

Oracle Database 12c: Develop PL/SQL Program Units

Student Guide – Volume II

D80170GC10
Edition 1.0
August 2013
D83231

ORACLE®

Author

Supriya Ananth

**Technical Contributors
and Reviewers**

Wayne Abbott
Anjulaponni Azhagulekshmi
Christopher Burandt
Yanti Chang
Diganta Choudhury
Salome Clement
Laszlo Czinkoczki
Steve Friedberg
Nancy Greenberg
KimSeong Loh
Miyuki Osato
Manish Pawar
Brian Pottle
Swarnapriya Shridhar

Graphic Designer

Seema Bopaiyah

Editors

Raj Kumar
Richard Wallis

Publishers

Glenn Austin
Sumesh Koshy

Copyright © 2013, 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
Suggested Course Agenda	1-5
Lesson Agenda	1-7
The Human Resources (HR) Schema That Is Used in This Course	1-8
Class Account Information	1-9
Appendices Used in This Course	1-10
PL/SQL Development Environments	1-11
What Is Oracle SQL Developer?	1-12
Coding PL/SQL in SQL*Plus	1-13
Enabling Output of a PL/SQL Block	1-14
Lesson Agenda	1-15
Oracle Cloud	1-16
Oracle Cloud Services	1-17
Cloud Deployment Models	1-18
Lesson Agenda	1-19
Oracle SQL and PL/SQL Documentation	1-20
Additional Resources	1-21
Summary	1-22
Practice 1 Overview: Getting Started	1-23

2 Creating Procedures

Objectives	2-2
Lesson Agenda	2-3
Creating a Modularized Subprogram Design	2-4
Creating a Layered Subprogram Design	2-5
Modularizing Development with PL/SQL Blocks	2-6
Anonymous Blocks: Overview	2-7
PL/SQL Run-time Architecture	2-8
What Are PL/SQL Subprograms?	2-9
The Benefits of Using PL/SQL Subprograms	2-10
Differences Between Anonymous Blocks and Subprograms	2-11
Lesson Agenda	2-12

What Are Procedures?	2-13
Creating Procedures: Overview	2-14
Creating Procedures with the SQL CREATE OR REPLACE Statement	2-15
Creating Procedures Using SQL Developer	2-16
Compiling Procedures and Displaying Compilation Errors in SQL Developer	2-17
Correcting Compilation Errors in SQL Developer	2-18
Naming Conventions of PL/SQL Structures Used in This Course	2-19
What Are Parameters and Parameter Modes?	2-20
Formal and Actual Parameters	2-21
Procedural Parameter Modes	2-22
Comparing the Parameter Modes	2-23
Using the IN Parameter Mode: Example	2-24
Using the OUT Parameter Mode: Example	2-25
Using the IN OUT Parameter Mode: Example	2-26
Viewing the OUT Parameters: Using the DBMS_OUTPUT.PUT_LINE Subroutine	2-27
Viewing OUT Parameters: Using SQL*Plus Host Variables	2-28
Available Notations for Passing Actual Parameters	2-29
Passing Actual Parameters: Creating the add_dept Procedure	2-30
Passing Actual Parameters: Examples	2-31
Using the DEFAULT Option for the Parameters	2-32
Calling Procedures	2-34
Calling Procedures Using SQL Developer	2-35
Lesson Agenda	2-36
Handled Exceptions	2-37
Handled Exceptions: Example	2-38
Exceptions Not Handled	2-39
Exceptions Not Handled: Example	2-40
Removing Procedures: Using the DROP SQL Statement or SQL Developer	2-41
Viewing Procedure Information Using the Data Dictionary Views	2-42
Viewing Procedures Information Using SQL Developer	2-43
Lesson Agenda	2-44
PL/SQL Bind Types	2-45
Subprogram with a BOOLEAN Parameter	2-46
Quiz	2-47
Summary	2-48
Practice 2 Overview: Creating, Compiling, and Calling Procedures	2-49

3 Creating Functions and Debugging Subprograms

Objectives	3-2
Lesson Agenda	3-3

Overview of Stored Functions	3-4
Creating Functions	3-5
The Difference Between Procedures and Functions	3-6
Creating and Running Functions: Overview	3-7
Creating and Invoking a Stored Function Using the CREATE FUNCTION Statement: Example	3-8
Using Different Methods for Executing Functions	3-9
Creating and Compiling Functions Using SQL Developer	3-11
Executing Functions Using SQL Developer	3-12
Advantages of User-Defined Functions in SQL Statements	3-13
Using a Function in a SQL Expression: Example	3-14
Calling User-Defined Functions in SQL Statements	3-15
Restrictions When Calling Functions from SQL Expressions	3-16
Controlling Side Effects When Calling Functions from SQL Expressions	3-17
Restrictions on Calling Functions from SQL: Example	3-18
Named and Mixed Notation from SQL	3-19
Named and Mixed Notation from SQL: Example	3-20
Viewing Functions Using Data Dictionary Views	3-21
Viewing Functions Information Using SQL Developer	3-22
Removing Functions: Using the DROP SQL Statement or SQL Developer	3-23
Quiz	3-24
Practice 3-1: Overview	3-25
Lesson Agenda	3-26
Debugging PL/SQL Subprograms Using the SQL Developer Debugger	3-27
Debugging a Subprogram: Overview	3-28
The Procedure or Function Code Editing Tab	3-29
The Procedure or Function Tab Toolbar	3-30
The Debugging – Log Tab Toolbar	3-31
Additional Tabs	3-33
Debugging a Procedure Example: Creating a New emp_list Procedure	3-34
Debugging a Procedure Example: Creating a New get_location Function	3-35
Setting Breakpoints and Compiling emp_list for Debug Mode	3-36
Compiling the get_location Function for Debug Mode	3-37
Debugging emp_list and Entering Values for the PMAXROWS Parameter	3-38
Debugging emp_list: Step Into (F7) the Code	3-39
Viewing the Data	3-41
Modifying the Variables While Debugging the Code	3-42
Debugging emp_list: Step Over the Code	3-43
Debugging emp_list: Step Out of the Code (Shift + F7)	3-44
Debugging emp_list: Run to Cursor (F4)	3-45
Debugging emp_list: Step to End of Method	3-46

Debugging a Subprogram Remotely: Overview 3-47
Practice 3-2 Overview: Introduction to the SQL Developer Debugger 3-48
Summary 3-49

4 Creating Packages

Objectives 4-2
Lesson Agenda 4-3
What Are PL/SQL Packages? 4-4
Advantages of Using Packages 4-5
Components of a PL/SQL Package 4-7
Internal and External Visibility of a Package's Components 4-8
Developing PL/SQL Packages: Overview 4-9
Lesson Agenda 4-10
Creating the Package Specification: Using the CREATE PACKAGE Statement 4-11
Creating the Package Specification: Using SQL Developer 4-12
Creating the Package Body: Using SQL Developer 4-13
Example of a Package Specification: comm_pkg 4-14
Creating the Package Body 4-15
Example of a Package Body: comm_pkg 4-16
Invoking the Package Subprograms: Examples 4-17
Invoking the Package Subprograms: Using SQL Developer 4-18
Creating and Using Bodiless Packages 4-19
Viewing Packages by Using the Data Dictionary 4-20
Viewing Packages by Using SQL Developer 4-21
Removing Packages by Using SQL Developer or the SQL DROP Statement 4-22
Guidelines for Writing Packages 4-23
Quiz 4-24
Summary 4-25
Practice 4 Overview: Creating and Using Packages 4-26

5 Working with Packages

Objectives 5-2
Lesson Agenda 5-3
Overloading Subprograms in PL/SQL 5-4
Overloading Procedures Example: Creating the Package Specification 5-6
Overloading Procedures Example: Creating the Package Body 5-7
Overloading and the STANDARD Package 5-8
Illegal Procedure Reference 5-9
Using Forward Declarations to Solve Illegal Procedure Reference 5-10
Initializing Packages 5-11
Using Package Functions in SQL 5-12

Controlling Side Effects of PL/SQL Subprograms	5-13
Package Function in SQL: Example	5-14
Lesson Agenda	5-15
Persistent State of Packages	5-16
Persistent State of Package Variables: Example	5-18
Persistent State of a Package Cursor: Example	5-19
Executing the CURS_PKG Package	5-21
Using Associative Arrays in Packages	5-22
Quiz	5-23
Summary	5-24
Practice 5 Overview: Working with Packages	5-25

6 Using Oracle-Supplied Packages in Application Development

Objectives	6-2
Lesson Agenda	6-3
Using Oracle-Supplied Packages	6-4
Examples of Some Oracle-Supplied Packages	6-5
Lesson Agenda	6-7
How the DBMS_OUTPUT Package Works	6-8
Using the UTL_FILE Package to Interact with Operating System Files	6-9
Some of the UTL_FILE Procedures and Functions	6-10
File Processing Using the UTL_FILE Package: Overview	6-11
Using the Available Declared Exceptions in the UTL_FILE Package	6-12
FOPEN and IS_OPEN Functions: Example	6-13
Using UTL_FILE: Example	6-16
What Is the UTL_MAIL Package?	6-18
Setting Up and Using the UTL_MAIL: Overview	6-20
Summary of UTL_MAIL Subprograms	6-21
Installing and Using UTL_MAIL	6-22
The SEND Procedure Syntax	6-23
The SEND_ATTACH_RAW Procedure	6-24
Sending Email with a Binary Attachment: Example	6-25
The SEND_ATTACH_VARCHAR2 Procedure	6-27
Sending Email with a Text Attachment: Example	6-28
Quiz	6-30
Summary	6-31
Practice 6 Overview: Using Oracle-Supplied Packages in Application Development	6-32

7 Using Dynamic SQL

Objectives	7-2
Lesson Agenda	7-3

Execution Flow of SQL	7-4
Working with Dynamic SQL	7-5
Using Dynamic SQL	7-6
Native Dynamic SQL (NDS)	7-7
Using the EXECUTE IMMEDIATE Statement	7-8
Available Methods for Using NDS	7-9
Dynamic SQL with a DDL Statement: Examples	7-11
Dynamic SQL with DML Statements	7-12
Dynamic SQL with a Single-Row Query: Example	7-13
Executing a PL/SQL Anonymous Block Dynamically	7-14
Using Native Dynamic SQL to Compile PL/SQL Code	7-15
Lesson Agenda	7-16
Using the DBMS_SQL Package	7-17
Using the DBMS_SQL Package Subprograms	7-18
Using DBMS_SQL with a DML Statement: Deleting Rows	7-20
Using DBMS_SQL with a Parameterized DML Statement	7-22
Quiz	7-23
Summary	7-24
Practice 7 Overview: Using Native Dynamic SQL	7-25

8 Design Considerations for PL/SQL Code

Objectives	8-2
Lesson Agenda	8-3
Standardizing Constants and Exceptions	8-4
Standardizing Exceptions	8-5
Standardizing Exception Handling	8-6
Standardizing Constants	8-7
Local Subprograms	8-8
Definer's Rights Versus Invoker's Rights	8-9
Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER	8-10
Autonomous Transactions	8-11
Features of Autonomous Transactions	8-12
Using Autonomous Transactions: Example	8-13
Lesson Agenda	8-15
Granting Roles to PL/SQL Packages and Stand-Alone Stored Subprograms	8-16
Lesson Agenda	8-17
Using the NOCOPY Hint	8-18
Effects of the NOCOPY Hint	8-19
When Does the PL/SQL Compiler Ignore the NOCOPY Hint?	8-20
Using the PARALLEL_ENABLE Hint	8-21
Using the Cross-Session PL/SQL Function Result Cache	8-22

Enabling Result-Caching for a Function	8-23
Declaring and Defining a Result-Cached Function: Example	8-24
Using the DETERMINISTIC Clause with Functions	8-26
Lesson Agenda	8-27
Using the RETURNING Clause	8-28
Using Bulk Binding	8-29
Bulk Binding: Syntax and Keywords	8-30
Bulk Binding FORALL: Example	8-32
Using BULK COLLECT INTO with Queries	8-34
Using BULK COLLECT INTO with Cursors	8-35
Using BULK COLLECT INTO with a RETURNING Clause	8-36
Using Bulk Binds in Sparse Collections	8-37
Using Bulk Bind with Index Array	8-40
Quiz	8-41
Summary	8-42
Practice 8 Overview: Design Considerations for PL/SQL Code	8-43

9 Creating Triggers

Objectives	9-2
What Are Triggers?	9-3
Defining Triggers	9-4
Trigger Event Types	9-5
Application and Database Triggers	9-6
Business Application Scenarios for Implementing Triggers	9-7
Available Trigger Types	9-8
Trigger Event Types and Body	9-9
Creating DML Triggers Using the CREATE TRIGGER Statement	9-10
Specifying the Trigger Firing (Timing)	9-12
Statement-Level Triggers Versus Row-Level Triggers	9-13
Creating DML Triggers Using SQL Developer	9-14
Trigger-Firing Sequence: Single-Row Manipulation	9-15
Trigger-Firing Sequence: Multirow Manipulation	9-16
Creating a DML Statement Trigger Example: SECURE_EMP	9-17
Testing Trigger SECURE_EMP	9-18
Using Conditional Predicates	9-19
Creating a DML Row Trigger	9-20
Using OLD and NEW Qualifiers	9-21
Using OLD and NEW Qualifiers: Example	9-22
Using the WHEN Clause to Fire a Row Trigger Based on a Condition	9-24
Summary of the Trigger Execution Model	9-25
Implementing an Integrity Constraint with an After Trigger	9-26

INSTEAD OF Triggers	9-27
Creating an INSTEAD OF Trigger: Example	9-28
Creating an INSTEAD OF Trigger to Perform DML on Complex Views	9-29
The Status of a Trigger	9-31
Creating a Disabled Trigger	9-32
Managing Triggers Using the ALTER and DROP SQL Statements	9-33
Managing Triggers Using SQL Developer	9-34
Testing Triggers	9-35
Viewing Trigger Information	9-36
Using USER_TRIGGERS	9-37
Quiz	9-38
Summary	9-39
Practice 9 Overview: Creating Statement and Row Triggers	9-40

10 Creating Compound, DDL, and Event Database Triggers

Objectives	10-2
What Is a Compound Trigger?	10-3
Working with Compound Triggers	10-4
The Benefits of Using a Compound Trigger	10-5
Timing-Point Sections of a Table Compound Trigger	10-6
Compound Trigger Structure for Tables	10-7
Compound Trigger Structure for Views	10-8
Compound Trigger Restrictions	10-9
Trigger Restrictions on Mutating Tables	10-10
Mutating Table: Example	10-11
Using a Compound Trigger to Resolve the Mutating Table Error	10-13
Creating Triggers on DDL Statements	10-15
Creating Database-Event Triggers	10-16
Creating Triggers on System Events	10-17
LOGON and LOGOFF Triggers: Example	10-18
CALL Statements in Triggers	10-19
Benefits of Database-Event Triggers	10-20
System Privileges Required to Manage Triggers	10-21
Guidelines for Designing Triggers	10-22
Quiz	10-23
Summary	10-24
Practice 10 Overview: Creating Compound, DDL, and Event Database Triggers	10-25

11 Using the PL/SQL Compiler

Objectives	11-2
Lesson Agenda	11-3

Initialization Parameters for PL/SQL Compilation	11-4
Using the Initialization Parameters for PL/SQL Compilation	11-5
The Compiler Settings	11-7
Displaying the PL/SQL Initialization Parameters	11-8
Displaying and Setting the PL/SQL Initialization Parameters	11-9
Changing PL/SQL Initialization Parameters: Example	11-10
Lesson Agenda	11-11
Overview of PL/SQL Compile-Time Warnings for Subprograms	11-12
Benefits of Compiler Warnings	11-14
Categories of PL/SQL Compile-Time Warning Messages	11-15
Setting the Warning Messages Levels	11-16
Setting Compiler Warning Levels: Using PLSQL_WARNINGS	11-17
Setting Compiler Warning Levels: Using PLSQL_WARNINGS, Examples	11-18
Setting Compiler Warning Levels: Using PLSQL_WARNINGS in SQL Developer	11-19
Viewing the Current Setting of PLSQL_WARNINGS	11-20
Viewing the Compiler Warnings: Using SQL Developer, SQL*Plus, or Data Dictionary Views	11-22
SQL*Plus Warning Messages: Example	11-23
Guidelines for Using PLSQL_WARNINGS	11-24
Lesson Agenda	11-25
Setting Compiler Warning Levels: Using the DBMS_WARNING Package	11-26
Using the DBMS_WARNING Package Subprograms	11-28
The DBMS_WARNING Procedures: Syntax, Parameters, and Allowed Values	11-29
The DBMS_WARNING Procedures: Example	11-30
The DBMS_WARNING Functions: Syntax, Parameters, and Allowed Values	11-31
The DBMS_WARNING Functions: Example	11-32
Using DBMS_WARNING: Example	11-33
Using the PLW 06009 Warning Message	11-35
The PLW 06009 Warning: Example	11-36
Quiz	11-37
Summary	11-38
Practice 11 Overview: Using the PL/SQL Compiler	11-39

12 Managing Dependencies

Objectives	12-2
Overview of Schema Object Dependencies	12-3
Dependencies	12-4
Direct Local Dependencies	12-5

Querying Direct Object Dependencies: Using the USER_DEPENDENCIES View	12-6
Querying an Object's Status	12-7
Invalidation of Dependent Objects	12-8
Schema Object Change That Invalidates Some Dependents: Example	12-9
Displaying Direct and Indirect Dependencies	12-11
Displaying Dependencies Using the DEPTREE View	12-12
More Precise Dependency Metadata from Oracle Database 11g	12-13
Fine-Grained Dependency Management	12-14
Fine-Grained Dependency Management: Example 1	12-15
Fine-Grained Dependency Management: Example 2	12-17
Changes to Synonym Dependencies	12-18
Maintaining Valid PL/SQL Program Units and Views	12-19
Another Scenario of Local Dependencies	12-20
Guidelines for Reducing Invalidation	12-21
Object Revalidation	12-22
Remote Dependencies	12-23
Concepts of Remote Dependencies	12-24
Setting the REMOTE_DEPENDENCIES_MODE Parameter	12-25
Remote Procedure B Compiles at 8:00 AM	12-26
Local Procedure A Compiles at 9:00 AM	12-27
Execute Procedure A	12-28
Remote Procedure B Recompiled at 11:00 AM	12-29
Execute Procedure A	12-30
Signature Mode	12-31
Recompiling a PL/SQL Program Unit	12-32
Unsuccessful Recompilation	12-33
Successful Recompilation	12-34
Recompiling Procedures	12-35
Packages and Dependencies: Subprogram References the Package	12-36
Packages and Dependencies: Package Subprogram References Procedure	12-37
Quiz	12-38
Summary	12-39
Practice 12 Overview: Managing Dependencies in Your Schema	12-40

A Table Descriptions

B Using SQL Developer

Objectives	B-2
What Is Oracle SQL Developer?	B-3
Specifications of SQL Developer	B-4

SQL Developer 3.2 Interface	B-5
Creating a Database Connection	B-7
Browsing Database Objects	B-10
Displaying the Table Structure	B-11
Browsing Files	B-12
Creating a Schema Object	B-13
Creating a New Table: Example	B-14
Using the SQL Worksheet	B-15
Executing SQL Statements	B-19
Saving SQL Scripts	B-20
Executing Saved Script Files: Method 1	B-21
Executing Saved Script Files: Method 2	B-22
Formatting the SQL Code	B-23
Using Snippets	B-24
Using Snippets: Example	B-25
Using Recycle Bin	B-26
Debugging Procedures and Functions	B-27
Database Reporting	B-28
Creating a User-Defined Report	B-29
Search Engines and External Tools	B-30
Setting Preferences	B-31
Resetting the SQL Developer Layout	B-33
Data Modeler in SQL Developer	B-34
Summary	B-35

C Using SQL*Plus

Objectives	C-2
SQL and SQL*Plus Interaction	C-3
SQL Statements Versus SQL*Plus Commands	C-4
Overview of SQL*Plus	C-5
Logging In to SQL*Plus	C-6
Displaying the Table Structure	C-7
SQL*Plus Editing Commands	C-9
Using LIST, n, and APPEND	C-11
Using the CHANGE Command	C-12
SQL*Plus File Commands	C-13
Using the SAVE, START Commands	C-14
SERVERROUTPUT Command	C-15
Using the SQL*Plus SPOOL Command	C-16
Using the AUTOTRACE Command	C-17
Summary	C-18

D REF Cursors

- Cursor Variables D-2
- Using Cursor Variables D-3
- Defining REF CURSOR Types D-4
- Using the OPEN-FOR, FETCH, and CLOSE Statements D-7
- Example of Fetching D-10

E Commonly Used SQL Commands

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

Right Outer Join E-35
Full Outer Join E-36
Self-Join: Example E-37
Cross Join E-38
Summary E-39

F Managing PL/SQL Code

Objectives F-2
Agenda F-3
Conditional Compilation F-4
How Does Conditional Compilation Work? F-5
Using Selection Directives F-6
Using Predefined and User-Defined Inquiry Directives F-7
The PLSQL_CCFLAGS Parameter and the Inquiry Directive F-8
Displaying the PLSQL_CCFLAGS Initialization Parameter Setting F-9
The PLSQL_CCFLAGS Parameter and the Inquiry Directive: Example F-10
Using Conditional Compilation Error Directives to Raise User-Defined Errors F-11
Using Static Expressions with Conditional Compilation F-12
DBMS_DB_VERSION Package: Boolean Constants F-13
DBMS_DB_VERSION Package Constants F-14
Using Conditional Compilation with Database Versions: Example F-15
Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text F-17
Agenda F-18
Obfuscation F-19
Benefits of Obfuscating F-20
What's New in Dynamic Obfuscating Since Oracle 10g? F-21
Nonobfuscated PL/SQL Code: Example F-22
Obfuscated PL/SQL Code: Example F-23
Dynamic Obfuscation: Example F-24
PL/SQL Wrapper Utility F-25
Running the PL/SQL Wrapper Utility F-26
Results of Wrapping F-27
Guidelines for Wrapping F-28
DBMS_DDL Package Versus wrap Utility F-29
Summary F-30

G Review of PL/SQL

Objectives G-2
Block Structure for Anonymous PL/SQL Blocks G-3
Declaring PL/SQL Variables G-4
Declaring Variables with the %TYPE Attribute: Examples G-5

Creating a PL/SQL Record	G-6
%ROWTYPE Attribute: Examples	G-7
Creating a PL/SQL Table	G-8
SELECT Statements in PL/SQL: Example	G-9
Inserting Data: Example	G-10
Updating Data: Example	G-11
Deleting Data: Example	G-12
Controlling Transactions with the COMMIT and ROLLBACK Statements	G-13
IF, THEN, and ELSIF Statements: Example	G-14
Basic Loop: Example	G-16
FOR Loop: Example	G-17
WHILE Loop: Example	G-18
SQL Implicit Cursor Attributes	G-19
Controlling Explicit Cursors	G-20
Controlling Explicit Cursors: Declaring the Cursor	G-21
Controlling Explicit Cursors: Opening the Cursor	G-22
Controlling Explicit Cursors: Fetching Data from the Cursor	G-23
Controlling Explicit Cursors: Closing the Cursor	G-24
Explicit Cursor Attributes	G-25
Cursor FOR Loops: Example	G-26
FOR UPDATE Clause: Example	G-27
WHERE CURRENT OF Clause: Example	G-28
Trapping Predefined Oracle Server Errors	G-29
Trapping Predefined Oracle Server Errors: Example	G-30
Non-Predefined Error	G-31
User-Defined Exceptions: Example	G-32
RAISE_APPLICATION_ERROR Procedure	G-33
Summary	G-35

H Studies for Implementing Triggers

Objectives	H-2
Controlling Security Within the Server	H-3
Controlling Security with a Database Trigger	H-4
Enforcing Data Integrity Within the Server	H-5
Protecting Data Integrity with a Trigger	H-6
Enforcing Referential Integrity Within the Server	H-7
Protecting Referential Integrity with a Trigger	H-8
Replicating a Table Within the Server	H-9
Replicating a Table with a Trigger	H-10
Computing Derived Data Within the Server	H-11
Computing Derived Values with a Trigger	H-12

Logging Events with a Trigger H-13
Summary H-15

I Using the DBMS_SCHEDULER and HTP Packages

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

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only

11

Using the PL/SQL Compiler

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

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

Note that the content in this chapter is specific to 11g and later releases. Some of the examples listed may or may not work in a 10g environment.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

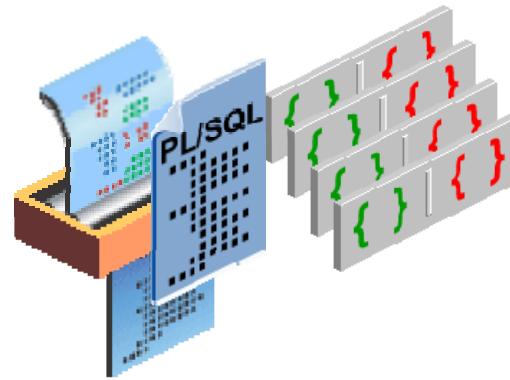
- Using the `PLSQL_CODE_TYPE` and `PLSQL_OPTIMIZE_LEVEL` PL/SQL compilation initialization parameters
- Using the PL/SQL compile-time warnings:
 - Using the `PLSQL_WARNING` initialization parameter
 - Using the `DBMS_WARNING` package subprograms



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Initialization Parameters for PL/SQL Compilation

- PLSQL_CODE_TYPE
- PLSQL_OPTIMIZE_LEVEL
- PLSQL_CCFLAGS
- PLSQL_WARNINGS



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In releases before Oracle Database 10g, the PL/SQL compiler translated your code to machine code without applying many changes for performance. Oracle Database 11g onwards, PL/SQL uses an optimizing compiler that can rearrange code for better performance. You do not need to do anything to get the benefits of this new optimizer; it is enabled by default.

Note

- The PLSQL_CCFLAGS initialization parameter is covered in the lesson titled “Managing PL/SQL Code.”
- The PLSQL_WARNINGS initialization parameter is covered later in this lesson.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

The `PLSQL_CODE_TYPE` Parameter

This parameter specifies the compilation mode for PL/SQL library units. If you choose `INTERPRETED`, PL/SQL library units will be compiled to PL/SQL bytecode format. Such modules are executed by the PL/SQL interpreter engine. If you choose `NATIVE`, PL/SQL library units (with the possible exception of top-level anonymous PL/SQL blocks) will be compiled to native (machine) code. Such modules will be executed natively without incurring any interpreter overhead. 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. From Oracle Database 11g, native compilation is easier and more integrated, with fewer initialization parameters to set.

In rare cases, if the overhead of the optimizer makes compilation of very large applications take too long, you might lower the optimization level by setting the initialization parameter `PLSQL_OPTIMIZE_LEVEL` to 1 instead of its default value 2. In even rarer cases, you might see a change in exception behavior—either an exception that is not raised at all or one that is raised earlier than expected. Setting `PLSQL_OPTIMIZE_LEVEL` to 0 prevents the code from being rearranged at all.

The PLSQL_OPTIMIZE_LEVEL Parameter

This parameter specifies the optimization level that will be used to compile PL/SQL library units. The higher the setting of this parameter, the more effort the compiler makes to optimize PL/SQL library units. The available values are (0, 1, and 2 were available starting with Oracle 10g release 2):

0: Maintains the evaluation order and hence the pattern of side effects, exceptions, and package initializations of Oracle9*i* and earlier releases. Also removes the new semantic identity of `BINARY_INTEGER` and `PLS_INTEGER` and restores the earlier rules for the evaluation of integer expressions. Although code will run somewhat faster than it did in Oracle9*i*, use of level 0 will forfeit most of the performance gains of PL/SQL starting with Oracle Database 10g.

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

2: Applies a wide range of modern optimization techniques beyond those of level 1 including changes which may move source code relatively far from its original location

3: This value was introduced in Oracle Database 11g. It applies a wide range of optimization techniques beyond those of level 2, automatically including techniques not specifically requested. This enables procedure inlining, which is an optimization process that replaces procedure calls with a copy of the body of the procedure to be called. The copied procedure almost always runs faster than the original call. To allow subprogram inlining, either accept the default value of the `PLSQL_OPTIMIZE_LEVEL` initialization parameter (which is 2) or set it to 3. With `PLSQL_OPTIMIZE_LEVEL = 2`, you must specify each subprogram to be inlined. With `PLSQL_OPTIMIZE_LEVEL = 3`, the PL/SQL compiler seeks opportunities to inline subprograms beyond those that you specify.

Note: For additional information about inlining, refer to *Oracle Database PL/SQL Language Reference* and the *Oracle Database Advanced PL/SQL* instructor-led course.

Generally, setting this parameter to 2 pays off in terms of better execution performance. If, however, the compiler runs slowly on a particular source module or if optimization does not make sense for some reason (for example, during rapid turnaround development), then setting this parameter to 1 results in almost as good a compilation with less use of compile-time resources. The value of this parameter is stored persistently with the library unit.

Note

The `PLSQL_CODE_TYPE` parameter in Oracle Database 10g replaced the following obsolete parameters:

- `PLSQL_NATIVE_C_COMPILER`
- `PLSQL_NATIVE_MAKE_FILE_NAME`
- `PLSQL_NATIVE_MAKE.Utility`
- `PLSQL_NATIVE_LINKER`

The `PLSQL_DEBUG` parameter is deprecated from Oracle Database 11g onwards. The parameter `PLSQL_DEBUG` no longer controls the generation of debugging information by the PL/SQL compiler; debugging information is always generated and no special parameter is needed.

The Compiler Settings

Compiler Option	Description
PLSQL_CODE_TYPE	Specifies the compilation mode for PL/SQL library units.
PLSQL_OPTIMIZE_LEVEL	Specifies the optimization level to be used to compile PL/SQL library units.
PLSQL_WARNINGS	Enables or disables the reporting of warning messages by the PL/SQL compiler.
PLSQL_CCFLAGS	Controls conditional compilation of each PL/SQL library unit independently.

In general, for the fastest performance, use the following setting:

```
PLSQL_CODE_TYPE = NATIVE  
PLSQL_OPTIMIZE_LEVEL = 2
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The new compiler increases the performance of PL/SQL code and allows it to execute approximately two times faster than an Oracle8i database and 1.5 times to 1.75 times as fast as Oracle9i Database Release 2.

To get the fastest performance, the compiler setting must be:

```
PLSQL_CODE_TYPE = NATIVE  
PLSQL_OPTIMIZE_LEVEL = 2
```

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
```

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)



Copyright © 2013, 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, or TYPE BODY.

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 (introduced from Oracle Database 11g)

Displaying and Setting the PL/SQL Initialization Parameters

```
SELECT name, type, plsql_code_type AS CODE_TYPE,  
       plsql_optimize_level AS OPT_LVL  
  FROM user_plsql_object_settings;
```

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	INTERPRETED	1

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Note

- For additional information about the ALTER SYSTEM or ALTER SESSION statements, refer to *Oracle Database SQL Reference*.
- The DBA_STORED_SETTINGS data dictionary view family is deprecated in Oracle Database 10g and is replaced with the DBA_PLSQL_OBJECT_SETTINGS data dictionary view family.

Changing PL/SQL Initialization Parameters: Example

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

session SET altered.
session SET altered.

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

@code_11_09_sa.sql

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 © 2013, Oracle and/or its affiliates. All rights reserved.

To change a compiled PL/SQL object from interpreted code type to native code type, you must first set the `PLSQL_CODE_TYPE` parameter to `NATIVE` (optionally set the other parameters) and then, recompile the program. To enforce native compilation to all PL/SQL code, you must recompile each one. Scripts (in the `rdmbs/admin` directory) are provided for you to achieve conversion to full native compilation (`dbmsupgnv.sql`) or full interpreted compilation (`dbmsupgini.sql`). The `add_job_history` procedure is created as follows:

```
CREATE OR REPLACE PROCEDURE add_job_history
( p_emp_id          job_history.employee_id%type
, p_start_date      job_history.start_date%type
, p_end_date        job_history.end_date%type
, p_job_id          job_history.job_id%type
, p_department_id   job_history.department_id%type )
IS
BEGIN
  INSERT INTO job_history (employee_id, start_date,
                           end_date, job_id, department_id)
  VALUES(p_emp_id, p_start_date, p_end_date,
         p_job_id, p_department_id);
END add_job_history;
/
```

Lesson Agenda

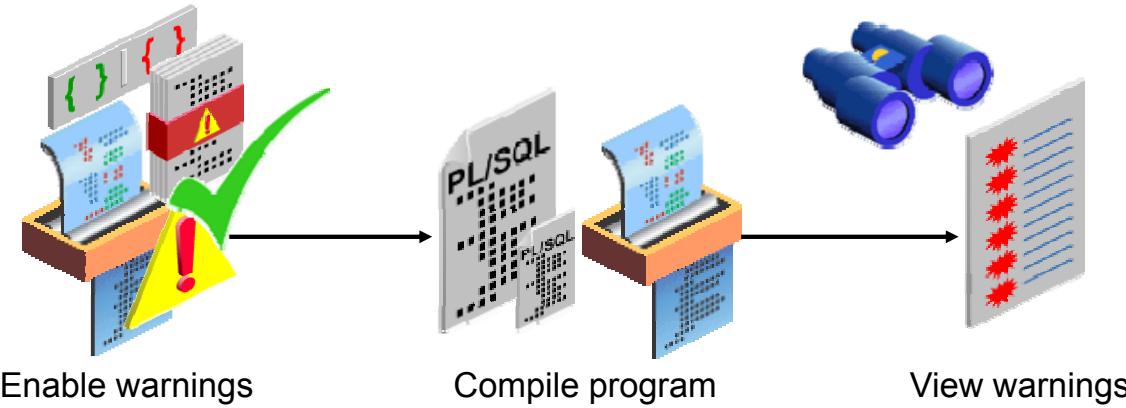
- Using the `PLSQL_CODE_TYPE` and `PLSQL_OPTIMIZE_LEVEL` PL/SQL compilation initialization parameters
- Using the PL/SQL compile-time warnings:
 - Using the `PLSQL_WARNING` initialization parameter
 - Using the `DBMS_WARNING` package subprograms



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Overview of PL/SQL Compile-Time Warnings for Subprograms

Starting with Oracle 10g, the PL/SQL compiler has been enhanced to produce warnings for subprograms.



