

# **Oracle Database 11g: Advanced PL/SQL**

**Volume II • Student Guide**

D52601GC20

Edition 2.0

September 2010

D66096

**ORACLE®**

**Author**

Tulika Srivastava

**Technical Contributors and Reviewers**

Maria Bilings

Laszlo Czinkocski

Todd Bao

Claire Bennett

Yanti Chang

Ken Cooper

Francesco Ferla

Nancy Greenberg

Rick Green

Laura Garza

Bryn Llewelyn

Timothy McGlue

Essi Parast

Nagavalli Pataballa

Alan Paulson

Surya Rekha

Lauran Serhal

Clinton Shaffer

Anjulaponni Azhagulekshmi Sub

Jenny Tsai

Ted Witiuk

Marcie Young

Jin Zhang

**Editors**

Daniel Milne

Raj Kumar

**Graphic Designer**

Asha Thampy

**Publishers**

Sujatha Nagendra

Veena Narasimhan

**Copyright © 2010, 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 is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

# Contents

## Preface

### 1 Introduction

- Course Objectives 1-2
- Lesson Agenda 1-3
- Course Agenda 1-4
- Appendixes Used in This Course 1-6
- Lesson Agenda 1-7
- Development Environments: Overview 1-8
- What Is Oracle SQL Developer? 1-9
- Coding PL/SQL in SQL\*Plus 1-10
- Lesson Agenda 1-11
- Tables Used in This Course 1-12
- The Order Entry Schema 1-13
- The Human Resources Schema 1-15
- Oracle 11g SQL and PL/SQL Documentation 1-16
- Summary 1-17
- Practice 1 Overview: Getting Started 1-18

### 2 PL/SQL Programming Concepts: Review

- Objectives 2-2
- Lesson Agenda 2-3
- PL/SQL Block Structure 2-4
- Naming Conventions 2-5
- Procedures 2-6
- Procedure: Example 2-7
- Stored Functions 2-8
- Function: Example 2-9
- Ways to Execute Functions 2-10
- Lesson Agenda 2-11
- Restrictions on Calling Functions from SQL Expressions 2-12
- Lesson Agenda 2-14
- PL/SQL Packages: Review 2-15
- Components of a PL/SQL Package 2-16
- Creating the Package Specification 2-17

- Creating the Package Body 2-18
- Lesson Agenda 2-19
- Cursor 2-20
- Processing Explicit Cursors 2-22
- Explicit Cursor Attributes 2-23
- Cursor FOR Loops 2-24
- Cursor: Example 2-25
- Lesson Agenda 2-26
- Handling Exceptions 2-27
- Exceptions: Example 2-29
- Predefined Oracle Server Errors 2-30
- Trapping Non-Predefined Oracle Server Errors 2-33
- Trapping User-Defined Exceptions 2-34
- Lesson Agenda 2-35
- The RAISE\_APPLICATION\_ERROR Procedure 2-36
- Lesson Agenda 2-38
- Dependencies 2-39
- Displaying Direct and Indirect Dependencies 2-41
- Lesson Agenda 2-42
- Using Oracle-Supplied Packages 2-43
- Some of the Oracle-Supplied Packages 2-44
- DBMS\_OUTPUT Package 2-45
- UTL\_FILE Package 2-46
- Summary 2-47
- Practice 2: Overview 2-48

### **3 Designing PL/SQL Code**

- Objectives 3-2
- Lesson Agenda 3-3
- Guidelines for Cursor Design 3-4
- Lesson Agenda 3-9
- Cursor Variables: Overview 3-10
- Working with Cursor Variables 3-11
- Strong Versus Weak REF CURSOR Variables 3-12
- Step 1: Defining a REF CURSOR Type 3-13
- Step 1: Declaring a Cursor Variable 3-14
- Step 1: Declaring a REF CURSOR Return Type 3-15
- Step 2: Opening a Cursor Variable 3-16
- Step 3: Fetching from a Cursor Variable 3-18
- Step 4: Closing a Cursor Variable 3-19

- Passing Cursor Variables as Arguments 3-20
- Using the `SYS_REFCURSOR` Predefined Type 3-23
- Rules for Cursor Variables 3-25
- Comparing Cursor Variables with Static Cursors 3-26
- Lesson Agenda 3-27
- Predefined PL/SQL Data Types 3-28
- Subtypes: Overview 3-29
- Benefits of Subtypes 3-31
- Declaring Subtypes 3-32
- Using Subtypes 3-33
- Subtype Compatibility 3-34
- Quiz 3-35
- Summary 3-38
- Practice 3: Overview 3-39

## **4 Working with Collections**

- Objectives 4-2
- Lesson Agenda 4-3
- Understanding Collections 4-4
- Collection Types 4-5
- Lesson Agenda 4-7
- Using Associative Arrays 4-8
- Creating the Array 4-10
- Traversing the Array 4-11
- Lesson Agenda 4-13
- Using Nested Tables 4-14
- Nested Table Storage 4-15
- Creating Nested Tables 4-16
- Declaring Collections: Nested Table 4-17
- Using Nested Tables 4-18
- Referencing Collection Elements 4-20
- Using Nested Tables in PL/SQL 4-21
- Lesson Agenda 4-23
- Understanding Varrays 4-24
- Declaring Collections: Varray 4-25
- Using Varrays 4-26
- Lesson Agenda 4-28
- Working with Collections in PL/SQL 4-29
- Initializing Collections 4-32
- Referencing Collection Elements 4-34
- Using Collection Methods 4-35

Manipulating Individual Elements 4-39  
Lesson Agenda 4-41  
Avoiding Collection Exceptions 4-42  
Avoiding Collection Exceptions: Example 4-43  
Lesson Agenda 4-44  
Listing Characteristics for Collections 4-45  
Guidelines for Using Collections Effectively 4-46  
Quiz 4-47  
Summary 4-51  
Practice 4: Overview 4-52

## **5 Manipulating Large Objects**

Objectives 5-2  
Lesson Agenda 5-3  
What Is a LOB? 5-4  
Components of a LOB 5-6  
Internal LOBs 5-7  
Managing Internal LOBs 5-8  
Lesson Agenda 5-9  
What Are BFILEs? 5-10  
Securing BFILEs 5-11  
What Is a DIRECTORY? 5-12  
Using the DBMS\_LOB Package 5-13  
DBMS\_LOB.READ and DBMS\_LOB.WRITE 5-14  
Managing BFILEs 5-15  
Preparing to Use BFILEs 5-16  
Populating BFILE Columns with SQL 5-17  
Populating a BFILE Column with PL/SQL 5-18  
Using DBMS\_LOB Routines with BFILEs 5-19  
Lesson Agenda 5-20  
Initializing LOB Columns Added to a Table 5-21  
Populating LOB Columns 5-23  
Writing Data to a LOB 5-24  
Reading LOBs from the Table 5-28  
Updating LOB by Using DBMS\_LOB in PL/SQL 5-30  
Checking the Space Usage of a LOB Table 5-31  
Selecting CLOB Values by Using SQL 5-33  
Selecting CLOB Values by Using DBMS\_LOB 5-34  
Selecting CLOB Values in PL/SQL 5-35

- Removing LOBs 5-36
- Quiz 5-37
- Lesson Agenda 5-40
- Temporary LOBs 5-41
- Creating a Temporary LOB 5-42
- Lesson Agenda 5-43
- SecureFile LOBs 5-44
- Storage of SecureFile LOBs 5-45
- Creating a SecureFile LOB 5-46
- Comparing Performance 5-47
- Enabling Deduplication and Compression 5-48
- Enabling Deduplication and Compression: Example 5-49
- Step 1: Checking Space Usage 5-50
- Step 1: Checking Space Usage 5-52
- Step 2: Enabling Deduplication and Compression 5-53
- Step 3: Rechecking LOB Space Usage 5-54
- Step 4: Reclaiming the Free Space 5-55
- Using Encryption 5-56
- Using Encryption: Example 5-58
- Migrating from BasicFile to SecureFile Format 5-59
- Quiz 5-62
- Summary 5-65
- Practice 5: Overview 5-66

## **6 Using Advanced Interface Methods**

- Objectives 6-2
- Calling External Procedures from PL/SQL 6-3
- Benefits of External Procedures 6-4
- External C Procedure Components 6-5
- How PL/SQL Calls a C External Procedure 6-6
- The `extproc` Process 6-7
- Development Steps for External C Procedures 6-8
- The Call Specification 6-12
- Publishing an External C Routine 6-15
- Executing the External Procedure 6-16
- Java: Overview 6-17
- Calling a Java Class Method by Using PL/SQL 6-18
- Development Steps for Java Class Methods 6-19
- Loading Java Class Methods 6-20
- Publishing a Java Class Method 6-21

Executing the Java Routine 6-23  
Creating Packages for Java Class Methods 6-24  
Quiz 6-25  
Summary 6-28  
Practice 6: Overview 6-29

## **7 Performance and Tuning**

Objectives 7-1  
Lesson Agenda 7-3  
Native and Interpreted Compilation 7-4  
Deciding on a Compilation Method 7-5  
Setting the Compilation Method 7-6  
Viewing the Compilation Settings 7-8  
Setting Up a Database for Native Compilation 7-10  
Compiling a Program Unit for Native Compilation 7-11  
Lesson Agenda 7-12  
Tuning PL/SQL Code 7-13  
Avoiding Implicit Data Type Conversion 7-14  
Understanding the `NOT NULL` Constraint 7-15  
Using the `PLS_INTEGER` Data Type for Integers 7-16  
Using the `SIMPLE_INTEGER` Data Type 7-17  
Modularizing Your Code 7-18  
Comparing SQL with PL/SQL 7-19  
Using Bulk Binding 7-22  
Using `SAVE EXCEPTIONS` 7-28  
Handling `FORALL` Exceptions 7-29  
Rephrasing Conditional Control Statements 7-30  
Passing Data Between PL/SQL Programs 7-32  
Lesson Agenda 7-35  
Introducing Intraunit Inlining 7-36  
Using Inlining 7-37  
Inlining Concepts 7-38  
Inlining: Example 7-41  
Inlining: Guidelines 7-43  
Quiz 7-44  
Summary 7-47  
Practice 7: Overview 7-48



