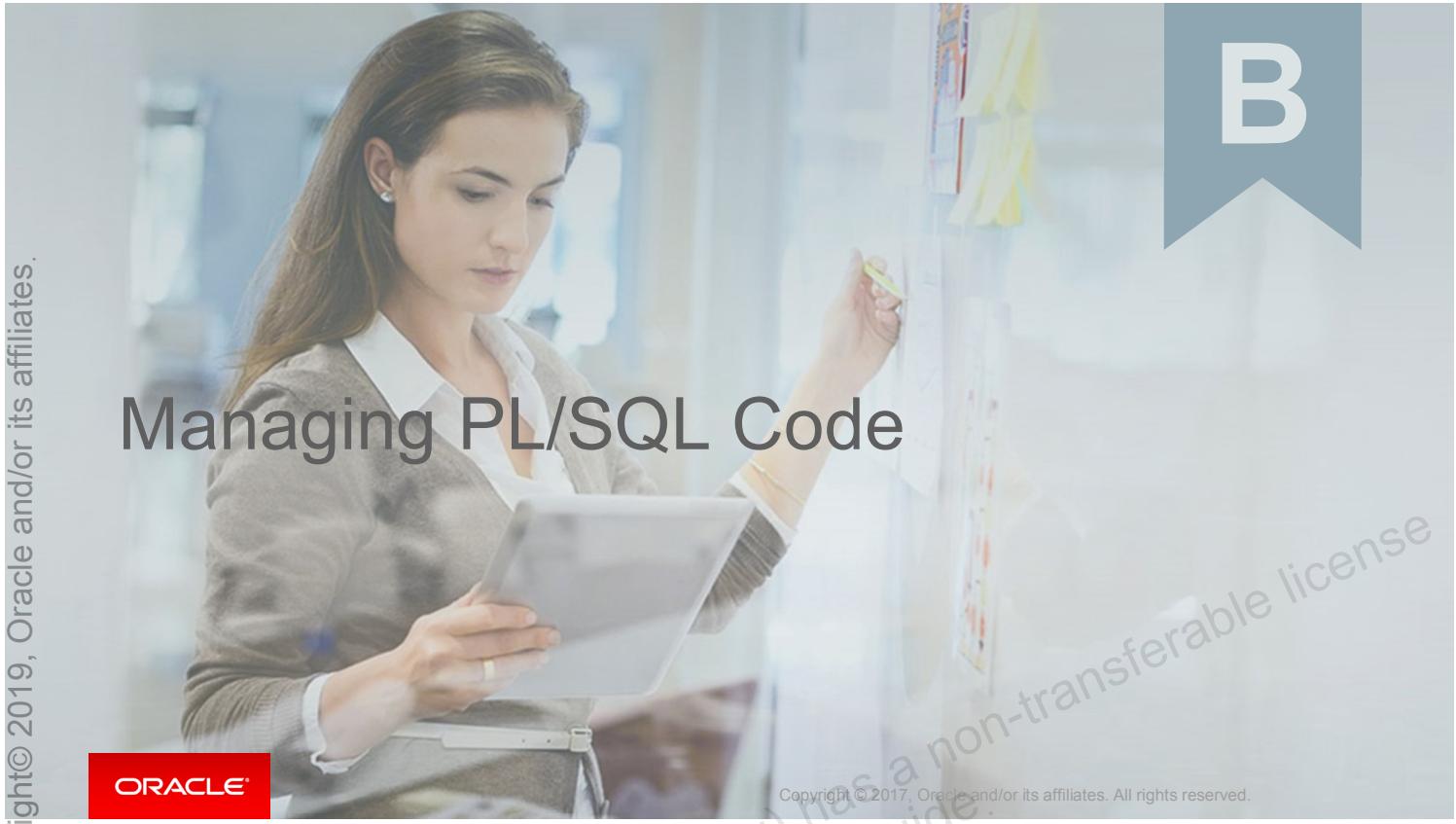
- Execute a basic SELECT statement
- Create, alter, and drop a table using the data definition language (DDL) statements
- Insert, update, and delete rows from one or more tables using data manipulation language (DML) statements
- Commit, roll back, and create save points using the transaction control statements
- Perform join operations on one or more tables



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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

GANG LIU (gangl@baylorhealth.edu) has a non-transferable license
to use this Student Guide.



Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

Managing PL/SQL Code

ORACLE®

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

B

Objectives

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

- Describe and use conditional compilation
- Hide PL/SQL source code by using dynamic obfuscation and the wrap utility

This appendix introduces the conditional compilation and obfuscating (or wrapping) PL/SQL code.

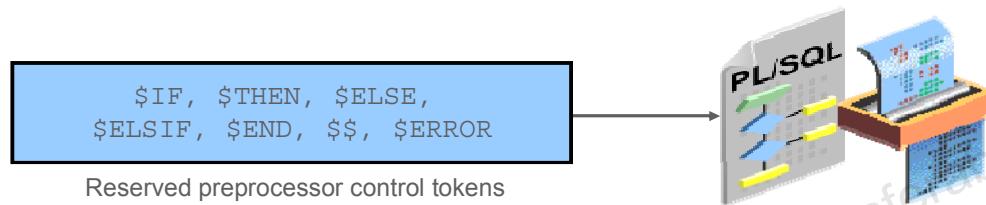
Agenda

- Using conditional compilation
- Obfuscating PL/SQL code

Conditional Compilation

Enables you to customize the functionality in a PL/SQL application without removing any source code:

- Use the latest functionality with the latest database release, or disable the new features to run the application against an older release of the database.
- Activate the debugging or tracing functionality in the development environment and hide that functionality in the application while it runs at a production site.



ORACLE

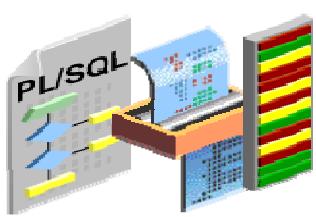
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Conditional compilation enables you to selectively include code depending on the values of the conditions evaluated during compilation. For example, conditional compilation enables you to determine which PL/SQL features in a PL/SQL application are used for specific database releases. The latest PL/SQL features in an application can be run on a new database release; at the same time, those features can be conditional so that the same application is compatible with a previous database release. Conditional compilation is also useful when you want to execute debugging procedures in a development environment but want to turn off the debugging routines in a production environment.

Benefits of Conditional Compilation

- Support for multiple versions of the same program in one source code
- Easy maintenance and debugging of code
- Easy migration of code to a different release of the database

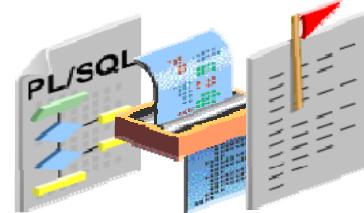
How Does Conditional Compilation Work?