Copyright © 2013, 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.

In releases prior to Oracle Database 10g, compiling a PL/SQL program had two possible outcomes:

- Success, producing a valid compiled unit
- Failure, with compilation errors indicating that the program had either syntax or semantic errors

However, even when compilation of a program was successful, the program may have violated recommended best practices or could have been coded to be more efficient. Oracle Database 10g introduced a new ease-of-use feature that allows the PL/SQL compiler to communicate warning messages in these situations. Compiler warnings allow developers to avoid common coding pitfalls, thus improving productivity.

PL/SQL supports passing of `IN OUT` and `OUT` parameters by value or by reference through the `NOCOPY` compiler hint. Passing parameters by value is inherently less efficient because it involves making multiple copies of the data. From Oracle Database 11g release, the compiler automatically detects and recommends the use of the `NOCOPY` hint, where the parameter types are large object, record, or collection types.

With the PL/SQL compiler-warning feature, compiling a PL/SQL program could have additional 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 © 2013, Oracle and/or its affiliates. All rights reserved.

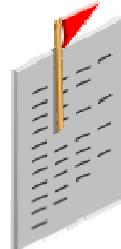
Using compiler warnings can help you to:

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

Note

- You can enable checking for certain warning conditions when these conditions are not serious enough to produce an error and keep you from compiling a subprogram.
- Warning messages can be issued during compilation of PL/SQL subprograms; anonymous blocks do not produce any warnings.
- All PL/SQL warning messages use the prefix PLW.

Categories of PL/SQL Compile-Time Warning Messages



SEVERE



PERFORMANCE



INFORMATIONAL



ALL

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

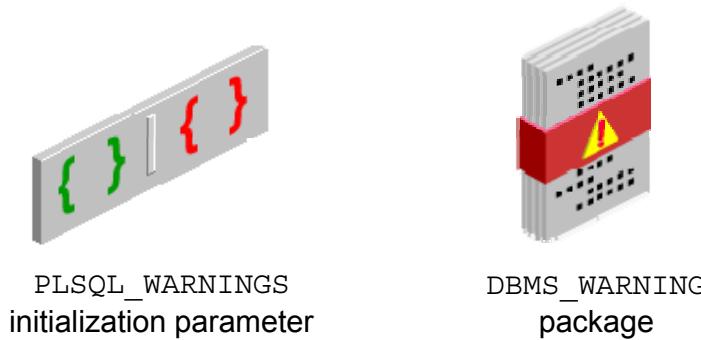
PL/SQL warning messages are divided into categories, so that you can suppress or display groups of similar warnings during compilation. The categories are:

- **SEVERE:** Messages for conditions that may cause unexpected behavior or wrong results, such as aliasing problems with parameters
- **PERFORMANCE:** Messages for conditions that may cause performance problems, such as passing a `VARCHAR2` value to a `NUMBER` column in an `INSERT` statement
- **INFORMATIONAL:** Messages for conditions that do not have an effect on performance or correctness, but that you may want to change to make the code more maintainable, such as unreachable code that can never be executed

Setting the Warning Messages Levels

You can set warning levels using one of the following methods:

- Declaratively:
 - Using the `PLSQL_WARNINGS` initialization parameter
- Programmatically:
 - Using the `DBMS_WARNING` package



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can set the compiler warning messages levels using one of the following methods:

Using the `PLSQL_WARNINGS` Initialization Parameter

The `PLSQL_WARNINGS` setting enables or disables the reporting of warning messages 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.
- Any valid combination of the preceding

The keyword `All` is a shorthand way to refer to all warning messages: `SEVERE`, `PERFORMANCE`, and `INFORMATIONAL`.

Using 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. This package is covered later in this lesson.

Setting Compiler Warning Levels: Using PLSQL_WARNINGS

```
ALTER [SESSION|SYSTEM]
PLSQL_WARNINGS = 'value_clause1' [, 'value_clause2'] ...
```

```
value_clause = Qualifier Value : Modifier Value
```

```
Qualifier Value = { ENABLE | DISABLE | ERROR }

Modifier Value =
{ ALL | SEVERE | INFORMATIONAL | PERFORMANCE |
{ integer | (integer [, integer] ...) } }
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Modifying Compiler Warning Settings

The parameter value comprises a comma-separated list of quoted qualifier and modifier keywords, where the keywords are separated by colons. The qualifier values are: **ENABLE**, **DISABLE**, and **ERROR**. The modifier value **ALL** applies to all warning messages. **SEVERE**, **INFORMATIONAL**, and **PERFORMANCE** apply to messages in their own category, and an integer list for specific warning messages.

Possible values for **ENABLE**, **DISABLE**, and **ERROR**:

- ALL
- SEVERE
- INFORMATIONAL
- PERFORMANCE
- numeric_value

Values for **numeric_value** are in:

- Range 5000-5999 for severe
- Range 6000-6249 for informational
- Range 7000-7249 for performance

Setting Compiler Warning Levels: Using PLSQL_WARNINGS, Examples

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

session SET altered.

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

session SET altered.

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

session SET altered.



Copyright © 2013, 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.

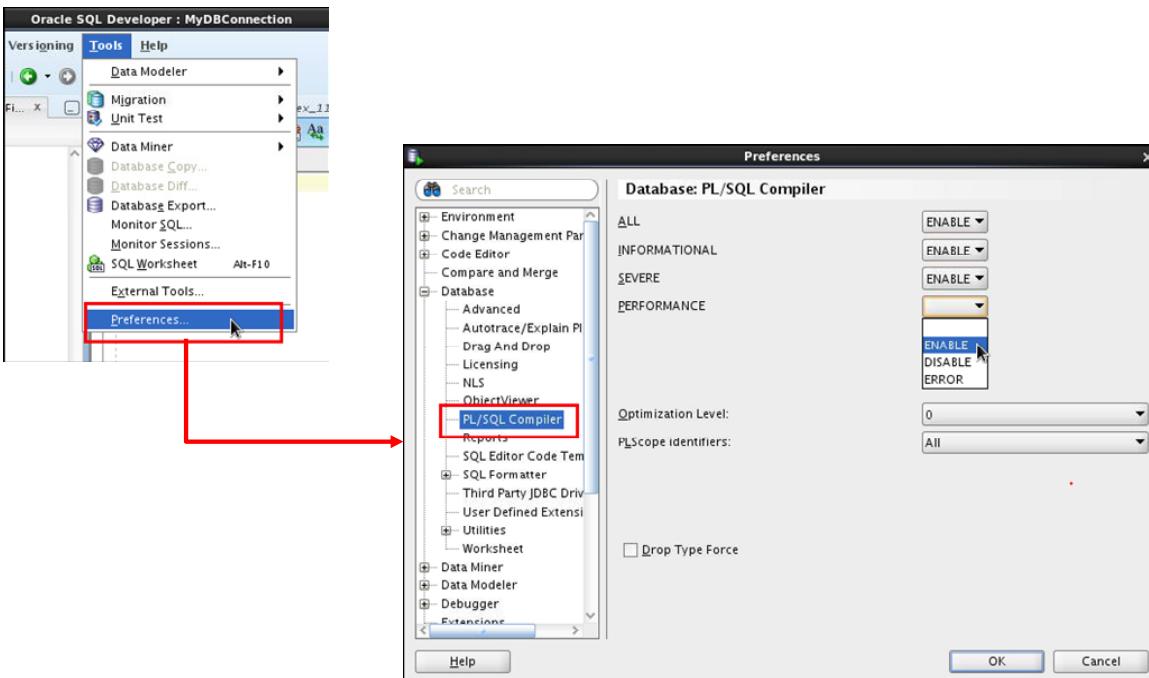
Example 2

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

Example 3

You can also 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, including '`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.

Setting Compiler Warning Levels: Using PLSQL_WARNINGS in SQL Developer



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

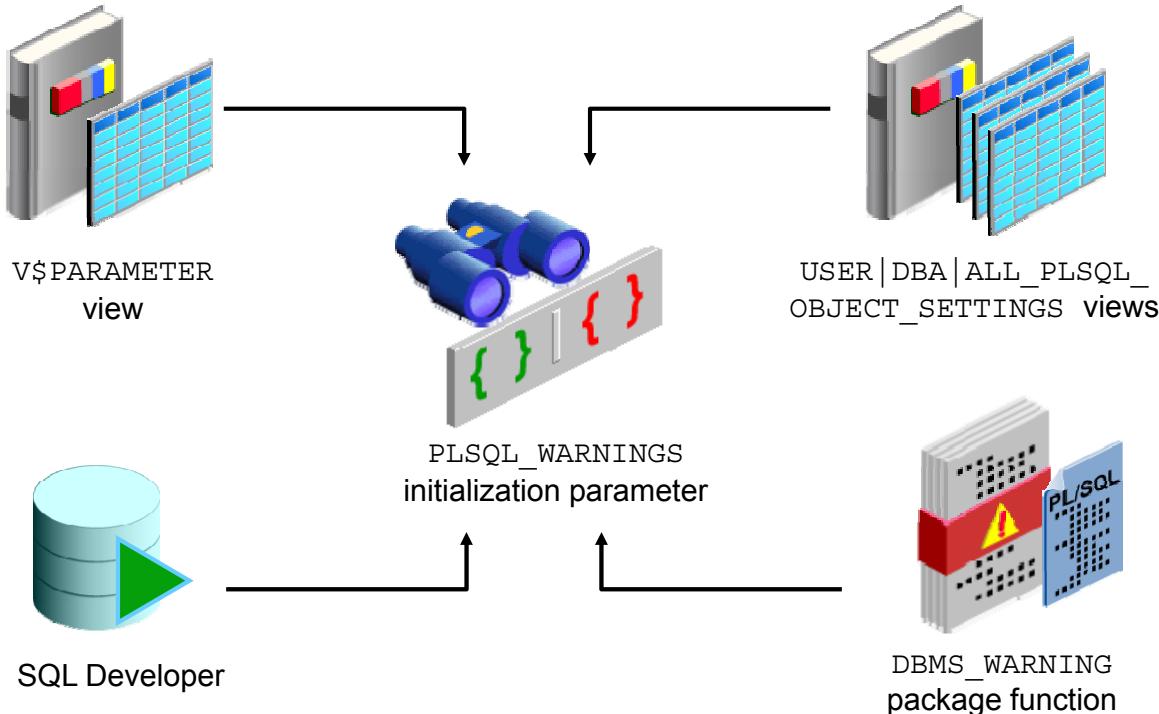
The PL/SQL Compiler pane specifies options for compilation of PL/SQL subprograms. If the **Generate PL/SQL Debug Information** check box is selected, PL/SQL debug information is included in the compiled code; if this option is not selected, this debug information is not included. The ability to stop on individual code lines and debugger access to variables are allowed only in code compiled with debug information generated.

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



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Viewing the Current Value of the PLSQL_WARNINGS Parameter

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

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

Session altered.

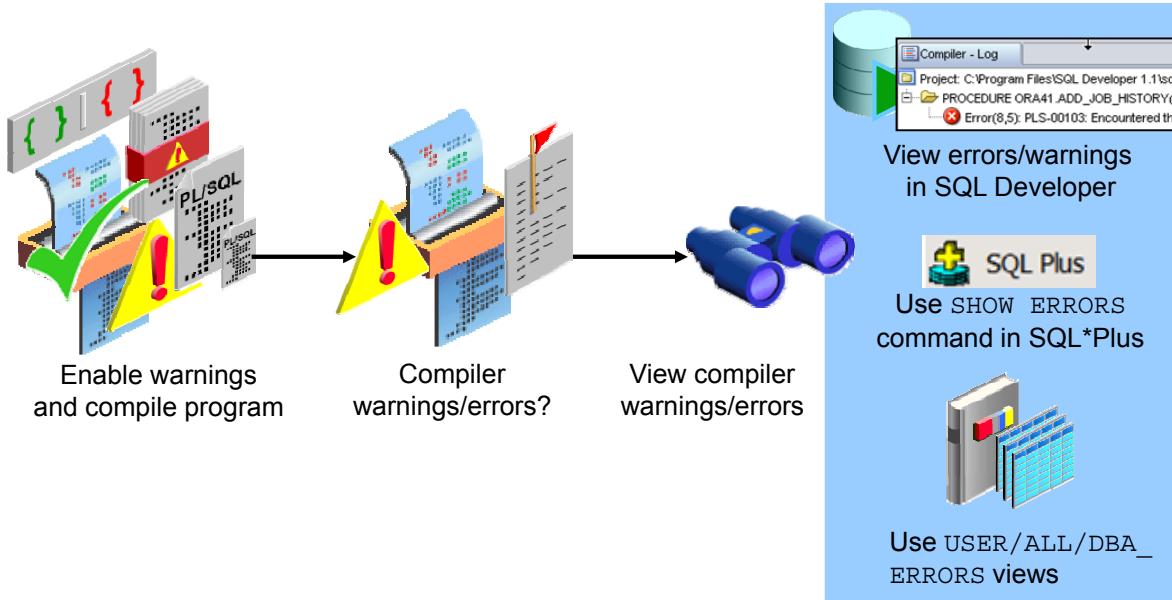
SELECT value FROM v$parameter WHERE name='plsql_warnings';
VALUE
-----
ENABLE:ALL
```

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;
/
```

```
anonymous block completed
ENABLE:ALL
```

Viewing the Compiler Warnings: Using SQL Developer, SQL*Plus, or Data Dictionary Views



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Viewing the Compiler Warnings

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 with the PERFORMANCE, INFORMATIONAL, or SEVERE modifiers. There is no differentiation between the three. You must enable the compiler warnings before compiling the program. You can display the compiler warning messages using one 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 the 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.

```
SHOW ERRORS;
Errors for PROCEDURE BAD_PROC:

LINE/COL      ERROR
-----
6/24          PLW-07203: parameter 'p_out' may benefit
from use of the NOCOPY compiler hint
```



Copyright © 2013, 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 the 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 using the `USER_ALL_DBAs_ERRORS` data dictionary views.

Guidelines for Using PLSQL_WARNINGS

- 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 © 2013, Oracle and/or its affiliates. All rights reserved.

As already stated, the PLSQL_WARNINGS parameter can be set at the session level or the system level.

The settings for the PLSQL_WARNINGS parameter are stored along with each compiled subprogram. If you recompile the subprogram with a CREATE OR REPLACE statement, the current settings for that session are used. If you recompile the subprogram with an ALTER . . . COMPILE statement, then the current session setting is used unless you specify the REUSE SETTINGS clause in the statement, which uses the original setting that is stored with the subprogram.

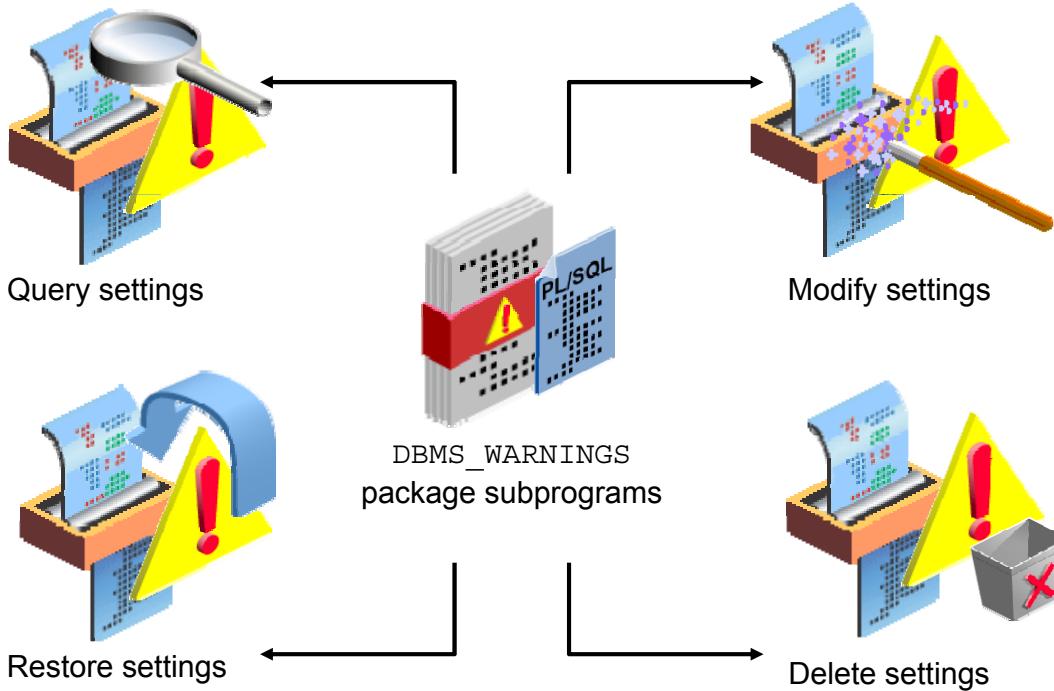
Lesson Agenda

- Using the `PLSQL_CODE_TYPE` and `PLSQL_OPTIMIZE_LEVEL` PL/SQL Compilation Initialization Parameters
- Using the PL/SQL Compile-Time Warnings:
 - Using the `PLSQL_WARNING` Initialization Parameter
 - Using the `DBMS_WARNING` Package subprograms



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Setting Compiler Warning Levels: Using the DBMS_WARNING Package



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Use the DBMS_WARNING package to programmatically manipulate the behavior of current system or session PL/SQL warning settings. 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.

The DBMS_WARNING package is valuable if you are writing a development environment that compiles PL/SQL subprograms. Using the package interface routines, you can control PL/SQL warning messages programmatically to suit your requirements.

Overview of PL/SQL Compile-Time Warnings for Subprograms: Example

Assume that you write some code to compile PL/SQL code. You know that the compiler issues performance warnings when passing collection variables as OUT or IN OUT parameters without specifying the NOCOPY hint. The general environment that calls your compilation utility may or may not have appropriate warning-level settings. In any case, your business rules indicate that the calling environment set must be preserved and that your compilation process should suppress the warnings. By calling subprograms in the DBMS_WARNING package, you can detect the current warning settings, change the settings to suit your business requirements, and restore the original settings when your processing has completed.

When you use the ALTER SESSION or ALTER SYSTEM command to set the PLSQL_WARNINGS parameter, the new value specified completely replaces the previous value. A new package, DBMS_WARNING, is available from Oracle Database 10g onwards that has interfaces to query and incrementally change the setting for the PLSQL_WARNINGS parameter and make it more specific to your requirements.

The DBMS_WARNING package can be used to change the PLSQL_WARNINGS parameter incrementally, so that you can set the warnings that you want to set, without having to work out how to preserve the values of any warnings not of direct interest to you. For example, the DBA may only enable severe warnings for the entire database in the initialization parameter file, but a developer who is testing new code may want to view specific performance and informational messages. The developer can then use the DBMS_WARNING package to incrementally add the specific warnings that he or she wishes to see. This allows the developer to view the messages that he or she wants to see without replacing the DBA's settings.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

Using the DBMS_WARNING Subprograms

The following is a list of the DBMS_WARNING subprograms:

- ADD_WARNING_SETTING_CAT: Modifies the current session or system warning settings of the warning_category previously supplied
- ADD_WARNING_SETTING_NUM: Modifies the current session or system warning settings of the warning_number previously supplied
- GET_CATEGORY: Returns the category name, given the message number
- GET_WARNING_SETTING_CAT: Returns the specific warning category in the session
- GET_WARNING_SETTING_NUM: Returns the specific warning number in the session
- GET_WARNING_SETTING_STRING: Returns the entire warning string for the current session
- SET_WARNING_SETTING_STRING: Replaces previous settings with the new value

Note: For additional information about the preceding subprograms, refer to *Oracle Database PL/SQL Packages and Types Reference* or *Oracle Database PL/SQL Packages and Types Reference* (as applicable to the version you are using).

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 © 2013, 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.

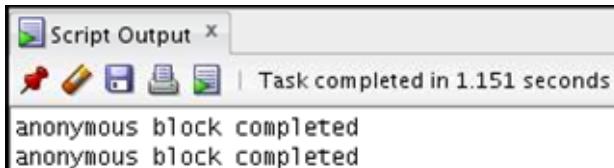
The `scope` specifies whether the changes are being performed in the session context or the system context. The allowed values are SESSION or SYSTEM. Using SYSTEM requires the ALTER SYSTEM privilege.

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');
```



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using DBMS_WARNING Procedures: Example

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 © 2013, 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 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) ;
```

```
anonymous block 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)) ;
```

```
anonymous block completed  
PERFORMANCE
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Note

The message numbers must be specified as positive integers, because the data type for the GET_CATEGORY parameter is PLS_INTEGER (allowing positive integer values).

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 © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the `compile_code` procedure is designed to compile a named PL/SQL package. The code suppresses the PERFORMANCE category warnings. The calling environment's warning settings must be restored after the compilation is performed. The code does not know what the calling environment warning settings are; it uses the `GET_WARNING_SETTING_STRING` function to save the current setting. This value is used to restore the calling environment setting using the `DBMS_WARNING.SET_WARNING_SETTING_STRING` procedure in the last line of the example code. Before compiling the package using Native Dynamic SQL, the `compile_code` procedure alters the current session-warning level by disabling warnings for the PERFORMANCE category. The code also prints the original, modified, and the restored warning settings.

Using DBMS_WARNING: Example

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

```
anonymous block completed
```

```
@/home/oracle/labs/plpu/code_ex/code_ex_scripts/code_11_33_s.sql
```

```
PROCEDURE compile_code compiled
```

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

```
anonymous block completed
Current warning settings: ENABLE:ALL
Modified warning settings: ENABLE:INFORMATIONAL,DISABLE:PERFORMANCE,ENABLE:SEVERE
Restored warning settings: ENABLE:ALL
```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the example provided in the previous slide is tested. First, enable all compiler warnings. Next, run the script on the previous page. Finally, call the `compile_code` procedure and pass it an existing package name, `EMP_PKG`, as a parameter.

Using the PLW 06009 Warning Message

- The PLW warning indicates that the OTHERS handler of your PL/SQL subroutine can exit without executing:
 - Some form of RAISE, or
 - A call to the standard procedure RAISE_APPLICATION_ERROR
- A good programming practice suggests that OTHERS handlers must always pass an exception upward.



ORACLE

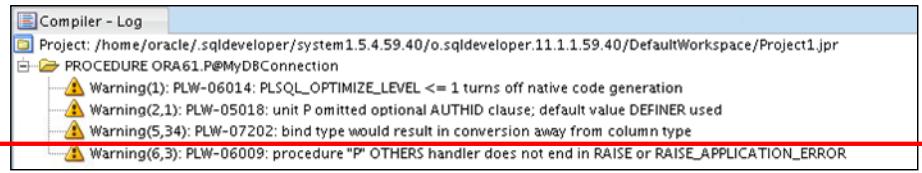
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

As a good programming practice, you should have your OTHERS exception handler pass the exception upward to the calling subroutine. If you fail to add this functionality, you run the risk of having exceptions go unnoticed. To avoid this flaw in your code, you can turn on warnings for your session and recompile the code that you want to verify. If the OTHERS handler does not handle the exception, the PLW 06009 warning will inform you. Please note that this warning is specific to Oracle Database 11g and further releases.

Note: PLW 06009 is not the only new warning message in Oracle Database 11g. For a complete list of all PLW warnings in Oracle Database 11g, see *Oracle Database Error Messages 11g Release 2 (11.2)*.

The PLW 06009 Warning: Example

```
CREATE OR REPLACE PROCEDURE p(i IN VARCHAR2)
IS
BEGIN
  INSERT INTO t(col_a) VALUES (i);
EXCEPTION
  WHEN OTHERS THEN null;
END p;
/
ALTER PROCEDURE P COMPILE
PLSQL_warnings = 'enable:all' REUSE SETTINGS;
```



```
SELECT *
FROM user_errors
WHERE name = 'P'
```

NAME	TYPE	SEQUENCE	LINE	POSITION	TEXT	ATTRIBUTE	MESSAGE_NUMBER
1 P	PROCEDURE	1	4	34	PLW-07202: bind type would result in conversion away from column type	WARNING	7202
2 P	PROCEDURE	2	1	1	PLW-05018: unit P omitted optional AUTHID clause; default value DEFINER used	WARNING	5018
3 P	PROCEDURE	3	0	0	PLW-06015: parameter PLSQL_DEBUG is deprecated; use PLSQL_OPTIMIZE_LEVEL = 1	WARNING	6015
4 P	PROCEDURE	4	0	0	PLW-06014: PLSQL_OPTIMIZE_LEVEL <= 1 turns off native code generation	WARNING	6014
5 P	PROCEDURE	5	5	3	PLW-06009: procedure \"P\" OTHERS handler does not end in RAISE or RAISE_APPLICATION_ERROR	WARNING	6009

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

After running the first code example in the slide and after compiling the procedure using the Object Navigation tree, the **Compiler – Log** tab displays the PLW-06009 warning.

You can also use the `user_error` data dictionary view to display the error.
The definition of table `t` that is used in the slide example is as follows:

```
CREATE TABLE t (col_a NUMBER);
```

Quiz

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

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, c, d

PL/SQL warning messages are divided into categories, so that you can suppress or display groups of similar warnings during compilation. The categories are:

- SEVERE: Messages for conditions that may cause unexpected behavior or wrong results, such as aliasing problems with parameters.
- PERFORMANCE: Messages for conditions that may cause performance problems, such as passing a VARCHAR2 value to a NUMBER column in an INSERT statement.
- INFORMATIONAL: Messages for conditions that do not have an effect on performance or correctness, but that you may want to change to make the code more maintainable, such as unreachable code that can never be executed.
- ALL: Displays all categories.

Summary

In this lesson, you should have learned how to:

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Practice 11 Overview: Using the 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you display the compiler initialization parameters. You then enable native compilation for your session and compile a procedure. You then suppress all compiler warnings categories and then restore the original session-warning settings. Finally, you identify the categories for some compiler-warning message numbers.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only

12

Managing Dependencies

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This lesson introduces you to object dependencies and implicit and explicit recompilation of invalid objects.

Overview of Schema Object Dependencies

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

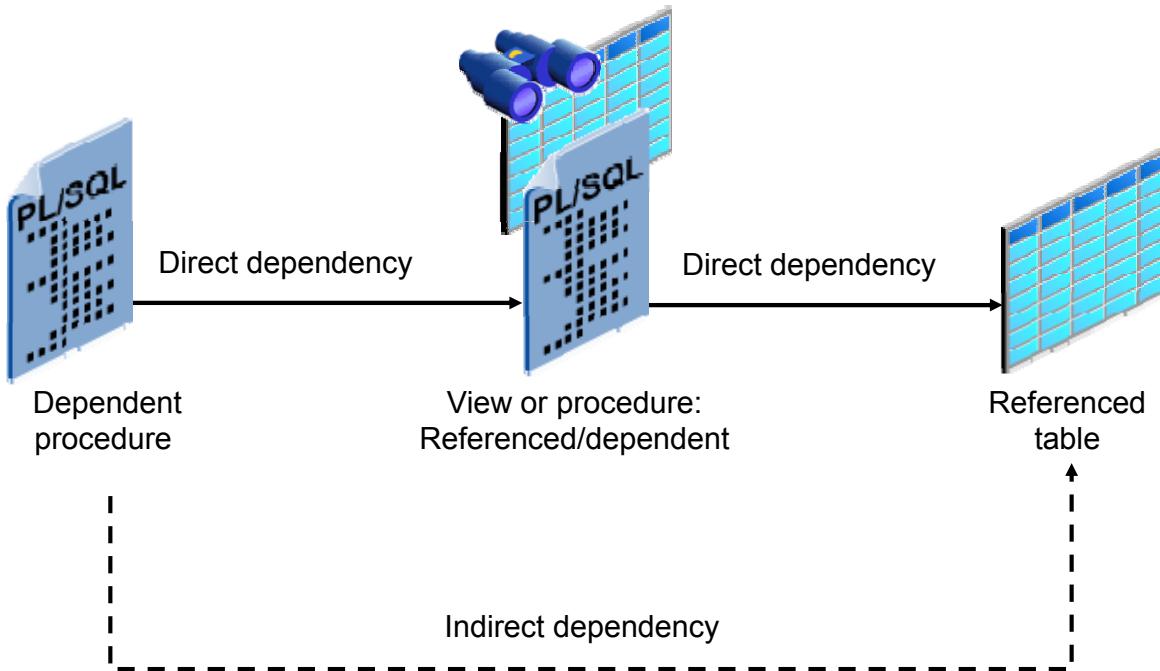
Dependent and Referenced Objects

Some types of schema objects can reference other objects in their definitions. For example, a view is defined by a query that references tables 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.

Dependencies



ORACLE

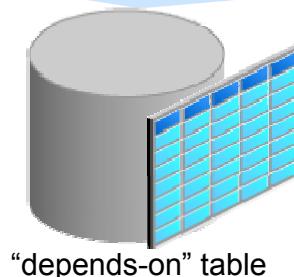
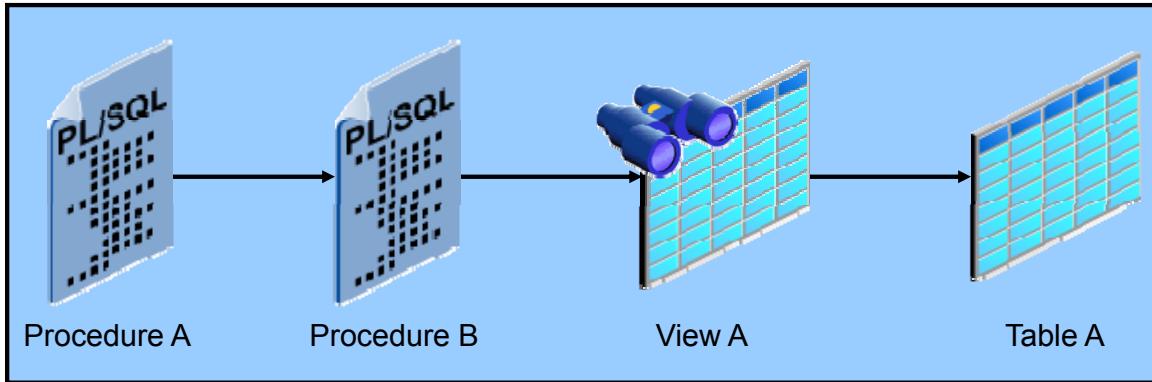
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Dependent and Referenced Objects

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

- Tables
- Views
- Sequences
- Procedures
- Functions
- Packaged procedures or functions

Direct Local Dependencies



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Managing Local Dependencies

In the case of local dependencies, the objects are on the same node in the same database. The Oracle server automatically manages all local dependencies, using the database's internal "depends-on" table. When a referenced object is modified, the dependent objects are sometimes invalidated. The next time an invalidated object is called, the Oracle server automatically recompiles it.

If you alter the definition of a referenced object, dependent objects might or might not continue to function without error, depending on the type of alteration. For example, if you drop a table, no view based on the dropped table is usable.

Starting with Oracle Database 10g, the `CREATE OR REPLACE SYNONYM` command has been enhanced to minimize the invalidations to dependent PL/SQL program units and views that reference it. This is covered later in this lesson.

Starting with Oracle Database 11g, dependencies are tracked at the level of element within unit. This is referred to as fine-grained dependency. Fine-grained dependencies are covered later in this lesson.