## **8 Improving Performance with Caching**

Objectives 8-2

Lesson Agenda 8-3

What Is Result Caching? 8-4

Increasing Result Cache Memory Size 8-5

Setting `Result_Cache_Max_Size` 8-6

Enabling Query Result Cache 8-7

Using the `DBMS_RESULT_CACHE` Package 8-8

Lesson Agenda 8-9

SQL Query Result Cache 8-10

Clearing the Shared Pool and Result Cache 8-12

Examining the Memory Cache 8-13

Examining the Execution Plan for a Query 8-14

Examining Another Execution Plan 8-15

Executing Both Queries 8-17

Viewing Cache Results Created 8-18

Re-Executing Both Queries 8-19

Viewing Cache Results Found 8-20

Lesson Agenda 8-21

PL/SQL Function Result Cache 8-22

Marking PL/SQL Function Results to Be Cached 8-23

Clearing the Shared Pool and Result Cache 8-24

Lesson Agenda 8-25

Creating a PL/SQL Function Using the `RESULT_CACHE` Clause 8-26

Lesson Agenda 8-27

Calling the PL/SQL Function Inside a Query 8-28

Verifying Memory Allocation 8-29

Viewing Cache Results Created 8-30

Calling the PL/SQL Function Again 8-31

Viewing Cache Results Found 8-32

Confirming That the Cached Result Was Used 8-33

Quiz 8-34

Summary 8-38

Practice 8: Overview 8-39

## **9 Analyzing PL/SQL Code**

Objectives 9-2

Lesson Agenda 9-3

Finding Coding Information 9-4

- Using SQL Developer to Find Coding Information 9-9
- Using DBMS\_DESCRIBE 9-11
- Using ALL\_ARGUMENTS 9-14
- Using SQL Developer to Report on Arguments 9-16
- Using DBMS\_UTILITY.FORMAT\_CALL\_STACK 9-18
- Finding Error Information 9-20
- Lesson Agenda 9-25
- PL/Scope Concepts 9-26
- Collecting PL/Scope Data 9-27
- Using PL/Scope 9-28
- The USER/ALL/DBA\_IDENTIFIERS Catalog View 9-29
- Sample Data for PL/Scope 9-30
- Collecting Identifiers 9-32
- Viewing Identifier Information 9-33
- Performing a Basic Identifier Search 9-35
- Using USER\_IDENTIFIERS to Find All Local Variables 9-36
- Finding Identifier Actions 9-37
- Lesson Agenda 9-39
- DBMS\_METADATA Package 9-40
- Metadata API 9-41
- Subprograms in DBMS\_METADATA 9-42
- FETCH\_XXX Subprograms 9-43
- SET\_FILTER Procedure 9-44
- Filters 9-45
- Examples of Setting Filters 9-46
- Programmatic Use: Example 1 9-47
- Programmatic Use: Example 2 9-49
- Browsing APIs 9-51
- Browsing APIs: Examples 9-52
- Quiz 9-54
- Summary 9-57
- Practice 9: Overview 9-58

## **10 Profiling and Tracing PL/SQL Code**

- Objectives 10-2
- Lesson Agenda 10-3
- Tracing PL/SQL Execution 10-4
- Tracing PL/SQL: Steps 10-7
- Step 1: Enable Specific Subprograms 10-8

Steps 2 and 3: Identify a Trace Level and Start Tracing 10-9  
Step 4: Turn Off Tracing 10-10  
Step 5: Examine the Trace Information 10-11  
plsql\_trace\_runs and plsql\_trace\_events 10-12  
Lesson Agenda 10-14  
Hierarchical Profiling Concepts 10-15  
Using the PL/SQL Profiler 10-17  
Understanding Raw Profiler Data 10-21  
Using the Hierarchical Profiler Tables 10-22  
Using DBMS\_HPROF.ANALYZE 10-23  
Using DBMS\_HPROF.ANALYZE to Write to Hierarchical Profiler Tables 10-24  
Analyzer Output from the DBMSHP\_RUNS Table 10-25  
Analyzer Output from the DBMSHP\_FUNCTION\_INFO Table 10-26  
plshprof: A Simple HTML Report Generator 10-27  
Using plshprof 10-28  
Using the HTML Reports 10-31  
Quiz 10-35  
Summary 10-38  
Practice 10: Overview 10-39

## **11 Implementing Fine-Grained Access Control for VPD**

Objectives 11-2  
Lesson Agenda 11-3  
Fine-Grained Access Control: Overview 11-4  
Identifying Fine-Grained Access Features 11-5  
How Fine-Grained Access Works 11-6  
Why Use Fine-Grained Access? 11-8  
Lesson Agenda 11-9  
Using an Application Context 11-10  
Creating an Application Context 11-12  
Setting a Context 11-13  
Implementing a Policy 11-15  
Step 2: Creating the Package 11-16  
Step 3: Defining the Policy 11-18  
Step 4: Setting Up a Logon Trigger 11-21  
Example Results 11-22  
Data Dictionary Views 11-23  
Using the ALL\_CONTEXT Dictionary View 11-24  
Policy Groups 11-25  
More About Policies 11-26

Quiz 11-28  
Summary 11-31  
Practice 11: Overview 11-32

## **12 Safeguarding Your Code Against SQL Injection Attacks**

Objectives 12-2  
Lesson Agenda 12-3  
Understanding SQL Injection 12-4  
Identifying Types of SQL Injection Attacks 12-5  
SQL Injection: Example 12-6  
Assessing Vulnerability 12-7  
Avoidance Strategies Against SQL Injection 12-8  
Protecting Against SQL Injection: Example 12-9  
Lesson Agenda 12-10  
Reducing the Attack Surface 12-11  
Expose the Database Only Via PL/SQL API 12-12  
Using Invoker's Rights 12-13  
Reducing Arbitrary Inputs 12-15  
Strengthen Database Security 12-16  
Lesson Agenda 12-17  
Using Static SQL 12-18  
Using Dynamic SQL 12-21  
Lesson Agenda 12-22  
Using Bind Arguments with Dynamic SQL 12-23  
Using Bind Arguments with Dynamic PL/SQL 12-24  
What If You Cannot Use Bind Arguments? 12-25  
Lesson Agenda 12-26  
Understanding DBMS\_ASSERT 12-27  
Formatting Oracle Identifiers 12-28  
Working with Identifiers in Dynamic SQL 12-29  
Choosing a Verification Route 12-30  
Avoiding Injection by Using DBMS\_ASSERT.ENQUOTE\_LITERAL 12-31  
Avoiding Injection by Using DBMS\_ASSERT.SIMPLE\_SQL\_NAME 12-34  
DBMS\_ASSERT Guidelines 12-36  
Lesson Agenda 12-39  
Using Bind Arguments 12-40  
Avoiding Privilege Escalation 12-41  
Beware of Filter Parameters 12-42  
Trapping and Handling Exceptions 12-43  
Lesson Agenda 12-44

Coding Review and Testing Strategy 12-45  
Reviewing Code 12-46  
Running Static Code Analysis 12-47  
Testing with Fuzzing Tools 12-48  
Generating Test Cases 12-49  
Quiz 12-51  
Summary 12-55  
Practice 12: Overview 12-56

## **Appendix A: Practices and Solutions**

## **Appendix B: Table Descriptions and Data**

## **Appendix C: Using SQL Developer**

Objectives C-2  
What Is Oracle SQL Developer? C-3  
Specifications of SQL Developer C-4  
SQL Developer 1.5 Interface C-5  
Creating a Database Connection C-7  
Browsing Database Objects C-10  
Displaying the Table Structure C-11  
Browsing Files C-12  
Creating a Schema Object C-13  
Creating a New Table: Example C-14  
Using the SQL Worksheet C-15  
Executing SQL Statements C-18  
Saving SQL Scripts C-19  
Executing Saved Script Files: Method 1 C-20  
Executing Saved Script Files: Method 2 C-21  
Formatting the SQL Code C-22  
Using Snippets C-23  
Using Snippets: Example C-24  
Debugging Procedures and Functions C-25  
Database Reporting C-26  
Creating a User-Defined Report C-27  
Search Engines and External Tools C-28  
Setting Preferences C-29  
Resetting the SQL Developer Layout C-30  
Summary C-31

## **Appendix D: Using SQL\*Plus**

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

## **Appendix E: PL/SQL Server Pages**

- Objectives E-2
- PSP Uses and Features E-3
- Format of the PSP File E-4
- Development Steps for PSP E-6
- Printing the Table Using a Loop E-12
- Specifying a Parameter E-13
- Using an HTML Form to Call a PSP E-16
- Debugging PSP Problems E-18
- Summary E-20

---

# Preface

---





## **Profile**

### **Before You Begin This Course**

Before you begin this course, you should have a thorough knowledge of SQL, SQL\*Plus, and have working experience on developing applications with PL/SQL. The prerequisites are *Oracle Database 11g: Develop PL/SQL Program Units* and *Oracle Database 11g: Introduction to SQL*.

### **How This Course Is Organized**

*Oracle Database 11g: Advanced PL/SQL* is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills.

## Related Publications

### Oracle Publications

<b>Title</b>	<b>Part Number</b>
<i>Oracle Database Concepts 11g Release 2 (11.2)</i>	<i>B28318-03</i>
<i>Oracle Database SQL Language Reference 11g Release 2 (11.2)</i>	<i>E10592-04</i>
<i>Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)</i>	<i>E10577-05</i>
<i>Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)</i>	<i>E10472-05</i>
<i>Oracle Database Advanced Developer's Guide 11g Release 2 (11.2)</i>	<i>E10471-04</i>
<i>Oracle Database Object-Relational Developer's Guide 11g Release 2 (11.2)</i>	<i>E11822-01</i>
<i>Oracle Database Performance Tuning Guide 11g Release 2 (11.2)</i>	<i>E10822-02</i>

### Additional Publications

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

## Typographic Conventions

The following table lists the typographical conventions that are used in text and code.

### Typographical Conventions in Text

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

## Typographic Conventions (continued)

### Typographical Conventions in Code

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

# 9

## Analyzing PL/SQL Code

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Use the supplied packages and dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications. You learn about the dictionary views and the packages that you can use to find information within your code and to generate information about your code.

# Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using `DBMS_METADATA` to retrieve object definitions

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Finding Coding Information

- Use the dictionary views:
  - ALL\_ARGUMENTS
  - ALL\_OBJECTS
  - ALL\_SOURCE
  - ALL\_PROCEDURES
  - ALL\_DEPENDENCIES
- Use the supplied packages:
  - dbms\_describe
  - dbms\_utility



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Finding Information About Your PL/SQL Code

The Oracle dictionary views store information about your compiled PL/SQL code. You can write SQL statements against the views to find information about your code.

Dictionary View	Description
ALL_SOURCE	Includes the lines of source code for all programs that you modify
ALL_ARGUMENTS	Includes information about the parameters for the procedures and functions that you can call
ALL_PROCEDURES	Contains the list of procedures and functions that you can execute
ALL_DEPENDENCIES	Is one of the several views that give you information about the dependencies between database objects

You can also use the Oracle-supplied DBMS\_DESCRIBE package to obtain information about a PL/SQL object. The package contains the DESCRIBE\_PROCEDURE procedure, which provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

You can use the DBMS\_UTILITY supplied package to follow a call stack and an exception stack.