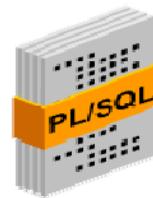
Selection directives:
Use the \$IF token.



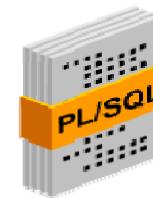
Inquiry directives:
Use the \$\$ token.



Error directives:
Use the \$ERROR token.



DBMS_PREPROCESSOR
package



DBMS_DB_VERSION
package



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

Using Selection Directives

```
$IF <Boolean-expression> $THEN Text  
$ELSIF <Boolean-expression> $THEN Text  
. . .  
$ELSE Text  
$END
```

```
DECLARE  
CURSOR cur IS SELECT employee_id FROM  
employees WHERE  
$IF myapp_tax_package.new_tax_code $THEN  
    salary > 20000;  
$ELSE  
    salary > 50000;  
$END  
BEGIN  
    OPEN cur;  
. . .  
END;
```

ORACLE

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

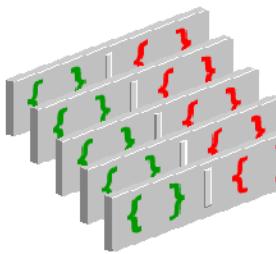
The conditional selection directive looks and operates like the IF-THEN-ELSE mechanism in PL/SQL proper. When the preprocessor encounters \$THEN, it verifies that the text between \$IF and \$THEN is a static expression. If the check succeeds and the result of the evaluation is TRUE, the PL/SQL program text between \$THEN and \$ELSE (or \$ELSIF) is selected for compilation.

The selection condition (the expression between \$IF and \$THEN) can be constructed by referring to constants defined in another package or an inquiry directive (or some combination of the two).

In the example in the slide, the conditional selection directive chooses between two versions of the cursor (cur) on the basis of the value of MYAPP_TAX_PACKAGE.NEW_TAX_CODE.

If the value is TRUE, employees with salary > 20000 are selected; otherwise, employees with salary > 50000 are selected.

Using Predefined and User-Defined Inquiry Directives



```
PLSQL_CCFLAGS  
PLSQL_CODE_TYPE  
PLSQL_OPTIMIZE_LEVEL  
PLSQL_WARNINGS  
NLS_LENGTH_SEMANTICS  
PLSQL_LINE  
PLSQL_UNIT
```

Predefined inquiry directives

```
PLSQL_CCFLAGS = ['plsql_ccflags:true,debug:true,debug:0'];
```

User-defined inquiry directives

ORACLE

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

An inquiry directive can be predefined or user-defined. Following is the order of the processing flow when conditional compilation attempts to resolve an inquiry directive:

1. The ID is used as an inquiry directive in the form `$$id` for the search key.
 2. The two-pass algorithm proceeds as follows:
 - a. The string in the `PLSQL_CCFLAGS` initialization parameter is scanned from right to left, searching by ID for a matching name (not case sensitive). Processing is done if an ID is found.
 - b. The predefined inquiry directives are searched. Processing is done if an ID is found.
 3. If the `$$ID` cannot be resolved to a value, the `PLW-6003` warning message is reported if the source text is not wrapped. The literal `NULL` is substituted as the value for undefined inquiry directives.
- Note:** If the PL/SQL code is wrapped, the warning message is disabled so that the undefined inquiry directive is not revealed.

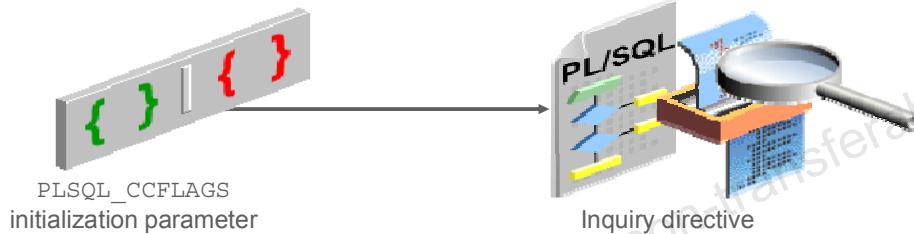
In the example in the slide, the value of `$$debug` is 0 and the value of `$$plsql_ccflags` is TRUE. Note that the value of `$$plsql_ccflags` resolves to the user-defined `plsql_ccflags` inside the value of the `PLSQL_CCFLAGS` compiler parameter. This occurs because a user-defined directive overrides the predefined one.

The PLSQL_CCFLAGS Parameter and the Inquiry Directive

Use the PLSQL_CCFLAGS parameter to control conditional compilation of each PL/SQL library unit independently.

```
PLSQL_CCFLAGS = '<v1>:<c1>,<v2>:<c2>,...,<vn>:<cn>'
```

```
ALTER SESSION SET  
PLSQL_CCFLAGS = 'plsql_ccflags:true, debug:true, debug:0';
```



ORACLE

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

Oracle Database 10g Release 2 introduced a new Oracle initialization parameter PLSQL_CCFLAGS for use with conditional compilation. This dynamic parameter enables you to set up name-value pairs. The names (called “flag names”) can then be referenced in inquiry directives.

PLSQL_CCFLAGS provides a mechanism that enables PL/SQL programmers to control the conditional compilation of each PL/SQL library unit independently.

Values

- *vi*: Has the form of an unquoted PL/SQL identifier that is unrestricted and can be a reserved word or a keyword. The text is not case sensitive. Each is known as a flag or a flag name. Each *vi* can occur more than once in the string, each occurrence can have a different flag value, and the flag values can be of different kinds.
- *ci*: Can be any of the following:
 - A PL/SQL Boolean literal
 - A PLS_INTEGER literal
 - The literal NULL (default). The text is not case sensitive. Each is known as a flag value and corresponds to a flag name.