Querying Direct Object Dependencies: Using the `USER_DEPENDENCIES` View

```
desc user_dependencies
Name          Null    Type
-----        ----   -----
NAME          NOT NULL VARCHAR2(128)
TYPE          VARCHAR2(18)
REFERENCED_OWNER VARCHAR2(128)
REFERENCED_NAME VARCHAR2(128)
REFERENCED_TYPE VARCHAR2(18)
REFERENCED_LINK_NAME VARCHAR2(128)
SCHEMAID      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 EMP_PKG	PACKAGE	EMPLOYEES	TABLE
2 EMP_PKG	PACKAGE BODY	EMPLOYEES	TABLE
3 EMPLOYEE_INITJOBS_TRG	TRIGGER	EMPLOYEES	TABLE
4 CHECK_SALARY_TRG	TRIGGER	EMPLOYEES	TABLE
5 EMP_DETAILS	VIEW	EMPLOYEES	TABLE
6 EMPLOYEE_REPORT	PROCEDURE	EMPLOYEES	TABLE
...			



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can determine which database objects to recompile manually by displaying direct dependencies from the `USER_DEPENDENCIES` data dictionary view.

The `ALL_DEPENDENCIES` and `DBA_DEPENDENCIES` views contain the additional `OWNER` column, which references the owner of the object.

The `USER_DEPENDENCIES` Data Dictionary View Columns

The columns of the `USER_DEPENDENCIES` data dictionary view are as follows:

- NAME: The name of the dependent object
- TYPE: The type of the dependent object (PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, or VIEW)
- REFERENCED_OWNER: The schema of the referenced object
- REFERENCED_NAME: The name of the referenced object
- REFERENCED_TYPE: The type of the referenced object
- REFERENCED_LINK_NAME: The database link used to access the referenced object

Querying an Object's Status

Every database object has one of the following status values:

Status	Description
VALID	The object was successfully compiled, 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.)

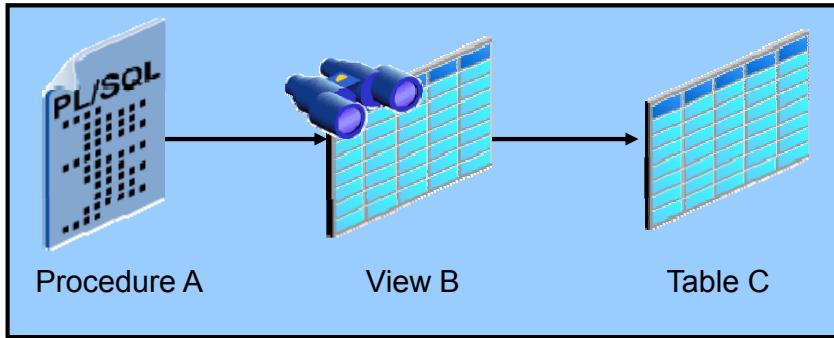


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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.

Invalidation of Dependent Objects



- Procedure A is a direct dependent of View B. View B is a direct dependent of Table C. Procedure A is an indirect dependent of Table C.
- Direct dependents are invalidated only by changes to the referenced object that affect them.
- Indirect dependents can be invalidated by changes to the reference object that do not affect them.

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

If object A depends on object B, which depends on object C, then A is a direct dependent of B, B is a direct dependent of C, and A is an indirect dependent of C.

Starting with Oracle Database 11g, direct dependents are invalidated only by changes to the referenced object that affect them (changes to the signature of the referenced object).

Indirect dependents can be invalidated by changes to the reference object that do not affect them: If a change to Table C invalidates View B, it invalidates Procedure A (and all other direct and indirect dependents of View B). This is called cascading invalidation.

Assume that the structure of the table on which a view is based is modified. When you describe the view by using the SQL*Plus DESCRIBE command, you get an error message that states that the object is invalid to describe. This is because the command is not a SQL command; at this stage, the view is invalid because the structure of its base table is changed. If you query the view now, then the view is recompiled automatically and you can see the result if it is successfully recompiled.

Schema Object Change That Invalidates Some Dependents: Example

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

```
CREATE VIEW six_figure_salary AS  
SELECT * FROM employees  
WHERE salary >= 100000;
```

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

OBJECT_NAME	STATUS
1 EMP_DETAILS_VIEW	VALID
2 EMP_DETAILS	VALID
3 COMMISSIONED	VALID
4 SIX FIGURE SALARY	VALID



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide demonstrates an example of a schema object change that invalidates some dependents but not others. The two newly created views are based on the EMPLOYEES table in the HR schema. The status of the newly created views is VALID.

Schema Object Change That Invalidates Some Dependents: Example

```
ALTER TABLE employees MODIFY email VARCHAR2(50);  
  
SELECT object_name, status  
FROM user_objects  
WHERE object_type = 'VIEW';
```

OBJECT_NAME	STATUS
1 EMP_DETAILS_VIEW	VALID
2 EMP_DETAILS	VALID
3 COMMISSIONED	VALID
4 SIX FIGURE SALARY	INVALID



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Suppose you determine that the `EMAIL` column in the `EMPLOYEES` table needs to be lengthened from 25 to 50, you alter the table as shown in the slide above.

Because the `COMMISSIONED` view does not include the `EMAIL` column in its select list, it is not invalidated. However, the `SIXFIGURES` view is invalidated because all columns in the table are selected.

Displaying Direct and Indirect Dependencies

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

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

2. Execute the `DEPTREE_FILL` procedure.

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle

Display direct and indirect dependencies from additional user views called `DEPTREE` and `IDEPTREE`; these views are provided by Oracle.

Example

1. Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/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.)

2. 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:

`object_type` Type of the referenced object

`object_owner` Schema of the referenced object

`object_name` Name of the referenced object

Displaying Dependencies Using the DEPTREE View

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

NESTED_LEVEL	TYPE	NAME
1	0 TABLE	EMPLOYEES
2	1 VIEW	EMP_DETAILS_VIEW
3	1 TRIGGER	UPDATE_JOB_HISTORY
4	1 PROCEDURE	EMP_LIST
5	1 FUNCTION	GET_ANNUAL_COMP
6	1 PROCEDURE	ADD_EMPLOYEE
7	1 PACKAGE	EMP_PKG
8	2 PACKAGE BODY	EMP_PKG
...		



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

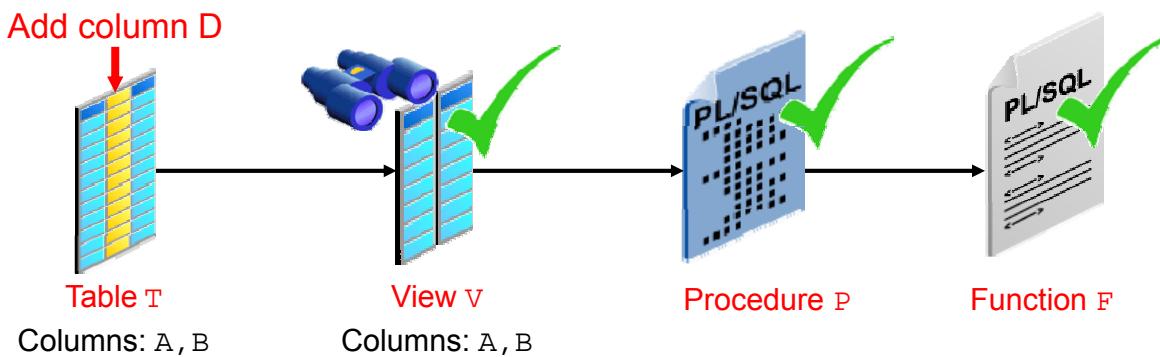
You can display a tabular representation of all dependent objects by querying the DEPTREE view. You can display an indented representation of the same information by querying the IDEPTREE view, which consists of a single column named DEPENDENCIES as follows:

```
SELECT *
FROM    ideptree;
```

DEPENDENCIES
VIEW ORA61.EMP_DETAILS_VIEW
PACKAGE ORA61.EMP_PKG
VIEW ORA61.EMP_DETAILS
TRIGGER ORA61.DELETE_EMP_TRG
TRIGGER ORA61.UPDATE_JOB_HISTORY
TRIGGER ORA61.EMPLOYEE_INITJOBS_TRG
FUNCTION ORA61.GET_ANNUAL_COMP
...

More Precise Dependency Metadata from Oracle Database 11g

- Before 11g, adding column D to table T invalidated the dependent objects.
- Oracle Database 11g records additional, finer-grained dependency management:
 - Adding column D to table T does not impact view V and does not invalidate the dependent objects



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Fine-Grained Dependencies

Starting with Oracle Database 11g, you have access to records that describe more precise dependency metadata. This is called fine-grained dependency and it enables you to see when the dependent objects are not invalidated without logical requirement.

Earlier Oracle Database releases record dependency metadata—for example, PL/SQL unit P depends on PL/SQL unit F, or that view V depends on table T—with the precision of the whole object. This means that dependent objects are sometimes invalidated without logical requirement. For example, if view V depends only on columns A and B in table T, and column D is added to table T, the validity of view V is not logically affected. Nevertheless, before Oracle Database Release 11.1, view V is invalidated by the addition of column D to table T. Starting with Oracle Database Release 11.1, adding column D to table T does not invalidate view V. Similarly, if procedure P depends only on elements E1 and E2 within a package, adding element E99 to the package does not invalidate procedure P.

Reducing the invalidation of dependent objects in response to changes to the objects on which they depend increases application availability, both in the development environment and during online application upgrade.

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 © 2013, 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 © 2013, 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 the 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 the view V is dependent because element-based dependency tracking realizes that the 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the view is invalidated because its element (`COL_A`) is modified in the table on which the view is dependent.

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 © 2013, 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`.

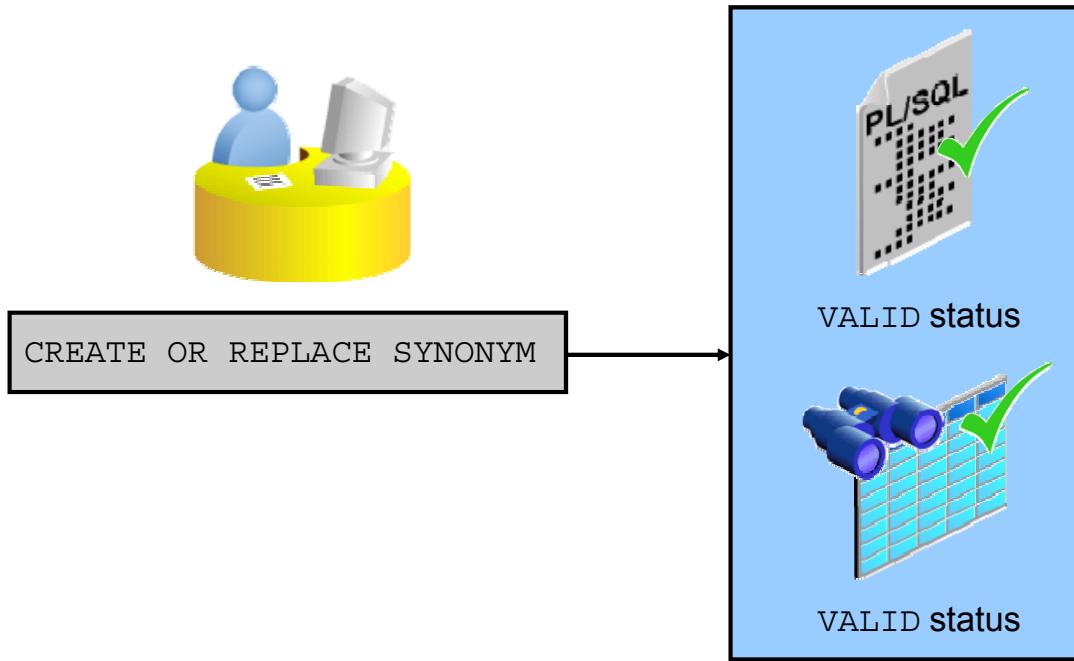
The definition of the `PKG` package is modified and another subroutine is added to the package declaration.

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

Changes to Synonym Dependencies



ORACLE®

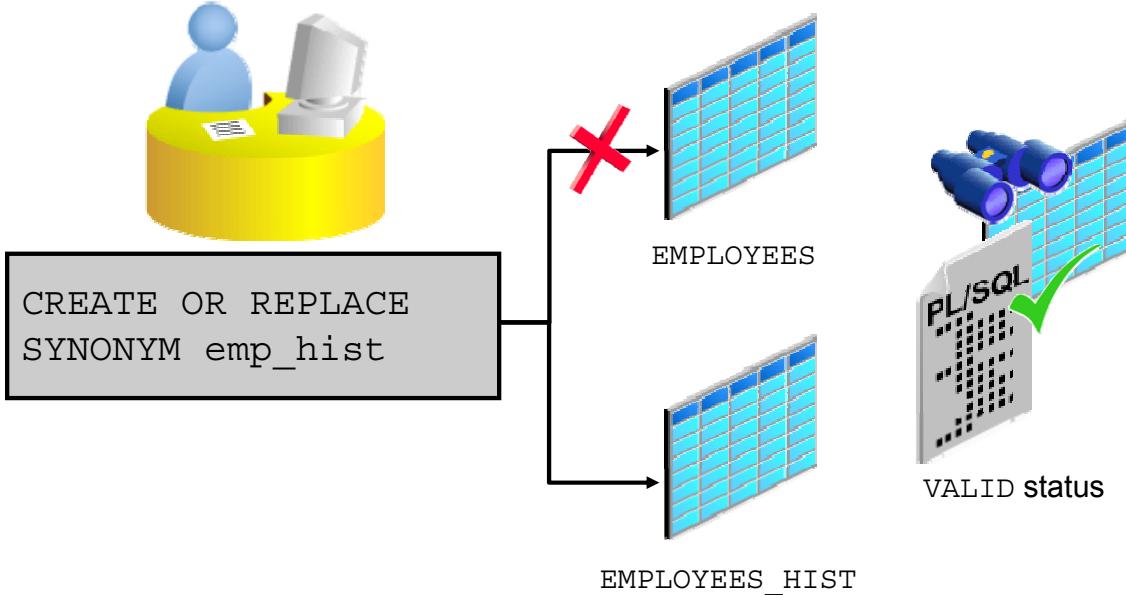
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The Oracle Database minimizes down time during code upgrades or schema merges. When certain conditions on columns, privileges, partitions, and so on are met, a table or object type is considered equivalent and dependent objects are no longer invalidated.

In Oracle Database 10g, the CREATE OR REPLACE SYNONYM command has been enhanced to minimize the invalidations to dependent PL/SQL program units and views that reference it. This eliminates the need for time-consuming recompilation of the program units after redefinition of the synonyms or during execution. You do not have to set any parameters or issue any special commands to enable this functionality; invalidations are minimized automatically.

Note: This enhancement applies only to synonyms pointing to tables.

Maintaining Valid PL/SQL Program Units and Views



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Maintaining Valid PL/SQL Program Units

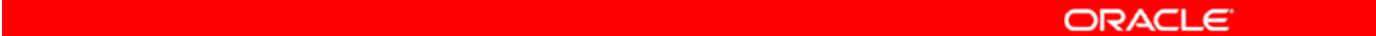
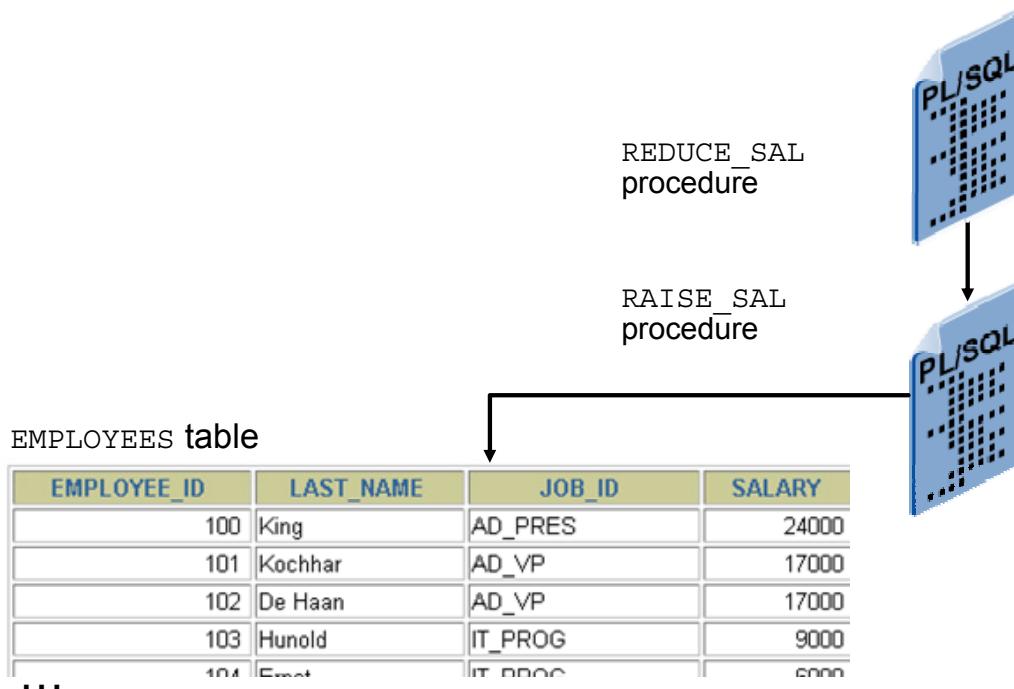
Starting with Oracle Database 10g Release 2, you can change the definition of a synonym, and the dependent PL/SQL program units are not invalidated under the following conditions:

- The column order, column names, and column data types of the tables are identical.
- The privileges on the newly referenced table and its columns are a superset of the set of privileges on the original table. These privileges must not be derived through roles alone.
- The names and types of partitions and subpartitions are identical.
- The tables are of the same organization type.
- Object type columns are of the same type.

Maintaining Valid Views: As with dependent PL/SQL program units, you can change the definition of a synonym, and the dependent views are not invalidated under the conditions listed in the preceding paragraph. In addition, the following must be true to preserve the VALID status of dependent views, but not of dependent PL/SQL program units, when you redefine a synonym:

- Columns and order of columns defined for primary key and unique indexes, NOT NULL constraints, and primary key and unique constraints must be identical.
- The dependent view cannot have any referential constraints.

Another Scenario of Local Dependencies



ORACLE

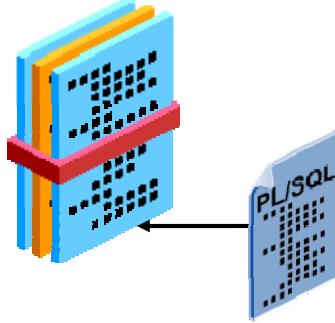
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide illustrates the effect that a change in the definition of a procedure has on the recompilation of a dependent procedure. Assume that the `RAISE_SAL` procedure updates the `EMPLOYEES` table directly, and that the `REDUCE_SAL` procedure updates the `EMPLOYEES` table indirectly by way of `RAISE_SAL`.

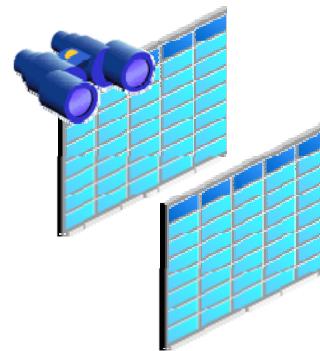
- If the internal logic of the `RAISE_SAL` procedure is modified, `REDUCE_SAL` will successfully recompile if `RAISE_SAL` has successfully compiled.
- If the formal parameters for the `RAISE_SAL` procedure are eliminated, `REDUCE_SAL` will not successfully recompile.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

Add New Items to End of 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, 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, 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.
- If an object is not valid, its status is either COMPILED WITH ERRORS, UNAUTHORIZED, or INVALID.



Copyright © 2013, 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.

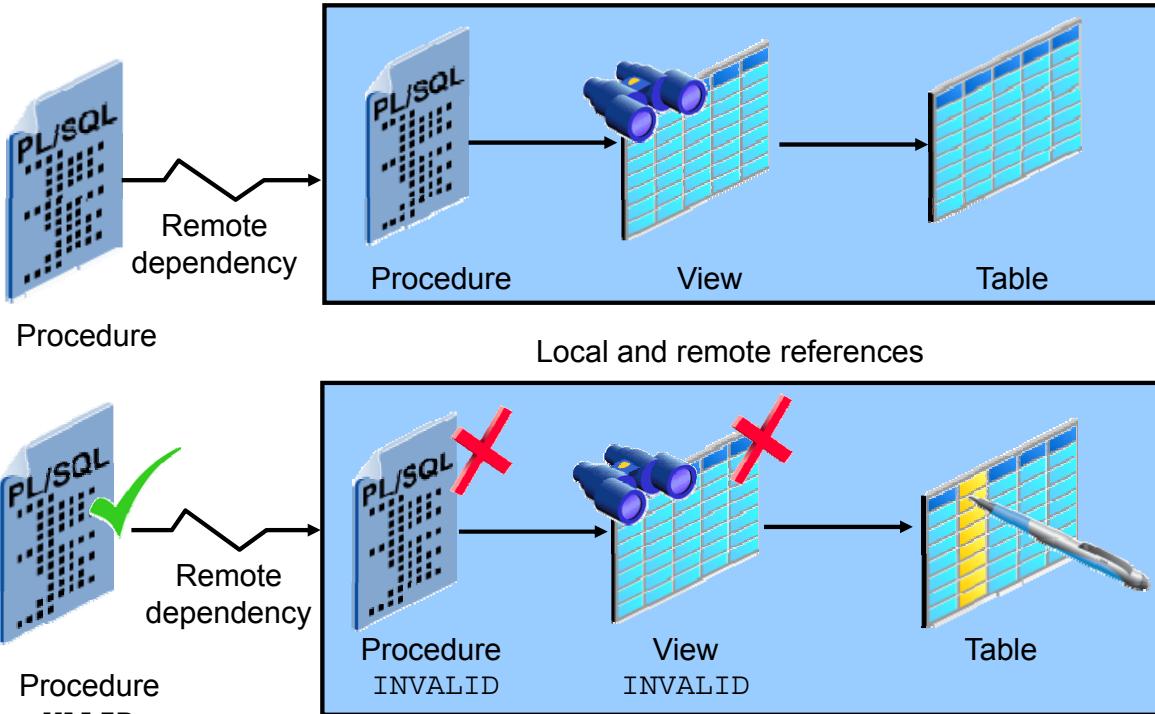
The compiler checks whether the unauthorized object has access privileges to all of its referenced objects. If so, the compiler revalidates the unauthorized object without recompiling it. If not, the compiler issues appropriate error messages.

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 changed in a way that affects the 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.
- 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.

Remote Dependencies



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

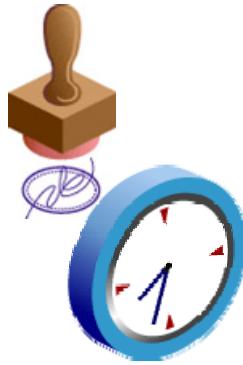
In the case of remote dependencies, the objects are on separate nodes. The Oracle server does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies (including functions, packages, and triggers). The local stored procedure and all its dependent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

Recompilation of Dependent Objects: Local and Remote

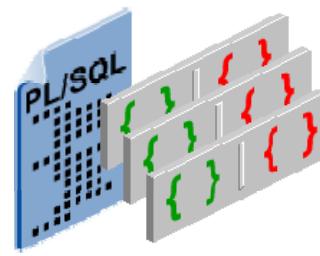
- Verify successful explicit recompilation of the dependent remote procedures and implicit recompilation of the dependent local procedures by checking the status of these procedures within the `USER_OBJECTS` view.
- If an automatic implicit recompilation of the dependent local procedures fails, the status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

Concepts of Remote Dependencies

Remote dependencies are governed by the mode that is chosen by the user:



TIMESTAMP checking



SIGNATURE checking



Copyright © 2013, 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 (RPC) layer performs a simple comparison of the signature to determine whether the call is safe or not. If the signature has not been changed in an incompatible manner, execution continues; otherwise, an error is returned.

Setting the `REMOTE_DEPENDENCIES_MODE` Parameter

- As an `init.ora` parameter:

```
REMOTE_DEPENDENCIES_MODE = TIMESTAMP /  
SIGNATURE
```

- At the system level:

```
ALTER SYSTEM SET REMOTE_DEPENDENCIES_MODE =  
TIMESTAMP | SIGNATURE
```

- At the session level:

```
ALTER SESSION SET REMOTE_DEPENDENCIES_MODE =  
TIMESTAMP | SIGNATURE
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

`REMOTE_DEPENDENCIES_MODE` Parameter

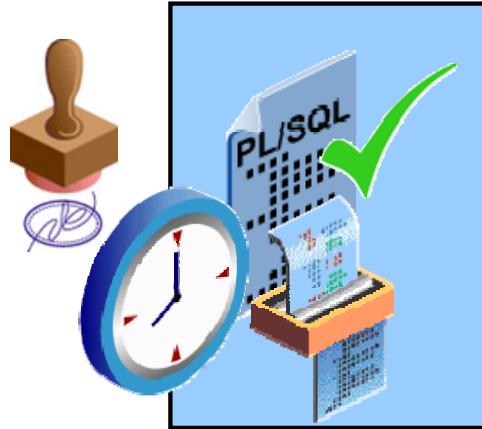
Setting the `REMOTE_DEPENDENCIES_MODE`

value `TIMESTAMP`
 `SIGNATURE`

Specify the value of the `REMOTE_DEPENDENCIES_MODE` parameter using one of the three methods described in the slide.

Note: The calling site determines the dependency model.

Remote Procedure B Compiles at 8:00 AM



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

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Local Procedures Referencing Remote Procedures

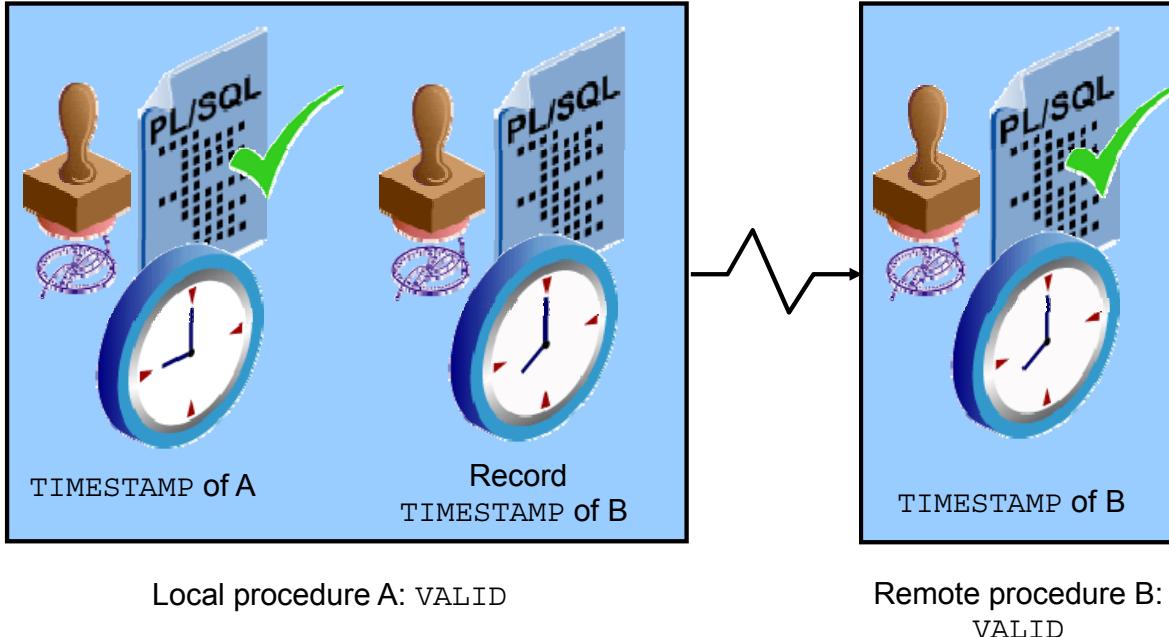
A local procedure that references a remote procedure is invalidated by the Oracle server if the remote procedure is recompiled after the local procedure is compiled.

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.

Local Procedure A Compiles at 9:00 AM



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

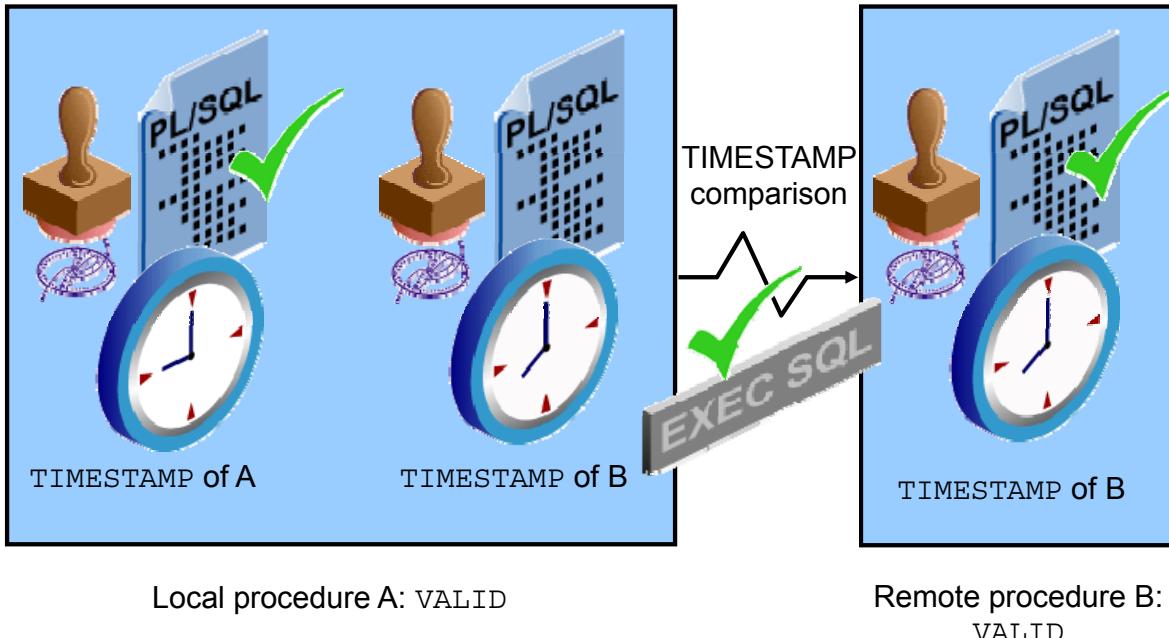
Local Procedures Referencing Remote Procedures

Automatic Remote Dependency Mechanism (continued)

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

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

Execute Procedure A



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

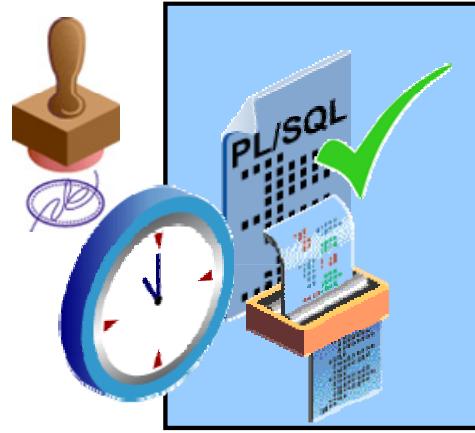
Automatic Remote Dependency

When the local procedure is invoked at run time, the Oracle server compares the two time stamps of the referenced remote procedure.

If the time stamps are equal (indicating that the remote procedure has not recompiled), then the Oracle server executes the local procedure.

In the example in the slide, the time stamp recorded with the `P` code of remote procedure B is the same as that recorded with local procedure A. Therefore, local procedure A is valid.

Remote Procedure B Recompiled at 11:00 AM



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

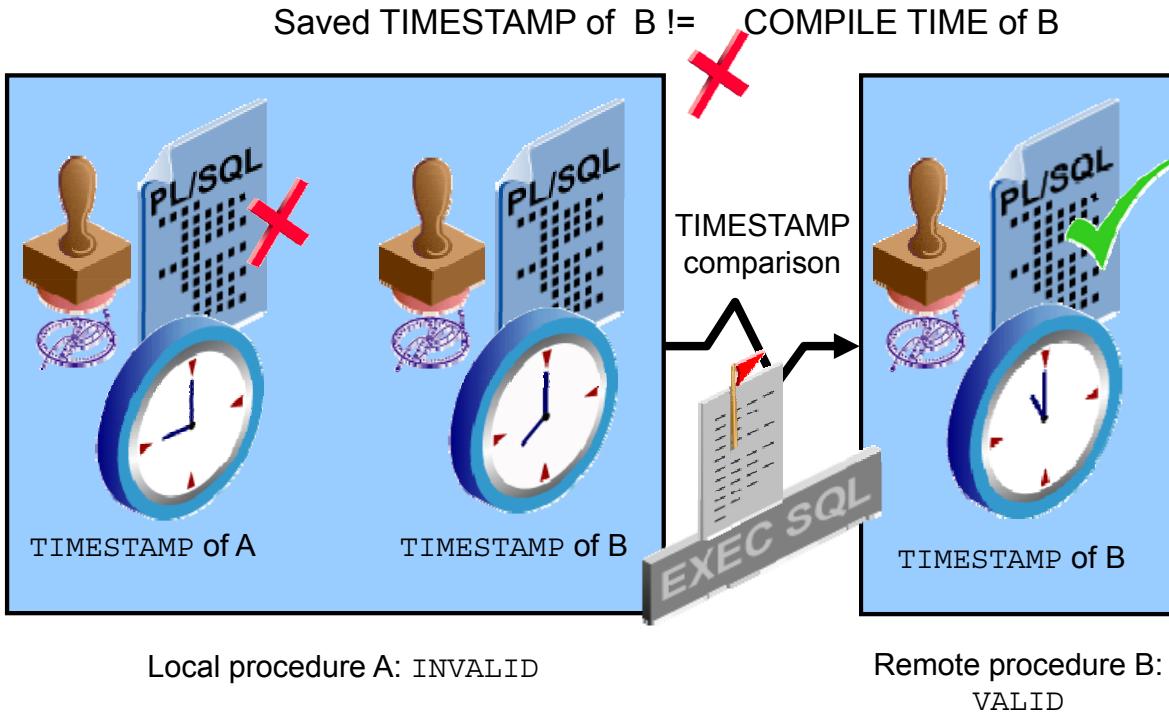
ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Local Procedures Referencing Remote Procedures

Assume that remote procedure B is successfully recompiled at 11:00 AM. The new time stamp is recorded along with its P code.

Execute Procedure A



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

Automatic Remote Dependency

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

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

In the example in the slide, the remote procedure is recompiled at 11:00 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 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 Mode

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Signatures

To alleviate some of the problems with the time stamp-only dependency model, you can use the signature model. This allows the remote procedure to be recompiled without affecting the local procedures. This is important if the database is distributed.

The signature of a subprogram contains the following information:

- The name of the subprogram
- The data types of the parameters
- The modes of the parameters
- The number of parameters
- The data type of the return value for a function

If a remote program is changed and recompiled but the signature does not change, then the local procedure can execute the remote procedure. With the time stamp method, an error would have been raised because the time stamps would not have matched.