# Finding Coding Information

Find all instances of CHAR in your code:

```
SELECT NAME, line, text
FROM      user_source
WHERE     INSTR (UPPER(text), ' CHAR') > 0
          OR INSTR (UPPER(text), ' CHAR(') > 0
          OR INSTR (UPPER(text), ' CHAR (') > 0;

NAME                                LINE TEXT
-----
CUST_ADDRESS_TYP                   6      , country_id          CHAR(2)
...
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Finding Data Types

You may want to find all occurrences of the CHAR data type. The CHAR data type is fixed in length and can cause false negatives on comparisons with VARCHAR2 strings. By finding the CHAR data type, you can modify the object, if appropriate, and change it to VARCHAR2.

## Finding Coding Information

Create a package with various queries that you can easily call:

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
    PROCEDURE find_text_in_code (str IN VARCHAR2);
    PROCEDURE encap_compliance ;
END query_code_pkg;
/
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Creating a Package to Query Code

A better idea is to create a package to hold various queries that you can easily call. The `QUERY_CODE_PKG` will hold two validation procedures:

The `FIND_TEXT_IN_CODE` procedure displays all programs with a specified character string. It queries `USER_SOURCE` to find occurrences of any text string. The text string is passed as a parameter. For efficiency, the `BULK COLLECT` statement is used to retrieve all matching rows into the collection variable.

The `ENCAP_COMPLIANCE` procedure identifies those programs that reference a table directly. This procedure queries the `ALL_DEPENDENCIES` view to find the PL/SQL code objects that directly reference a table or a view.

You can also include a procedure to validate a set of standards for exception handling.

## Creating a Package to Query Code (continued)

### QUERY\_CODE\_PKG Code

```
CREATE OR REPLACE PACKAGE BODY query_code_pkg IS
  PROCEDURE find_text_in_code (str IN VARCHAR2)
  IS
    TYPE info_rt IS RECORD (NAME user_source.NAME%TYPE,
      text user_source.text%TYPE );
    TYPE info_aat IS TABLE OF info_rt INDEX BY PLS_INTEGER;
    info_aa info_aat;
  BEGIN
    SELECT NAME || '-' || line, text
    BULK COLLECT INTO info_aa FROM user_source
      WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
      AND NAME != 'VALSTD' AND NAME != 'ERRNUMS';
    DBMS_OUTPUT.PUT_LINE ('Checking for presence of ' ||
      str || ':');
    FOR indx IN info_aa.FIRST .. info_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        info_aa (indx).NAME || ',' || info_aa (indx).text);
    END LOOP;
  END find_text_in_code;

  PROCEDURE encap_compliance IS
    SUBTYPE qualified_name_t IS VARCHAR2 (200);
    TYPE refby_rt IS RECORD (NAME qualified_name_t,
      referenced_by qualified_name_t );
    TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
    refby_aa refby_aat;
  BEGIN
    SELECT owner || '.' || NAME refs_table
      , referenced_owner || '.' || referenced_name
      AS table_referenced
    BULK COLLECT INTO refby_aa
      FROM all_dependencies
      WHERE owner = USER
      AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
        'PROCEDURE', 'FUNCTION')
      AND referenced_type IN ('TABLE', 'VIEW')
      AND referenced_owner NOT IN ('SYS', 'SYSTEM')
      ORDER BY owner, NAME, referenced_owner, referenced_name;
    DBMS_OUTPUT.PUT_LINE ('Programs that reference
      tables or views');
    FOR indx IN refby_aa.FIRST .. refby_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (refby_aa (indx).NAME || ',' ||
        refby_aa (indx).referenced_by);
    END LOOP;
  END encap_compliance;
END query_code_pkg;
/
```

# Finding Coding Information

```
EXECUTE query_code_pkg.encap_compliance
Programs that reference tables or views
OE.CREDIT_CARD_PKG,OE.CUSTOMERS
OE.LOAD_PRODUCT_IMAGE,OE.PRODUCT_INFORMATION
OE.SET_VIDEO,OE.CUSTOMERS
OE.WRITE_LOB,OE.PRODUCT_DESCRIPTIONS
...

PL/SQL procedure successfully completed.
```

1

```
EXECUTE query_code_pkg.find_text_in_code('customers')

Checking for presence of customers:
CREDIT_CARD_PKG-12,      FROM customers
CREDIT_CARD_PKG-36,      UPDATE customers
CREDIT_CARD_PKG-41,      UPDATE customers
SET VIDEO-6,      SELECT cust_first_name FROM customers
SET_VIDEO-13,      UPDATE customers SET video = file_ptr
...

PL/SQL procedure successfully completed.
```

2

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

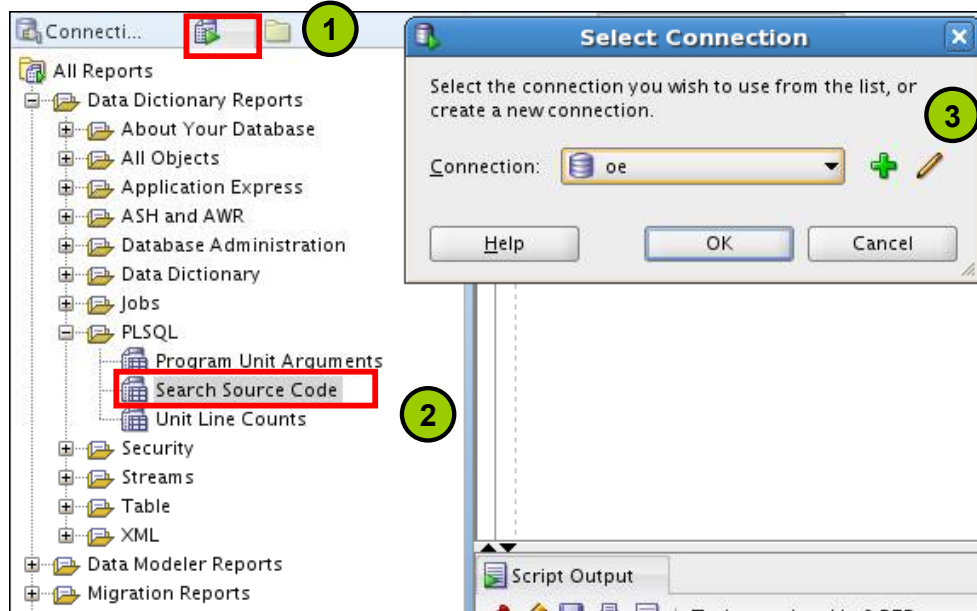
## QUERY\_CODE\_PKG: Examples

In the first example, the ENCAP\_COMPLIANCE procedure displays all PL/SQL code objects that reference a table or view directly. Both the code name and table or view name are listed in the output.

In the second example, the FIND\_TEXT\_IN\_CODE procedure returns all PL/SQL code objects that contain the “customers” text string. The code name, line number, and line are listed in the output.

# Using SQL Developer to Find Coding Information

Use Reports:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using SQL Developer to Find Coding Information

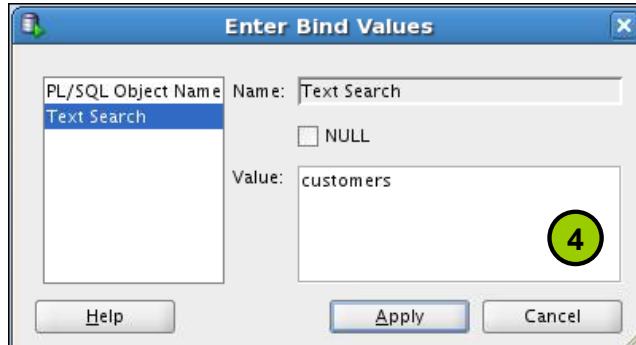
SQL Developer comes with predefined reports that you can use to find information about PL/SQL coding.

If you want to find the occurrence of a text string or an object name, use the Reports feature.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select the Search Source Code option.
3. Enter the connection information.

# Using SQL Developer to Find Coding Information

Enter a text string or an object name.

The 'Enter Bind Values' dialog box in SQL Developer. It has a title bar with a close button. Inside, there's a section for 'PL/SQL Object Name' with a list box containing 'Text Search'. To the right, there's a 'Name' field with 'Text Search' and a 'Value' field with 'customers'. There's also a 'NULL' checkbox. At the bottom are 'Help', 'Apply', and 'Cancel' buttons. A green circle with the number '4' is next to the 'Value' field.

Enter Bind Values

PL/SQL Object Name: Text Search

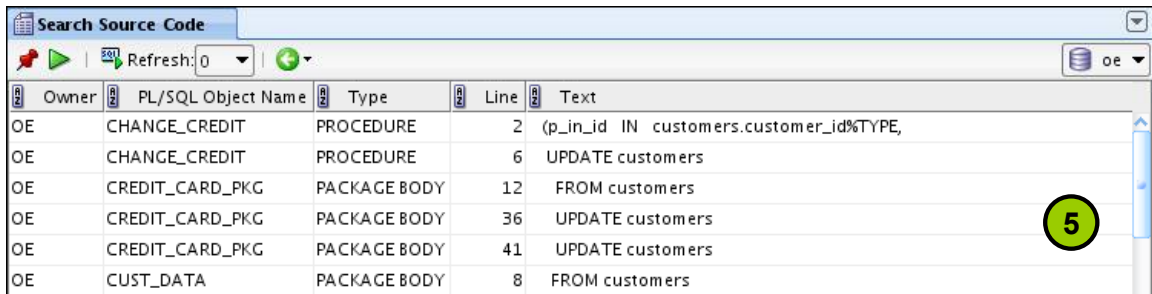
Name: Text Search

☐ NULL

Value: customers

Help Apply Cancel

Results

The 'Search Source Code' window in SQL Developer. It shows a table of search results. The table has columns: Owner, PL/SQL Object Name, Type, Line, and Text. The results are for the 'customers' table. A green circle with the number '5' is next to the table.

Owner	PL/SQL Object Name	Type	Line	Text
OE	CHANGE_CREDIT	PROCEDURE	2	(p_in_id IN customers.customer_id%TYPE,
OE	CHANGE_CREDIT	PROCEDURE	6	UPDATE customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	12	FROM customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	36	UPDATE customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	41	UPDATE customers
OE	CUST_DATA	PACKAGE BODY	8	FROM customers

ORACLE

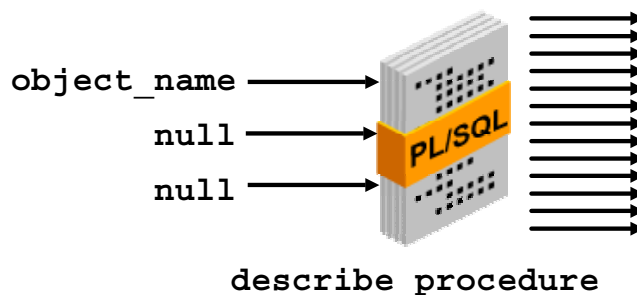
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using SQL Developer to Find Coding Information (continued)

4. Select either an Object Name or a Text Search for the search type. In the Value field, enter the text string that you want found in your PL/SQL source code for the connection that you specified in step 3.
5. View the results of your search.

## Using DBMS\_DESCRIBE

- Can be used to retrieve information about a PL/SQL object
- Contains one procedure: DESCRIBE\_PROCEDURE
- Includes:
  - Three scalar IN parameters
  - One scalar OUT parameter
  - 12 associative array OUT parameters



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### The DBMS\_DESCRIBE Package

You can use the DBMS\_DESCRIBE package to find information about your procedures. It contains one procedure named DESCRIBE\_PROCEDURE. The DESCRIBE\_PROCEDURE routine accepts the name of the procedure that you are enquiring about. There are two other IN parameters. Both must be either NULL or an empty string. These two parameters are reserved.

The DBMS\_DESCRIBE returns detailed parameter information in a set of associative arrays. The details are numerically coded. You can find the following information from the results returned:

- **Overload:** If overloaded, it holds a value for each version of the procedure.
- **Position:** Position of the argument in the parameter list. 0 is reserved for the RETURN information of a function.
- **Level:** For composite types only; it holds the level of the data type.
- **Argument name:** Name of the argument
- **Data type:** A numerically coded value representing a data type
- **Default value:** 0 for no default value, 1 if the argument has a default value
- **Parameter mode:** 0 = IN, 1 = OUT, 2 = IN OUT

**Note:** This is not the complete list of values returned from the DESCRIBE\_PROCEDURE routine. For a complete list, see the *PL/SQL Packages and Types Reference 11g Release 1* reference manual.

# Using DBMS\_DESCRIBE

Create a package to call the  
DBMS\_DESCRIBE.DESCRIBE\_PROCEDURE routine:

```
CREATE OR REPLACE PACKAGE use_dbms_describe
IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
```

1

```
EXEC use_dbms_describe.get_data('query_code_pkg.find_text_in_code')
```

2

anonymous block completed

Name	Mode	Position	Datatype
STR	0	1	1

Name	Mode	Position	Datatype
STR	0	1	1

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## The DESCRIBE\_PROCEDURE Routine

Because the DESCRIBE\_PROCEDURE returns information about your parameters in a set of associative arrays, it is easiest to define a package to call and handle the information returned from it.

In the first example in the slide, the specification for the USE\_DBMS\_DESCRIBE package is defined. This package holds one procedure, GET\_DATA. The GET\_DATA routine calls the DBMS\_DESCRIBE.DESCRIBE\_PROCEDURE routine. The implementation of the USE\_DBMS\_DESCRIBE package is shown on the next page. Note that several associative array variables are defined to hold the values returned via the OUT parameters from the DESCRIBE\_PROCEDURE routine. Each of these arrays uses the predefined package types:

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
```

```
INDEX BY BINARY_INTEGER;
```

```
TYPE NUMBER_TABLE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

In the call to the DESCRIBE\_PROCEDURE routine, you need to pass three parameters: the name of the procedure that you are enquiring about and two null values. These null values are reserved for future use.

In the second example in the slide, the results are displayed for the parameters of the query\_code\_pkg.find\_text\_in\_code function. A data type of 1 indicates that it is a VARCHAR2 data type.



## The DESCRIBE\_PROCEDURE Routine (continued)

### Calling DBMS\_DESCRIBE.DESCRIBE\_PROCEDURE

```
CREATE OR REPLACE PACKAGE use_dbms_describe IS
    PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
CREATE OR REPLACE PACKAGE BODY use_dbms_describe IS
    PROCEDURE get_data (p_obj_name VARCHAR2)
    IS
        v_overload      DBMS_DESCRIBE.NUMBER_TABLE;
        v_position      DBMS_DESCRIBE.NUMBER_TABLE;
        v_level         DBMS_DESCRIBE.NUMBER_TABLE;
        v_arg_name       DBMS_DESCRIBE.VARCHAR2_TABLE;
        v_datatype       DBMS_DESCRIBE.NUMBER_TABLE;
        v_def_value      DBMS_DESCRIBE.NUMBER_TABLE;
        v_in_out         DBMS_DESCRIBE.NUMBER_TABLE;
        v_length         DBMS_DESCRIBE.NUMBER_TABLE;
        v_precision      DBMS_DESCRIBE.NUMBER_TABLE;
        v_scale          DBMS_DESCRIBE.NUMBER_TABLE;
        v_radix          DBMS_DESCRIBE.NUMBER_TABLE;
        v_spare          DBMS_DESCRIBE.NUMBER_TABLE;
    BEGIN
        DBMS_DESCRIBE.DESCRIBE_PROCEDURE
        (p_obj_name, null, null, -- these are the 3 in parameters
         v_overload, v_position, v_level, v_arg_name,
         v_datatype, v_def_value, v_in_out, v_length,
         v_precision, v_scale, v_radix, v_spare, null);
        IF v_in_out.FIRST IS NULL THEN
            DBMS_OUTPUT.PUT_LINE ('No arguments to report.');
        ELSE
            DBMS_OUTPUT.PUT
            ('Name                                     Mode');
            DBMS_OUTPUT.PUT_LINE('  Position      Datatype ');
            FOR i IN v_arg_name.FIRST .. v_arg_name.LAST LOOP
                IF v_position(i) = 0 THEN
                    DBMS_OUTPUT.PUT('This is the RETURN data for
                    the function: ');
                ELSE
                    DBMS_OUTPUT.PUT (
                        rpad(v_arg_name(i), LENGTH(v_arg_name(i)) +
                            42-LENGTH(v_arg_name(i)), ' ');
                END IF;
                DBMS_OUTPUT.PUT( '      ' ||
                    v_in_out(i) || '      ' || v_position(i) ||
                    '      ' || v_datatype(i) );
                DBMS_OUTPUT.NEW_LINE;
            END LOOP;
        END IF;
    END get_data;
END use_dbms_describe;
```

## Using ALL\_ARGUMENTS

Query the ALL\_ARGUMENTS view to find information about arguments for procedures and functions:

```
SELECT object_name, argument_name, in_out, position, data_type
FROM   all_arguments
WHERE  package_name = 'CREDIT_CARD_PKG';
```

OBJECT_NAME	ARGUMENT_NAME	IN_OUT	POSITION	DATA_TYPE
DISPLAY_CARD_INFO	P_CUST_ID	IN	1	NUMBER
UPDATE_CARD_INFO		OUT	1	OBJECT
UPDATE_CARD_INFO	O_CARD_INFO	OUT	4	TABLE
UPDATE_CARD_INFO	P_CARD_NO	IN	3	VARCHAR2
UPDATE_CARD_INFO	P_CARD_TYPE	IN	2	VARCHAR2
UPDATE_CARD_INFO	P_CUST_ID	IN	1	NUMBER
CUST_CARD_INFO		IN/OUT	1	OBJECT
CUST_CARD_INFO	P_CARD_INFO	IN/OUT	2	TABLE
CUST_CARD_INFO	P_CUST_ID	IN	1	NUMBER
CUST_CARD_INFO		OUT	0	PL/SQL BOOLEAN

10 rows selected.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the ALL\_ARGUMENTS Dictionary View

You can also query the ALL\_ARGUMENTS dictionary view to find information about the arguments of procedures and functions to which you have access. Like DBMS\_DESCRIBE, the ALL\_ARGUMENTS view returns information in textual rather than numeric form. Though there is overlap between the two, there is unique information to be found both in DBMS\_DESCRIBE and ALL\_ARGUMENTS.

In the example shown in the slide, the argument name, mode, position, and data type are returned for ORDERS\_APP\_PKG. Note the following:

- A position of 1 and a sequence and level of 0 indicates that the procedure has no arguments.
- A function that has no arguments is displayed as a single row for the RETURN clause, with a position of 0.
- The argument name for the RETURN clause is NULL.
- If the programs are overloaded, the OVERLOAD column (not shown in the slide) indicates the Nth overloading; otherwise, it is NULL.
- The DATA\_LEVEL column (not shown in the slide) value of 0 identifies a parameter as it appears in the program specification.

## Using ALL\_ARGUMENTS

Other column information:

- Details about the data type are found in the DATA\_TYPE and TYPE\_ columns.
- All arguments in the parameter list are at level 0.
- For composite parameters, the individual elements of the composite are assigned levels, starting at 1.
- The POSITION-DATA\_LEVEL column combination is unique only for a level 0 argument (the actual parameter, not its subtypes if it is a composite).

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the ALL\_ARGUMENTS Dictionary View (continued)

The DATA\_TYPE column holds the generic PL/SQL data type. To find more information about the data type, query the TYPE\_ columns.

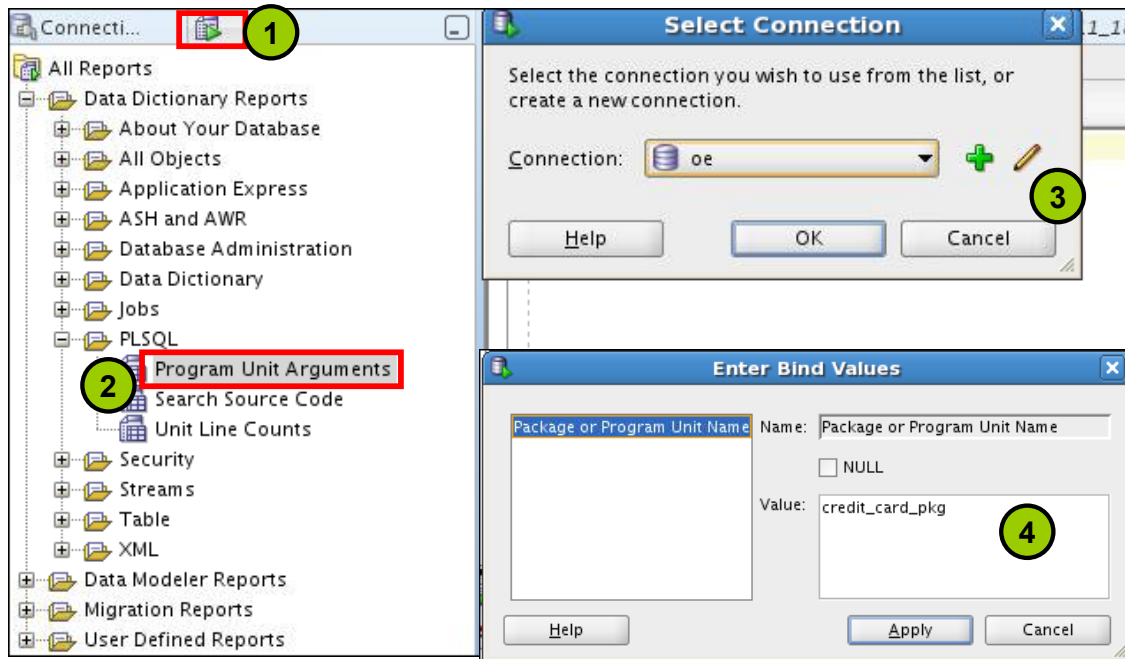
- TYPE\_NAME: Holds the name of the type of the argument. If the type is a package local type (that is, it is declared in a package specification), this column displays the name of the package.
- TYPE\_SUBNAME: Is relevant only for package local types. Displays the name of the type declared in the package identified in the TYPE\_NAME column. For example, if the data type is a PL/SQL table, you can find out the type of the table only by looking at the TYPE\_SUBNAME column.

**Note:** The DEFAULT\_VALUE and DEFAULT\_LENGTH columns are reserved for future use and do not currently contain information about a parameter's default value. However, you can use DBMS\_DESCRIBE to find some default value information. In this package, the parameter DEFAULT\_VALUE returns 1 if there is a default value; otherwise, it returns 0.

By combining the information from DBMS\_DESCRIBE and ALL\_ARGUMENTS, you can find valuable information about parameters, as well as about how your PL/SQL routines are overloaded.

# Using SQL Developer to Report on Arguments

Use Reports:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using SQL Developer to Find Argument Information

SQL Developer comes with predefined reports that you can use to find PL/SQL coding information on your program unit arguments.

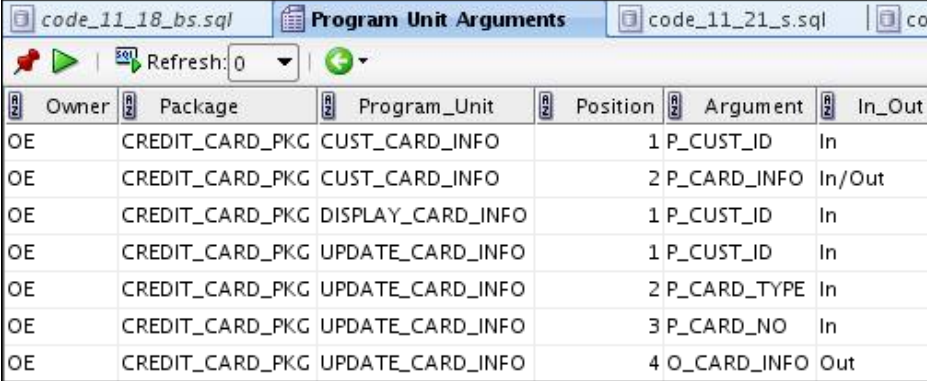
Use the Reports feature to find information on program unit arguments.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select Program Unit Arguments.
3. When prompted, enter the connection information.
4. Enter the name of the program unit on which you want the arguments reported.

# Using SQL Developer to Report on Arguments

Results:

5



Owner	Package	Program_Unit	Position	Argument	In_Out
OE	CREDIT_CARD_PKG	CUST_CARD_INFO	1	P_CUST_ID	In
OE	CREDIT_CARD_PKG	CUST_CARD_INFO	2	P_CARD_INFO	In/Out
OE	CREDIT_CARD_PKG	DISPLAY_CARD_INFO	1	P_CUST_ID	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	1	P_CUST_ID	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	2	P_CARD_TYPE	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	3	P_CARD_NO	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	4	O_CARD_INFO	Out

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using SQL Developer to Find Argument Information (continued)

5. View the results of your search.

In the example shown in the slide, the output displays the owner of the stored PL/SQL package, the name of the PL/SQL package, the names of the subroutines within the PL/SQL package, the position of the arguments within the package, the argument names, and the argument types (IN, OUT, or IN OUT).

## Using DBMS\_UTILITY.FORMAT\_CALL\_STACK

- This function returns the formatted text string of the current call stack.
- Use it to find the line of code being executed.