Displaying the PLSQL_CCFLAGS Initialization Parameter Setting

```
SELECT name, type, plsql_ccflags  
FROM user_plsql_object_settings
```

Results:			
#	NAME	TYPE	PLSQL_CCFLAGS
1	DEPT_PKG	PACKAGE	(null)
2	DEPT_PKG	PACKAGE BODY	(null)
3	TAXES_PKG	PACKAGE	(null)
4	TAXES_PKG	PACKAGE BODY	(null)
5	EMP_PKG	PACKAGE	(null)
6	EMP_PKG	PACKAGE BODY	(null)
7	SECURE_DML	PROCEDURE	(null)
8	SECURE_EMPLOYEES	TRIGGER	(null)
9	ADD_JOB_HISTORY	PROCEDURE	plsql_ccflags:true, debug:true, debug:0
10	UPDATE_JOB_HISTORY	TRIGGER	(null)

ORACLE

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

The PLSQL_CCFLAGS Parameter and the Inquiry Directive: Example

```
ALTER SESSION SET PLSQL_CCFLAGS = 'Tracing:true';
CREATE OR REPLACE PROCEDURE P IS
BEGIN
$IF $$tracing $THEN
    DBMS_OUTPUT.PUT_LINE ('TRACING');
$END
END P;
```

```
ALTER SESSION SET succeeded.
PROCEDURE P Compiled.
```

```
SELECT name, plsql_ccflags
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE name = 'P';
```

Results:

	NAME	PLSQL_CCFLAGS
1	P	Tracing:true

ORACLE

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

Using Conditional Compilation Error Directives to Raise User-Defined Errors

```
$ERROR varchar2_static_expression $END  
  
ALTER SESSION SET Plsql_CCFlags = ' Trace_Level:3 '  
/  
CREATE PROCEDURE P IS  
BEGIN  
  $IF    $$Trace_Level = 0 $THEN ...;  
  $ELSIF $$Trace_Level = 1 $THEN ...;  
  $ELSIF $$Trace_Level = 2 $THEN ...;  
  $else $error 'Bad: '||$$Trace_Level $END -- error  
                                -- directive  
  $END -- selection directive ends  
END P;
```

```
SHOW ERRORS  
Errors for PROCEDURE P:  
LINE/COL ERROR  
-----  
6/9      PLS-00179: $ERROR: Bad: 3
```



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

Using Static Expressions with Conditional Compilation

- Boolean static expressions:
 - TRUE, FALSE, NULL, IS NULL, IS NOT NULL
 - >, <, >=, <=, =, <>, NOT, AND, OR
- PLS_INTEGER static expressions:
 - -2147483648 to 2147483647, NULL
- VARCHAR2 static expressions include:
 - ||, NULL, TO_CHAR
- Static constants:

```
static_constant CONSTANT datatype := static_expression;
```



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

As described earlier, a preprocessor processes conditional directives before proper compilation begins. Consequently, only those expressions that can be fully evaluated at compile time are permitted in conditional compilation directives. Expressions that contain references to variables or functions that require the execution of the PL/SQL are not available during compilation and cannot be evaluated.

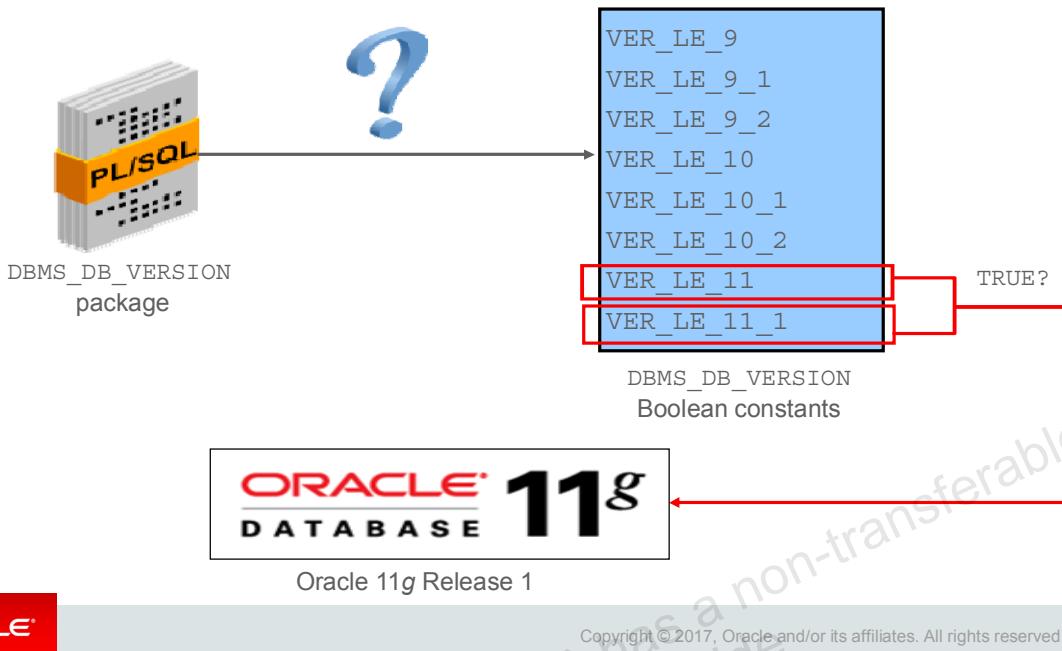
These PL/SQL expressions that are allowed in conditional compilation directives are referred to as “static expressions.” Static expressions are carefully defined to guarantee that if a unit is automatically recompiled without any changes to the values it depends on, the expressions evaluate in the same way and the same source is compiled.

Static expressions are typically composed of three sources:

- Inquiry directives marked with \$\$
- Constants defined in PL/SQL packages such as DBMS_DB_VERSION. These values can be combined and compared using the ordinary operations of PL/SQL.
- Literals such as TRUE, FALSE, 'CA', 123, and NULL

Static expressions can also contain operations that include comparisons, logical Boolean operations (such as OR and AND), or concatenations of static character expression.

DBMS_DB_VERSION Package: Boolean Constants



Oracle Database 10g Release 2 introduced the DBMS_DB_VERSION package. This package specifies the Oracle Database version and release numbers that are useful when making simple selections for conditional compilation.

The constants represent a Boolean condition that evaluates to less than or equal to the version and the release, if present.

Example

VER_LE_11 indicates that the database version ≤ 11 . The values of the constants are either TRUE or FALSE. For example, in an Oracle Database 11g Release 1 database, VER_LE_11 and VER_LE_11_1 are TRUE and all other constants are FALSE.

DBMS_DB_VERSION Package Constants

Name	Description
VER_LE_9	Version <= 9.
VER_LE_9_1	Version <= 9 and release <= 1.
VER_LE_9_2	Version <= 9 and release <= 2.
VER_LE_10	Version <= 10.
VER_LE_10_1	Version <= 10 and release <= 1.
VER_LE_10_2	Version <= 10 and release <= 2.
VER_LE_11	Version <= 11.
VER_LE_11_1	Version <= 11 and release <= 1.



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

The package for the Oracle Database 11g Release 1 version is shown as follows:

```
PACKAGE DBMS_DB_VERSION IS
    VERSION CONSTANT PLS_INTEGER := 11; -- RDBMS version
                                         -- number
    RELEASE CONSTANT PLS_INTEGER := 1;   -- RDBMS release
                                         -- number
    ver_le_9_1      CONSTANT BOOLEAN := FALSE;
    ver_le_9_2      CONSTANT BOOLEAN := FALSE;
    ver_le_9        CONSTANT BOOLEAN := FALSE;
    ver_le_10_1     CONSTANT BOOLEAN := FALSE;
    ver_le_10_2     CONSTANT BOOLEAN := FALSE;
    ver_le_10       CONSTANT BOOLEAN := FALSE;
    ver_le_11_1     CONSTANT BOOLEAN := TRUE;
    ver_le_11       CONSTANT BOOLEAN := TRUE;
END DBMS_DB_VERSION;
```

The DBMS_DB_VERSION package contains different constants for different Oracle Database releases. The Oracle Database 11g Release 1 version of the DBMS_DB_VERSION package uses the constants shown in the slide.