Recompiling a PL/SQL Program Unit

Recompilation:

- Is handled automatically through implicit run-time recompilation
- Is handled through explicit recompilation with the ALTER statement

```
ALTER PROCEDURE [SCHEMA.] procedure_name COMPILE;
```

```
ALTER FUNCTION [SCHEMA.] function_name COMPILE;
```

```
ALTER PACKAGE [SCHEMA.] package_name
    COMPILE [PACKAGE | SPECIFICATION | BODY];
```

```
ALTER TRIGGER trigger_name [COMPILE [DEBUG]] ;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Recompiling PL/SQL Objects

If the recompilation is successful, the object becomes valid. If not, the Oracle server returns an error and the object remains invalid. When you recompile a PL/SQL object, the Oracle server first recompiles any invalid object on which it depends.

Procedure: Any local objects that depend on a procedure (such as procedures that call the recompiled procedure or package bodies that define the procedures that call the recompiled procedure) are also invalidated.

Packages: The COMPILE PACKAGE option recompiles both the package specification and the body, regardless of whether it is invalid. The COMPILE SPECIFICATION option recompiles the package specification. Recompiling a package specification invalidates any local objects that depend on the specification, such as subprograms that use the package. Note that the body of a package also depends on its specification. The COMPILE BODY option recompiles only the package body.

Triggers: Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The DEBUG option instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

Unsuccessful Recompilation

Recompiling dependent procedures and functions is unsuccessful when:

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Sometimes a recompilation of dependent procedures is unsuccessful (for example, when a referenced table is dropped or renamed).

The success of any recompilation is based on the exact dependency. If a referenced view is re-created, any object that is dependent on the view needs to be recompiled. The success of the recompilation depends on the columns that the view now contains, as well as the columns that the dependent objects require for their execution. If the required columns are not part of the new view, then the object remains invalid.

Successful Recompilation

Recompiling dependent procedures and functions is successful if:

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



Copyright © 2013, 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 becomes invalid. When the procedure is recompiled (either explicitly or implicitly) and a public table exists, the procedure can recompile successfully but is now dependent on the public table. The recompilation is successful only if the public table contains the columns that the procedure requires; otherwise, the status of the procedure remains invalid.

Recompiling Procedures

Minimize dependency failures by:

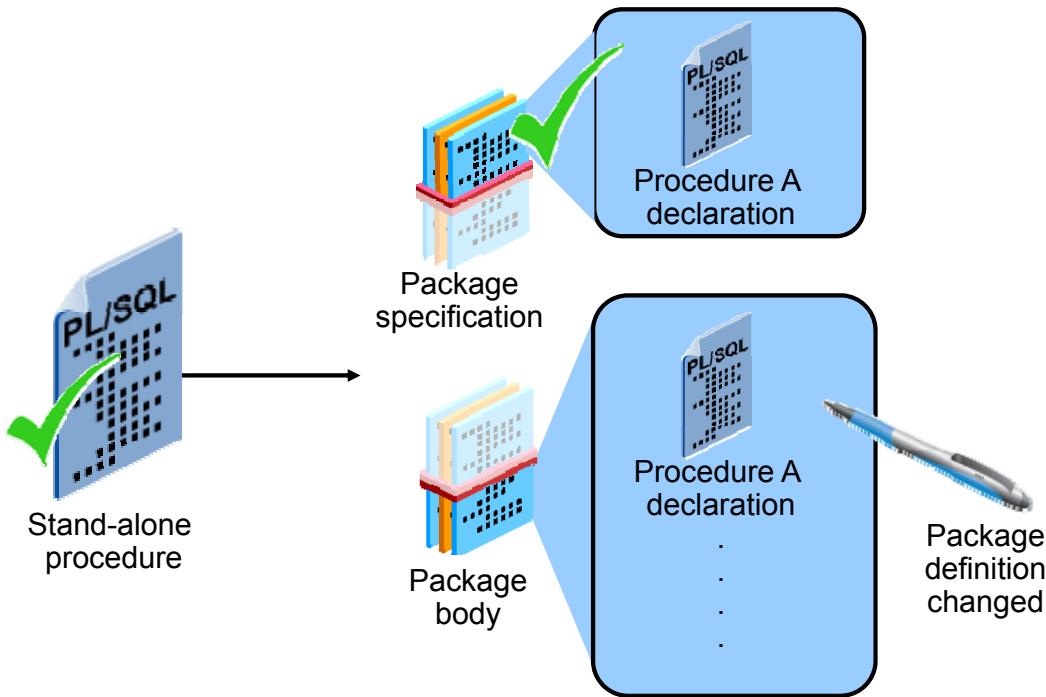
- Declaring records with the %ROWTYPE attribute
- Declaring variables with the %TYPE attribute
- Querying with the SELECT * notation
- Including a column list with INSERT statements



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can minimize recompilation failure by following the guidelines that are shown in the slide.

Packages and Dependencies: Subprogram References the Package



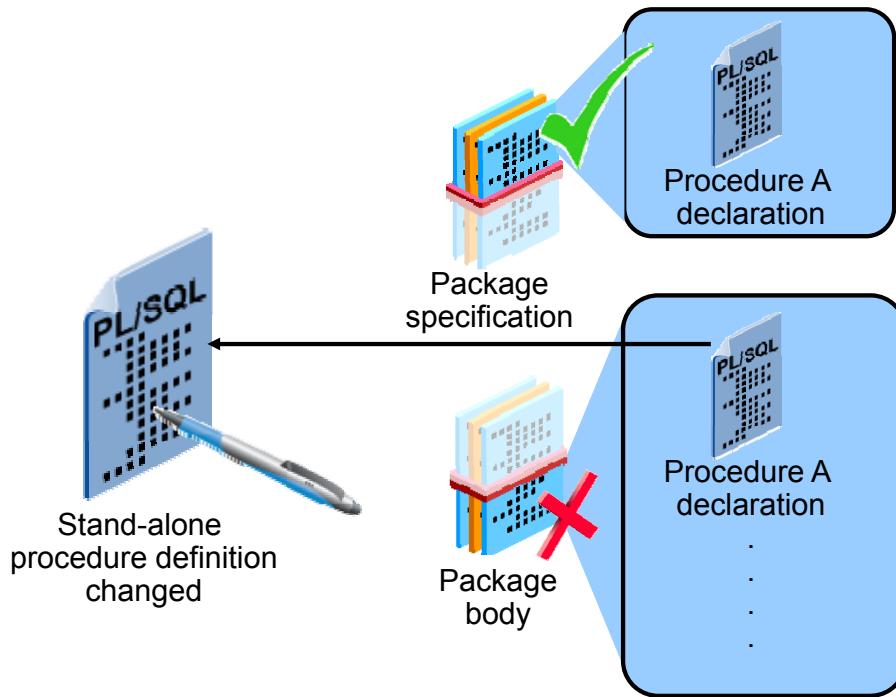
ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

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

Packages and Dependencies: Package Subprogram References Procedure



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

If a stand-alone procedure that is referenced within the package changes, then the entire package body is invalidated, but the package specification remains valid. Therefore, it is recommended that you bring the procedure into the package.

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: a

Displaying Direct and Indirect Dependencies

You can display direct and indirect dependencies as follows:

1. Run the `utldtree.sql` script, which creates the objects that enable you to display the direct and indirect dependencies.
2. Populate the `DEPTREE_TEMP TAB` table with information for a particular referenced object by executing the `DEPTREE_FILL` procedure.
3. Query the `DEPTREE` or `IDEPTREE` views.

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Avoid disrupting production by keeping track of dependent procedures and recompiling them manually as soon as possible after the definition of a database object changes.

Practice 12 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you use the `DEPTREE_FILL` procedure and the `IDEPTREE` view to investigate dependencies in your schema. In addition, you recompile invalid procedures, functions, packages, and views.

Table Descriptions

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Schema Description

Overall Description

The Oracle Database sample schemas portray a fictional company that operates worldwide to fill orders for several different products. The company has three divisions:

- **Human Resources:** Tracks information about the employees and facilities
- **Order Entry:** Tracks product inventories and sales through various channels
- **Sales History:** Tracks business statistics to facilitate business decisions

Each of these divisions is represented by a schema. In this course, you have access to the objects in all the schemas. However, the emphasis of the examples, demonstrations, and practices is on the Human Resources (HR) schema.

All scripts necessary to create the sample schemas reside in the \$ORACLE_HOME/demo/schema/ folder.

Human Resources (HR)

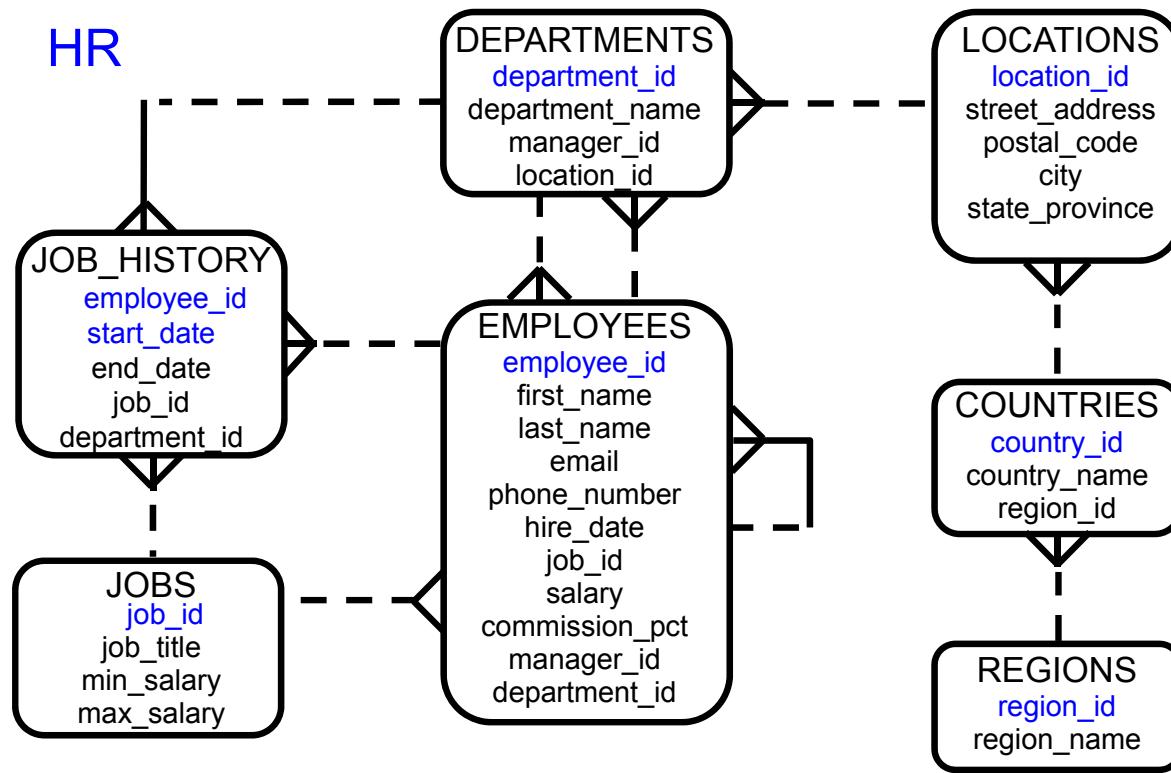
This is the schema that is used in this course. In the Human Resource (HR) records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee resigns, the duration the employee was working, the job identification number, and the department are recorded.

The sample company is regionally diverse, so it tracks the locations of its warehouses and departments. Each employee is assigned to a department, and each department is identified either by a unique department number or a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and country code.

In places where the departments and warehouses are located, the company records details such as the country name, currency symbol, and currency name, as well as the region where the country is located geographically.

HR Entity Relationship Diagram



Human Resources (HR) Table Descriptions

DESCRIBE countries

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

SELECT * FROM countries;

#	COUNTRY_ID	COUNTRY_NAME	REGION_ID
1	AR	Argentina	2
2	AU	Australia	3
3	BE	Belgium	1
4	BR	Brazil	2
5	CA	Canada	2
6	CH	Switzerland	1
7	CN	China	3
8	DE	Germany	1
9	DK	Denmark	1
10	EG	Egypt	4
11	FR	France	1
12	HK	HongKong	3
13	IL	Israel	4
14	IN	India	3
15	IT	Italy	1
16	JP	Japan	3
17	KW	Kuwait	4
18	LB	Lebanon	4
19	MX	Mexico	2
20	NG	Nigeria	4
21	NL	Netherlands	1
22	SG	Singapore	3
23	UK	United Kingdom	1
24	US	United States of America	2
25	ZM	Zambia	4
26	ZW	Zimbabwe	4

DESCRIBE departments

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

SELECT * FROM departments;

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	280	TRAINING	(null)	2500
2	290	Unknown	(null)	1700
3	300	ADVERTISING	(null)	1200
4	310	Unknown	(null)	1200
5	980	Education	(null)	2500
6	460	Training	(null)	2400
7	10	Administration	200	1700
8	20	Marketing	201	1800
9	30	Purchasing	114	1700
10	40	Human Resources	203	2400
11	50	Shipping	121	1500
12	60	IT	103	1400
13	70	Public Relations	204	2700
14	80	Sales	145	2500
15	90	Executive	100	1700
16	100	Finance	108	1700
17	110	Accounting	205	1700
18	120	Treasury	(null)	1700
19	130	Corporate Tax	(null)	1700
20	140	Control And Credit	(null)	1700
21	150	Shareholder Serv...	(null)	1700
22	160	Benefits	(null)	1700

23	170	Manufacturing	(null)	1700
24	180	Construction	(null)	1700
25	190	Contracting	(null)	1700
26	200	Operations	(null)	1700
27	210	IT Support	(null)	1700
28	220	NOC	(null)	1700
29	230	IT Helpdesk	(null)	1700
30	240	Government Sales	(null)	1700
31	250	Retail Sales	(null)	1700
32	260	Recruiting	(null)	1700
33	270	Payroll	(null)	1700

DESCRIBE employees

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

SELECT * FROM employees;

#	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	TOTSAL
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	29040	(null)	(null)	90	(null)
2	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-05	AD_VP	17000	(null)	100	90	(null)
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-01	AD_VP	20570	(null)	100	90	(null)
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-06	IT_PROG	9000	(null)	102	60	(null)
5	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-07	IT_PROG	7260	(null)	103	60	(null)
6	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-05	IT_PROG	5000	(null)	103	60	(null)
7	106	Vallii	Pataballa	VPATABAL	590.423.4560	05-FEB-06	IT_PROG	5000	(null)	103	60	(null)
8	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-07	IT_PROG	5000	(null)	103	60	(null)
9	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-02	FI_MGR	13208.8	(null)	101	100	(null)
10	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-02	FI_ACCOUNT	9000	(null)	108	100	(null)
11	110	John	Chen	JCHEN	515.124.4269	28-SEP-05	FI_ACCOUNT	9922	(null)	108	100	(null)
12	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-05	FI_ACCOUNT	7700	(null)	108	100	(null)
13	112	Jose Manuel	Urman	JURMAN	515.124.4469	07-MAR-06	FI_ACCOUNT	7800	(null)	108	100	(null)
14	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-07	FI_ACCOUNT	6900	(null)	108	100	(null)
15	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-02	PU_MAN	12100	(null)	100	30	(null)
16	115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-03	PU_CLERK	2800	(null)	114	30	(null)
17	116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-05	PU_CLERK	3190	(null)	114	30	(null)
18	117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-05	PU_CLERK	3080	(null)	114	30	(null)
19	118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-06	PU_CLERK	2860	(null)	114	30	(null)
20	119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-07	PU_CLERK	2750	(null)	114	30	(null)

#	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	TOTSAL
21	120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-04	ST_MAN	8000	(null)	100	50	(null)
22	121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-05	ST_MAN	8200	(null)	100	50	(null)
23	122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-03	ST_MAN	7900	(null)	100	50	(null)
24	123	Shanta	Voillman	SVOLLMAN	650.123.4234	10-OCT-05	ST_MAN	6500	(null)	100	50	(null)
25	124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-07	ST_MAN	5800	(null)	100	50	(null)
26	125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-05	ST_CLERK	3200	(null)	120	50	(null)
27	126	Irene	Mikkilineni	IMIKILINI	650.124.1224	28-SEP-06	ST_CLERK	2700	(null)	120	50	(null)
28	127	James	Landry	JLANDRY	650.124.1334	14-JAN-07	ST_CLERK	2400	(null)	120	50	(null)
29	128	Steven	Markle	SWARCKLE	650.124.1434	08-MAR-08	ST_CLERK	2200	(null)	120	50	(null)
30	129	Laura	Bisot	LBISOTT	650.124.5234	20-AUG-05	ST_CLERK	3300	(null)	121	50	(null)
31	130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-05	ST_CLERK	2800	(null)	121	50	(null)
32	131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-05	ST_CLERK	2500	(null)	121	50	(null)
33	132	TJ	Olson	TJOLSON	650.124.8234	10-APR-07	ST_CLERK	2100	(null)	121	50	(null)
34	133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-04	ST_CLERK	3300	(null)	122	50	(null)
35	134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-06	ST_CLERK	2900	(null)	122	50	(null)
36	135	Ki	Gee	KGEE	650.127.1734	12-DEC-07	ST_CLERK	2400	(null)	122	50	(null)
37	136	Hazel	Philtanker	PHILKTAN	650.127.1634	06-FEB-08	ST_CLERK	2200	(null)	122	50	(null)
38	137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-03	ST_CLERK	3600	(null)	123	50	(null)
39	138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-05	ST_CLERK	3400	(null)	123	50	(null)
40	139	John	Seo	JSEO	650.121.2019	12-FEB-06	ST_CLERK	2700	(null)	123	50	(null)

Employees (continued)

#	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	TOTSAL
41	140	Joshua	Patel	JPATEL	650.121.1834	06-APR-06	ST_CLERK	2500	(null)	123	50	(null)
42	141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-03	ST_CLERK	3500	(null)	124	50	(null)
43	142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-05	ST_CLERK	3100	(null)	124	50	(null)
44	143	Randall	Matos	RMATOS	650.121.2874	15-MAR-06	ST_CLERK	2600	(null)	124	50	(null)
45	144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-06	ST_CLERK	2500	(null)	124	50	(null)
46	145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-04	SA_MAN	14000	0.4	100	80	(null)
47	146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-05	SA_MAN	13500	0.3	100	80	(null)
48	147	Alberto	Errazuriz	AERRAZUR	011.44.1344.429278	10-MAR-05	SA_MAN	12000	0.3	100	80	(null)
49	148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-07	SA_MAN	11000	0.3	100	80	(null)
50	149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-08	SA_MAN	10500	0.2	100	80	(null)
51	150	Peter	Tucker	PTUCKER	011.44.1344.129268	30-JAN-05	SA REP	10000	0.3	145	80	(null)
52	151	David	Bernstein	DBERNSTE	011.44.1344.345268	24-MAR-05	SA REP	9500	0.25	145	80	(null)
53	152	Peter	Hall	PHALL	011.44.1344.478968	20-AUG-05	SA REP	9000	0.25	145	80	(null)
54	153	Christopher	Olsen	COLSEN	011.44.1344.498718	30-MAR-06	SA REP	8000	0.2	145	80	(null)
55	154	Nanette	Cambrault	NCAMBRAU	011.44.1344.987668	09-DEC-06	SA REP	7500	0.2	145	80	(null)
56	155	Oliver	Tuvault	OTUVAU	011.44.1344.486508	23-NOV-07	SA REP	7000	0.15	145	80	(null)
57	156	Janette	King	JKING	011.44.1345.429268	30-JAN-04	SA REP	10000	0.35	146	80	(null)
58	157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-04	SA REP	9500	0.35	146	80	(null)
59	158	Allan	McEwen	AMCEWEN	011.44.1345.829268	01-AUG-04	SA REP	9000	0.35	146	80	(null)
60	159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-05	SA REP	8000	0.3	146	80	(null)

#	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	TOTSAL
61	160	Louise	Doran	LDORAN	011.44.1345.629268	15-DEC-05	SA REP	7500	0.3	146	80	(null)
62	161	Sarath	Sewall	SSEWALL	011.44.1345.529268	03-NOV-06	SA REP	7000	0.25	146	80	(null)
63	162	Clara	Vishney	CVISHNEY	011.44.1346.129268	11-NOV-05	SA REP	10500	0.25	147	80	(null)
64	163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-07	SA REP	9500	0.15	147	80	(null)
65	164	Mattiea	Marvins	MMARVINS	011.44.1346.329268	24-JAN-08	SA REP	7200	0.1	147	80	(null)
66	165	David	Lee	DLEE	011.44.1346.529268	23-FEB-08	SA REP	6800	0.1	147	80	(null)
67	166	Sundar	Ande	SANDE	011.44.1346.629268	24-MAR-08	SA REP	6400	0.1	147	80	(null)
68	167	Amit	Banda	ABANDA	011.44.1346.729268	21-APR-08	SA REP	6200	0.1	147	80	(null)
69	168	Lisa	Ozer	LOZER	011.44.1343.929268	11-MAR-05	SA REP	11500	0.25	148	80	(null)
70	169	Harrison	Bloom	HBL00M	011.44.1343.829268	23-MAR-06	SA REP	10000	0.2	148	80	(null)
71	170	Tayler	Fox	TFOX	011.44.1343.729268	24-JAN-06	SA REP	9600	0.2	148	999	(null)
72	171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-07	SA REP	7400	0.15	148	80	(null)
73	172	Elizabeth	Bates	EBATES	011.44.1343.529268	24-MAR-07	SA REP	7300	0.15	148	80	(null)
74	173	Sundita	Kumar	SKUMAR	011.44.1343.329268	21-APR-08	SA REP	6100	0.1	148	80	(null)
75	174	Eileen	Abel	EABEL	011.44.1644.429267	11-MAY-04	SA REP	11000	0.3	149	80	(null)
76	175	Alyssa	Hutton	AHUTTON	011.44.1644.429266	10-MAR-05	SA REP	8800	0.25	149	80	(null)
77	176	Jonathon	Taylor	JTAYL0R	011.44.1644.429265	24-MAR-06	SA REP	10406	0.2	149	80	(null)
78	177	Jack	Livingston	JLIVINGS	011.44.1644.429264	23-APR-06	SA REP	8400	0.2	149	80	(null)
79	179	Charles	Johnson	CJOHNSON	011.44.1644.429262	04-JAN-08	SA REP	6200	0.1	149	80	(null)
80	180	Winston	Taylor	WTAYL0R	650.507.9876	24-JAN-06	SH CLERK	3200	(null)	120	50	(null)

Employees (continued)

#	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	TOTSAL
81	181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-06	SH_CLERK	3100	(null)	120	50	(null)
82	182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-07	SH_CLERK	2500	(null)	120	50	(null)
83	183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-08	SH_CLERK	2800	(null)	120	50	(null)
84	184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-04	SH_CLERK	4200	(null)	121	50	(null)
85	185	Alexis	Bull	ABULL	650.509.2876	20-FEB-05	SH_CLERK	4100	(null)	121	50	(null)
86	186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-06	SH_CLERK	3400	(null)	121	50	(null)
87	187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-07	SH_CLERK	3000	(null)	121	50	(null)
88	188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-05	SH_CLERK	3800	(null)	122	50	(null)
89	189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-05	SH_CLERK	3600	(null)	122	50	(null)
90	190	Timothy	Gates	TGATES	650.505.3876	11-JUL-06	SH_CLERK	2900	(null)	122	50	(null)
91	191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-07	SH_CLERK	2500	(null)	122	50	(null)
92	192	Sarah	Bell	SBELL	650.501.1876	04-FEB-04	SH_CLERK	4000	(null)	123	50	(null)
93	193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-05	SH_CLERK	3900	(null)	123	50	(null)
94	194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-06	SH_CLERK	3200	(null)	123	50	(null)
95	195	Vance	Jones	VJONES	650.501.4876	17-MAR-07	SH_CLERK	2800	(null)	123	50	(null)
96	196	Alana	Walsh	AWALSH	650.507.9811	24-APR-06	SH_CLERK	3100	(null)	124	50	(null)
97	197	Kevin	Feehey	KFEENEY	650.507.9822	23-MAY-06	SH_CLERK	3000	(null)	124	50	(null)
98	198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-07	SH_CLERK	2600	(null)	124	50	(null)
99	199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-08	SH_CLERK	2600	(null)	124	50	(null)
100	200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-03	AD_ASST	4400	(null)	101	10	(null)

#	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	TOTSAL
101	201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-04	MK_MAN	13000	(null)	100	20	(null)
102	202	Pat	Fay	PFAY	603.123.6666	17-AUG-05	MK_REP	6000	(null)	201	20	(null)
103	203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-02	HR REP	6500	(null)	101	40	(null)
104	204	Hermann	Baer	HBAER	515.123.8888	07-JUN-02	PR REP	10000	(null)	101	70	(null)
105	205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-02	AC_MGR	12008	(null)	101	110	(null)
106	206	William	Gietz	WGRIETZ	515.123.8181	07-JUN-02	AC_ACCOUNT	8300	(null)	205	110	(null)
107	207	Joe	Harris	JAHARRIS	(null)	16-NOV-12	SA REP	1000	0	145	80	(null)
108	208	David	Smith	DASMITH	(null)	18-NOV-12	SA REP	1000	0	145	80	(null)
109	209	Samuel	Joplin	SJOPLIN	(null)	18-NOV-12	SA REP	1100	0	145	30	(null)

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

#	EMPLOYEE_ID	START_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

DESCRIBE jobs

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

SELECT * FROM jobs;

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1 AD_PRES	President	20080	40000
2 AD_VP	Administration Vice President	15000	30000
3 AD_ASST	Administration Assistant	3000	6000
4 FI_MGR	Finance Manager	8200	16000
5 FI_ACCOUNT	Accountant	4200	9000
6 AC_MGR	Accounting Manager	8200	16000
7 AC_ACCOUNT	Public Accountant	4200	9000
8 SA_MAN	Sales Manager	10000	20080
9 SA_REP	Sales Representative	6000	12008
10 PU_MAN	Purchasing Manager	8000	15000
11 PU_CLERK	Purchasing Clerk	2500	5500
12 ST_MAN	Stock Manager	5500	8500
13 ST_CLERK	Stock Clerk	2008	5000
14 SH_CLERK	Shipping Clerk	2500	5500
15 IT_PROG	Programmer	5000	10000
16 MK_MAN	Marketing Manager	9000	15000
17 MK_REP	Marketing Representative	4000	9000
18 HR_REP	Human Resources Representative	4000	9000
19 PR_REP	Public Relations Representative	4500	10500

DESCRIBE locations

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

SELECT * FROM locations;

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	COUNTRY_ID
1	1000 1297 Via Cola di Rie	00989	Roma	(null)	IT
2	1100 93091 Calle della Testa	10934	Venice	(null)	IT
3	1200 2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
4	1300 9450 Kamiya-cho	6823	Hiroshima	(null)	JP
5	1400 2014 Jabberwocky Rd	26192	Southlake	Texas	US
6	1500 2011 Interiors Blvd	99236	South San Francisco	California	US
7	1600 2007 Zagora St	50090	South Brunswick	New Jersey	US
8	1700 2004 Charade Rd	98199	Seattle	Washington	US
9	1800 147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
10	1900 6092 Boxwood St	Y5W 9T2	Whitehorse	Yukon	CA
11	2000 40-5-12 Laogianggen	190518	Beijing	(null)	CN
12	2100 1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
13	2200 12-98 Victoria Street	2901	Sydney	New South Wales	AU
14	2300 198 Clementi North	540198	Singapore	(null)	SG
15	2400 8204 Arthur St	(null)	London	(null)	UK
16	2500 Magdalene Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
17	2600 9702 Chester Road	09629850293	Stretford	Manchester	UK
18	2700 Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
19	2800 Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
20	2900 20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
21	3000 Murtenstrasse 921	3095	Bern	BE	CH
22	3100 Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
23	3200 Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