```
EXECUTE third_one
----- PL/SQL Call Stack -----
  object      line  object
  handle      number name
0x566ce8e0      4  procedure OE.FIRST_ONE
0x5803f7a8      5  procedure OE.SECOND_ONE
0x569c3770      6  procedure OE.THIRD_ONE
0x567ee3d0      1  anonymous block

PL/SQL procedure successfully completed.
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### The DBMS\_UTILITY.FORMAT\_CALL\_STACK Function

Another tool that is available to you is the FORMAT\_CALL\_STACK function within the DBMS\_UTILITY supplied package. It returns the call stack in a formatted character string. The results shown above were generated based on the following routines:

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
    dbms_output.put_line(
        substr(dbms_utility.format_call_stack, 1, 255));
END;
/

CREATE OR REPLACE PROCEDURE second_one
IS
BEGIN
    null;
    first_one;
END;
/
-- continued on next page
```

## The DBMS\_UTILITY.FORMAT\_CALL\_STACK Function (continued)

-- continued from previous page

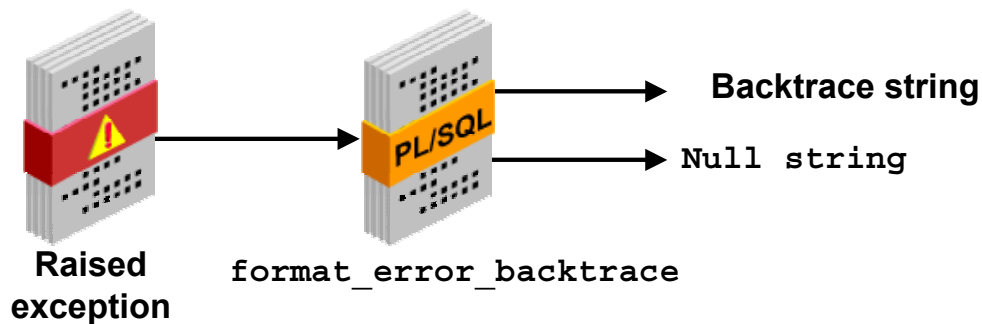
```
CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
    null;
    null;
    second_one;
END;
/
```

The output from the FORMAT\_CALL\_STACK function shows the object handle number, the line number where a routine is called from, and the routine that is called. Note that the NULL; statements added to the procedures are used to emphasize the line number where the routine is called from.

## Finding Error Information

`DBMS_UTILITY.FORMAT_ERROR_BACKTRACE:`

- Shows you the call stack at the point where an exception is raised
- Returns:
  - The backtrace string
  - A null string if no errors are being handled



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE`

You can use this function to display the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output returned is similar to the output of the `SQLERRM` function but not subject to the same size limitation.

### Using `DBMS_UTILITY.FORMAT_ERROR_STACK`

You can use this function to format the current error stack. It can be used in exception handlers to view the full error stack. The function returns the error stack up to 2,000 bytes.



## Finding Error Information

```
CREATE OR REPLACE PROCEDURE top_with_logging IS
  -- NOTE: SQLERRM in principle gives the same info
  -- as format_error_stack.
  -- But SQLERRM is subject to some length limits,
  -- while format_error_stack is not.
BEGIN
  P5(); -- this procedure, in turn, calls others,
        -- building a stack. P0 contains the exception
EXCEPTION
  WHEN OTHERS THEN
    log_errors ( 'Error Stack...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_STACK() );
    log_errors ( 'Error Backtrace...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '-----' );
END top_with_logging;
/
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using `FORMAT_ERROR_STACK` and `FORMAT_ERROR_BACKTRACE`

To show you the functionality of the `FORMAT_ERROR_STACK` and `FORMAT_ERROR_BACKTRACE` functions, a `TOP_WITH_LOGGING` procedure is created. This procedure calls the `LOG_ERRORS` procedure and passes to it the results of the `FORMAT_ERROR_STACK` and `FORMAT_ERROR_BACKTRACE` functions.

The `LOG_ERRORS` procedure is shown on the next page.

# Finding Error Information

```
CREATE OR REPLACE PROCEDURE log_errors ( i_buff IN VARCHAR2 ) IS
  g_start_pos PLS_INTEGER := 1;
  g_end_pos   PLS_INTEGER;
  FUNCTION output_one_line RETURN BOOLEAN IS
  BEGIN
    g_end_pos := INSTR ( i_buff, CHR(10), g_start_pos );
    CASE g_end_pos > 0
      WHEN TRUE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff,
                                         g_start_pos, g_end_pos-g_start_pos ));
        g_start_pos := g_end_pos+1;
        RETURN TRUE;
      WHEN FALSE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff, g_start_pos,
                                         (LENGTH(i_buff)-g_start_pos)+1 ));
        RETURN FALSE;
    END CASE;
  END output_one_line;
BEGIN
  WHILE output_one_line() LOOP NULL;
  END LOOP;
END log_errors;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## LOG\_ERRORS Example

This procedure takes the return results of the `FORMAT_ERROR_STACK` and `FORMAT_ERROR_BACKTRACE` functions as an IN string parameter, and reports it back to you using `DBMS_OUTPUT.PUT_LINE`. The `LOG_ERRORS` procedure is called twice from the `TOP_WITH_LOGGING` procedure. The first call passes the results of `FORMAT_ERROR_STACK`, and the second procedure passes the results of `FORMAT_ERROR_BACKTRACE`.

**Note:** You can use `UTL_FILE` instead of `DBMS_OUTPUT` to write and format the results to a file.

## LOG\_ERRORS Example (continued)

Next, several procedures are created and one procedure calls another so that a stack of procedures is built. The P0 procedure raises a zero divide exception when it is invoked. The call stack is:

TOP\_WITH\_LOGGING > P5 > P4 > P3 > P2 > P1 > P0

```
SET DOC OFF
SET FEEDBACK OFF
SET ECHO OFF

CREATE OR REPLACE PROCEDURE P0 IS
    e_01476 EXCEPTION;
    pragma exception_init ( e_01476, -1476 );
BEGIN
    RAISE e_01476;  -- this is a zero divide error
END P0;
/
CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
    P0();
END P1;
/
CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
    P1();
END P2;
/
CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
    P2();
END P3;
/
CREATE OR REPLACE PROCEDURE P4 IS
    BEGIN P3();
END P4;
/
CREATE OR REPLACE PROCEDURE P5 IS
    BEGIN P4();
END P5;
/
CREATE OR REPLACE PROCEDURE top IS
BEGIN
    P5(); -- this procedure is used to show the results
        -- without using the TOP_WITH_LOGGING routine.
END top;
/
SET FEEDBACK ON
```

## Finding Error Information

### Results:

```
EXECUTE top_with_logging
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP_WITH_LOGGING", line 7
-----
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Finding Error Information Results

The results from executing the TOP\_WITH\_LOGGING procedure is shown. Note that the error stack displays the exception encountered. The backtrace information traces the flow of the exception to its origin.

If you execute the TOP procedure without using the TOP\_WITH\_LOGGING procedure, these are the results:

```
EXECUTE top
BEGIN top; END;
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP", line 3
ORA-06512: at line 1
```

Note that the line number reported is misleading.

# Lesson Agenda

- Running reports on source code
- **Determining identifier types and usages**
- Using `DBMS_METADATA` to retrieve object definitions

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# PL/Scope

- **Definition**
  - Is a tool that is used for extracting, organizing, and storing user-supplied identifiers from PL/SQL source code
  - Works with the PL/SQL compiler
  - Gathers data on scoping and overloading
- **Benefits**
  - Can potentially increase developer productivity
  - Is a valuable resource in helping developers understand source code
  - Can be used to build a PL/SQL IDE

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## PL/Scope

With PL/Scope, you can produce a cross-referenced repository of PL/SQL identifiers to gather information about your PL/SQL applications.