Using Conditional Compilation with Database Versions: Example

```
ALTER SESSION SET PLSQL_CCFLAGS = 'my_debug:FALSE, my_tracing:FALSE';
CREATE PACKAGE my_pkg AS
  SUBTYPE my_real IS
    -- Check the database version, if >= 10g, use BINARY_DOUBLE data type,
    -- else use NUMBER data type
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN    NUMBER;
    $ELSE    BINARY_DOUBLE;
    $END
    my_pi my_real; my_e my_real;
END my_pkg;
/
CREATE PACKAGE BODY my_pkg AS
BEGIN
  $IF DBMS_DB_VERSION.VERSION < 10 $THEN
    my_pi := 3.14016408289008292431940027343666863227;
    my_e   := 2.71828182845904523536028747135266249775;
  $ELSE
    my_pi := 3.14016408289008292431940027343666863227d;
    my_e   := 2.71828182845904523536028747135266249775d;
  $END
END my_pkg;
/
```



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

This example also shows the use of the PLSQL_CCFLAGS parameter. You first set the PLSQL_CCFLAGS parameter flag to display debugging code and tracing information.

In the example in the slide on this page and the next page, conditional compilation is used to specify code for database versions. Conditional compilation is used to determine whether the BINARY_DOUBLE data type can be used in the calculations for PL/SQL units in the database. The BINARY_DOUBLE data type can be used only in Oracle Database 10g or later. If you are using Oracle Database 10g, the data type for my_real is BINARY_DOUBLE; otherwise, the data type for my_real is NUMBER.

In the specification of the new package (my_pkg), conditional compilation is used to check for the database version. In the body definition of the package, conditional compilation is used again to set the values of my_pi and my_e for future calculations based on the database version.

The result of the example code in the slide is as follows:

```
PACKAGE my_pkg compiled
PACKAGE BODY my_pkg compiled
session SET altered.
```

Using Conditional Compilation with Database Versions: Example

```
CREATE OR REPLACE PROCEDURE circle_area(p_radius my_pkg.my_real) IS
  v_my_area my_pkg.my_real;
  v_my_datatype VARCHAR2(30);
BEGIN
  v_my_area := my_pkg.my_pi * p_radius * p_radius;
  DBMS_OUTPUT.PUT_LINE('Radius: ' || TO_CHAR(p_radius)
    || ' Area: ' || TO_CHAR(v_my_area));
  $IF $$my_debug $$THEN -- if my_debug is TRUE, run some debugging code
    SELECT DATA_TYPE INTO v_my_datatype FROM USER_ARGUMENTS
      WHERE OBJECT_NAME = 'CIRCLE_AREA' AND ARGUMENT_NAME = 'P_RADIUS';
    DBMS_OUTPUT.PUT_LINE('Datatype of the RADIUS argument is: ' ||
      v_my_datatype);
  $END
END; /
```

PROCEDURE circle_area compiled

```
CALL circle_area(50); -- Using Oracle Database 11g Release 2
```

```
CALL circle_area(50) succeeded.
Radius: 5.0E+001 Area: 7.8504102072252062E+003
```

ORACLE

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

In the example in the slide, a new procedure called `circle_area` is defined. This procedure calculates the area of a circle based on the values of the variables in the `my_pkg` package defined on the previous page. The procedure has one informal parameter: `radius`.

The procedure declares two variables:

- `my_area`, which is the same data type as `my_real` in `my_pkg`
- `my_datatype`, which is a `VARCHAR2 (30)`

In the procedure's body, `my_area` becomes equal to the value of `my_pi` set in `my_pkg` multiplied by the value that is passed to the procedure as a radius. A message is printed displaying the radius and the area of the circle, as shown in the second code example in the slide.

Note: If you want to set `my_debug` to TRUE, you can make this change only for the `circle_area` procedure with the `REUSE SETTINGS` clause as follows:

```
ALTER PROCEDURE circle_area COMPILE PLSQL_CCFLAGS = 'my_debug:TRUE' REUSE
SETTINGS;
```

Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text

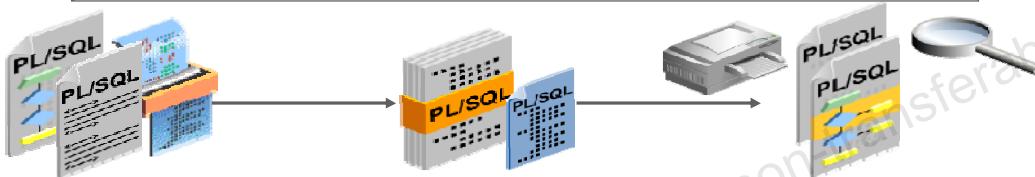
CALL

```
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE('PACKAGE',
'ORA61', 'MY_PKG');
```

```
CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE('PACKAGE', succeeded.
PACKAGE my_pkg AS
SUBTYPE my_real IS
-- Check the database version, if >= 10g, use BINARY_DOUBLE data type,
-- else use NUMBER data type

BINARY_DOUBLE;

my_pi my_real; my_e my_real;
END my_pkg;
```



ORACLE

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

DBMS_PREPROCESSOR subprograms print or retrieve the postprocessed source text of a PL/SQL unit after processing the conditional compilation directives. This postprocessed text is the actual source that is used to compile a valid PL/SQL unit. The example in the slide shows how to print the postprocessed form of `my_pkg` by using the `PRINT_POST_PROCESSED_SOURCE` procedure.

When `my_pkg` is compiled on an Oracle Database 10g release (or later) database using the HR account, the resulting output is shown in the slide.

The `PRINT_POST_PROCESSED_SOURCE` removes unselected text. The lines of code that are not included in the postprocessed text are removed. The arguments for the `PRINT_POST_PROCESSED_SOURCE` procedure are object type, schema name (using student account ORA61), and object name.

Note: For additional information about the `DBMS_PREPROCESSOR` package, refer to the *Oracle Database PL/SQL Packages and Types Reference*.

Agenda

- Using conditional compilation
- Obfuscating PL/SQL code

Obfuscation

- The obfuscation (or wrapping) of a PL/SQL unit is the process of hiding the PL/SQL source code.
- Wrapping can be done with the `wrap` utility and `DBMS_DDL` subprograms.
- The `wrap` utility is run from the command line. It processes an input SQL file, such as a SQL*Plus installation script.
- The `DBMS_DDL` subprograms wrap a single PL/SQL unit, such as a single `CREATE PROCEDURE` command, that has been generated dynamically.