```
DESCRIBE regions
```

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

```
SELECT * FROM regions
```

	REGION_ID	REGION_NAME
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

Using SQL Developer

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this appendix, you should be able to:

- List the key features of Oracle SQL Developer
- Identify the menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports
- Browse the Data Modeling options in SQL Developer

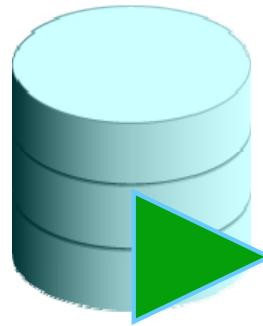


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this appendix, you are introduced to the graphical tool called SQL Developer. You learn how to use SQL Developer for your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts.

What Is Oracle SQL Developer?

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



SQL Developer

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

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

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

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

SQL Developer is the interface to administer the Oracle Application Express Listener. The new interface enables you to specify global settings and multiple database settings with different database connections for the Application Express Listener. SQL Developer provides the option to drag and drop objects by table or column name onto the worksheet. It provides improved DB Diff comparison options, GRANT statements support in the SQL editor, and DB Doc reporting. Additionally, SQL Developer includes support for Oracle Database 12c features.

Specifications of SQL Developer

- Is shipped along with Oracle Database 12c Release 1
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Enables default connectivity using the JDBC Thin driver
- Connects to Oracle Database version 9.2.0.1 and later



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle SQL Developer is shipped along with Oracle Database 12c Release 1 by default. SQL Developer is developed in Java, leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms.

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

Note

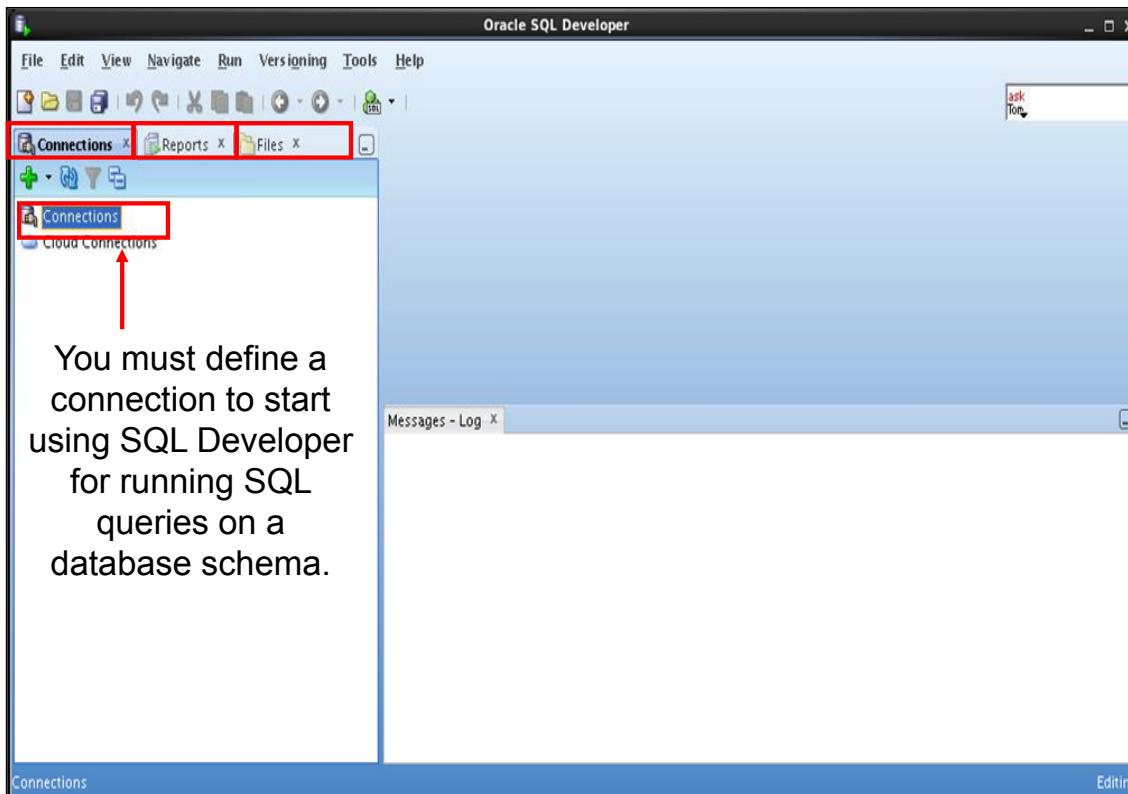
For Oracle Database 12c Release 1, you will have to download and install SQL Developer. SQL Developer is freely downloadable from the following link:

<http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html>

For instructions on how to install SQL Developer, see the website at:

<http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>

SQL Developer 3.2 Interface



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

- **Connections tab:** By using this tab, you can browse database objects and users to which you have access.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.
- **Files tab:** Identified by the Files folder icon, this tab enables you to access files from your local machine without having to use the File > Open menu.

General Navigation and Use

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures and functions.

Menus

The following menus contain standard entries, plus entries for features that are specific to SQL Developer:

- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to panes and for executing subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **Versioning:** Provides integrated support for the following versioning and source control systems – Concurrent Versions System (CVS) and Subversion
- **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet. It also contains options related to migrating third-party databases to Oracle.

Note: The Run menu also contains options that are relevant when a function or procedure is selected for debugging.

Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for:
 - Multiple databases
 - Multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

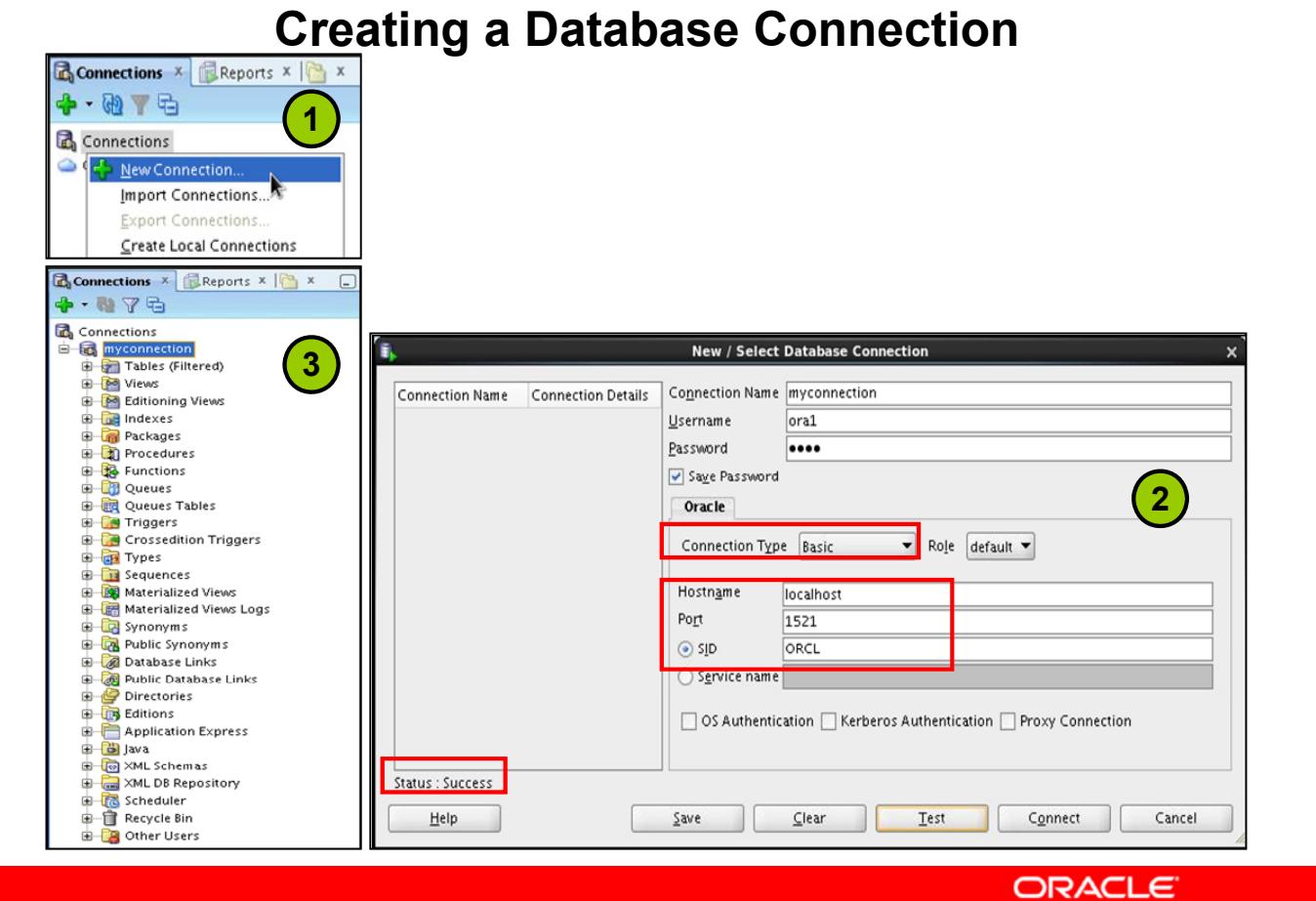
You can create and test connections for multiple databases and for multiple schemas.

By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and open the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows, if the `tnsnames.ora` file exists, but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it.

You can create additional connections as different users to the same database or to connect to the different databases.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click Connections and select New Connection.
2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.
 - a. From the Role drop-down list, you can select either *default* or *SYSDBA*. (You choose *SYSDBA* for the *sys* user or any user with database administrator privileges.)
 - b. You can select the connection type as:
 - Basic:** In this type, enter host name and SID for the database that you want to connect to. Port is already set to 1521. You can also choose to enter the Service name directly if you use a remote database connection.
 - TNS:** You can select any one of the database aliases imported from the *tnsnames.ora* file.
 - LDAP:** You can look up database services in Oracle Internet Directory, which is a component of Oracle Identity Management.
 - Advanced:** You can define a custom Java Database Connectivity (JDBC) URL to connect to the database.

Local/Bequeath: If the client and database exist on the same computer, a client connection can be passed directly to a dedicated server process without going through the listener.

- c. Click Test to ensure that the connection has been set correctly.
- d. Click Connect.

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you are not prompted for the password.

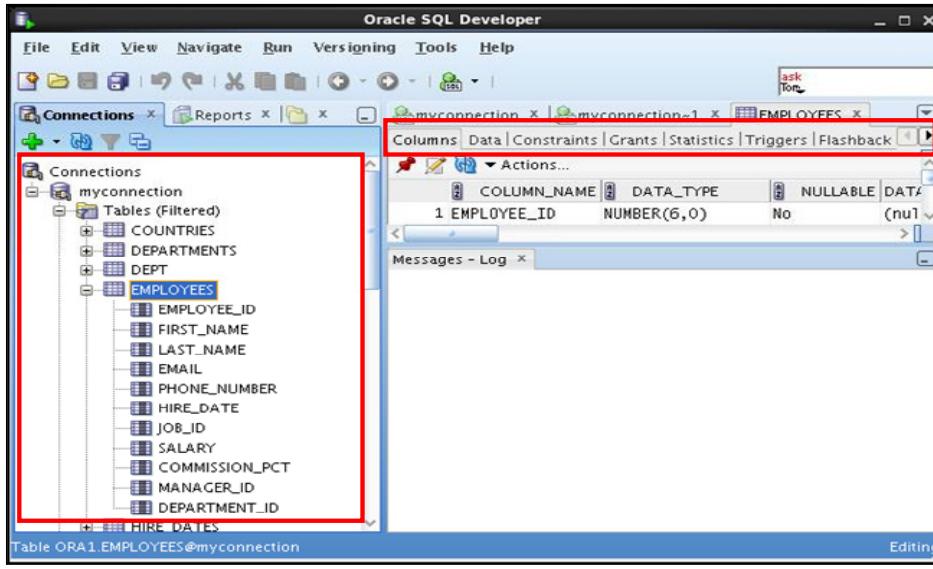
3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions (dependencies, details, statistics, and so on).

Note: From the same New>Select Database Connection window, you can define connections to non-Oracle data sources using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.

Browsing Database Objects

Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

After you create a database connection, you can use the Connections Navigator to browse through many objects in a database schema, including Tables, Views, Indexes, Packages, Procedures, Triggers, and Types.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

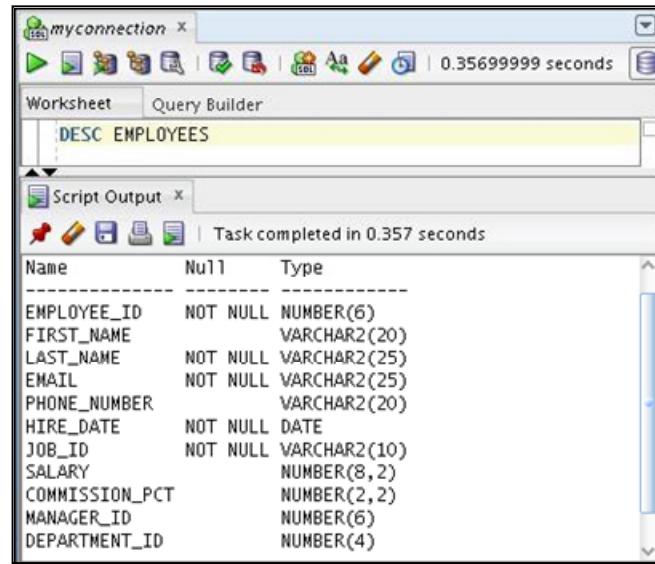
You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the EMPLOYEES table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click EMPLOYEES. By default, the Columns tab is selected. It shows the column description of the table. Using the Data tab, you can view the table data and also enter new rows, update data, and commit these changes to the database.

Displaying the Table Structure

Use the DESCRIBE command to display the structure of a table:



The screenshot shows the Oracle SQL Developer interface. In the top-left pane, the connection name is "myconnection". Below it, the "Worksheet" tab is active, showing the command "DESC EMPLOYEES" in yellow. In the bottom-right pane, the "Script Output" tab is active, displaying the results of the DESCRIBE command. The output shows the column names, whether they can be null, and their data types:

Name	Null	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

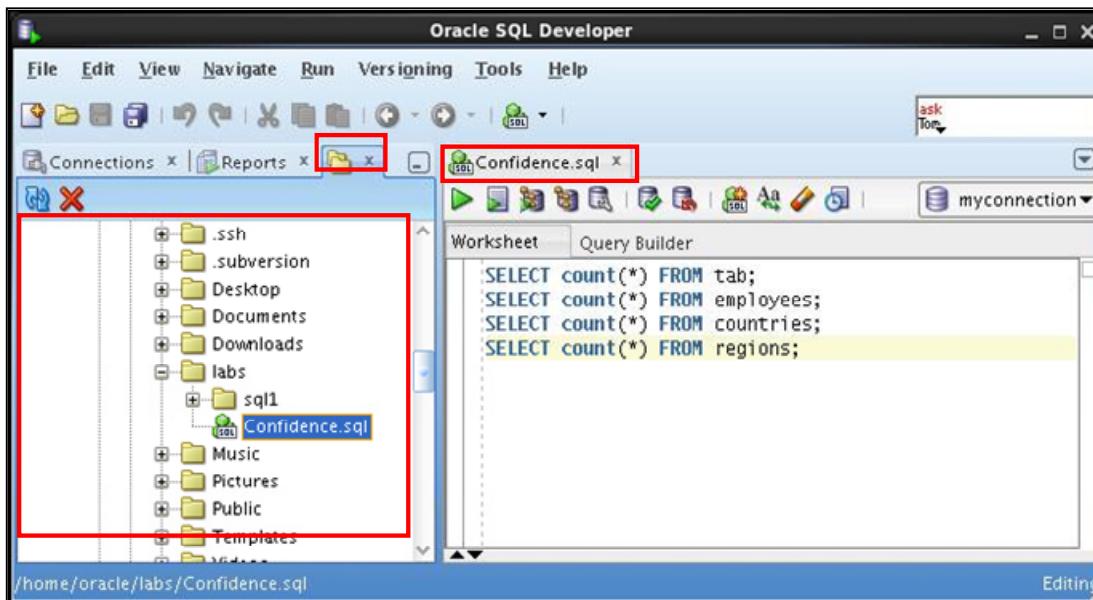
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In SQL Developer, you can also display the structure of a table using the DESCRIBE command. The result of the command is a display of column names and data types, as well as an indication of whether a column must contain data.

Browsing Files

Use the File Navigator to explore the file system and open system files.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

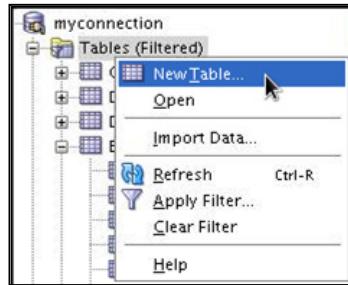
Browsing Database Objects

You can use the File Navigator to browse and open system files.

- To view the File Navigator, click the View tab and select Files, or select View > Files.
- To view the contents of a file, double-click a file name to display its contents in the SQL Worksheet area.

Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
 - Executing a SQL statement in SQL Worksheet
 - Using the context menu
- Edit the objects by using an edit dialog box or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.



ORACLE

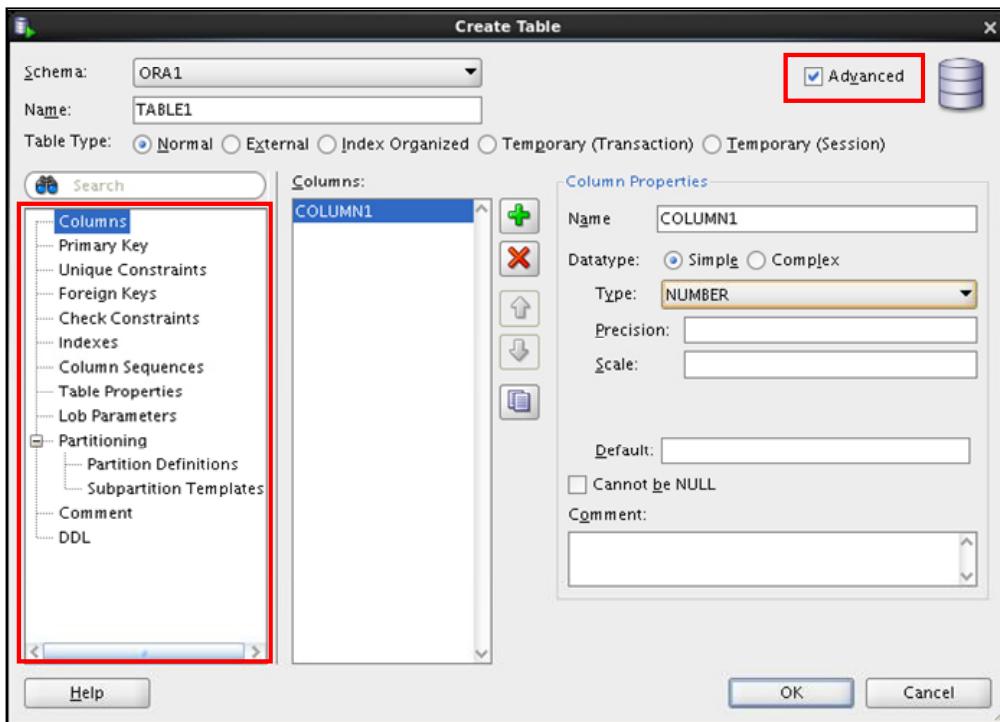
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL Developer supports the creation of any schema object by executing a SQL statement in SQL Worksheet. Alternatively, you can create objects by using the context menus. When created, you can edit objects using an edit dialog box or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the DDL for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows how to create a table using the context menu. To open a dialog box for creating a new table, right-click Tables and select New Table. The dialog boxes to create and edit database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the Advanced check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while you create the table.

The example in the slide shows how to create the DEPENDENTS table by selecting the Advanced check box.

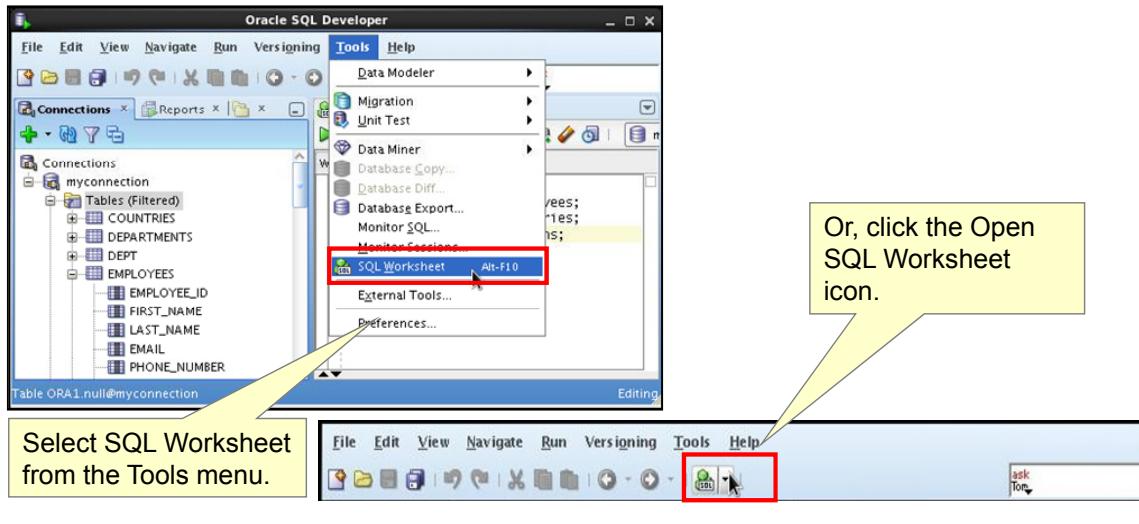
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables and select Create TABLE.
2. In the Create Table dialog box, select Advanced.
3. Specify the column information.
4. Click OK.

Although it is not required, you should also specify a primary key by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created; to do so, right-click the table in the Connections Navigator and select Edit.

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL *Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

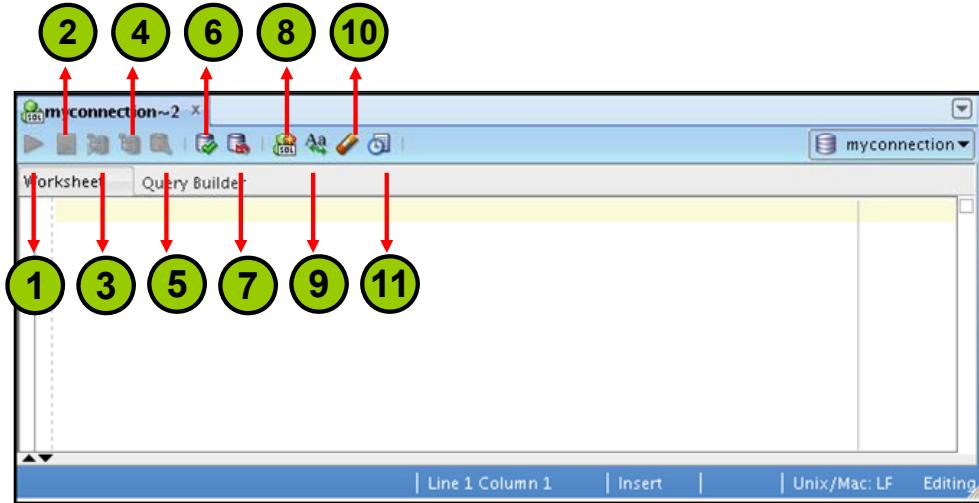
You can specify the actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL Worksheet by using one of the following:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Using the SQL Worksheet

The Oracle logo, which consists of the word "ORACLE" in white capital letters inside a red horizontal bar.

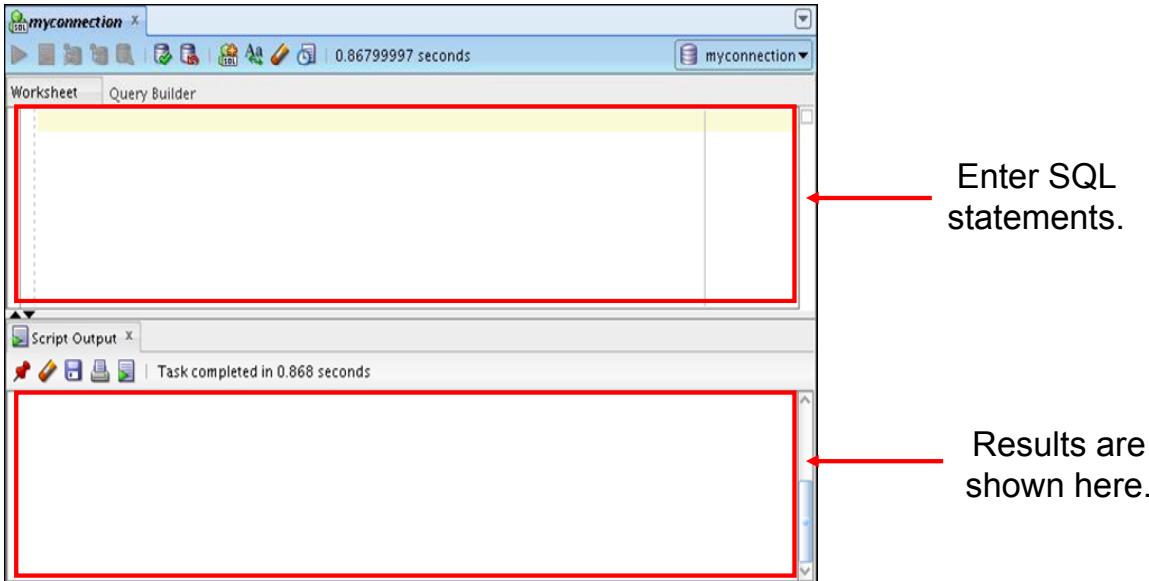
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You may want to use the shortcut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of the SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Run Statement:** Executes the statement where the cursor is located in the Enter SQL Statement box. You can use bind variables in the SQL statements, but not substitution variables.
2. **Run Script:** Executes all the statements in the Enter SQL Statement box by using the Script Runner. You can use substitution variables in the SQL statements, but not bind variables.
3. **Autotrace:** Generates trace information for the statement
4. **Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
5. **SQL Tuning Advisory:** Analyzes high-volume SQL statements and offers tuning recommendations
6. **Commit:** Writes any changes to the database and ends the transaction
7. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction

8. **Unshared SQL Worksheet:** Creates a separate unshared SQL Worksheet for a connection
9. **To Upper/Lower/InitCap:** Changes the selected text to uppercase, lowercase, or initcap, respectively
10. **Clear:** Erases the statement or statements in the Enter SQL Statement box
11. **SQL History:** Displays a dialog box with information about the SQL statements that you have executed

Using the SQL Worksheet



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL Worksheet to the Oracle database. The SQL*Plus commands that are used in SQL Developer must be interpreted by the SQL Worksheet before being passed to the database.

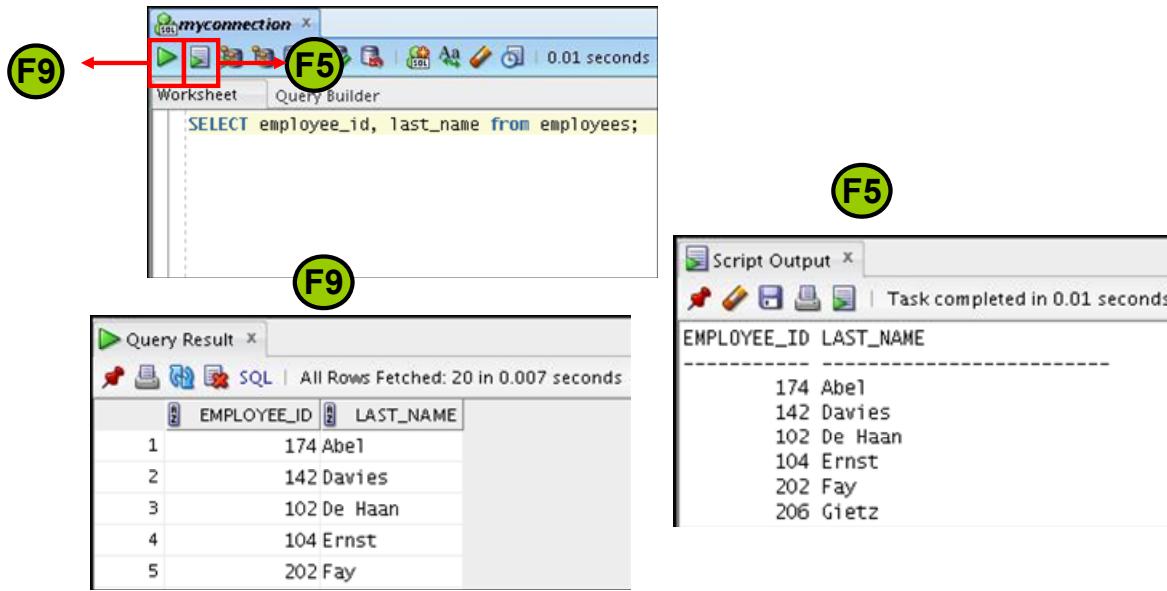
The SQL Worksheet currently supports a number of SQL*Plus commands. Commands that are not supported by the SQL Worksheet are ignored and not sent to the Oracle database. Through the SQL Worksheet, you can execute the SQL statements and some of the SQL*Plus commands.

You can display a SQL Worksheet by using any of the following options:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

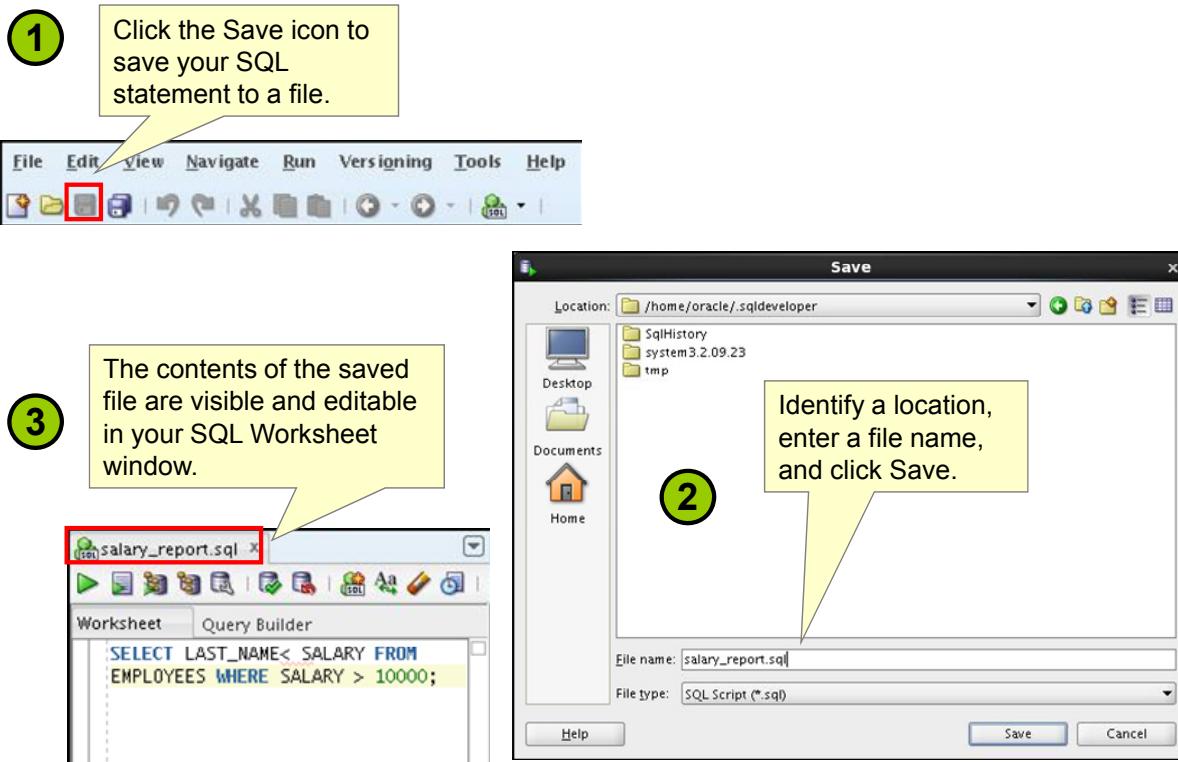


ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows the difference in output for the same query when the F9 key or Execute Statement is used versus the output when F5 or Run Script is used.

Saving SQL Scripts



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

You can save your SQL statements from the SQL Worksheet to a text file. To save the contents of the Enter SQL Statement box, perform the following steps:

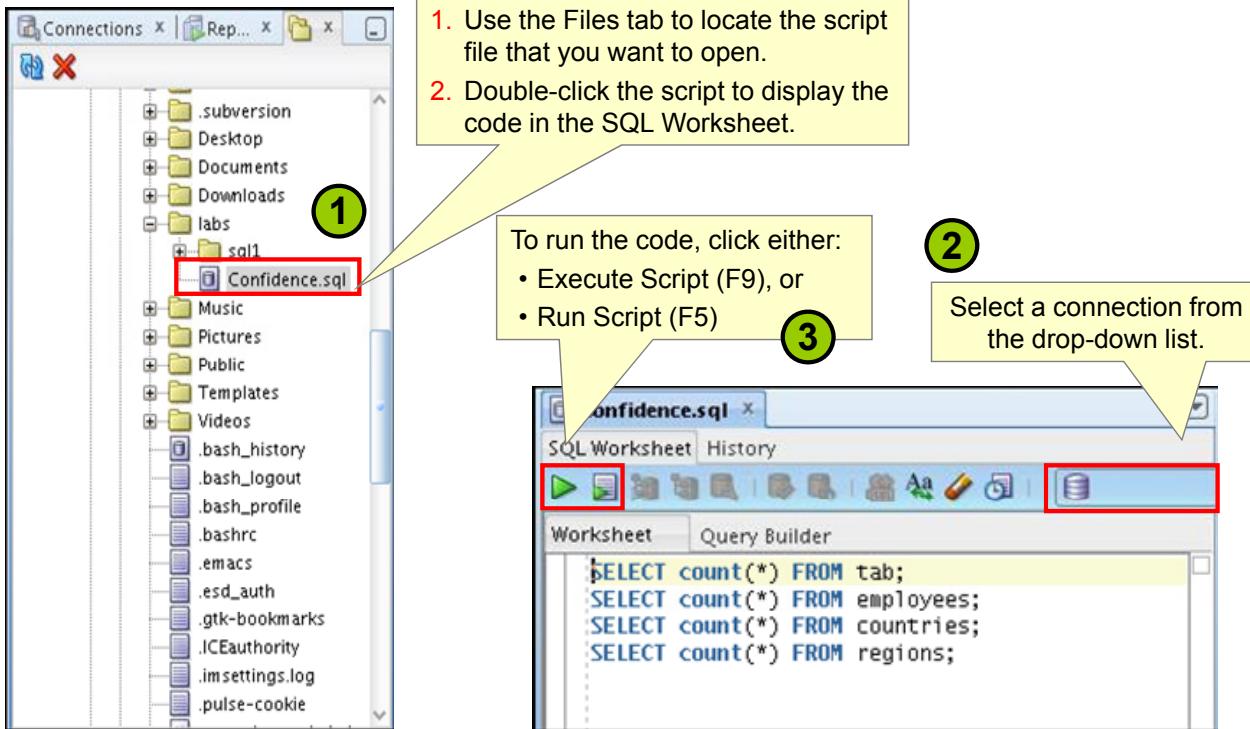
1. Click the Save icon or use the File > Save menu item.
2. In the Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

After you save the contents to a file, the Enter SQL Statement window displays a tabbed page of your file contents. You can have multiple files open at the same time. Each file displays as a tabbed page.

Script Pathing

You can select a default path to look for scripts and to save scripts. Under Tools > Preferences > Database > Worksheet Parameters, enter a value in the “Select default path to look for scripts” field.