PL/Scope has two targeted audiences:

- PL/SQL application developers: Targeted to use PL/Scope through IDEs such as JDeveloper and SQL Developer
- IDE and Tools developers: Provides the means to build a comprehensive cross-referenced repository by extracting, categorizing, organizing, and storing all identifiers discovered in the PL/SQL source code

## Collecting PL/Scope Data

Collected data includes:

- Identifier types
- Usages
  - Declaration
  - Definition
  - Reference
  - Call
  - Assignment
- Location of usage



ORACLE

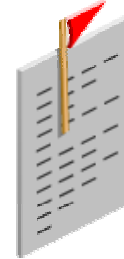
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Collecting PL/Scope Data

You can use PL/Scope to collect data about the user-defined identifiers found in the PL/SQL source code. The data is collected at compilation time and made available in the static data dictionary views.

# Using PL/Scope

- Set the PL/SQL compilation parameter `PLSCOPE_SETTINGS`.
- Valid values for `IDENTIFIERS`:
  - `ALL` – Collect all PL/SQL identifier actions found in compiled source
  - `NONE` – Do not collect any identifier actions (the default)
- Set at the session, system, or per library unit basis:
  - `ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
  - `ALTER SYSTEM SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
  - `ALTER functionname COMPILE PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
- The `USER/ALL/DBA_IDENTIFIERS` catalog view holds the collected identifier values.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using PL/Scope

To use PL/Scope, you need to set the PL/SQL compilation parameter `PLSCOPE_SETTINGS` to either `'IDENTIFIERS:ALL'` or `'IDENTIFIERS:NONE'`. You can use the `ALTER SESSION`, `ALTER SYSTEM`, or `ALTER COMPILE` statements to set this parameter. After the parameter is set, any code that you compile is analyzed for PL/Scope. The information collected is gathered in the `USER/ALL/DBA_IDENTIFIERS` dictionary view.

The identifier action describes how the identifier is used, such as in a declaration, definition, reference, assignment, or call.

Query the `USER|ALL|DBA_PLSQL_OBJECT_SETTINGS` views to view what the `PLSCOPE_SETTINGS` are set to. This view contains a column called `PLSCOPE_SETTINGS`. When you query this column, by default all objects are set to `'IDENTIFIERS:NONE'`, unless you reset the PL/SQL compilation parameter and recompile the code.



## The USER/ALL/DBA\_IDENTIFIERS Catalog View

Column	Description
OWNER	Owner of the identifier
NAME	Name of the identifier (may not be unique)
SIGNATURE	A unique signature for this identifier
TYPE	The type of the identifier, such as variable, formal, varray
OBJECT_NAME	The object name where the action occurred
OBJECT_TYPE	The type of object where the action occurred
USAGE	The action performed on the identifier, such as declaration, definition, reference, assignment, or call
USAGE_ID	The unique key for the identifier usage
LINE	The line where the identifier action occurred
COL	The column where the identifier action occurred
USAGE_CONTEXT_ID	The context USAGE_ID of the identifier action

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### New Dictionary View

The USER/ALL/DBA\_IDENTIFIERS dictionary view is added in Oracle Database 11g to collect information about your identifiers for any code that is compiled or altered when the PL/SQL compilation parameter is set to `PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL'`.

# Sample Data for PL/Scope

Sample data for scoping:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      ...
  END update_card_info;
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      ...
  END display_card_info;
END credit_card_pkg; -- package body
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Sample Data

The CREDIT\_CARD\_PKG created earlier is used for PL/SQL scoping. The complete code is shown on the following page.

The code is a simple package that contains one type, two procedures, and one function. Identifier information will be collected on this package in the following pages.

## Sample Data (continued)

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
                DBMS_OUTPUT.PUT('Card Type: ' || v_card_info(idx).card_type
                                || ' ');
                DBMS_OUTPUT.PUT_LINE('/ Card No: ' || v_card_info(idx).card_num );
            END LOOP;
        ELSE
            DBMS_OUTPUT.PUT_LINE('Customer has no credit cards. ');
        END IF;
    END display_card_info;
END credit_card_pkg; -- package body
/
```

# Collecting Identifiers

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';  
ALTER PACKAGE credit_card_pkg COMPILE;
```

- Identifier information is collected in the USER\_IDENTIFIERS dictionary view, where you can:
  - Perform a basic identifier search
  - Use contexts to describe identifiers
  - Find identifier actions
  - Describe identifier actions

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Collecting Identifiers

In the example shown in the slide, the PL/SQL compilation parameter PLSCOPE\_SETTINGS is turned on to collect information about all identifiers. The CREDIT\_CARD\_PKG is then recompiled. After it is recompiled, the identifier information is available in the ALL | USER | DBA\_IDENTIFIER views.

To verify that you have enabled identifier information to be collected on the CREDIT\_CARD\_PKG package, you can issue the following statement:

```
SELECT PLSCOPE_SETTINGS  
FROM USER_PLSQL_OBJECT_SETTINGS  
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';  
  
PLSCOPE_SETTINGS  
-----  
IDENTIFIERS:ALL
```

## Viewing Identifier Information

Create a hierarchical report of identifier information:

```
WITH v AS
  (SELECT      Line,
               Col,
               INITCAP(NAME) Name,
               LOWER(TYPE)    Type,
               LOWER(USAGE)   Usage,
               USAGE_ID, USAGE_CONTEXT_ID
  FROM USER_IDENTIFIERS
 WHERE Object_Name = 'CREDIT_CARD_PKG'
       AND Object_Type = 'PACKAGE BODY' )
  SELECT RPAD(LPAD(' ', 2*(Level-1)) ||
             Name, 20, '.') || ' ' ||
         RPAD(Type, 20) || RPAD(Usage, 20)
  FROM v
  START WITH USAGE_CONTEXT_ID = 0
  CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
  ORDER SIBLINGS BY Line, Col;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Viewing Identifier Information

The sample SQL statement shown above retrieves and formats the identifier information that is collected for the CREDIT\_CARD\_PKG package body. Note that the inline view retrieves values from the Name, Type, and Usage columns in the USER\_IDENTIFIERS dictionary view.

## Viewing Identifier Information

Results:

### IDENTIFIER\_USAGE\_CONTEXTS

```
-----
Credit_Card_Pkg..... package          definition
  Cust_Card_Info.... function          definition
    P_Cust_Id..... formal in          declaration
    P_Card_Info..... formal in out    declaration
    V_Card_Info_Exis variable          declaration
    P_Card_Info..... formal in out    assignment
    P_Cust_Id..... formal in          reference
    Plitblm..... synonym              call
      P_Card_Info... formal in out    reference
      V_Card_Info_Ex variable          assignment
      V_Card_Info_Exis variable        assignment
...
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Viewing Identifier Information (continued)

The results are formatted and display the name of the identifier, type, and how the identifier is used.

**Note:** PLITBLM is an Oracle-supplied package that contains subprograms to help implement the basic language features. PLITBLM handles index-table operations.

## Performing a Basic Identifier Search

Display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions:

```
SELECT NAME, SIGNATURE, TYPE
FROM USER_IDENTIFIERS
WHERE USAGE='DECLARATION'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

ID	NAME	ID	SIGNATURE	ID	TYPE
1	TAX_AMT		C857A22DF0C8ED7946088BD2E91CD833		FUNCTION
2	GET_CREDIT		72EBEDA9DD0C3A46F11DA088EDE076D		FUNCTION
3	ORD_COUNT		017E03E4C553FC8A091125490D8E5208		FUNCTION
4	GET_FILESIZE		B76901C0A7047545D96589337D670EC2		FUNCTION
5	X		A4E010EB5FF197741D14C266748AF704		FORMAL IN
6	V_ID		2684544E2F2F867F6774638C4E8451C6		FORMAL IN
7	CUST_NO		63E48F5F621EDC75E7A338D885156DB6		FORMAL IN
8	P_FILE_PTR		258954A3FA37084EE4A7C1196CC8FC6E		FORMAL IN OUT
9	V_CREDIT		9C6689C39824408734C6CC28309F4CE7		VARIABLE
10	V_COUNT		BF3EC8187C9F044D6CB086125C28E4CA		VARIABLE

...

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Performing a Basic Identifier Search

You can display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions. Valid actions are:

- **DECLARATION:** All identifiers have one and only one declaration. Each declaration may also have an associated type.
- **TYPE:** Has the value of either packages, function or procedures, object types, triggers, or exceptions
- **SIGNATURE:** A unique value that distinguishes the identifier from other identifiers with the same name, whether they are defined in the same program unit or different program units

# Using USER\_IDENTIFIERS to Find All Local Variables

Find all local variables:

```
SELECT a.NAME variable_name, b.NAME context_name, a.SIGNATURE
FROM USER_IDENTIFIERS a, USER_IDENTIFIERS b
WHERE a.USAGE_CONTEXT_ID = b.USAGE_ID
      AND a.TYPE = 'VARIABLE'
      AND a.USAGE = 'DECLARATION'
      AND a.OBJECT_NAME = 'CREDIT_CARD_PKG'
      AND a.OBJECT_NAME = b.OBJECT_NAME
      AND a.OBJECT_TYPE = b.OBJECT_TYPE
      AND (b.TYPE = 'FUNCTION' or b.TYPE = 'PROCEDURE')
ORDER BY a.OBJECT_TYPE, a.USAGE_ID;
```

VARIABLE_NAME	CONTEXT_NAME	SIGNATURE
V_CARD_INFO	UPDATE_CARD_INFO	5FE3409823709E61A12314F2667949CA
I	UPDATE_CARD_INFO	F5008E5A79542F08A3A8FF4AFC986C1E
V_CARD_INFO	DISPLAY_CARD_INFO	5ACFED98136068B5A8BC8C9063F974E4
I	DISPLAY_CARD_INFO	E5E48887FD19043F4240B84ECA77E916

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Finding Information About Local Variables

In the example shown in the slide, all local variables belonging to procedures or functions are found for the CREDIT\_CARD\_PKG package.



## Finding Identifier Actions

Find all usages performed on the local variable:

```
SELECT USAGE, USAGE_ID, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='5FE3409B23709E61A12314F2667949CA'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

USAGE	USAGE_ID	OBJECT_NAME	OBJECT_TYPE
DECLARATION	9	CREDIT_CARD_PKG	PACKAGE BODY
ASSIGNMENT	14	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	16	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	19	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	21	CREDIT_CARD_PKG	PACKAGE BODY
ASSIGNMENT	22	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	28	CREDIT_CARD_PKG	PACKAGE BODY

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Finding Identifier Actions

You can find all actions performed on an identifier. In the example in the slide, all actions performed on a variable are found by searching on the signature of the variable. The variable is called V\_CARD\_INFO, and it is used in the DISPLAY\_CARD\_INFO procedure. Variable V\_CARD\_INFO's signature is found in the previous query. It is available to you in the SIGNATURE column of the USER\_IDENTIFIERS dictionary view.