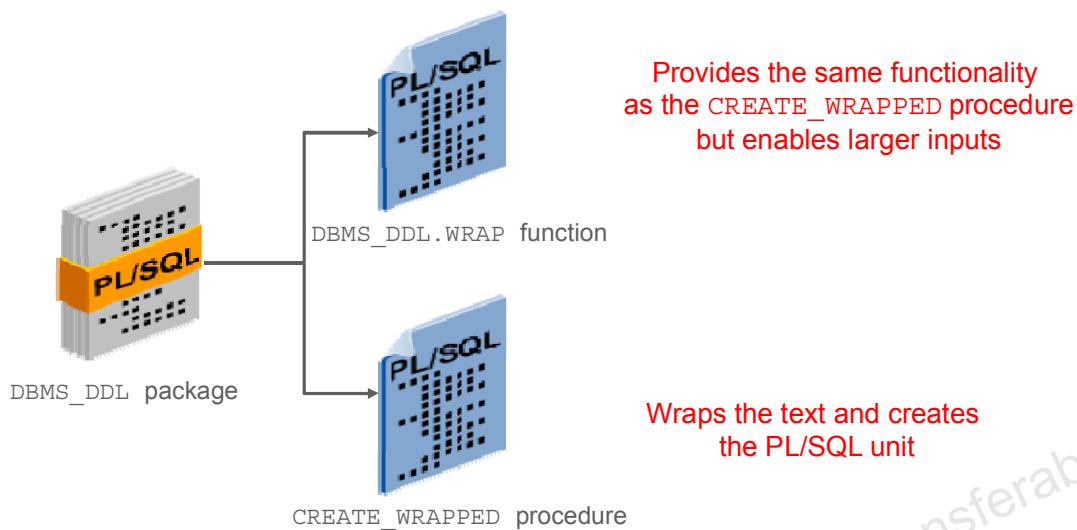
ORACLE

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

Benefits of Obfuscating

- It prevents others from seeing your source code.
- Your source code is not visible through the `USER_SOURCE`, `ALL_SOURCE`, or `DBA_SOURCE` data dictionary views.
- SQL*Plus can process the obfuscated source files.
- The Import and Export utilities accept wrapped files.

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



ORACLE

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

CREATE_WWRAPPED Procedure

Does the following:

1. Takes as input a single CREATE OR REPLACE statement that specifies creation of a PL/SQL package specification, package body, function, procedure, type specification, or type body
2. Generates a CREATE OR REPLACE statement in which the PL/SQL source text has been obfuscated
3. Executes the generated statement

WRAP Function

Does the following:

1. Takes as input a CREATE OR REPLACE statement that specifies the creation of a PL/SQL package specification, package body, function, procedure, type specification, or type body
2. Returns a CREATE OR REPLACE statement in which the text of the PL/SQL unit has been obfuscated

Nonobfuscated PL/SQL Code: Example

```
SET SERVEROUTPUT ON
BEGIN -- The ALL_SOURCE view family shows source code
  EXECUTE IMMEDIATE '
    CREATE OR REPLACE PROCEDURE P1 IS
      BEGIN
        DBMS_OUTPUT.PUT_LINE ('I am not wrapped');
      END P1;
    ';
END;
/
CALL p1();
```

```
anonymous block completed
CALL p1() succeeded.
I'm not wrapped
```

```
SELECT text FROM user_source
WHERE name = 'P1' ORDER BY line;
```

TEXT
1 PROCEDURE P1 IS
2 BEGIN
3 DBMS_OUTPUT.PUT_LINE ('I am not wrapped');
4 END P1;



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

Obfuscated PL/SQL Code: Example

```
BEGIN -- ALL_SOURCE view family obfuscates source code
DBMS_DDL.CREATE_WRAPPED (
    CREATE OR REPLACE PROCEDURE P1 IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE (''I am wrapped now'');
        END P1;
    );
END;
/
CALL p1();
```

```
anonymous block completed
p1 ) succeeded.
```

```
SELECT text FROM user_source
WHERE name = 'P1' ORDER BY line;
```

```
TEXT
-----
PROCEDURE P1 wrapped
a000000
b2
abcd
...

```



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

Dynamic Obfuscation: Example

```
SET SERVEROUTPUT ON

DECLARE
c_code CONSTANT VARCHAR2(32767) :=
' CREATE OR REPLACE PROCEDURE new_proc AS
  v_VDATE DATE;
BEGIN
  v_VDATE := SYSDATE;
  DBMS_OUTPUT.PUT_LINE(v_VDATE) ;
END; '
BEGIN
  DBMS_DDL.CREATE_WRAPPED (c_CODE);
END;
/
```



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

The example in the slide displays the creation of a dynamically obfuscated procedure called NEW_PROC. To verify that the code for NEW_PROC is obfuscated, you can query from the DBA|ALL|USER_SOURCE dictionary views:

```
SELECT text FROM user_source
WHERE name = 'NEW_PROC';
```

TEXT

PROCEDURE new_proc wrapped a000000

7 71 9e hBWmpGeSsd58b4jCP3/0d04xof0wg5nnm7+fMr2ywFyFDGLQ1haXriu4dCuPCWnnx1J0Ulxp pvc8nsr7Seq/riQvHRsXAQovdoh0K6ZvM1Kbskr+KLK957KzHQYwLK4k6rJLC55EyJ7qJB/2 RDmm3j79Uw==
--

PL/SQL Wrapper Utility

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



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

Running the PL/SQL Wrapper Utility

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

- Do not use spaces around the equal signs.
- The `INAME` argument is required.
- The default extension for the input file is `.sql`, unless it is specified with the name.
- The `ONAME` argument is optional.
- The default extension for output file is `.plb`, unless specified with the `ONAME` argument.

Examples

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



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

The PL/SQL Wrapper utility is an operating system executable called `WRAP`.

Note: This is 11g-only code.

To run the wrapper, enter the following command at your operating system prompt:

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

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

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

Note

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

Results of Wrapping

```
-- Original PL/SQL source code in input file:  
  
CREATE PACKAGE banking IS  
    min_bal := 100;  
    no_funds EXCEPTION;  
    ...  
END banking;  
/
```

```
-- Wrapped code in output file:  
  
CREATE PACKAGE banking  
    wrapped  
012abc463e ...  
/
```



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

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

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

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

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

Guidelines for Wrapping

- You must wrap only the package body and not the package specification.
- The wrapper can detect syntactic errors but cannot detect semantic errors.
- The output file should not be edited. You maintain the original source code and wrap again as required.
- To ensure that all the important parts of your source code are obfuscated, view the wrapped file in a text editor before distributing it.