Executing Saved Script Files: Method 1



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

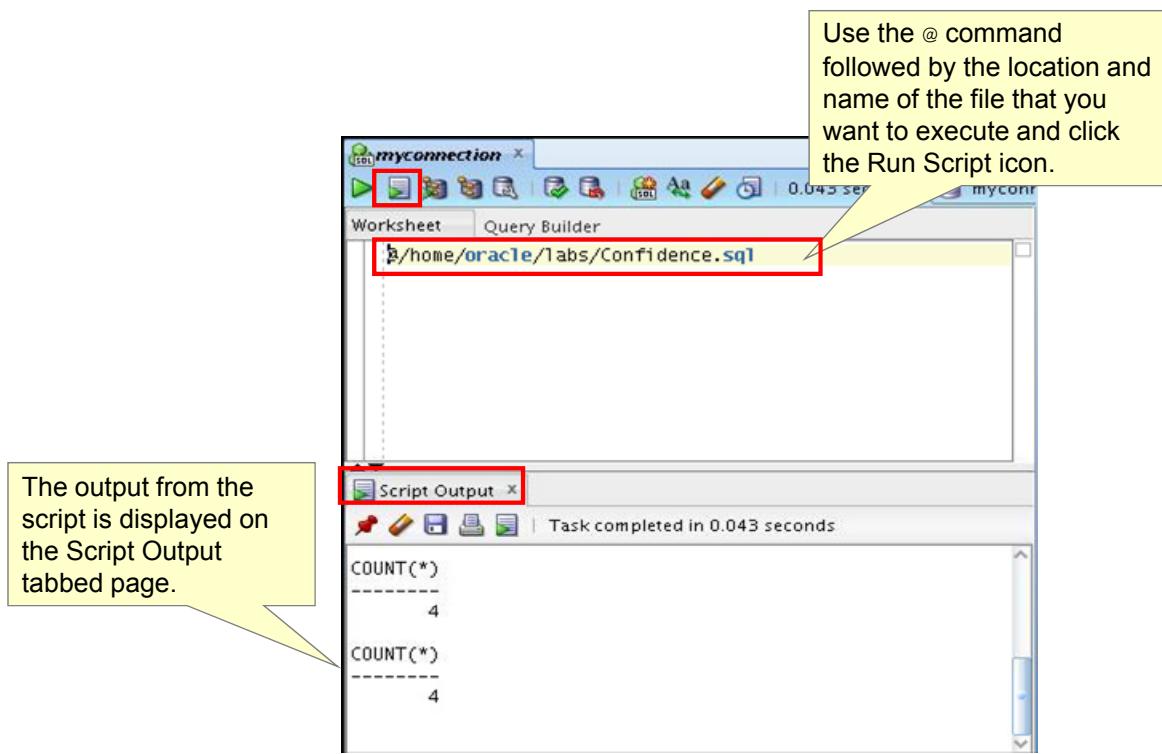
To open a script file and display the code in the SQL Worksheet area, perform the following steps:

1. In the files navigator, select (or navigate to) the script file that you want to open.
2. Double-click the file to open it. The code of the script file is displayed in the SQL Worksheet area.
3. Select a connection from the connection drop-down list.
4. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection that you want to use for the script execution.

Alternatively, you can also do the following:

1. Select File > Open. The Open dialog box is displayed.
2. In the Open dialog box, select (or navigate to) the script file that you want to open.
3. Click Open. The code of the script file is displayed in the SQL Worksheet area.
4. Select a connection from the connection drop-down list.
5. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection that you want to use for the script execution.

Executing Saved Script Files: Method 2



ORACLE

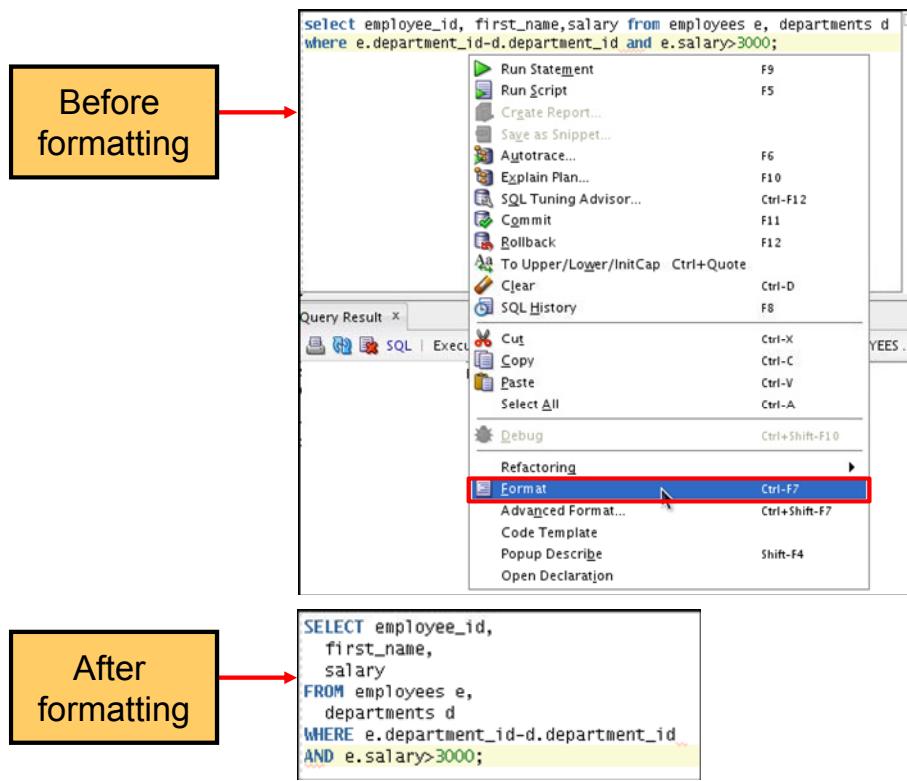
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To run a saved SQL script, perform the following steps:

1. Use the @ command followed by the location and the name of the file that you want to run in the Enter SQL Statement window.
2. Click the Run Script icon.

The results from running the file are displayed on the Script Output tabbed page. You can also save the script output by clicking the Save icon on the Script Output tabbed page. The File Save dialog box appears and you can identify a name and location for your file.

Formatting the SQL Code



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

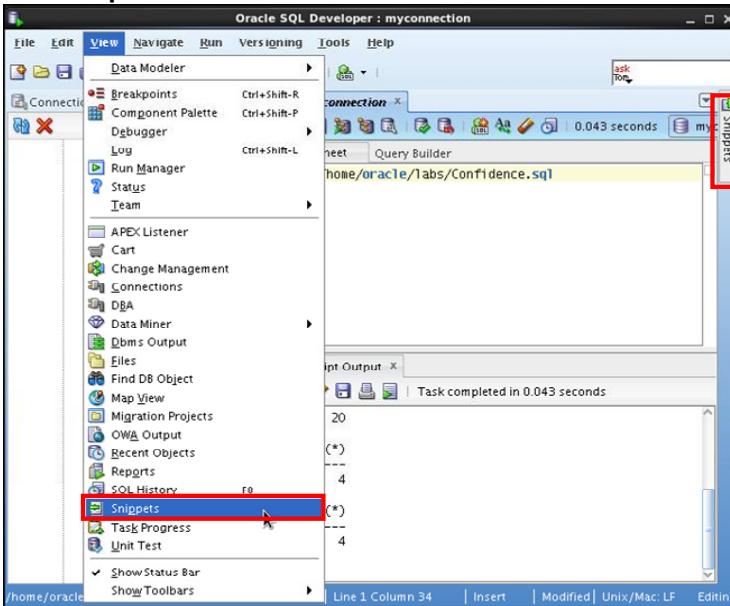
You may want to format the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

To format the SQL code, right-click in the statement area and select Format.

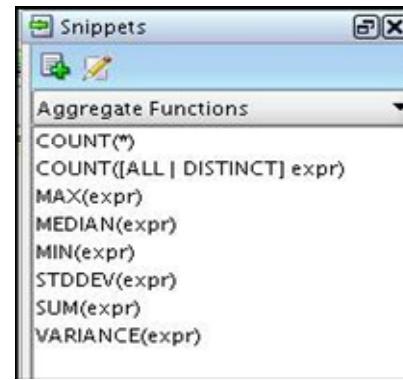
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

Using Snippets

Snippets are code fragments that may be just syntax or examples.



When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category that you want.



ORACLE

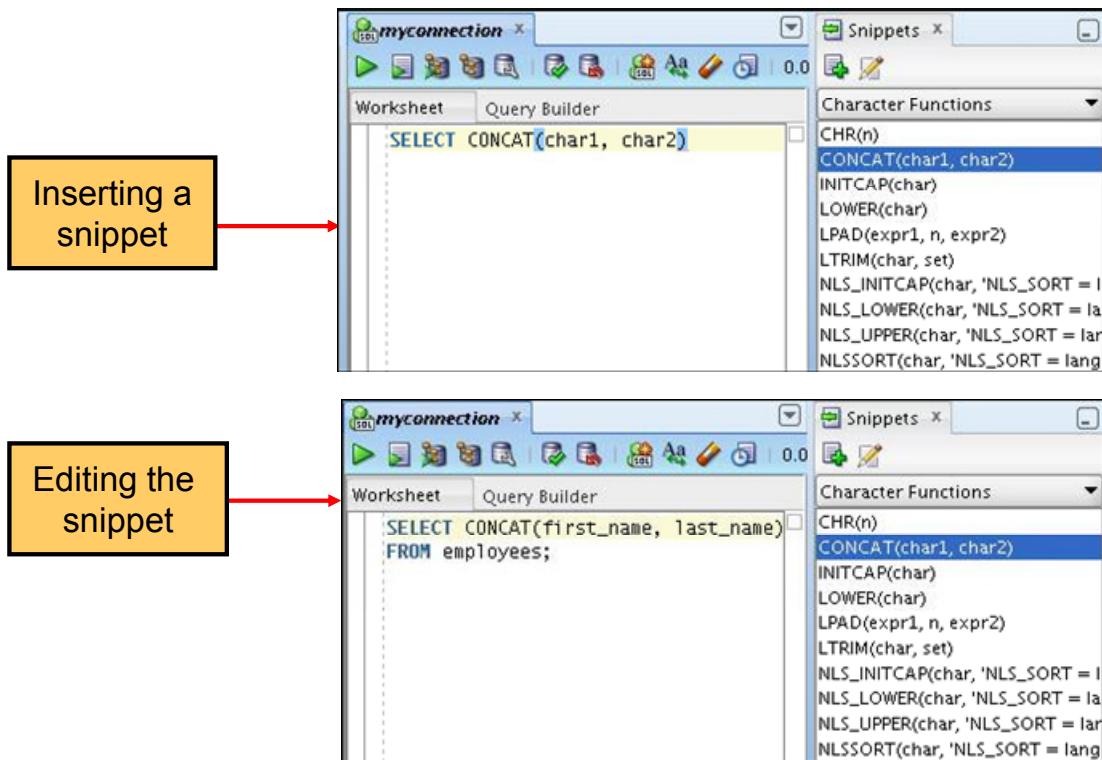
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You may want to use certain code fragments when you use the SQL Worksheet or create or edit a PL/SQL function or procedure. SQL Developer has a feature called Snippets. Snippets are code fragments such as SQL functions, optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets to the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed on the right. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Using Snippets: Example



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

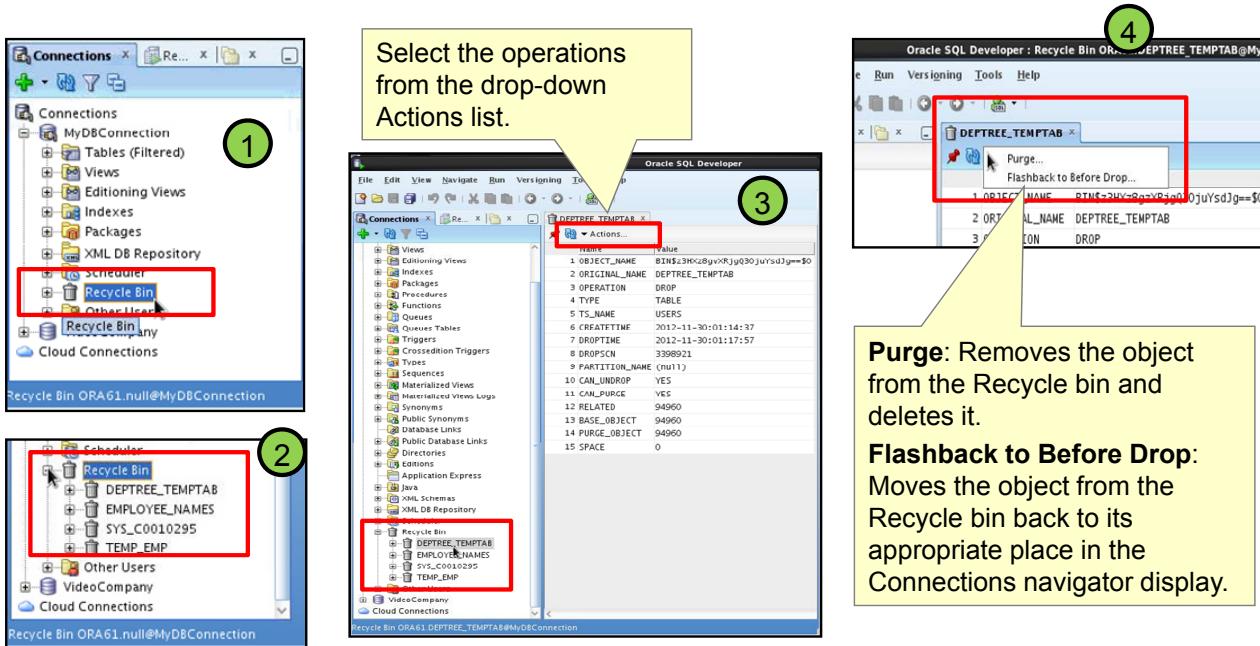
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window to the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT (char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added as in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Using Recycle Bin

The Recycle bin holds objects that have been dropped.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

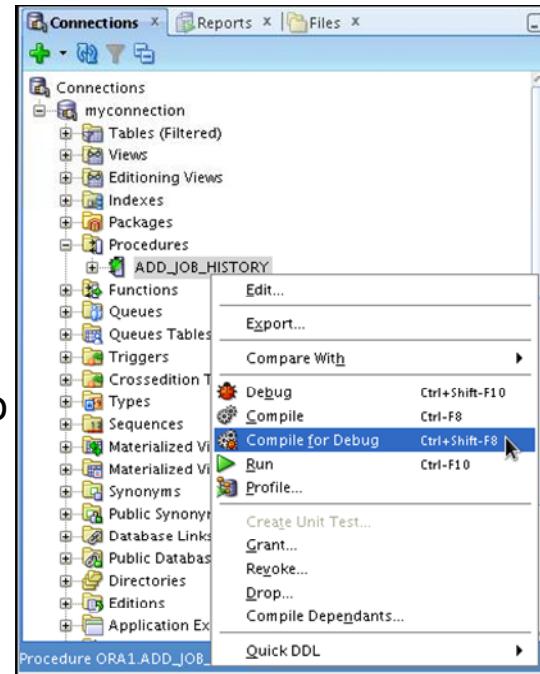
The recycle bin is a data dictionary table containing information about dropped objects. Dropped tables and any associated objects such as indexes, constraints, nested tables, and the likes are not removed and still occupy space. They continue to count against user space quotas, until specifically purged from the recycle bin or the unlikely situation where they must be purged by the database because of tablespace space constraints.

To use the Recycle Bin, perform the following steps:

1. In the Connections navigator, select (or navigate to) the Recycle Bin.
2. Expand Recycle Bin and click the object name. The object details are displayed in the SQL Worksheet area.
3. Click the Actions drop-down list and select the operation you want to perform on the object.

Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the Compile for Debug option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use the Debug menu options to set breakpoints, and to perform step into, step over tasks.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution, but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon on the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of more frequently accessed and more valid objects.

These options are also available as icons on the Debugging tab of the output window.

Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.

Owner	Name	Type	Referenced_Owner	Referenced_Name	Referenced_Type
APEX_040100	APEX	PROCEDURE	APEX_040100	WWV_FLOW	PACKAGE
APEX_040100	APEX	PROCEDURE	APEX_040100	WWV_FLOW_ISC	PACKAGE
APEX_040100	APEX	PROCEDURE	APEX_040100	WWV_FLOW_SECURITY	PACKAGE
APEX_040100	APEX	PROCEDURE	SYS	STANDARD	PACKAGE
APEX_040100	APEX	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
APEX_040100	APEXWS	PACKAGE	SYS	STANDARD	PACKAGE
APEX_040100	APEXADMIN	PROCEDURE	APEX_040100	F	PROCEDURE
APEX_040100	APEX_ADMIN	PROCEDURE	SYS	STANDARD	PACKAGE
APEX_040100	APEX_ADMIN	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	NV	FUNCTION
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOWS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_APPLICATION_GROUPS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_AUTHENTIFICATIONS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_COMPANIES	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_COMPANY_SCHEMAS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_COMPUTATIONS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_ICON_BAR	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_INSTALL_SCRIPTS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_ITEMS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WWV_FLOW_LANGUAGE_MAP	TABLE



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

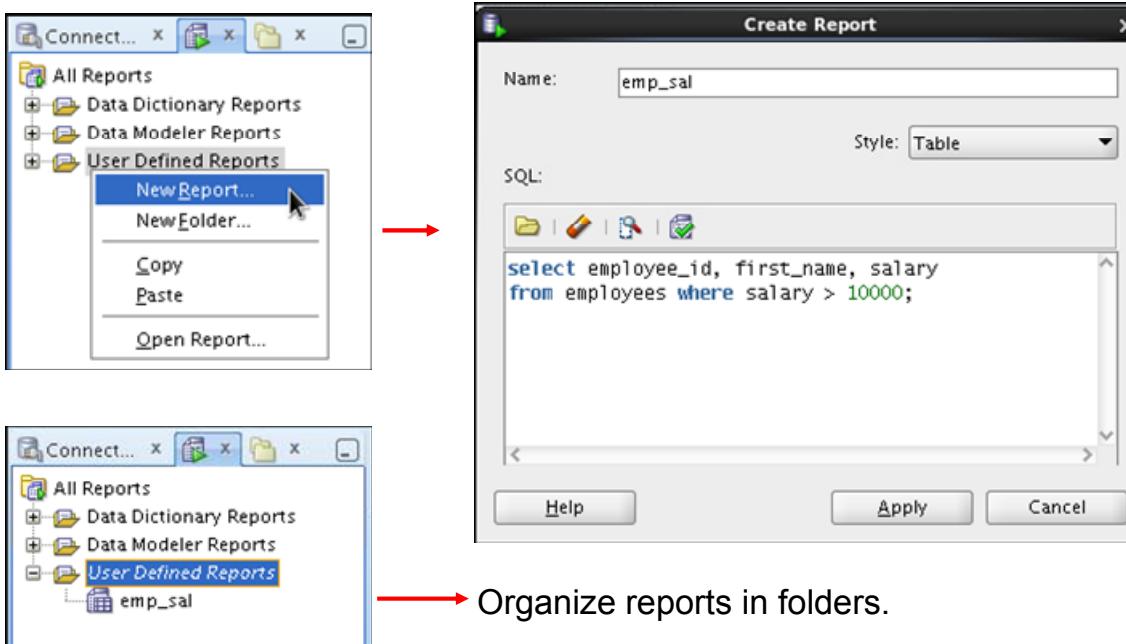
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-Defined reports

To display reports, click the Reports tab on the left of the window. Individual reports are displayed in tabbed panes on the right of the window; and for each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

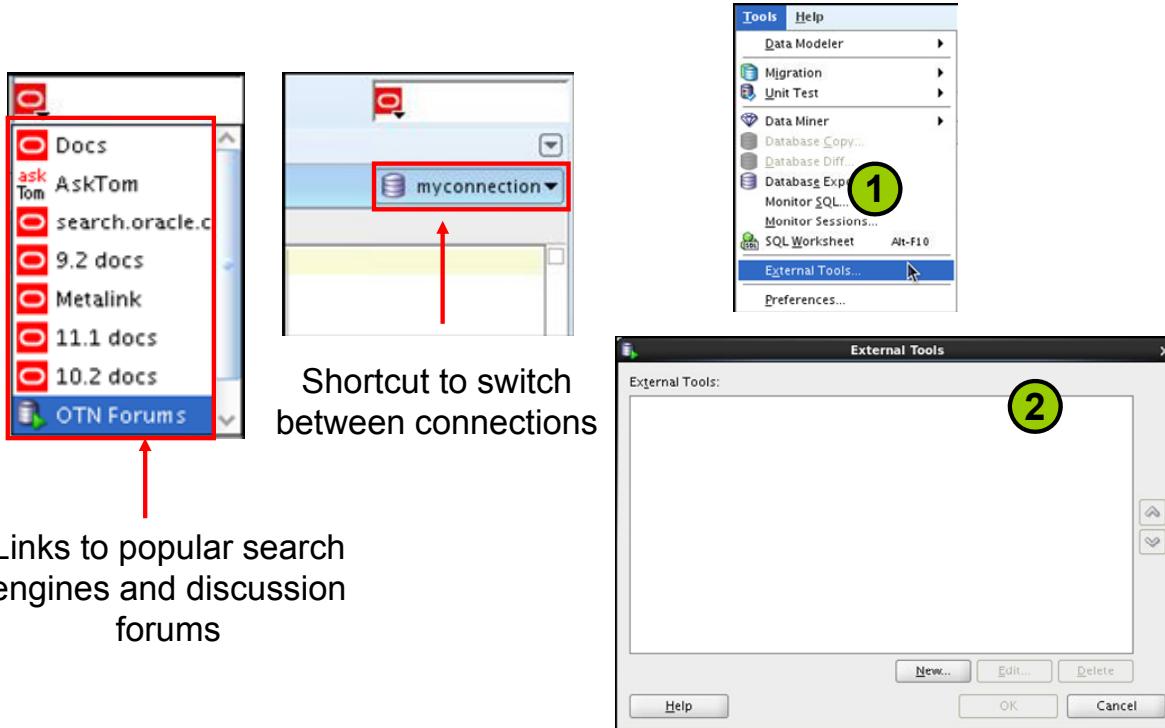
1. Right-click the User Defined Reports node under Reports and select Add Report.
2. In the Create Report dialog box, specify the report name and the SQL query to retrieve information for the report. Then click Apply.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details of employees with `salary >= 10000`.

The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the cursor stays briefly over the report name in the Reports navigator display.

You can organize user-defined reports in folders and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined Reports node or any folder name under that node and select Add Folder. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` in the directory for user-specific information.

Search Engines and External Tools



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

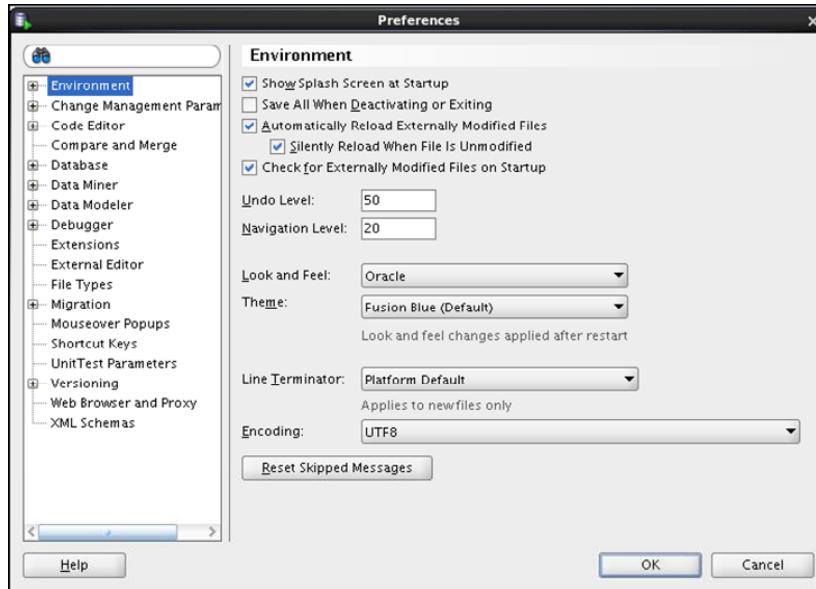
To enhance the productivity of developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to the tools that you do not use frequently. To do so, perform the following steps:

1. From the Tools menu, select External Tools.
2. In the External Tools dialog box, select New to add new tools. Select Delete to remove any tool from the list.

Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

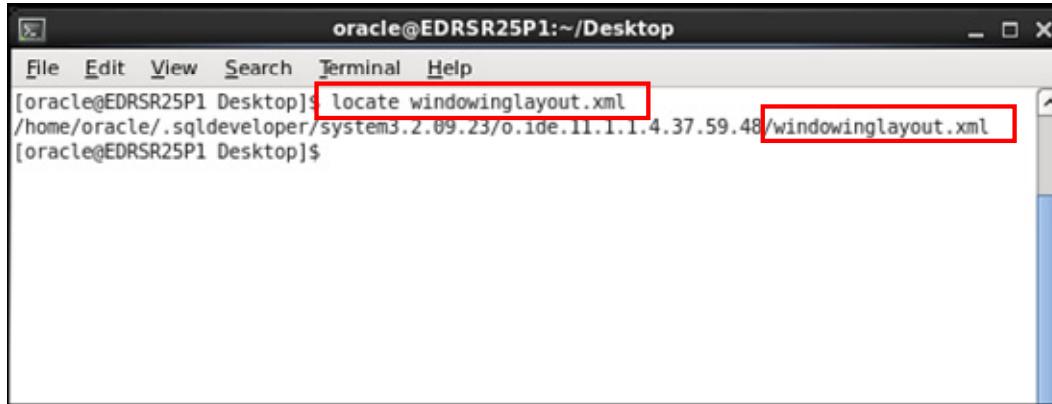
You can customize many aspects of the SQL Developer interface and environment by modifying SQL Developer preferences according to your needs. To modify SQL Developer preferences, select Tools, and then Preferences.

The preferences are grouped into the following categories:

- Environment
- Change Management parameter
- Code Editors
- Compare and Merge
- Database
- Data Miner
- Data Modeler
- Debugger
- Extensions
- External Editor
- File Types
- Migration

- Mouseover Popups
- Shortcut Keys
- Unit Test Parameters
- Versioning
- Web Browser and Proxy
- XML Schemas

Resetting the SQL Developer Layout



A screenshot of a terminal window titled "oracle@EDRSR25P1:~/Desktop". The window shows the command "locate windowinglayout.xml" being run, and the output is "/home/oracle/.sqldeveloper/system3.2.09.23/o.ide.11.1.1.4.37.59.48/windowinglayout.xml". The line containing the file path is highlighted with a red rectangle.



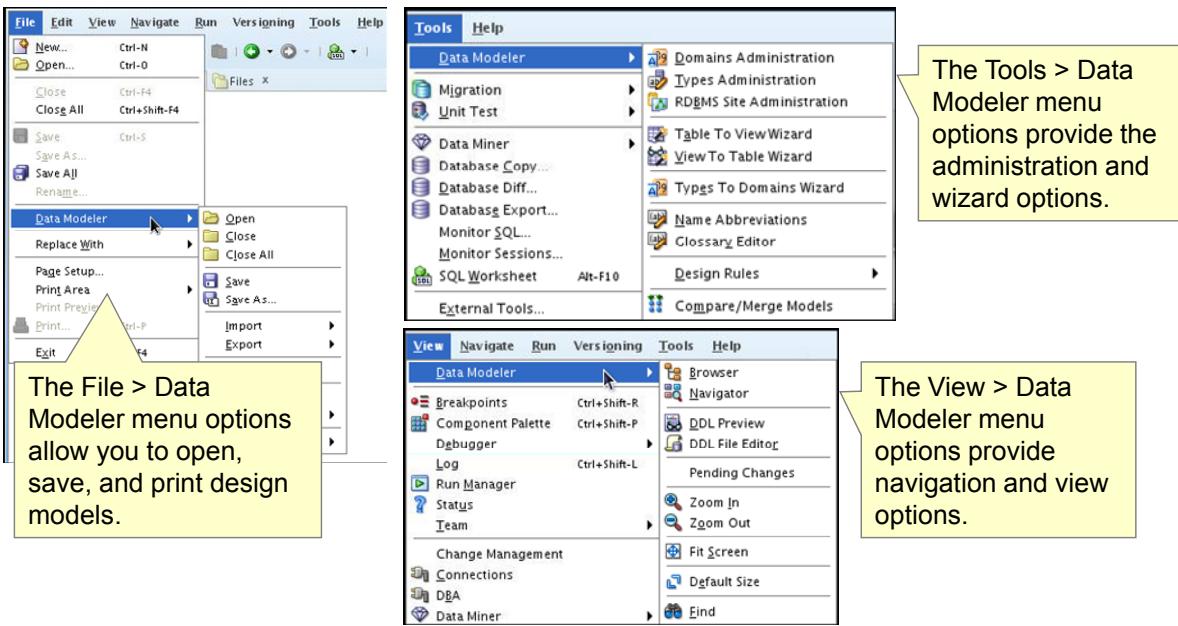
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

While working with SQL Developer, if the Connections Navigator disappears or if you cannot dock the Log window in its original place, perform the following steps to fix the problem:

1. Exit SQL Developer.
2. Open a terminal window and use the locate command to find the location of `windowinglayout.xml`.
3. Go to the directory that has `windowinglayout.xml` and delete it.
4. Restart SQL Developer.

Data Modeler in SQL Developer

SQL Developer includes an integrated version of SQL Developer Data Modeler.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the integrated version of the SQL Developer Data Modeler, you can:

- Create, open, import, and save a database design
- Create, modify, and delete Data Modeler objects

To display Data Modeler in a pane, click Tools, and then Data Modeler. The Data Modeler menu under Tools includes additional commands, for example, that enable you to specify design rules and preferences.

Summary

In this appendix, you should have learned how to use SQL Developer to do:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports
- Browse the Data Modeling options in SQL Developer



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use SQL Worksheet to run SQL statements and scripts. SQL Developer enables you to create and save your own special set of reports for repeated use.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only



Using SQL*Plus



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

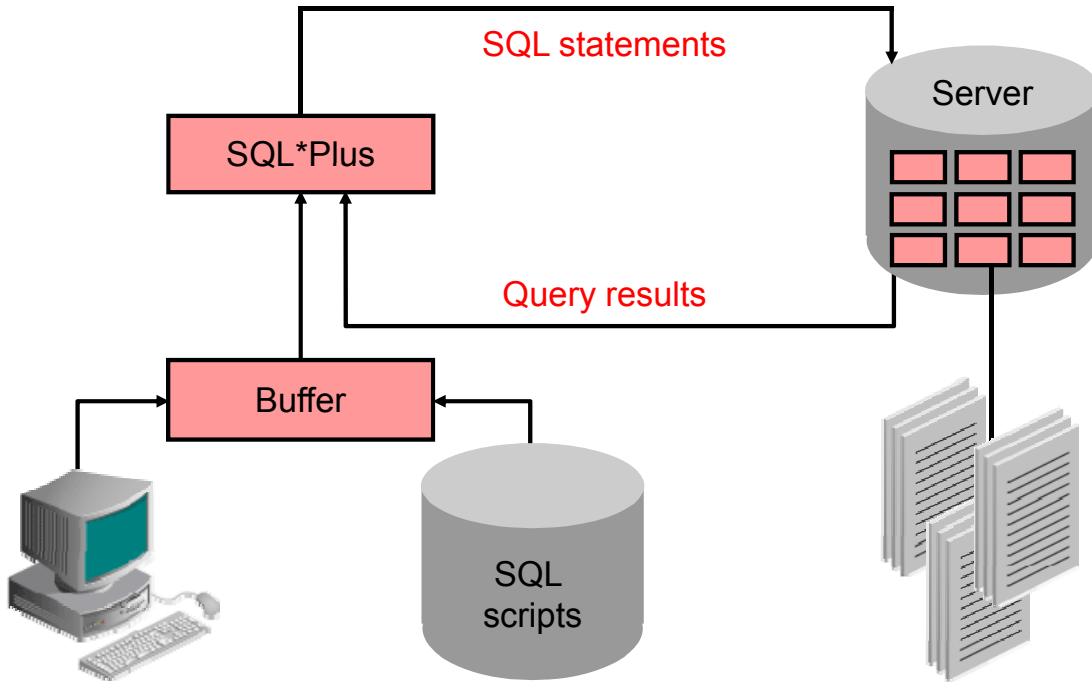
- Log in to SQL*Plus
- Edit SQL commands
- Format the output using SQL*Plus commands
- Interact with script files



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You might want to create `SELECT` statements that can be used repeatedly. This appendix covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL and SQL*Plus

SQL is a command language that is used for communication with the Oracle server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the *SQL buffer* and remains there until you enter a new SQL statement. SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle9i Server for execution. It contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

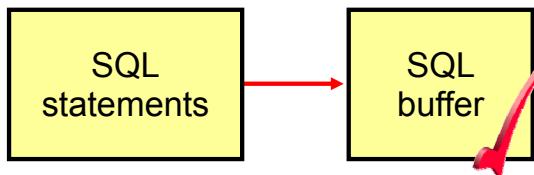
Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases

SQL Statements Versus SQL*Plus Commands

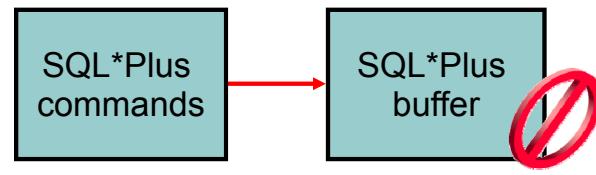
SQL

- A language
- ANSI-standard
- Keywords cannot be abbreviated.
- Statements manipulate data and table definitions in the database.



SQL*Plus

- An environment
- Oracle-proprietary
- Keywords can be abbreviated.
- Commands do not allow manipulation of values in the database.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The following table compares SQL and SQL*Plus:

SQL	SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI)-standard SQL	Is the Oracle-proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time, not stored in the SQL buffer
Does not have a continuation character	Uses a dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute commands immediately	Does not require termination characters; executes commands immediately
Uses functions to perform some formatting	Uses commands to format data

Overview of SQL*Plus

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from the file to buffer to edit.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL*Plus

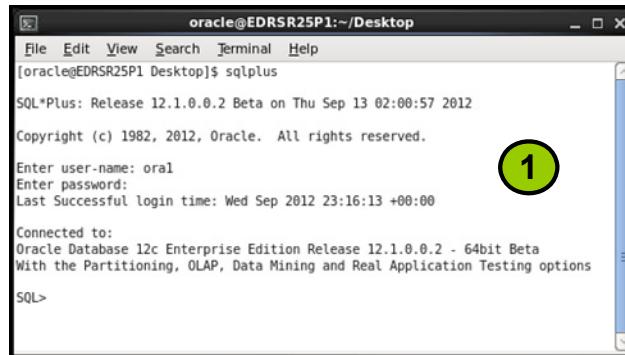
SQL*Plus is an environment in which you can:

- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repeated use in the future

SQL*Plus commands can be divided into the following main categories:

Category	Purpose
Environment	Affect the general behavior of SQL statements for the session
Format	Format query results
File manipulation	Save, load, and run script files
Execution	Send SQL statements from the SQL buffer to the Oracle server
Edit	Modify SQL statements in the buffer
Interaction	Create and pass variables to SQL statements, print variable values, and print messages to the screen
Miscellaneous	Connect to the database, manipulate the SQL*Plus environment, and display column definitions

Logging In to SQL*Plus



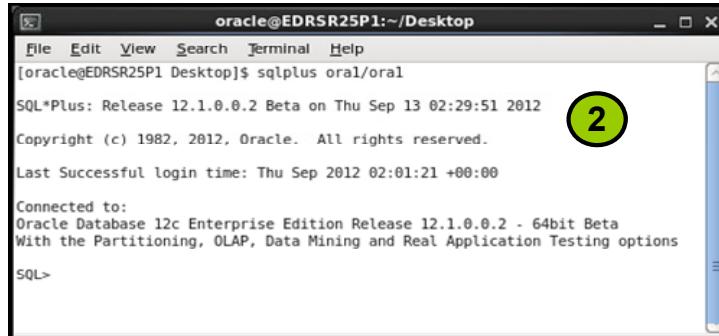
```
oracle@EDRSR25P1:~/Desktop
[oracle@EDRSR25P1 Desktop]$ sqlplus
SQL*Plus: Release 12.1.0.0.2 Beta on Thu Sep 13 02:00:57 2012
Copyright (c) 1982, 2012, Oracle. All rights reserved.