The different types of the identifier usage are:

- DECLARATION
- DEFINITION
- CALL
- REFERENCE
- ASSIGNMENT

## Finding Identifier Actions

Find out where the assignment to the local identifier `i` occurred:

```
SELECT LINE, COL, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='5ACFED9813606BB5A8BCBC9063F974E4'
AND USAGE='ASSIGNMENT';
```

LINE	COL	OBJECT_NAME	OBJECT_TYPE
35	12	CREDIT_CARD_PKG	PACKAGE BODY

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Finding Identifier Actions (continued)

The local `V_CARD_INFO` identifier is found on line 35 of the `CREDIT_CARD_PKG` body.

# Lesson Agenda

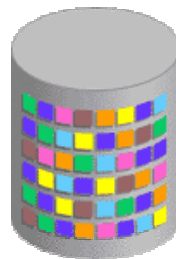
- Running reports on source code
- Determining identifier types and usages
- **Using DBMS\_METADATA to retrieve object definitions**

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## DBMS\_METADATA Package

The DBMS\_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### DBMS\_METADATA Package

You can invoke DBMS\_METADATA to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

You can use DBMS\_METADATA for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column data types, and so on), and then converting the metadata to data definition language (DDL) so that the object can be re-created on the same or another database. In the past, you needed to do this programmatically with problems resulting in each new release.

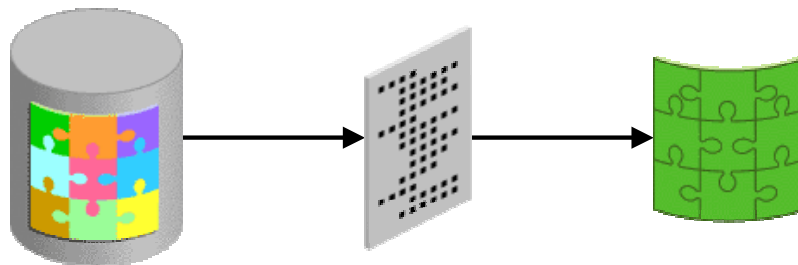
The DBMS\_METADATA functionality is used for the Oracle 11g Export/Import replacement, commonly called “the Data Pump.”

**Note:** For more information about the DBMS\_DATAPUMP package, refer to the online course titled *Oracle Database 11g: Reduce Management - Tools and Utilities*.

# Metadata API

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Metadata API

Every entity in the database is modeled as an object that belongs to an object type. For example, the `ORDERS` table is an object; its object type is `TABLE`. When you fetch an object's metadata, you must specify the object type.

Every object type is implemented by using three entities:

- A user-defined type (UDT) whose attributes comprise all metadata for objects of the type. An object's XML representation is a translation of a type instance into XML with the XML tag names derived from the type attribute names. (In the case of tables, several UDTs are needed to represent the different varieties of the object type.)
- An object view of the UDT that populates the instances of the object type
- An Extensible Style Sheet Language (XSL) script that converts the XML representation of an object into SQL DDL

## Subprograms in DBMS\_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved, such as the object name or schema
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single FETCH_XXX call
GET_QUERY	Returns the text of the queries that will be used by FETCH_XXX
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned
ADD_TRANSFORM	Specifies a transform that FETCH_XXX applies to the XML representation of the retrieved objects
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by transform_handle
FETCH_XXX	Returns metadata for objects that meet the criteria established by OPEN, SET_FILTER
CLOSE	Invalidates the handle returned by OPEN and cleans up the associated state

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Subprograms in DBMS\_METADATA

The table provides an overview of the procedures and functions that are available in the DBMS\_METADATA package. To retrieve metadata, you can specify the:

- Kind of object to be retrieved—either an object type (a table, index, procedure) or a heterogeneous collection of object types that form a logical unit (such as database export and schema export)
- Selection criteria (owner, name, and so on)
- “parse items” attributes of objects to be parsed and returned separately
- Transformations on the output, implemented by XSLT scripts

The package provides two types of retrieval interfaces for two types of usage:

- **For programmatic use:** OPEN, SET\_FILTER, SET\_COUNT, GET\_QUERY, SET\_PARSE\_ITEM, ADD\_TRANSFORM, SET\_TRANSFORM\_PARAM, SET\_REMAP\_PARAM, FETCH\_XXX, and CLOSE. These enable a flexible selection criteria and the extraction of a stream of objects.
- **For use in SQL queries and for ad hoc browsing:** The GET\_XXX interfaces (GET\_XML and GET\_DDL) return metadata for a single named object. The GET\_DEPENDENT\_XXX and GET\_GRANTED\_XXX interfaces return metadata for one or more dependent or granted objects. None of these APIs supports heterogeneous object types.

## **FETCH\_xxx Subprograms**

Name	Description
FETCH_XML	This function returns the XML metadata for an object as an XMLType.
FETCH_DDL	This function returns the DDL (either to create or to drop the object) into a predefined nested table.
FETCH_CLOB	This function returns the objects (transformed or not) as a CLOB.
FETCH_XML_CLOB	This procedure returns the XML metadata for the objects as a CLOB in an IN OUT NOCOPY parameter to avoid expensive LOB copies.

**ORACLE**

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### **FETCH\_xxx Subprograms**

These functions and procedures return metadata for objects meeting the criteria established by the call to the OPEN function that returned the handle, and the subsequent calls to SET\_FILTER, SET\_COUNT, ADD\_TRANSFORM, and so on. Each call to FETCH\_xxx returns the number of objects specified by SET\_COUNT (or a smaller number, if fewer objects remain in the current cursor) until all objects are returned.

## SET\_FILTER Procedure

- Syntax:

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- Example:

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME', 'OE');
...
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### SET\_FILTER Procedure

You use the SET\_FILTER procedure to identify restrictions on the objects that are to be retrieved. For example, you can specify restrictions on an object or schema that is being retrieved. This procedure is overloaded with the parameters that have the following meanings:

- `handle` is the handle returned from the OPEN function.
- `name` is the name of the filter. For each filter, the object type applies to its name, data type (text or Boolean), and meaning or effect (including its default value, if there is one).
- `value` is the value of the filter. It can be text, Boolean, or a numeric value.
- `object_type_path` is a path name designating the object types to which the filter applies.

By default, the filter applies to the object type of the OPEN handle.

If you use an expression filter, it is placed to the right of a SQL comparison, and the value is compared with it. The value must contain parentheses and quotation marks where appropriate. A filter value is combined with a particular object attribute to produce a WHERE condition in the query that fetches the objects.



# Filters

There are over 70 filters that are organized into object type categories such as:

- Named objects
- Tables
- Objects dependent on tables
- Index
- Dependent objects
- Granted objects
- Table data
- Index statistics
- Constraints
- All object types
- Database export

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Filters

There are over 70 filters that you can specify when using the `SET_FILTER` procedure. These filters are organized into object type categories. Some of the new object type categories in Oracle Database 11g are listed in the slide.

When using the `SET_FILTER` procedure, you specify the name of the filter and its respective value. For example, you can use the `SCHEMA` filter with a value to identify the schema whose objects are selected. Then use a second call to the `SET_FILTER` procedure and use a filter named `INCLUDE_USER` that has a Boolean data type for its value. If it is set to `TRUE`, objects that contain privileged information about the user are retrieved.

```
DBMS_METADATA.SET_FILTER(handle, SCHEMA, 'OE');  
DBMS_METADATA.SET_FILTER(handle, INCLUDE_USER, TRUE);
```

Each call to `SET_FILTER` causes a `WHERE` condition to be added to the underlying query that fetches the set of objects. The `WHERE` conditions are combined by using an `AND` operator, so that you can use multiple `SET_FILTER` calls to refine the set of objects to be returned.

## Examples of Setting Filters

Set the filter to fetch the OE schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain PAYROLL at the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',  
    'IN (''PAYROLL'', ''OE'')');  
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',  
    '=' 'FUNCTION' ');  
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',  
    '=' 'PROCEDURE' ');  
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',  
    '=' 'PACKAGE' ');  
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',  
    'LIKE ''PAYROLL%''', 'VIEW');
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Examples of Setting Filters

The example shown in the slide calls the SET\_FILTER procedure several times to create a WHERE condition that identifies which object types are to be fetched. First, the objects in the PAYROLL and OE schemas are identified as object types to be fetched. Subsequently, the SET\_FILTER procedure identifies certain object types (functions, procedures, and packages) and view object names that are to be excluded.

## Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
  v_hdl      NUMBER; v_th1  NUMBER; v_th2  NUMBER;
  v_doc      sys.ku$_ddl; ← 1
BEGIN
  v_hdl := DBMS_METADATA.OPEN('SCHEMA_EXPORT'); ← 2
  DBMS_METADATA.SET_FILTER (v_hdl, 'SCHEMA', 'OE'); ← 3
  v_th1 := DBMS_METADATA.ADD_TRANSFORM (v_hdl, ← 4
    'MODIFY', NULL, 'TABLE');
  DBMS_METADATA.SET_REMAP_PARAM(v_th1, ← 5
    'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
  v_th2 := DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL'); ← 4
  DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ← 6
    'SQLTERMINATOR', TRUE);
  DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ← 6
    'REF_CONSTRAINTS', FALSE, 'TABLE');
  LOOP
    v_doc := DBMS_METADATA.FETCH_DDL(v_hdl); ← 7
    EXIT WHEN v_doc IS NULL;
  END LOOP;
  DBMS_METADATA.CLOSE(v_hdl); ← 8
END;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Programmatic Use: Example 1

In this example, all objects are retrieved from the HR schema as creation DDL. The MODIFY transform is used to change the tablespaces for the tables.

1. The DBMS\_METADATA package has several predefined types that are owned by SYS. The `sys.ku$_ddl` stand-alone object type used in the DBMS\_METADATA package. It is a table type that holds the CLOB type of data.
2. You use the OPEN function to specify the type of object to be retrieved, the version of its metadata, and the object model. It returns a context handle for the set of objects. In this example, 'SCHEMA\_EXPORT' is the object type, which indicates all metadata objects in a schema. There are 85 predefined types of objects for the model that you can specify for this parameter. Both the version of metadata and the object model parameters are not identified in this example. The version of metadata parameter defaults to 'COMPATIBLE'. You can also specify 'LATEST' or a specific database version.
3. The SET\_FILTER procedure identifies restrictions on the objects that are to be retrieved.

## Programmatic Use: Example 1 (continued)

4. The `ADD_TRANSFORM` function specifies a transform that `FETCH_XXX` applies to the XML representation of the retrieved objects. You can have multiple transforms. In the example, two transforms occur, one for each of the `th1` and `th2` program variables. The `ADD_TRANSFORM` function accepts four parameters and returns a number representing the opaque handle to the transform. The parameters are the handle returned from the `OPEN` statement, the name of the transform (`DDL`, `DROP`, or `MODIFY`), the encoding name (which is the name of the national language support [NLS] character set in which the style sheet pointed to by the name is encoded), and the object type. If the object type is omitted, the transform applies to all objects; otherwise, it applies only to the object type specified.