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

Guidelines include the following:

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

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

However, the PL/SQL compiler resolves external references. Therefore, semantic errors are reported when the wrapper output file (`.plb` file) is compiled.

- Because its contents are not readable, the output file should not be edited. To change a wrapped object, you must modify the original source code and wrap the code again.

DBMS_DDL Package Versus wrap Utility

Functionality	DBMS_DDL Package	wrap Utility
Code obfuscation	Yes	Yes
Dynamic Obfuscation	Yes	No
Obfuscate multiple programs at a time	No	Yes



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

Both the `wrap` utility and the `DBMS_DDL` package have distinct uses.

wrap Utility

The `wrap` utility is useful for obfuscating multiple programs with one execution of the utility. In essence, a complete application can be wrapped. However, the `wrap` utility cannot be used to obfuscate dynamically generated code at run time. The `wrap` utility processes an input SQL file and obfuscates only the PL/SQL units in the file, such as:

- Package specification and body
- Function and procedure
- Type specification and body

The `wrap` utility does not obfuscate PL/SQL content in:

- Anonymous blocks
- Triggers
- Non-PL/SQL code

DBMS_DDL Package

The `DBMS_DDL` package is intended to obfuscate a dynamically generated program unit from within another program unit. The `DBMS_DDL` package methods cannot obfuscate multiple program units at one execution. Each execution of these methods accepts only one `CREATE OR REPLACE` statement at a time as argument.

Summary

In this appendix, you should have learned how to:

- Describe and use conditional compilation
- Hide PL/SQL source code by using dynamic obfuscation and the `wrap` utility

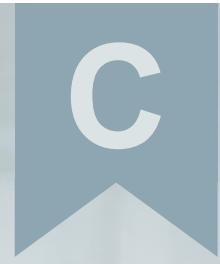


Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

Implementing Triggers

ORACLE®

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



Objectives

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

- Enhance database security with triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities by using triggers



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

Controlling Security Within the Server

Using database security with the GRANT statement

```
GRANT SELECT, INSERT, UPDATE, DELETE  
  ON employees  
  TO clerk;          -- database role  
GRANT clerk TO scott;
```



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

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
  dummy PLS_INTEGER;
BEGIN
  IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT', 'SUN')) THEN
    RAISE_APPLICATION_ERROR(-20506,'You may only
      change data during normal business hours.');
  END IF;
  SELECT COUNT(*) INTO dummy FROM holiday
  WHERE holiday_date = TRUNC (SYSDATE);
  IF dummy > 0 THEN
    RAISE_APPLICATION_ERROR(-20507,
      'You may not change data on a holiday.');
  END IF;
END;
/
```



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

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD  
CONSTRAINT ck_salary CHECK (salary >= 500);
```



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

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key.

Use these rules to:

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

Example

The code sample in the slide ensures that the salary is at least \$500.

Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
    'Do not decrease salary.');
END;
/
```



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

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

Example

The code sample in the slide ensures that the salary is never decreased.

Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
    FOREIGN KEY (department_id)
      REFERENCES departments(department_id)
ON DELETE CASCADE;
```



Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

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

Example

When a department is removed from the DEPARTMENTS parent table, cascade the deletion to the corresponding rows in the EMPLOYEES child table.

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
  AFTER UPDATE OF department_id ON departments
  FOR EACH ROW
BEGIN
  UPDATE employees
    SET employees.department_id=:NEW.department_id
    WHERE employees.department_id=:OLD.department_id;
  UPDATE job_history
    SET department_id=:NEW.department_id
    WHERE department_id=:OLD.department_id;
END;
/
```



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

The following referential integrity rules are not supported by declarative constraints:

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.

You can develop triggers to implement these integrity rules.

Example

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table.

For a complete referential integrity solution using triggers, a single trigger is not enough.

Replicating a Table Within the Server

```
CREATE MATERIALIZED VIEW emp_copy  
NEXT sysdate + 7  
AS SELECT * FROM employees@ny;
```



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

Creating a Materialized View

Materialized views enable you to maintain copies of remote data on your local node for replication purposes. You can select data from a materialized view as you would from a normal database table or view. A materialized view is a database object that contains the results of a query, or a copy of some database on a query. The `FROM` clause of the query of a materialized view can name tables, views, and other materialized views.

When a materialized view is used, replication is performed implicitly by the Oracle server. This performs better than user-defined PL/SQL triggers for replication. Materialized views:

- Copy data from local and remote tables asynchronously, at user-defined intervals
- Can be based on multiple master tables
- Are read-only by default, unless using the Oracle Advanced Replication feature
- Improve the performance of data manipulation on the master table

Alternatively, you can replicate tables using triggers.

The example in the slide creates a copy of the remote `EMPLOYEES` table from New York. The `NEXT` clause specifies a date-time expression for the interval between automatic refreshes.

Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
  BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN /* Proceed if user initiates data operation,
NOT through the cascading trigger.*/
  IF INSERTING THEN
    IF :NEW.flag IS NULL THEN
      INSERT INTO employees@sf
      VALUES (:new.employee_id, . . . , 'B');
      :NEW.flag := 'A';
    END IF;
  ELSE /* Updating. */
    IF :NEW.flag = :OLD.flag THEN
      UPDATE employees@sf
      SET ename=:NEW.last_name, . . . , flag=:NEW.flag
      WHERE employee_id = :NEW.employee_id;
    END IF;
    IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
    ELSE :NEW.flag := 'A';
  END IF;
END IF;
END;
```



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

You can replicate a table with a trigger. By replicating a table, you can:

- Copy tables synchronously, in real time
- Base replicas on a single master table
- Read from replicas as well as write to them

Note: Excessive use of triggers can impair the performance of data manipulation on the master table, particularly if the network fails.

Example

In New York, replicate the local EMPLOYEES table to San Francisco.

Computing Derived Data Within the Server

```
UPDATE departments
SET total_sal=(SELECT SUM(salary)
   FROM employees
  WHERE employees.department_id =
    departments.department_id);
```



By using the server, you can schedule batch jobs or use the database Scheduler for the following scenarios:

- Compute derived column values asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

Example

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

Computing Derived Values with a Trigger

```
CREATE PROCEDURE increment_salary
  (id NUMBER, new_sal NUMBER) IS
BEGIN
  UPDATE departments
  SET    total_sal = NVL (total_sal, 0)+ new_sal
  WHERE  department_id = id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE
ON employees FOR EACH ROW
BEGIN
  IF DELETING THEN      increment_salary(
    :OLD.department_id, (-1*:OLD.salary));
  ELSIF UPDATING THEN   increment_salary(
    :NEW.department_id, (:NEW.salary-:OLD.salary));
  ELSE                 increment_salary(
    :NEW.department_id,:NEW.salary); -- INSERT
  END IF;
END;
```



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

By using a trigger, you can perform the following tasks:

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a single pass to the database.