Enter user-name: oral
Enter password:
Last Successful login time: Wed Sep 2012 23:16:13 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

```
sqlplus [username [/password[@database]]]
```



```
oracle@EDRSR25P1:~/Desktop
[oracle@EDRSR25P1 Desktop]$ sqlplus oral/oral
SQL*Plus: Release 12.1.0.0.2 Beta on Thu Sep 13 02:29:51 2012
Copyright (c) 1982, 2012, Oracle. All rights reserved.

Last Successful login time: Thu Sep 2012 02:01:21 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

How you invoke SQL*Plus depends on the type of operating system that you are running Oracle Database on.

To log in from a Linux environment, perform the following steps:

1. Right-click your Linux desktop and select terminal.
2. Enter the `sqlplus` command shown in the slide.
3. Enter the username, password, and database name.

In the syntax:

<code>username</code>	Your database username
<code>password</code>	Your database password (Your password is visible if you enter it here.)
<code>@database</code>	The database connect string

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the password prompt.

Displaying the Table Structure

Use the SQL*Plus DESCRIBE command to display the structure of a table:

```
DESC [RIBE] tablename
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In SQL*Plus, you can display the structure of a table using the DESCRIBE command. The result of the command is a display of column names and data types, as well as an indication if a column must contain data.

In the syntax:

tablename The name of any existing table, view, or synonym that is accessible to the user

To describe the DEPARTMENTS table, use the following command:

```
SQL> DESCRIBE DEPARTMENTS
      Name          Null    Type
----- 
DEPARTMENT_ID           NOT NULL NUMBER(4)
DEPARTMENT_NAME          NOT NULL VARCHAR2(30)
MANAGER_ID                NUMBER(6)
LOCATION_ID                 NUMBER(4)
```

Displaying the Table Structure

```
DESCRIBE departments
```

Name	Null	Type
DEPARTMENT_ID	NOT NULL	NUMBER (4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2 (30)
MANAGER_ID		NUMBER (6)
LOCATION_ID		NUMBER (4)



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide displays information about the structure of the DEPARTMENTS table.
In the result:

- Null: Specifies whether a column must contain data (NOT NULL indicates that a column must contain data.)
- Type: Displays the data type for a column

SQL*Plus Editing Commands

- A [PPEND] *text*
- C [HANGE] / *old* / *new*
- C [HANGE] / *text* /
- CL [EAR] BUFF [ER]
- DEL
- DEL *n*
- DEL *m n*



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

Command	Description
A [PPEND] <i>text</i>	Adds <i>text</i> to the end of the current line
C [HANGE] / <i>old</i> / <i>new</i>	Changes <i>old</i> text to <i>new</i> in the current line
C [HANGE] / <i>text</i> /	Deletes <i>text</i> from the current line
CL [EAR] BUFF [ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL <i>n</i>	Deletes line <i>n</i>
DEL <i>m n</i>	Deletes lines <i>m</i> to <i>n</i> inclusive

Guidelines

- If you press Enter before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer either by entering one of the terminator characters (semicolon or slash) or by pressing Enter twice. The SQL prompt appears.

SQL*Plus Editing Commands

- I [NPUT]
- I [NPUT] *text*
- L [IST]
- L [IST] *n*
- L [IST] *m n*
- R [UN]
- *n*
- *n text*
- 0 *text*



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Command	Description
I [NPUT]	Inserts an indefinite number of lines
I [NPUT] <i>text</i>	Inserts a line consisting of <i>text</i>
L [IST]	Lists all lines in the SQL buffer
L [IST] <i>n</i>	Lists one line (specified by <i>n</i>)
L [IST] <i>m n</i>	Lists a range of lines (<i>m</i> to <i>n</i>) inclusive
R [UN]	Displays and runs the current SQL statement in the buffer
<i>n</i>	Specifies the line to make the current line
<i>n text</i>	Replaces line <i>n</i> with <i>text</i>
0 <i>text</i>	Inserts a line before line 1

Note: You can enter only one SQL*Plus command for each SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (-).

Using LIST, n, and APPEND

```
LIST
 1  SELECT last_name
 2* FROM employees
```

```
1
1* SELECT last_name
```

```
A , job_id
1* SELECT last_name, job_id
```

```
LIST
 1  SELECT last_name, job_id
 2* FROM employees
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- Use the L [IST] command to display the contents of the SQL buffer. The asterisk (*) beside line 2 in the buffer indicates that line 2 is the current line. Any edits that you made apply to the current line.
- Change the number of the current line by entering the number (n) of the line that you want to edit. The new current line is displayed.
- Use the A [PPEND] command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the LIST command.

Note: Many SQL*Plus commands, including LIST and APPEND, can be abbreviated to just their first letter. LIST can be abbreviated to L; APPEND can be abbreviated to A.

Using the CHANGE Command

```
LIST  
1* SELECT * from employees
```

```
c/employees/departments  
1* SELECT * from departments
```

```
LIST  
1* SELECT * from departments
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- Use `L[IST]` to display the contents of the buffer.
- Use the `C[HANGE]` command to alter the contents of the current line in the SQL buffer. In this case, replace the `employees` table with the `departments` table. The new current line is displayed.
- Use the `L[IST]` command to verify the new contents of the buffer.

SQL*Plus File Commands

- `SAVE filename`
- `GET filename`
- `START filename`
- `@ filename`
- `EDIT filename`
- `SPOOL filename`
- `EXIT`



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL statements communicate with the Oracle server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands described in the following table:

Command	Description
<code>SAV [E] filename [.ext]</code> [REP [LACE] APP [END]]	Saves the current contents of the SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.
<code>GET filename [.ext]</code>	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
<code>STA [RT] filename [.ext]</code>	Runs a previously saved command file
<code>@ filename</code>	Runs a previously saved command file (same as START)
<code>ED [IT]</code>	Invokes the editor and saves the buffer contents to a file named afiedt.buf
<code>ED [IT] [filename [.ext]]</code>	Invokes the editor to edit the contents of a saved file
<code>SPO [OL] [filename [.ext]] OFF OUT</code>	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
<code>EXIT</code>	Quits SQL*Plus

Using the **SAVE**, **START** Commands

```
LIST
```

```
1  SELECT last_name, manager_id, department_id
2* FROM employees
```

```
SAVE my_query
```

```
Created file my_query
```

```
START my_query
```

```
LAST_NAME
```

```
MANAGER_ID DEPARTMENT_ID
```

```
-----
```

```
King
```

```
90
```

```
Kochhar
```

```
100
```

```
90
```

```
...
```

```
107 rows selected.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SAVE

Use the **SAVE** command to store the current contents of the buffer in a file. Thus, you can store frequently used scripts for use in the future.

START

Use the **START** command to run a script in SQL*Plus. You can also, alternatively, use the symbol @ to run a script.

```
@my_query
```

SERVERTOUTPUT Command

- Use the SET SERVEROUT [PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

```
SET SERVEROUT [PUT] {ON | OFF} [SIZE {n | UNL [IMITED]}]
[FOR [MAT] {WRA [PPED] | WOR [D_WWRAPPED] | TRU [NCATED]}]
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Most of the PL/SQL programs perform input and output through SQL statements, to store data in database tables or query those tables. All other PL/SQL input/output is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures, such as PUT_LINE. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

Note

- SIZE sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is UNLIMITED. n cannot be less than 2000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see *Oracle Database PL/SQL User's Guide and Reference 12c*.

Using the SQL*Plus SPOOL Command

```
SPO [OL] [file_name [.ext]] [CRE [ATE] | REP [LACE] |
APP [END]] | OFF | OUT]
```

Option	Description
file_name [.ext]	Spools output to the specified file name
CRE [ATE]	Creates a new file with the name specified
REP [LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP [END]	Adds the contents of the buffer to the end of the file that you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The SPOOL command stores query results in a file or optionally sends the file to a printer. The SPOOL command has been enhanced. You can now append to, or replace an existing file, where previously you could use SPOOL only to create (and replace) a file. REPLACE is the default.

To spool the output generated by commands in a script without displaying the output on screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect the output from commands that run interactively.

You must use quotation marks around file names that contain white space. To create a valid HTML file using SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. SET SQLPLUSCOMPAT [IBILITY] to 9.2 or earlier to disable the CREATE, APPEND, and SAVE parameters.

Using the AUTOTRACE Command

- It displays a report after the successful execution of SQL DML statements such as SELECT, INSERT, UPDATE, or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT [RACE] {ON | OFF | TRACE [ONLY]} [EXP [LAIN]]  
[STATISTICS]
```

```
SET AUTOTRACE ON  
-- The AUTOTRACE report includes both the optimizer  
-- execution path and the SQL statement execution  
-- statistics
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

EXPLAIN shows the query execution path by performing an EXPLAIN PLAN. STATISTICS displays SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The DBMS_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several predefined formats.

Note

- For additional information about the package and subprograms, refer to *Oracle Database PL/SQL Packages and Types Reference 12c*.
- For additional information about the EXPLAIN PLAN, refer to *Oracle Database SQL Reference 12c*.
- For additional information about Execution Plans and the statistics, refer to *Oracle Database Performance Tuning Guide 12c*.

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL*Plus is an execution environment that you can use to send SQL commands to the database server and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.

REF Cursors

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Cursor Variables

- Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself.
- In PL/SQL, a pointer is declared as REF X, where REF is short for REFERENCE and X stands for a class of objects.
- A cursor variable has the data type REF CURSOR.
- A cursor is static, but a cursor variable is dynamic.
- Cursor variables give you more flexibility.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself. Thus, declaring a cursor variable creates a pointer, not an item. In PL/SQL, a pointer has the data type REF X, where REF is short for REFERENCE and X stands for a class of objects. A cursor variable has the REF CURSOR data type.

Like a cursor, a cursor variable points to the current row in the result set of a multirow query. However, cursors differ from cursor variables the way constants differ from variables. A cursor is static, but a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query. This gives you more flexibility.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, and then pass it as an input host variable (bind variable) to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side. The Oracle Server also has a PL/SQL engine. You can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs).

Using Cursor Variables

- You can use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients.
- PL/SQL can share a pointer to the query work area in which the result set is stored.
- You can pass the value of a cursor variable freely from one scope to another.
- You can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single roundtrip.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, an Oracle Forms application, and the Oracle Server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block that is embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from the client to the server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, and then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single roundtrip.

A cursor variable holds a reference to the cursor work area in the Program Global Area (PGA) instead of addressing it with a static name. Because you address this area by a reference, you gain the flexibility of a variable.

Defining REF CURSOR Types

Define a REF CURSOR type:

```
Define a REF CURSOR type  
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

Example:

```
DECLARE  
TYPE DeptCurTyp IS REF CURSOR RETURN  
departments%ROWTYPE;  
dept_cv DeptCurTyp;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To define a REF CURSOR, you perform two steps. First, you define a REF CURSOR type, and then you declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package using the following syntax:

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

where:

ref_type_name Is a type specifier used in subsequent declarations of cursor variables

return_type Represents a record or a row in a database table

In this example, you specify a return type that represents a row in the database table DEPARTMENT.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). As the next example shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
```

```
TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE; -- strong
```

```
TYPE GenericCurTyp IS REF CURSOR; -- weak
```

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Declaring Cursor Variables

After you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable DEPT_CV:

```
DECLARE
    TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
    dept_cv DeptCurTyp; -- declare cursor variable
```

Note: You cannot declare cursor variables in a package. Unlike packaged variables, cursor variables do not have persistent states. Remember, declaring a cursor variable creates a pointer, not an item. Cursor variables cannot be saved in the database; they follow the usual scoping and instantiation rules.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable, as follows:

```
DECLARE
    TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
    tmp_cv TmpCurTyp; -- declare cursor variable
    TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
    emp_cv EmpCurTyp; -- declare cursor variable
```

Similarly, you can use %TYPE to provide the data type of a record variable, as the following example shows:

```
DECLARE
    dept_rec departments%ROWTYPE; -- declare record variable
    TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
    dept_cv DeptCurTyp; -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```
DECLARE
    TYPE EmpRecTyp IS RECORD (
        empno NUMBER(4),
        ename VARCHAR2(10),
        sal    NUMBER(7,2));
    TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
    emp_cv EmpCurTyp; -- declare cursor variable
```

Cursor Variables as Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following example, you define the REF CURSOR type `EmpCurTyp`, and then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE  
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;  
    PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

Using the OPEN-FOR, FETCH, and CLOSE Statements

- The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, and positions the cursor to point to the first row of the result set.
- The FETCH statement returns a row from the result set of a multirow query, assigns the values of the select-list items to the corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row.
- The CLOSE statement disables a cursor variable.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You use three statements to process a dynamic multirow query: OPEN-FOR, FETCH, and CLOSE. First, you “open” a cursor variable “for” a multirow query. Then you “fetch” rows from the result set one at a time. When all the rows are processed, you “close” the cursor variable.

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, positions the cursor to point to the first row of the results set, and then sets the rows-processed count kept by %ROWCOUNT to zero. Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is:

```
OPEN {cursor_variable | :host_cursor_variable} FOR  
dynamic_string  
      [USING bind_argument [, bind_argument] ...];
```

where CURSOR_VARIABLE is a weakly typed cursor variable (one without a return type), HOST_CURSOR_VARIABLE is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multirow query.

In the following example, the syntax declares a cursor variable, and then associates it with a dynamic SELECT statement that returns rows from the EMPLOYEES table:

```

DECLARE
  TYPE EmpCurTyp IS REF CURSOR; -- define weak REF CURSOR      type
  emp_cv  EmpCurTyp; -- declare cursor variable
  my_ename VARCHAR2(15);
  my_sal   NUMBER := 1000;
BEGIN
  OPEN emp_cv FOR -- open cursor variable
    'SELECT last_name, salary FROM employees WHERE salary >
     :s'
    USING my_sal;
  ...
END;

```

Any bind arguments in the query are evaluated only when the cursor variable is opened. Thus, to fetch rows from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values each time.

Fetching from the Cursor Variable

The FETCH statement returns a row from the result set of a multirow query, assigns the values of the select-list items to the corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row. Use the following syntax:

```

FETCH {cursor_variable | :host_cursor_variable}
  INTO {define_variable[, define_variable]... | record};

```

Continuing the example, fetch rows from the cursor variable `emp_cv` into the define variables `MY_ENAME` and `MY_SAL`:

```

LOOP
  FETCH emp_cv INTO my_ename, my_sal; -- fetch next row
  EXIT WHEN emp_cv%NOTFOUND; -- exit loop when last row is
    fetched
  -- process row
END LOOP;

```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible variable or field in the INTO clause. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set. If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Closing the Cursor Variable

The CLOSE statement disables a cursor variable. After that, the associated result set is undefined. Use the following syntax:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this example, when the last row is processed, close the emp_cv cursor variable:

```
LOOP
  FETCH emp_cv INTO my_ename, my_sal;
  EXIT WHEN emp_cv%NOTFOUND;
  -- process row
END LOOP;
CLOSE emp_cv;  -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises INVALID_CURSOR.

Example of Fetching

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR;
  emp_cv    EmpCurTyp;
  emp_rec   employees%ROWTYPE;
  sql_stmt  VARCHAR2(200);
  my_job    VARCHAR2(10) := 'ST_CLERK';
BEGIN
  sql_stmt := 'SELECT * FROM employees
               WHERE job_id = :j';
  OPEN emp_cv FOR sql_stmt USING my_job;
  LOOP
    FETCH emp_cv INTO emp_rec;
    EXIT WHEN emp_cv%NOTFOUND;
    -- process record
  END LOOP;
  CLOSE emp_cv;
END;
/
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows that you can fetch rows from the result set of a dynamic multirow query into a record. You must first define a REF CURSOR type, `EmpCurTyp`. You then define a cursor variable `emp_cv`, of the type `EmpCurTyp`. In the executable section of the PL/SQL block, the `OPEN-FOR` statement associates the cursor variable `emp_cv` with the multirow query, `sql_stmt`. The `FETCH` statement returns a row from the result set of a multirow query and assigns the values of the select-list items to `EMP_REC` in the `INTO` clause. When the last row is processed, close the `emp_cv` cursor variable.

Commonly Used SQL Commands

ORACLE®

Copyright © 2013, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

This lesson explains how to obtain data from one or more tables using the SELECT statement, how to use DDL statements to alter the structure of data objects, how to manipulate data in the existing schema objects by using the DML statements, how to manage the changes made by DML statements, and how to use joins to display data from multiple tables using SQL:1999 join syntax.

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 © 2013, 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:

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

ORACLE

Copyright © 2013, 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 © 2013, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

The ORDER BY clause specifies the order in which the rows should be displayed. The rows can be sorted in ascending or descending fashion. By default, the rows are displayed in ascending order.

The example in the slide retrieves rows from the `employees` table ordered first by ascending order of `department_id` and then by descending order of `salary`.

GROUP BY Clause

- Use the optional GROUP BY clause to group columns that have matching values into subsets.
- Each group has no 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 © 2013, 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 © 2013, 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 © 2013, 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.
- Example1:

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

- Example 2:

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



Copyright © 2013, 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 © 2013, 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 © 2013, 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 using the GRANT statement:
 - ALL PRIVILEGES
 - DELETE
 - INSERT
 - REFERENCES
 - SELECT
 - UPDATE



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle Database provides a variety of privilege types to grant privileges to a user or role:

- Use the ALL PRIVILEGES privilege type to grant all privileges to the user or role for the specified table.
- Use the DELETE privilege type to grant permission to delete rows from the specified table.
- Use the INSERT privilege type to grant permission to insert rows into the specified table.
- Use the REFERENCES privilege type to grant permission to create a foreign key reference to the specified table.
- Use the SELECT privilege type to grant permission to perform SELECT statements on a table or view.
- Use the UPDATE privilege type to grant permission to use the UPDATE statement on the specified table.

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 © 2013, 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 © 2013, 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:
 - Re-grant object privileges
 - Re-create indexes, integrity constraints, and triggers.
 - Re-specify 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 © 2013, Oracle and/or its affiliates. All rights reserved.

Data manipulation language (DML) statements enable you to query or change the contents of an existing schema object. These statements are most frequently used to:

- Add new rows of data to a table or view by specifying a list of column values or by using a subquery to select and manipulate existing data
- Change column values in the existing rows of a table or view
- Remove rows from tables or views

A collection of DML statements that forms a logical unit of work is called a transaction. Unlike DDL statements, DML statements do not implicitly commit the current transaction.

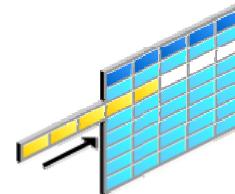
INSERT Statement

- 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 © 2013, 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

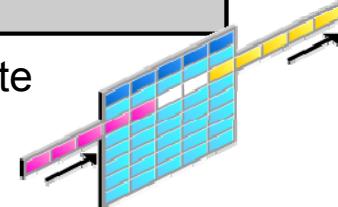
- Use the UPDATE statement to modify the existing rows in a table.
- Update more than one row at a time (if required).

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

- Example:

```
UPDATE      copy_emp  
SET         employee_id = 110;  
22 rows updated
```

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



ORACLE

Copyright © 2013, 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.

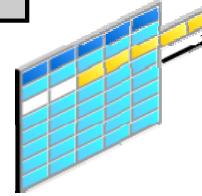
DELETE Statement

- 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 © 2013, 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A transaction is a sequence of SQL statements that Oracle Database treats as a single unit. Transaction control statements are used in a database to manage the changes made by DML statements and to group these statements into transactions.

Each transaction is assigned a unique `transaction_id` and it groups SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone from the database.

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 © 2013, 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 © 2013, 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 © 2013, 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 © 2013, 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:

<i>table1.column</i>	Denotes the table and column from which data is retrieved
<i>table1.column1</i> = <i>table2.column2</i>	Is the condition that joins (or relates) the tables together

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 © 2013, Oracle and/or its affiliates. All rights reserved.

To join tables, you can use Oracle's join syntax.

Note: Before the Oracle9*i* release, the join syntax was proprietary. The SQL:1999-compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the `DEPARTMENT_ID` column in the `SELECT` list could be from either the `DEPARTMENTS` table or the `EMPLOYEES` table. Therefore, it is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using a table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. Therefore, you can use *table aliases*, instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, thereby using less memory.

The table name is specified in full, followed by a space, and then the table alias. For example, the `EMPLOYEES` table can be given an alias of `e`, and the `DEPARTMENTS` table an alias of `d`.

Guidelines

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the `FROM` clause, that table alias must be substituted for the table name throughout the `SELECT` statement.
- Table aliases should be meaningful.
- A table alias is valid only for the current `SELECT` statement.

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	South San Francisco
3	US	1700	United States of America	Seattle
4	CA	1800	Canada	Toronto
5	UK	2500	United Kingdom	Oxford



Copyright © 2013, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. To determine an employee's department name, you compare the values in the `DEPARTMENT_ID` column in the `EMPLOYEES` table with the `DEPARTMENT_ID` values in the `DEPARTMENTS` table. The relationship between the `EMPLOYEES` and `DEPARTMENTS` tables is an *equijoin*; that is, values in the `DEPARTMENT_ID` column in both tables must be equal. Often, this type of join involves primary and foreign key complements.

Note: Equijoins are also called *simple joins*.

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	200 Whalen	10	10	1700
2	201 Hartstein	20	20	1800
3	202 Fay	20	20	1800
4	144 Vargas	50	50	1500
5	143 Matos	50	50	1500
6	142 Davies	50	50	1500
7	141 Rajs	50	50	1500
8	124 Mourgos	50	50	1500
9	103 Hunold	60	60	1400
10	104 Ernst	60	60	1400
11	107 Lorentz	60	60	1400
...				



Copyright © 2013, 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	Marketing	Toronto
2	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 © 2013, 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 additional 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
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 © 2013, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the COUNTRY_ID columns in the COUNTRIES and LOCATIONS tables are joined and thus the LOCATION_ID of the location where an employee works is shown.

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



Copyright © 2013, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

This query retrieves all the rows in the COUNTRIES table, which is the left table, even if there is no match in the LOCATIONS table.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

This query retrieves all the rows in the DEPARTMENTS table, which is the table at the right, even if there is no match in the EMPLOYEES table.

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
     ON (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 © 2013, Oracle and/or its affiliates. All rights reserved.

This query retrieves all the rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all the rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

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

...



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the EMPLOYEES table to itself, or perform a self-join. The example in the slide joins the EMPLOYEES table to itself. To simulate two tables in the FROM clause, there are two aliases, namely worker and manager, for the same table, EMPLOYEES.

In this example, the WHERE clause contains the join that means “where a worker's manager ID matches the employee ID for the manager.”

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

...



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The CROSS JOIN syntax specifies the cross product. It is also known as a Cartesian product. A cross join produces the cross product of two relations, and is essentially the same as the comma-delimited Oracle Database notation.

You do not specify any WHERE condition between the two tables in the CROSS JOIN.

Summary

In this appendix, you should have learned how to use:

- The SELECT statement to retrieve rows from one or more tables
- DDL statements to alter the structure of objects
- DML statements to manipulate data in the existing schema objects
- Transaction control statements to manage the changes made by DML statements
- Joins to display data from multiple tables



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This appendix covered commonly used SQL commands and statements, including DDL statements, DML statements, transaction control statements, and joins.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only

Managing PL/SQL Code

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This appendix introduces the conditional compilation and obfuscating (or wrapping) PL/SQL code.

Agenda

- Using conditional compilation
- Obfuscating PL/SQL code

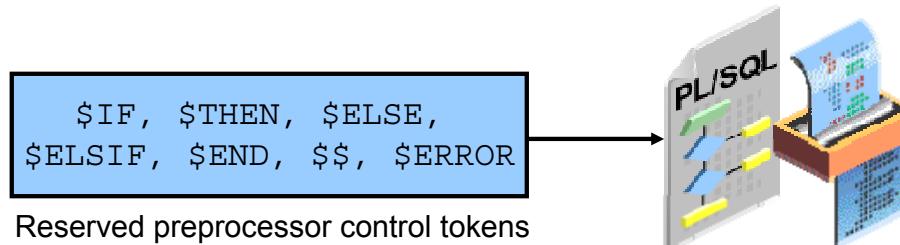


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

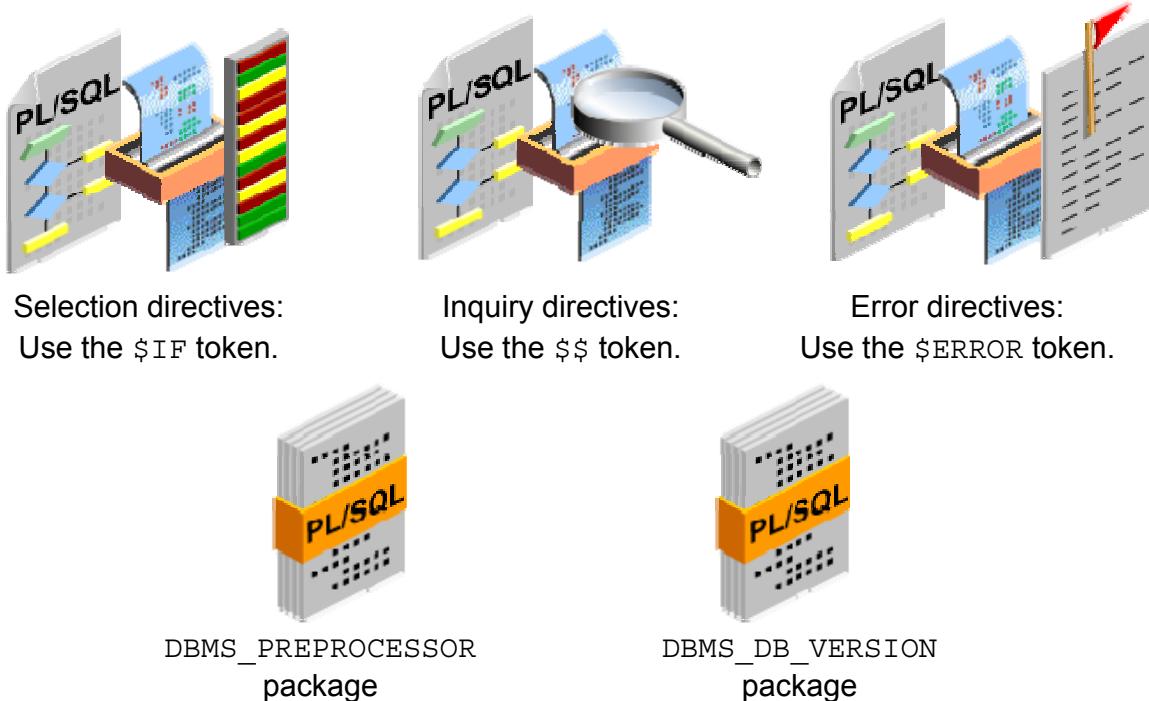
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 you 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?



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use conditional compilation by embedding directives in your PL/SQL source programs. When the PL/SQL program is submitted for compilation, a preprocessor evaluates these directives and selects parts of the program to be compiled. The selected program source is then handed off to the compiler for compilation.

Inquiry directives use the `$$` token to make inquiries about the compilation environment, such as the value of the PL/SQL compiler `PLSQL_CCFLAGS` or `PLSQL_OPTIMIZE_LEVEL` initialization parameters for the unit being compiled. This directive can be used in conjunction with the conditional selection directive to select the parts of the program to compile.

Selection directives can test inquiry directives or static package constants by using the `$IF` construct to branch sections of code for possible compilation if a condition is satisfied.

Error directives issue a compilation error if an unexpected condition is encountered during conditional compilation using the `$ERROR` token.

The `DBMS_DB_VERSION` package provides database version and release constants that can be used for conditional compilation.

The `DBMS_PREPROCESSOR` package provides subprograms for accessing the post-processed source text that is selected by conditional compilation directives in a PL/SQL unit.

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;
```



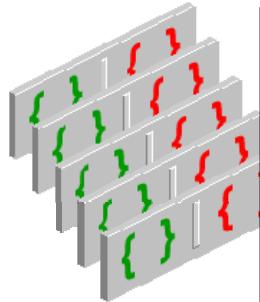
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

An inquiry directive can be predefined or user-defined. The following describes 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.

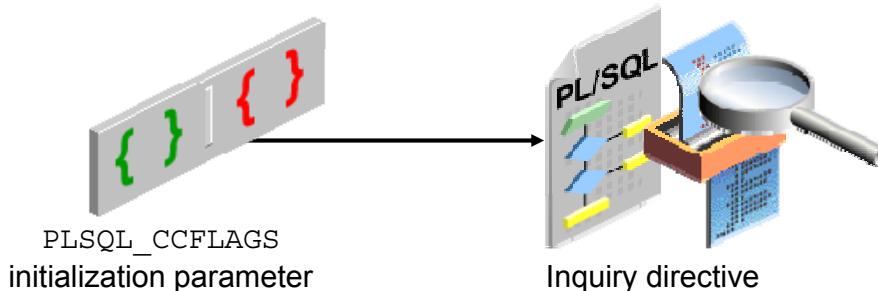
In the example in the slide, the value of `$$debug` is 0 and the value of `$$plsql_ccflags` is TRUE. Note that the value of `$$plsql_ccflags` resolves to the user-defined `plsql_ccflags` inside the value of the `PLSQL_CCFLAGS` compiler parameter. This occurs because a user-defined directive overrides 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 © 2013, 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)

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.



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

You can define any allowable value for `PLSQL_CCFLAGS`. However, Oracle recommends that this parameter be used for controlling the conditional compilation of debugging or tracing code.

The flag names can be set to any identifier, including reserved words and keywords. The values must be the literals `TRUE`, `FALSE`, or `NULL`, or a `PLS_INTEGER` literal. The flag names and values are not case sensitive. The `PLSQL_CCFLAGS` parameter is a PL/SQL compiler parameter (like other compiler parameters) and is stored with the PL/SQL program unit. Consequently, if the PL/SQL program is recompiled later with the `REUSE SETTINGS` clause (example, `ALTER PACKAGE ...REUSE SETTINGS`), the same value of `PLSQL_CCFLAGS` is used for the recompilation. Because the `PLSQL_CCFLAGS` parameter can be set to a different value for each PL/SQL unit, it provides a convenient method for controlling conditional compilation on a per-unit basis.

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';
```

NAME	PLSQL_CCFLAGS
P	Tracing:true



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the parameter is set and then the procedure is created. The setting is stored with each PL/SQL unit.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

The \$ERROR error directive raises a user-defined error and is of the following form:

```
$ERROR varchar2_static_expression $END
```

Note: varchar2_static_expression must be a VARCHAR2 static expression.

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 © 2013, 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.

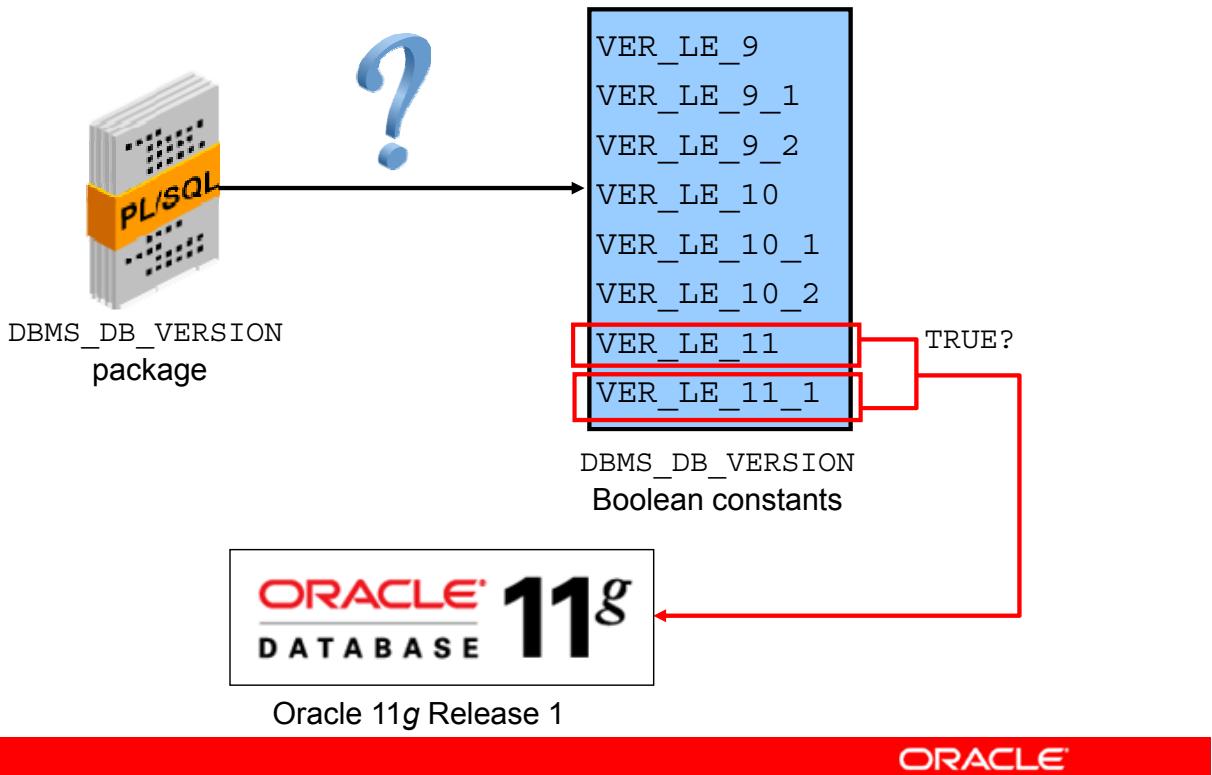
These PL/SQL expressions that are allowed in conditional compilation directives are referred to as “static expressions.” Static expressions are carefully defined to guarantee that if a unit is automatically recompiled without any changes to the values it depends on, the expressions evaluate in the same 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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 © 2013, 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
```



Copyright © 2013, 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)`

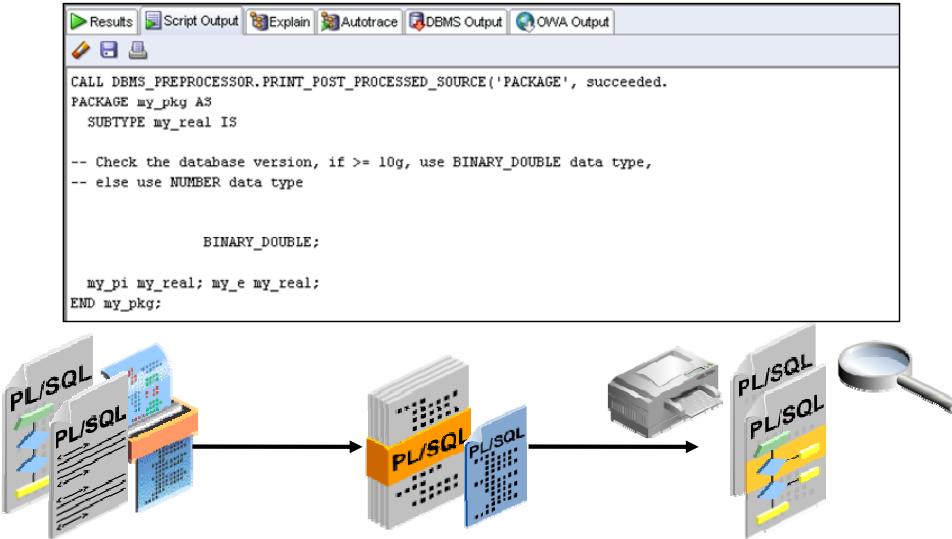
In the procedure's body, `my_area` becomes equal to the value of `my_pi` set in `my_pkg` multiplied by the value that is passed to the procedure as a radius. A message is printed displaying the radius and the area of the circle, as shown in the second code example in the slide.

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');
```



ORACLE

Copyright © 2013, 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Note

For additional information about obfuscation, refer to the *Oracle Database PL/SQL Language Reference*.

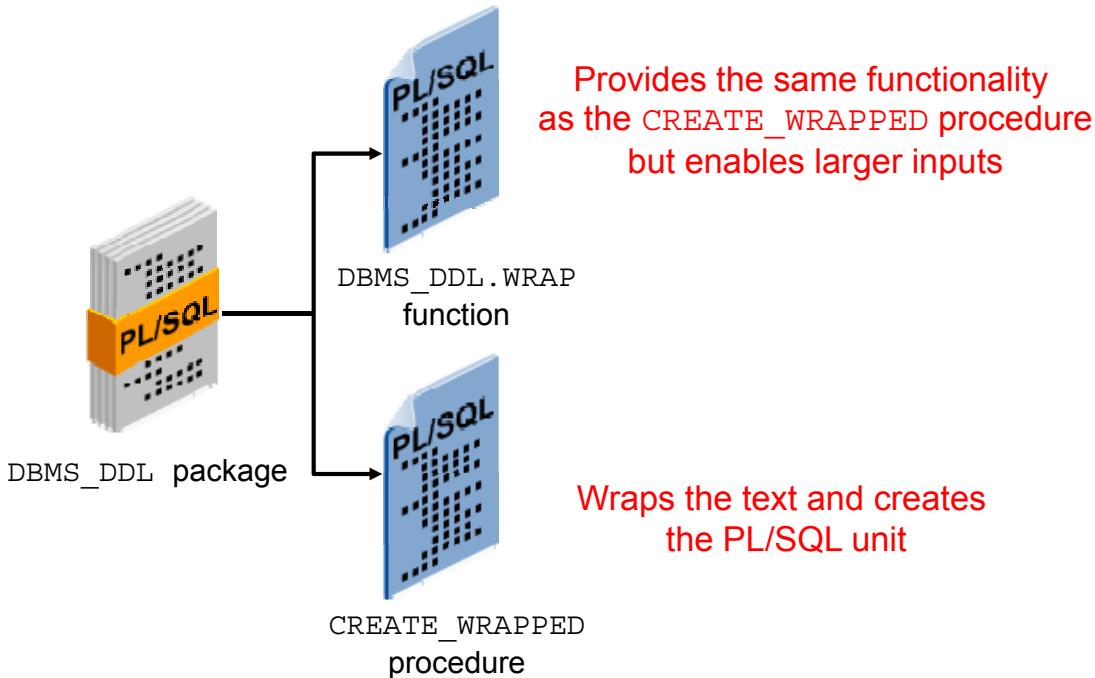
Benefits of Obfuscating

- 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.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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



ORACLE

Copyright © 2013, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

In the first example in the slide, the EXECUTE IMMEDIATE statement is used to create the P1 procedure. The code in the created procedure is not wrapped. The code is not hidden when you use any of the views from the ALL_SOURCE view family to display the procedure's code as shown in the slide.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the DBMS_DDL.CREATE_WRAPPED package procedure is used to create the P1 procedure. The code is obfuscated when you use any of the views from the ALL_SOURCE view family to display the procedure's code (as shown on the next page). When you check the *_SOURCE views, the source is wrapped, or hidden, so that others cannot view the code details shown in the output of the command in the slide.

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 © 2013, 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
```

```
hBWmpGeSsd58b4jCP3/0d04rof0wg5nnm7+fMr2ywFyFDGLQlhaXriu4dCuPCWnnx1J0Ulxp  
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 © 2013, Oracle and/or its affiliates. All rights reserved.

Using the PL/SQL Wrapper utility, you can deliver PL/SQL applications without exposing your source code, which may contain proprietary algorithms and data structures. The Wrapper utility converts the readable source code into unreadable code. By hiding application internals, it prevents misuse of your application.

Wrapped code (such as PL/SQL stored programs) has several features:

- It is platform independent, so you do not need to deliver multiple versions of the same compilation unit.
- It permits dynamic loading, so users do not need to shut down and restart to add a new feature.
- It permits dynamic binding, so external references are resolved at load time.
- It offers strict dependency checking, so that invalidated program units are recompiled automatically when they are invoked.
- It supports normal importing and exporting, so the import/export utility can process wrapped files.

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 © 2013, 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 © 2013, 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 © 2013, 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 © 2013, 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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

G

Review of PL/SQL

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Review the block structure for anonymous PL/SQL blocks
- Declare PL/SQL variables
- Create PL/SQL records and tables
- Insert, update, and delete data
- Use IF, THEN, and ELSIF statements
- Use basic, FOR, and WHILE loops
- Declare and use explicit cursors with parameters
- Use cursor FOR loops and FOR UPDATE and WHERE CURRENT OF clauses
- Trap predefined and user-defined exceptions



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You have learned about implicit cursors that are automatically created by PL/SQL when you execute a SQL SELECT or DML statement. In this lesson, you learn about explicit cursors. You learn to differentiate between implicit and explicit cursors. You also learn to declare and control simple cursors as well as cursors with parameters.

Block Structure for Anonymous PL/SQL Blocks

- **DECLARE** (optional)
 - Declare PL/SQL objects to be used within this block.
- **BEGIN** (mandatory)
 - Define the executable statements.
- **EXCEPTION** (optional)
 - Define the actions that take place if an error or exception arises.
- **END ;** (mandatory)



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Anonymous Blocks

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

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

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

Declaring PL/SQL Variables

- Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
[:= | DEFAULT expr];
```

- Examples:

```
Declare
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location       VARCHAR2(13) := 'Atlanta';
  c_comm           CONSTANT NUMBER := 1400;
  v_count          BINARY_INTEGER := 0;
  v_valid          BOOLEAN NOT NULL := TRUE;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You need to declare all PL/SQL identifiers within the declaration section before referencing them within the PL/SQL block. You have the option to assign an initial value. You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, you must be sure to declare them separately in a previous statement.

In the syntax:

Identifier is the name of the variable

CONSTANT constrains the variable so that its value cannot change; constants must be initialized.

Datatype is a scalar, composite, reference, or LOB data type (This course covers only scalar and composite data types.)

NOT NULL constrains the variable so that it must contain a value; *NOT NULL* variables must be initialized.

expr is any PL/SQL expression that can be a literal, another variable, or an expression involving operators and functions

Declaring Variables with the %TYPE Attribute: Examples

```
...
v_ename          employees.last_name%TYPE;
v_balance        NUMBER(7,2);
v_min_balance   v_balance%TYPE := 10;
...
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

```
...
v_ename          employees.last_name%TYPE;
...
```

Declare variables to store the balance of a bank account, as well as the minimum balance, which starts out as 10.

```
...
v_balance        NUMBER(7,2);
v_min_balance   v_balance%TYPE := 10;
...
```

A NOT NULL column constraint does not apply to variables declared using %TYPE. Therefore, if you declare a variable using the %TYPE attribute and a database column defined as NOT NULL, then you can assign the NULL value to the variable.

Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

```
...
TYPE emp_record_type IS RECORD
(ename      VARCHAR2 (25),
 job        VARCHAR2 (10),
 sal        NUMBER (8, 2));
emp_record    emp_record_type;
...
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Field declarations are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

The following example shows that you can use the %TYPE attribute to specify a field data type:

```
DECLARE
  TYPE emp_record_type IS RECORD
    (empid  NUMBER(6) NOT NULL := 100,
     ename   employees.last_name%TYPE,
     job     employees.job_id%TYPE);
  emp_record    emp_record_type;
...
```

Note: You can add the NOT NULL constraint to any field declaration to prevent the assigning of nulls to that field. Remember that fields declared as NOT NULL must be initialized.

%ROWTYPE Attribute: Examples

- Declare a variable to store the same information about a department as is stored in the DEPARTMENTS table.

```
dept_record    departments%ROWTYPE;
```

- Declare a variable to store the same information about an employee as is stored in the EMPLOYEES table.

```
emp_record    employees%ROWTYPE;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Examples

The first declaration in the slide creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID.

The second declaration in the slide creates a record with the same field names and field data types as a row in the EMPLOYEES table. The fields are EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID.

In the following example, you select column values into a record named job_record.

```
DECLARE
    job_record    jobs%ROWTYPE;
    ...
BEGIN
    SELECT * INTO job_record
    FROM   jobs
    WHERE  ...
```

Creating a PL/SQL Table

```
DECLARE
    TYPE ename_table_type IS TABLE OF
        employees.last_name%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE hiredate_table_type IS TABLE OF DATE
        INDEX BY BINARY_INTEGER;
    ename_table    ename_table_type;
    hiredate_table hiredate_table_type;
BEGIN
    ename_table(1) := 'CAMERON';
    hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
        INSERT INTO ...
    ...
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

There are no predefined data types for PL/SQL tables, as there are for scalar variables. Therefore, you must create the data type first and then declare an identifier using that data type.

Referencing a PL/SQL Table

Syntax

```
pl/sql_table_name(primary_key_value)
```

In this syntax, `primary_key_value` belongs to the `BINARY_INTEGER` type.

Reference the third row in a PL/SQL table `ENAME_TABLE`.

```
ename_table(3) ...
```

The magnitude range of a `BINARY_INTEGER` is -2,147,483,647 through 2,147,483,647. The primary key value can therefore be negative. Indexing need not start with 1.

Note: The `table.EXISTS(i)` statement returns TRUE if at least one row with index `i` is returned. Use the `EXISTS` statement to prevent an error that is raised in reference to a nonexistent table element.

SELECT Statements in PL/SQL: Example

The INTO clause is mandatory.

```
DECLARE
  v_deptid  NUMBER(4);
  v_loc     NUMBER(4);
BEGIN
  SELECT  department_id, location_id
  INTO    v_deptid, v_loc
  FROM    departments
  WHERE   department_name = 'Sales';
  ...
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables to hold the values that SQL returns from the SELECT clause. You must give one variable for each item selected, and the order of variables must correspond to the items selected.

You use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return One and Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of Embedded SQL, for which the following rule applies:

Queries must return one and only one row. More than one row or no row generates an error.

PL/SQL deals with these errors by raising standard exceptions, which you can trap in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. You should code SELECT statements to return a single row.

Inserting Data: Example

Add new employee information to the EMPLOYEES table.

```
DECLARE
    v_empid  employees.employee_id%TYPE;
BEGIN
    SELECT  employees_seq.NEXTVAL
    INTO    v_empno
    FROM    dual;
    INSERT INTO employees(employee_id, last_name,
                           job_id, department_id)
    VALUES (v_empid, 'HARDING', 'PU_CLERK', 30);
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Inserting Data

- Use SQL functions, such as `USER` and `SYSDATE`.
- Generate primary key values by using database sequences.
- Derive values in the PL/SQL block.
- Add column default values.

Note: There is no possibility for ambiguity with identifiers and column names in the `INSERT` statement. Any identifier in the `INSERT` clause must be a database column name.

Updating Data: Example

Increase the salary of all employees in the EMPLOYEES table who are purchasing clerks.

```
DECLARE
    v_sal_increase    employees.salary%TYPE := 2000;
BEGIN
    UPDATE employees
    SET      salary = salary + v_sal_increase
    WHERE    job_id = 'PU_CLERK';
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Updating Data

There may be ambiguity in the `SET` clause of the `UPDATE` statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable.

Remember that the `WHERE` clause is used to determine which rows are affected. If no rows are modified, no error occurs (unlike the `SELECT` statement in PL/SQL).

Note: PL/SQL variable assignments always use `:=` and SQL column assignments always use `=...`. Remember that if column names and identifier names are identical in the `WHERE` clause, the Oracle server looks to the database first for the name.

Deleting Data: Example

Delete rows that belong to department 190 from the EMPLOYEES table.

```
DECLARE
    v_deptid    employees.department_id%TYPE := 190;
BEGIN
    DELETE FROM employees
    WHERE department_id = v_deptid;
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Delete a specific job:

```
DECLARE
    v_jobid      jobs.job_id%TYPE := 'PR_REP';
BEGIN
    DELETE FROM jobs
    WHERE job_id = v_jobid;
END;
```

Controlling Transactions with the COMMIT and ROLLBACK Statements

- Initiate a transaction with the first DML command to follow a COMMIT or ROLLBACK statement.
- Use COMMIT and ROLLBACK SQL statements to terminate a transaction explicitly.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You control the logic of transactions with COMMIT and ROLLBACK SQL statements, rendering some groups of database changes permanent while discarding others. As with the Oracle server, data manipulation language (DML) transactions start at the first command to follow a COMMIT or ROLLBACK and end on the next successful COMMIT or ROLLBACK. These actions may occur within a PL/SQL block or as a result of events in the host environment. A COMMIT ends the current transaction by making all pending changes to the database permanent.

Syntax

```
COMMIT [WORK] ;  
ROLLBACK [WORK] ;
```

In this syntax, WORK is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as LOCK TABLE and SELECT ... FOR UPDATE) in a block. They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.

IF, THEN, and ELSIF Statements: Example

For a given value entered, return a calculated value.

```
.
.
.
IF v_start > 100 THEN
    v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
    v_start := 0.5 * v_start;
ELSE
    v_start := 0.1 * v_start;
END IF;
.
.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

IF, THEN, and ELSIF Statements

When possible, use the `ELSIF` clause instead of nesting `IF` statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the `ELSE` clause consists purely of another `IF` statement, it is more convenient to use the `ELSIF` clause. This makes the code clearer by removing the need for nested `END IF`s at the end of each further set of conditions and actions.

Example

```
IF condition1 THEN
    statement1;
ELSIF condition2 THEN
    statement2;
ELSIF condition3 THEN
    statement3;
END IF;
```

The statement in the slide is further defined as follows:

For a given value entered, return a calculated value. If the entered value is over 100, then the calculated value is two times the entered value. If the entered value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.

Basic Loop: Example

```
DECLARE
    v_ordid      order_items.order_id%TYPE := 101;
    v_counter    NUMBER(2)   := 1;
BEGIN
    LOOP
        INSERT INTO order_items(order_id,line_item_id)
        VALUES(v_ordid, v_counter);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 10;
    END LOOP;
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The basic loop example shown in the slide is defined as follows:

Insert the first 10 new line items for order number 101.

Note: A basic loop enables execution of its statements at least once, even if the condition has been met upon entering the loop.

FOR Loop: Example

Insert the first 10 new line items for order number 101.

```
DECLARE
    v_ordid      order_items.order_id%TYPE := 101;
BEGIN
    FOR i IN 1..10 LOOP
        INSERT INTO order_items(order_id,line_item_id)
        VALUES(v_ordid, i);
    END LOOP;
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide shows a `FOR` loop that inserts 10 rows into the `order_items` table.

WHILE Loop: Example

```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot -
PROMPT 'Enter the maximum total for purchase of item: '
DECLARE
...
v_qty          NUMBER(8) := 1;
v_running_total NUMBER(7,2) := 0;

BEGIN
...
WHILE v_running_total < &p_itemtot LOOP
...
v_qty := v_qty + 1;
v_running_total := v_qty * &p_price;
END LOOP;
...
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

SQL Implicit Cursor Attributes

You can use SQL cursor attributes to test the outcome of your SQL statements.

SQL Cursor Attributes	Description
SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value)
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows
SQL%ISOPEN	Boolean attribute that always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed

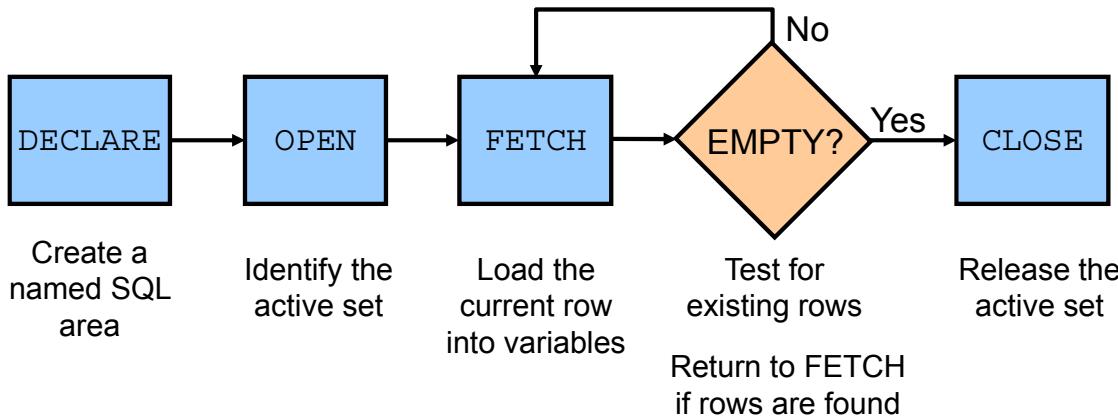


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL cursor attributes enable you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, and SQL%ISOPEN attributes in the exception section of a block to gather information about the execution of a DML statement. In PL/SQL, a DML statement that does not change any rows is not seen as an error condition, whereas the SELECT statement will return an exception if it cannot locate any rows.

Controlling Explicit Cursors



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Explicit Cursors

Controlling Explicit Cursors Using Four Commands

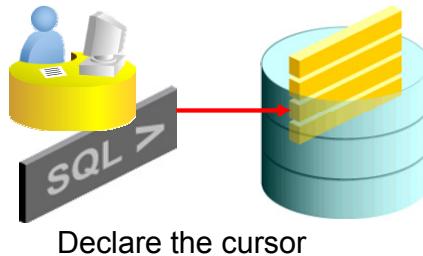
1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The **OPEN** statement executes the query and binds any variables that are referenced. Rows identified by the query are called the **active set** and are now available for fetching.
3. Fetch data from the cursor. The **FETCH** statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore, each fetch accesses a different row returned by the query. In the flow diagram in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise, it closes the cursor.
4. Close the cursor. The **CLOSE** statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Controlling Explicit Cursors: Declaring the Cursor

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM employees;

  CURSOR c2 IS
    SELECT *
    FROM departments
    WHERE department_id = 10;
BEGIN
  ...

```



Declare the cursor

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Explicit Cursor Declaration

Retrieve the employees one by one.

```
DECLARE
  v.empid  employees.employee_id%TYPE;
  v.ename   employees.last_name%TYPE;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM employees;
BEGIN
  ...

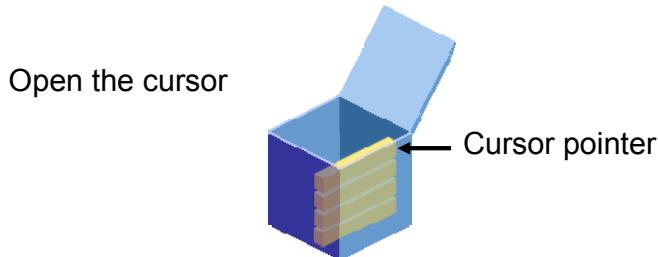
```

Note: You can reference variables in the query, but you must declare them before the CURSOR statement.

Controlling Explicit Cursors: Opening the Cursor

```
OPEN  cursor_name;
```

- Open the cursor to execute the query and identify the active set.
- If the query returns no rows, no exception is raised.
- Use cursor attributes to test the outcome after a fetch.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

OPEN Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax, `cursor_name` is the name of the previously declared cursor.

`OPEN` is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information
2. Parses the `SELECT` statement
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the `OPEN` statement is executed. Rather, the `FETCH` statement retrieves the rows.
5. Positions the pointer just before the first row in the active set

Note: If the query returns no rows when the cursor is opened, then PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared by using the `FOR UPDATE` clause, the `OPEN` statement also locks those rows.

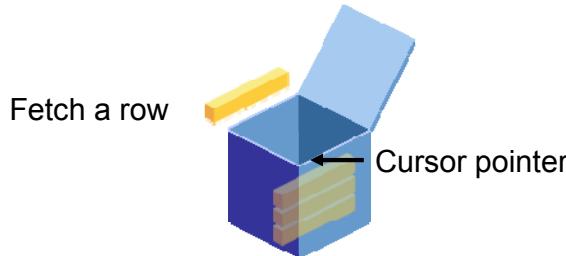
Controlling Explicit Cursors: Fetching Data from the Cursor

```

FETCH c1 INTO v.empid, v.ename;

...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;

```



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

FETCH Statement

You use the **FETCH** statement to retrieve the current row values into output variables. After the fetch, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the **INTO** list. Also, their data types must be compatible. Retrieve the first 10 employees one by one:

```

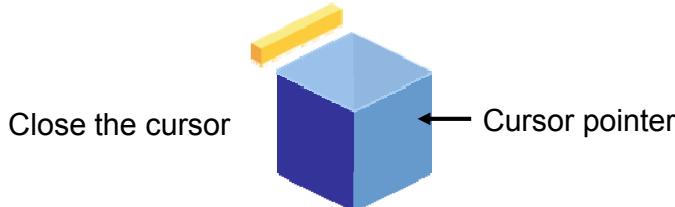
DECLARE
  v.empid  employees.employee_id%TYPE;
  v.ename   employees.last_name%TYPE;
  i         NUMBER := 1;
CURSOR c1 IS
  SELECT employee_id, last_name
  FROM   employees;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v.empid, v.ename;
    ...
  END LOOP;
END;

```

Controlling Explicit Cursors: Closing the Cursor

```
CLOSE    cursor_name;
```

- Close the cursor after completing the processing of the rows.
- Reopen the cursor, if required.
- Do not attempt to fetch data from a cursor after it has been closed.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore, you can establish an active set several times.

In the syntax, `cursor_name` is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor after it has been closed, or the `INVALID_CURSOR` exception will be raised.

Note: The CLOSE statement releases the context area. Although it is possible to terminate the PL/SQL block without closing cursors, you should always close any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the `OPEN_CURSORS` parameter in the database parameter field. By default, the maximum number of `OPEN_CURSORS` is 50.

```
...
FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename; ...
END LOOP;
CLOSE c1;
END;
```

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
ISOPEN	BOOLEAN	Evaluates to TRUE if the cursor is open
%NOTFOUND	BOOLEAN	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	BOOLEAN	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	NUMBER	Evaluates to the total number of rows returned so far



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a DML statement.

Note: Do not reference cursor attributes directly in a SQL statement.

Cursor FOR Loops: Example

Retrieve employees one by one until there are no more left.

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
    -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
    END LOOP; -- implicit close occurs
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. The cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched. In the example in the slide, `emp_record` in the cursor for loop is an implicitly declared record that is used in the FOR LOOP construct.

FOR UPDATE Clause: Example

Retrieve the orders for amounts over \$1,000 that were processed today.

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
           AND order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

FOR UPDATE Clause

If the database server cannot acquire the locks on the rows it needs in a `SELECT FOR UPDATE`, then it waits indefinitely. You can use the `NOWAIT` clause in the `SELECT FOR UPDATE` statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore, you can retry opening the cursor *n* times before terminating the PL/SQL block.

If you intend to update or delete rows by using the `WHERE CURRENT OF` clause, you must specify a column name in the `FOR UPDATE OF` clause.

If you have a large table, you can achieve better performance by using the `LOCK TABLE` statement to lock all rows in the table. However, when using `LOCK TABLE`, you cannot use the `WHERE CURRENT OF` clause and must use the notation `WHERE column = identifier`.

WHERE CURRENT OF Clause: Example

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
  ...
  END LOOP;
  COMMIT;
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your DELETE or UPDATE statement to contain the WHERE CURRENT OF cursor_name clause to refer to the latest row processed by the FETCH statement. When you use this clause, the cursor you reference must exist and must contain the FOR UPDATE clause in the cursor query; otherwise, you get an error. This clause enables you to apply updates and deletes to the currently addressed row without the need to explicitly reference the ROWID pseudocolumn.

Trapping Predefined Oracle Server Errors

- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Trap a predefined Oracle server error by referencing its standard name within the corresponding exception-handling routine.

Note: PL/SQL declares predefined exceptions in the STANDARD package.

It is a good idea to always consider the NO_DATA_FOUND and TOO_MANY_ROWS exceptions, which are the most common.

Trapping Predefined Oracle Server Errors: Example

```
BEGIN   SELECT ... COMMIT;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
      statement1;  
      statement2;  
    WHEN TOO_MANY_ROWS THEN  
      statement1;  
    WHEN OTHERS THEN  
      statement1;  
      statement2;  
      statement3;  
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, a message is printed out to the user for each exception. Only one exception is raised and handled at any time.

Non-Predefined Error

Trap for Oracle server error number –2292, which is an integrity constraint violation.

```
DECLARE
    1 e_products_invalid EXCEPTION;
    PRAGMA EXCEPTION_INIT (
        2 e_products_invalid, -2292);
    v_message VARCHAR2(50);
BEGIN
    . . .
EXCEPTION
    3 WHEN e_products_invalid THEN
        :g_message := 'Product ID
        specified is not valid.';
    . . .
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Trapping a Non-Predefined Oracle Server Exception

1. Declare the name for the exception within the declarative section.

Syntax

```
exception EXCEPTION;
```

In this syntax, *exception* is the name of the exception.

2. Associate the declared exception with the standard Oracle server error number, using the PRAGMA EXCEPTION_INIT statement.

Syntax

```
PRAGMA EXCEPTION_INIT(exception, error_number);
```

In this syntax:

exception Is the previously declared exception

error_number Is a standard Oracle server error number

3. Reference the declared exception within the corresponding exception-handling routine.

In the example in the slide: If there is product in stock, halt processing and print a message to the user.

User-Defined Exceptions: Example

```
[DECLARE]
    e_amount_remaining EXCEPTION;
    .
    .
BEGIN
    .
    .
    RAISE e_amount_remaining;
    .
    .
EXCEPTION
    WHEN e_amount_remaining THEN
        :g_message := 'There is still an amount
                      in stock.';
    .
    .
END;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Trapping User-Defined Exceptions

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.

Syntax: exception EXCEPTION;

where: exception Is the name of the exception

2. Use the RAISE statement to raise the exception explicitly within the executable section.

Syntax : RAISE exception;

where: exception Is the previously declared exception

3. Reference the declared exception within the corresponding exception-handling routine.

In the example in the slide: This customer has a business rule that states that a product cannot be removed from its database if there is any inventory left in stock for this product. Because there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a DELETE on the PRODUCT_INFORMATION table, the block queries the INVENTORIES table to see whether there is any stock for the product in question. If there is stock, raise an exception.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception back to the calling environment.

RAISE_APPLICATION_ERROR Procedure

```
raise_application_error (error_number,  
message[, {TRUE | FALSE}]);
```

- Enables you to issue user-defined error messages from stored subprograms
- Is called from an executing stored subprogram only



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax, *error_number* is a user-specified number for the exception between -20,000 and -20,999. The *message* is the user-specified message for the exception. It is a character string that is up to 2,048 bytes long.

TRUE | FALSE is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example:

```
...  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    RAISE_APPLICATION_ERROR (-20201,  
      'Manager is not a valid employee.');
```

```
END;
```

RAISE_APPLICATION_ERROR Procedure

- Is used in two different places:
 - Executable section
 - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

RAISE_APPLICATION_ERROR Procedure: Example

```
...
DELETE FROM employees
WHERE manager_id = v_mgr;
IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202,
                           'This is not a valid manager');
END IF;
...
```

Summary

In this appendix, you should have learned how to:

- Review the block structure for anonymous PL/SQL blocks
- Declare PL/SQL variables
- Create PL/SQL records and tables
- Insert, update, and delete data
- Use IF, THEN, and ELSIF Statements
- Use basic, FOR, and WHILE loops
- Declare and use explicit cursors with parameters
- Use cursor FOR loops and FOR UPDATE and WHERE CURRENT OF clauses
- Trap predefined and user-defined exceptions



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The Oracle server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor FOR loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor FOR loops do this implicitly. If you are updating or deleting rows, lock the rows by using a FOR UPDATE clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a WHERE CURRENT OF clause in conjunction with the FOR UPDATE clause to reference the current row fetched by the cursor.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only

H

Studies for Implementing Triggers

ORACLE®

Copyright © 2013, 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 using triggers



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn to develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server. In some cases, it may be sufficient to refrain from using triggers and accept the functionality provided by the Oracle server.

This lesson covers the following business application scenarios:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

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 © 2013, Oracle and/or its affiliates. All rights reserved.

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data-manipulation, and data-definition privileges.

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
  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 © 2013, Oracle and/or its affiliates. All rights reserved.

Develop triggers to handle more complex security requirements.

- Base privileges on any database values, such as the time of day, the day of the week, and so on.
- Determine access to tables only.
- Determine data-manipulation privileges only.

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD  
CONSTRAINT ck_salary CHECK (salary >= 500);
```



Copyright © 2013, 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 © 2013, 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;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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 © 2013, 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 using 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 IF;
END;
```



Copyright © 2013, 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);
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

Computing Derived Data Values with a Trigger

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 © 2013, Oracle and/or its affiliates. All rights reserved.

In the server, you can log events by querying data and performing operations manually. This sends an email message when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package `UTL_MAIL` to send the email message.

Logging Events Within the Server

1. Query data explicitly to determine whether an operation is necessary.
2. Perform the operation, such as sending a message.

Using Triggers to Log Events

1. Perform operations implicitly, such as firing off an automatic electronic memo.
2. Modify data and perform its dependent operation in a single step.
3. Log events automatically as data is changing.

Logging Events Transparently

In the trigger code:

- CHR(10) is a carriage return
- Reorder_point is not NULL
- 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 © 2013, Oracle and/or its affiliates. All rights reserved.

This lesson provides some detailed comparison of using the Oracle database server functionality to implement security, auditing, data integrity, replication, and logging. The lesson also covers how database triggers can be used to implement the same features but go further to enhance the features that the database server provides. In some cases, you must use a trigger to perform some activities (such as computation of derived data) because the Oracle server cannot know how to implement this kind of business rule without some programming effort.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only

Using the DBMS_SCHEDULER and HTP Packages

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Use the `HTP` package to generate a simple Web page
- Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution

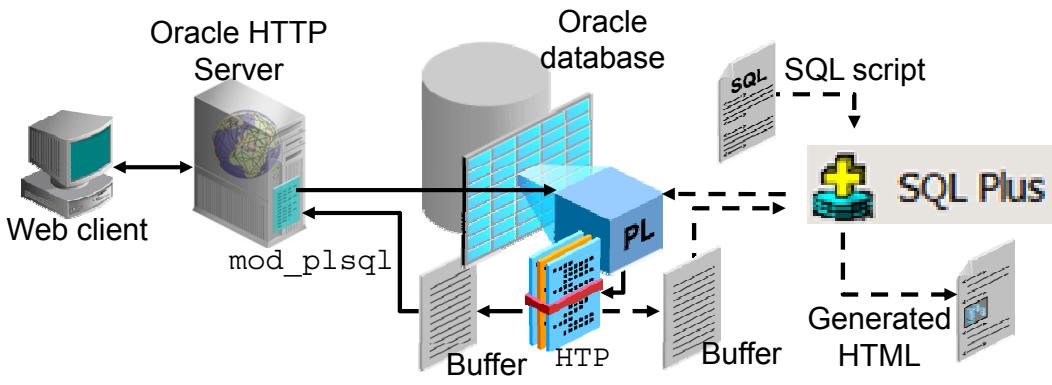


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

Generating Web Pages 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 © 2013, 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;      ----->
    htp.headOpen;     ----->
    htp.title('Welcome');   -->
    htp.headClose;    ----->
    htp.bodyOpen;     ----->
    htp.print('My home page');
    htp.bodyClose;   ----->
    htp.htmlClose;   ----->
END;
```

-- Generates :

```
<HTML>
<HEAD>
<TITLE>Welcome</TITLE>
</HEAD>
<BODY>
My home page
</BODY>
</HTML>
```



Copyright © 2013, 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 Web page, 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 © 2013, 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.

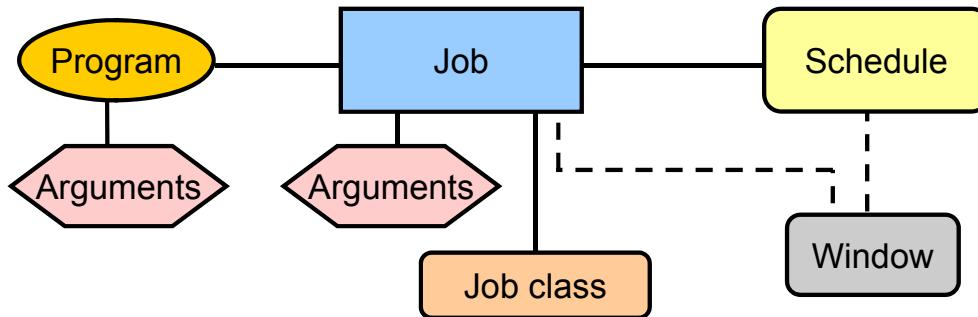
The `ACCEPT` command prompts for the name of the procedure to execute. The call to `OWA_UTIL.SHOWPAGE` displays the HTML tags in the browser window. You can then copy and paste the generated HTML tags from the browser window into an HTML file, typically with an `.htm` or `.html` extension.

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 © 2013, 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 © 2013, 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 © 2013, 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 © 2013, 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 © 2013, 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 © 2013, 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 jobs using the schedule. The job schedules are changed once, not multiple times.

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

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

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

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

Setting the Repeat Interval for a Job

- Using a calendaring expression:

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

- Using a PL/SQL expression:

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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 © 2013, 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 © 2013, 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 Web page
- Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

In this lesson, you should have learned how to schedule PL/SQL and external code for execution with the `DBMS_SCHEDULER` package.

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

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Ventara AG use only