Example

Keep a running total of the salary for each department in the special TOTAL_SALARY column of the DEPARTMENTS table.

Logging Events with a Trigger

```
CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
ON inventories FOR EACH ROW
DECLARE
  dsc product_descriptions.product_description%TYPE;
  msg_text VARCHAR2(2000);
BEGIN
  IF :NEW.quantity_on_hand <=
  :NEW.reorder_point THEN
    SELECT product_description INTO dsc
    FROM product_descriptions
    WHERE product_id = :NEW.product_id;
    msg_text := 'ALERT: INVENTORY LOW ORDER:' ||
      'Yours,' || CHR(10) || user || '.' || CHR(10);
  ELSIF :OLD.quantity_on_hand >=
  :NEW.quantity_on_hand THEN
    msg_text := 'Product #' || ... CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com','ord@oracle.com',
    message=>msg_text, subject='Inventory Notice');
END;
```



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

Logging Events Transparently

In the trigger code:

- CHR(10) is a carriage return
- Reorder_point is not NULL
- Another transaction can receive and read the message in the pipe

Example

```

CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF amount_in_stock, reorder_point
ON inventory FOR EACH ROW
DECLARE
    dsc product.descrip%TYPE;
    msg_text VARCHAR2(2000);
BEGIN
    IF :NEW.amount_in_stock <= :NEW.reorder_point THEN
        SELECT descrip INTO dsc
        FROM PRODUCT WHERE prodid = :NEW.product_id;
        msg_text := 'ALERT: INVENTORY LOW ORDER:' || CHR(10) ||
                    'It has come to my personal attention that, due to recent' ||
                    'transactions, our inventory for product #' ||
                    TO_CHAR(:NEW.product_id) || '-- ' || dsc ||
                    '-- has fallen below acceptable levels.' || CHR(10) ||
                    'Yours,' || CHR(10) || user || '.' || CHR(10) || CHR(10);
    ELSIF :OLD.amount_in_stock >= :NEW.amount_in_stock THEN
        msg_text := 'Product #' || TO_CHAR(:NEW.product_id) ||
                    ' ordered. ' || CHR(10) || CHR(10);
    END IF;
    UTL_MAIL.SEND('inv@oracle.com', 'ord@oracle.com',
                  message => msg_text, subject => 'Inventory Notice');
END;

```

Summary

In this appendix, you should have learned how to:

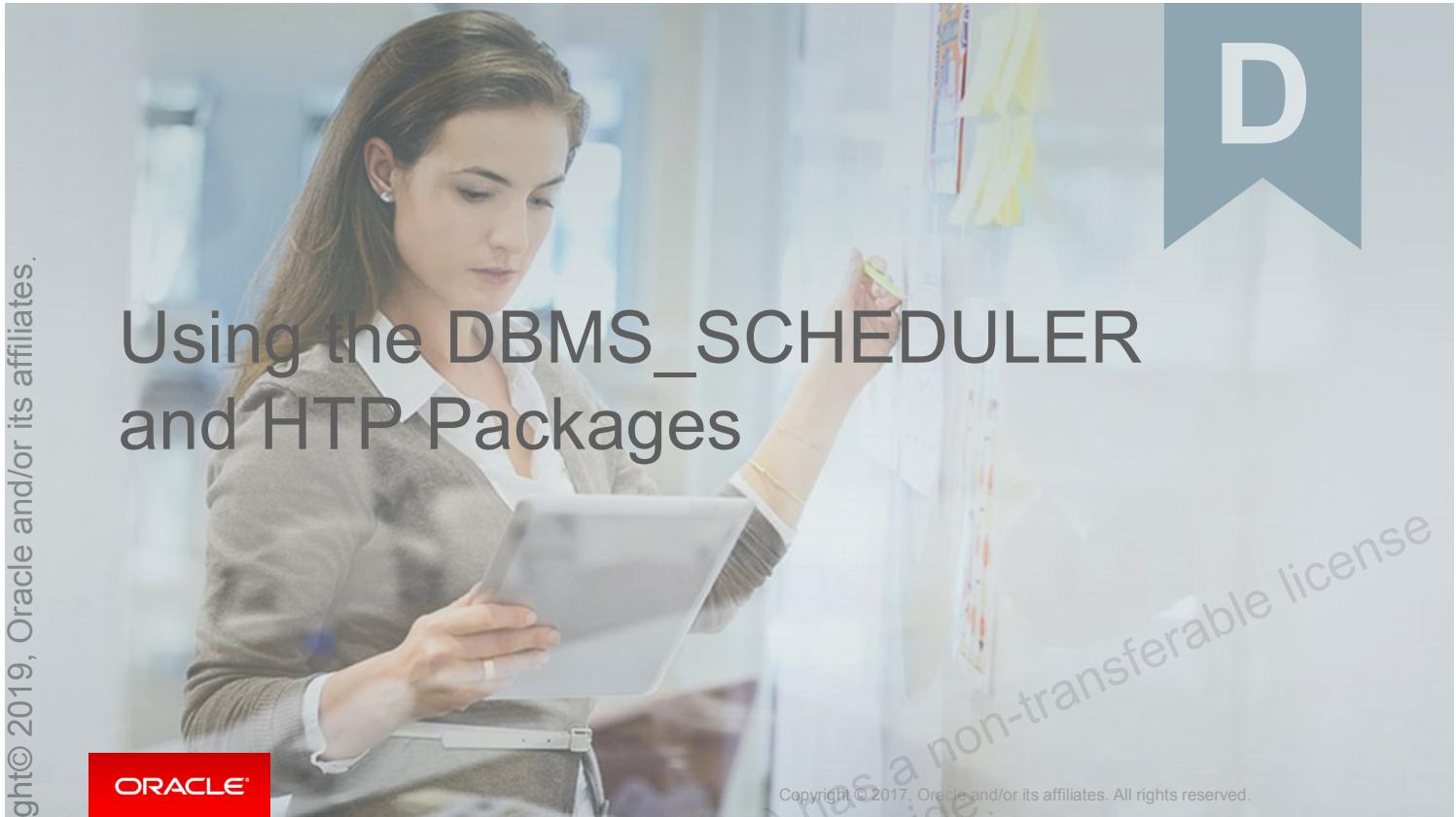
- Enhance database security with triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities using triggers



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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

GANG LIU (gangl@baylorhealth.edu) has a non-transferable license
to use this Student Guide.



Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

Using the DBMS_SCHEDULER and HTP Packages

ORACLE®

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

DANG LIU (gangli@baylorhealth.edu) has a non-transferable license
to use this Student Guide.

Objectives

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

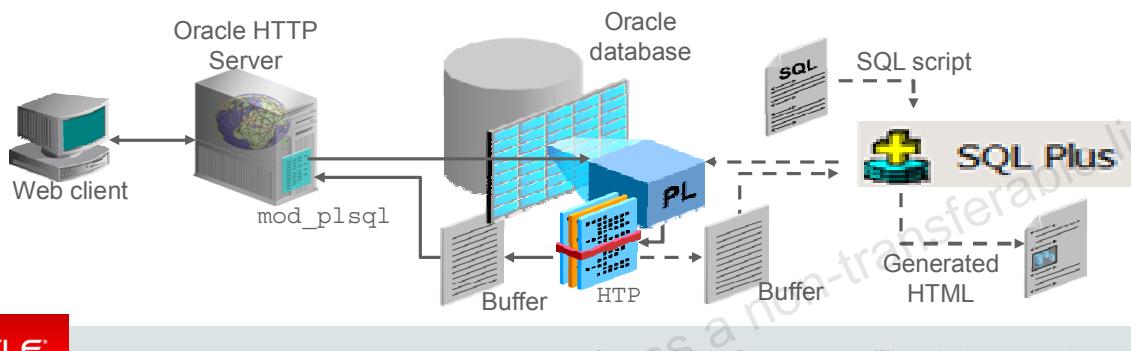
- Use the `HTP` package to generate a simple webpage
- Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution



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

Generating Webpages with the HTP Package

- The HTP package procedures generate HTML tags.
- The HTP package is used to generate HTML documents dynamically and can be invoked from:
 - A browser using Oracle HTTP Server and PL/SQL Gateway (`mod_plsql`) services
 - A SQL*Plus script to display HTML output



ORACLE

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

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

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

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

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

Using the HTP Package Procedures

- Generate one or more HTML tags. For example:

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

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

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

```
<HTML>
<HEAD>
<TITLE>Welcome</TITLE>
</HEAD>
<BODY>
My home page
</BODY>
</HTML>
```



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

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

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

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

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

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

Creating an HTML File with SQL*Plus

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

1. Create a SQL script with the following commands:

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

2. Load and execute the script in SQL*Plus, supplying values for substitution variables.
3. Select, copy, and paste the HTML text that is generated in the browser to an HTML file.
4. Open the HTML file in a browser.



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

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

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

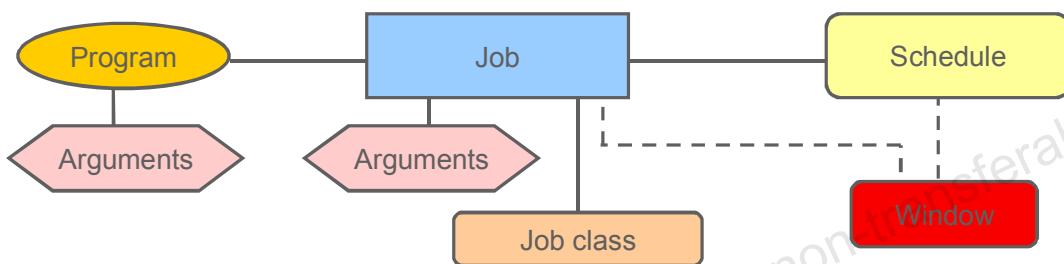
The ACCEPT command prompts for the name of the procedure to execute. The call to OWA_UTIL.SHOWPAGE displays the HTML tags in the browser window. You can then copy and paste the generated HTML tags from the browser window into an HTML file, typically with an .htm or .html extension.

Note: If you are using SQL*Plus, then you can use the SPOOL command to direct the HTML output directly to an HTML file.

The DBMS_SCHEDULER Package

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

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



ORACLE

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

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

The diagram shows the following architectural components of the Scheduler:

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

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

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

For more information about the Scheduler, see the online course titled *Oracle Database: Configure and Manage Jobs with the Scheduler*.

Creating a Job

- A job can be created in several ways by using a combination of inline parameters, named Programs, and named Schedules.
- You can create a job with the `CREATE_JOB` procedure by:
 - Using inline information with the “what” and the schedule specified as parameters
 - Using a named (saved) program and specifying the schedule inline
 - Specifying what should be done inline and using a named Schedule
 - Using named Program and Schedule components



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

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

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

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

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

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

Creating a Job with Inline Parameters

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

```
-- Schedule a PL/SQL block every hour:  
  
BEGIN  
  DBMS_SCHEDULER.CREATE_JOB (  
    job_name => 'JOB_NAME',  
    job_type => 'PLSQL_BLOCK',  
    job_action => 'BEGIN ...; END;',  
    start_date => SYSTIMESTAMP,  
    repeat_interval=>'FREQUENCY=HOURLY; INTERVAL=1',  
    enabled => TRUE);  
END;  
/
```



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

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

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

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

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

Creating a Job Using a Program

- Use CREATE_PROGRAM to create a program:

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

- Use overloaded CREATE_JOB procedure with its program_name parameter:

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



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

The DBMS_SCHEDULER.CREATE_PROGRAM procedure defines a program that must be assigned a unique name. Creating the program separately for a job enables you to:

- Define the action once and then reuse this action within multiple jobs
- Change the schedule for a job without having to re-create the PL/SQL block
- Change the program executed without changing all the jobs

The program action string specifies a procedure, executable name, or PL/SQL block depending on the value of the program_type parameter, which can be:

- PLSQL_BLOCK to execute an anonymous block or SQL statement
- STORED_PROCEDURE to execute a stored procedure, such as PL/SQL, Java, or C
- EXECUTABLE to execute operating system command-line programs

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

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

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

Creating a Job for a Program with Arguments

- Create a program:

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

- Define an argument:

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

- Create a job specifying the number of arguments:

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



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

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

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

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

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

Creating a Job Using a Schedule

- Use CREATE_SCHEDULE to create a schedule:

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

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

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



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

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

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

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

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

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

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

Setting the Repeat Interval for a Job

- Using a calendaring expression:

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

- Using a PL/SQL expression:

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

Creating a Job Using a Named Program and Schedule

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

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



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

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

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

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

For more information, refer to the online course titled *Oracle Database: Configure and Manage Jobs with the Scheduler*.

Managing Jobs

- Run a job:

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

- Stop a job:

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

- Drop a job even if it is currently running:

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



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

After a job has been created, you can:

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

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

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

Data Dictionary Views

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



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

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

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

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

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

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

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

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

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

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

Summary

In this appendix, you should have learned how to:

- Use the `HTP` package to generate a simple webpage
- Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution



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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

GANG LIU (gangl@baylorhealth.edu) has a non-transferable license
to use this Student Guide.