The first transform shown in the program code is the handle returned from the `OPEN` function. The second transform shown in the code has two parameter values specified. The first parameter is the handle identified from the `OPEN` function. The second parameter value is the `DDL`, which means the document is transformed to the DDL that creates the object. The output of this transform is not an XML document. The third and fourth parameters are not specified. Both take the default values for the encoding and object type parameters.

5. The `SET_REMAP_PARAM` procedure identifies the parameters to the XSLT style sheet identified by the transform handle, which is the first parameter passed to the procedure. In the example, the second parameter value '`REMAP_TABLESPACE`' means that the objects have their tablespaces renamed from an old value to a new value. In the `ADD_TRANSFORM` function, the choices are `DDL`, `DROP`, or `MODIFY`. For each of these values, the `SET_REMAP_PARAM` identifies the name of the parameter. `REMAP_TABLESPACE` means the objects in the document will have their tablespaces renamed from an old value to a new value. The third and fourth parameters identify the old value and new value. In this example, the old tablespace name is `SYSTEM`, and the new tablespace name is `TBS1`.
6. `SET_TRANSFORM_PARAM` works similarly to `SET_REMAP_PARAM`. In the code shown, the first call to `SET_TRANSFORM_PARAM` identifies the parameters for the `th2` variable. The `SQLTERMINATOR` and `TRUE` parameter values cause the SQL terminator (`;` or `/`) to be appended to each DDL statement.  
The second call to `SET_TRANSFORM_PARAM` identifies more characteristics for the `th2` variable. `REF_CONSTRAINTS`, `FALSE`, and `TABLE` means that the referential constraints on the tables are not copied to the document.
7. The `FETCH_DDL` function returns the metadata for objects that meet the criteria established by the `OPEN`, `SET_FILTER`, `ADD_TRANSFORM`, `SET_REMAP_PARAM`, and `SET_TRANSFORM_PARAM` subroutines.
8. The `CLOSE` function invalidates the handle returned by the `OPEN` function and cleans up the associated state. Use this function to terminate the stream of objects established by the `OPEN` function.

## Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
  v_hdl  NUMBER; -- returned by 'OPEN'
  v_th   NUMBER; -- returned by 'ADD_TRANSFORM'
  v_doc  CLOB;
BEGIN
  -- specify the OBJECT TYPE
  v_hdl := DBMS_METADATA.OPEN('TABLE');
  -- use FILTERS to specify the objects desired
  DBMS_METADATA.SET_FILTER(v_hdl, 'SCHEMA', 'OE');
  DBMS_METADATA.SET_FILTER
    (v_hdl, 'NAME', 'ORDERS');
  -- request to be TRANSFORMED into creation DDL
  v_th := DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL');
  -- FETCH the object
  v_doc := DBMS_METADATA.FETCH_CLOB(v_hdl);
  -- release resources
  DBMS_METADATA.CLOSE(v_hdl);
  RETURN v_doc;
END;
/
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Programmatic Use: Example 2

This example returns the metadata for the ORDERS table. The result is:

```
set pagesize 0
set long 1000000
SELECT get_table_md FROM dual;

CREATE TABLE "OE"."ORDERS"
(
  "ORDER_ID" NUMBER(12,0),
  "ORDER_DATE" TIMESTAMP (6) WITH LOCAL TIME ZONE
CONSTRAINT "ORDER_DATE_NN" NOT
NULL ENABLE,
  "ORDER_MODE" VARCHAR2(8),
  "CUSTOMER_ID" NUMBER(6,0) CONSTRAINT
"ORDER_CUSTOMER_ID_NN" NOT NULL ENABLE,
  "ORDER_STATUS" NUMBER(2,0),
  "ORDER_TOTAL" NUMBER(8,2),
  "SALES_REP_ID" NUMBER(6,0),
  "PROMOTION_ID" NUMBER(6,0),
  CONSTRAINT "ORDER_MODE_LOV" CHECK (order_mode in
...

```

## Programmatic Use: Example 2 (continued)

```
...
CONSTRAINT "ORDER_TOTAL_MIN" CHECK (order_total >= 0) ENABLE,
CONSTRAINT "ORDER_PK" PRIMARY KEY ("ORDER_ID")
    USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 NOLOGGING
        COMPUTE STATISTICS
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
    2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
DEFAULT)
TABLESPACE "EXAMPLE" ENABLE,
CONSTRAINT "ORDERS_CUSTOMER_ID_FK" FOREIGN KEY ("CUSTOMER_ID")
    REFERENCES "OE"."CUSTOMERS" ("CUSTOMER_ID")
    ON DELETE SET NULL ENABLE,
CONSTRAINT "ORDERS_SALES_REP_FK" FOREIGN KEY
    ("SALES_REP_ID") REFERENCES "HR"."EMPLOYEES"
    ("EMPLOYEE_ID") ON DELETE SET NULL ENABLE
)
PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
    NOCOMPRESS NOLOGGING
STORAGE(INITIAL 65536 NEXT 1048576
    MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
    BUFFER_POOL DEFAULT)
TABLESPACE "EXAMPLE"
```

You can accomplish the same effect with the browsing interface:

```
SELECT dbms_metadata.get_ddl
    ('TABLE', 'ORDERS', 'OE')
FROM dual;
```

## Browsing APIs

Name	Description
GET_XXX	The GET_XML and GET_DDL functions return metadata for a single named object.
GET_DEPENDENT_XXX	This function returns metadata for a dependent object.
GET_GRANTED_XXX	This function returns metadata for a granted object.
Where xxx is:	DDL or XML

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Browsing APIs

The browsing APIs are designed for use in SQL queries and ad hoc browsing. These functions allow you to fetch metadata for objects with a single call. They encapsulate calls to OPEN, SET\_FILTER, and so on. The function that you use depends on the characteristics of the object type and whether you want XML or DDL.

For some object types, you can use multiple functions. You can use GET\_XXX to fetch an index by name, or GET\_DEPENDENT\_XXX to fetch the same index by specifying the table on which it is defined.

GET\_XXX returns a single object name.

For GET\_DEPENDENT\_XXX and GET\_GRANTED\_XXX, an arbitrary number of granted or dependent objects may match the input criteria. However, you can specify an object count when fetching these objects.

If you invoke these functions from SQL\*Plus, you should use the SET LONG and SET PAGESIZE commands to retrieve the complete, uninterrupted output.

```
SET LONG 2000000
SET PAGESIZE 300
```

## Browsing APIs: Examples

1. Get the XML representation of OE.ORDERS:

```
SELECT DBMS_METADATA.GET_XML
        ('TABLE', 'ORDERS', 'OE')
FROM    dual;
```

2. Fetch the DDL for all object grants on OE.ORDERS:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
        ('OBJECT_GRANT', 'ORDERS', 'OE')
FROM    dual;
```

3. Fetch the DDL for all system grants granted to OE:

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
        ('SYSTEM_GRANT', 'OE')
FROM    dual;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Browsing APIs: Examples

1. Results for fetching the XML representation of OE.ORDERS:

```
DBMS_METADATA.GET_XML('TABLE', 'ORDERS', 'OE')
```

```
-----
<?xml version="1.0"?>
<ROWSET>
  <ROW>
    <TABLE_T>
      <VERS_MAJOR>1</VERS_MAJOR>
```

2. Results for fetching the DDL for all object grants on OE.ORDERS:

```
DBMS_METADATA.GET_DEPENDENT_DDL
('OBJECT_GRANT', 'ORDERS', 'OE')
```

```
-----
GRANT SELECT ON "OE"."ORDERS" TO "PM"
GRANT SELECT ON "OE"."ORDERS" TO "BI"
```

3. Results for fetching the DDL for all system grants granted to OE:

```
DBMS_METADATA.GET_GRANTED_DDL('SYSTEM_GRANT', 'OE')
```

```
-----
GRANT QUERY REWRITE TO "OE"
```

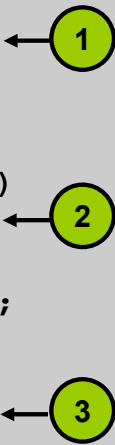
...



# Browsing APIs: Examples

```
BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM,
    'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
FROM   user_all_tables u
WHERE  u.nested = 'NO'
AND    (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT');
END;
/
```



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Browsing APIs: Examples (continued)

The example in the slide shows how to fetch creation DDL for all “complete” tables in the current schema, filtering out nested tables and overflow segments. The steps shown in the slide are as follows:

1. The `SET_TRANSFORM_PARAM` function specifies that the storage clauses are not to be returned in the SQL DDL. The `SESSION_TRANSFORM` function is interpreted to mean “for the current session.”
2. Use the `GET_DDL` function to retrieve DDL on all nonnested and non-IOT (index-organized table) tables.

```
CREATE TABLE "HR"."COUNTRIES"
( "COUNTRY_ID" CHAR(2)
  CONSTRAINT "COUNTRY_ID_NN" NOT NULL ENABLE,
  "COUNTRY_NAME" VARCHAR2(40),
  "REGION_ID" NUMBER,
  CONSTRAINT "COUNTRY_C_ID_PK"
  PRIMARY KEY ("COUNTRY_ID") ENABLE,
  CONSTRAINT "COUNTR_REG_FK" FOREIGN KEY
...

```

3. Reset the session-level parameters to their defaults.

## Quiz

Which one of the following SQL Developer predefined reports would you use to find the occurrence of a text string or an object name within a PL/SQL coding?

- a. Search Source Code
- b. Program Unit Arguments
- c. Unit Line Counts

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## Quiz

To find information about arguments for procedures and functions, use which of the following?

- a. The `ALL_ARGUMENTS` view
- b. The `DBMS_DESCRIBE.DESCRIBE_PROCEDURE` routine
- c. SQL Developer predefined report, Program Unit Arguments
- d. None of the above

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: b, c**

## Quiz

You can invoke `DBMS_METADATA` to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

- a. True
- b. False

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## Summary

In this lesson, you should have learned how to:

- Use the supplied packages and the dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Summary

This lesson showed you how to use the dictionary views and supplied PL/SQL packages to analyze your PL/SQL applications.

## Practice 09: Overview

This practice covers the following topics:

- Analyzing PL/SQL code
- Using PL/Scope
- Using DBMS\_METADATA

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Practice 09: Overview

Using the OE application that you have created, write code to analyze your application. You will perform the following:

- Find coding information
- Use PL/Scope
- Use DBMS\_METADATA

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

# 10

## Profiling and Tracing PL/SQL Code

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Trace PL/SQL program execution
- Profile PL/SQL applications

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications. You are introduced to tracing PL/SQL code and profiling PL/SQL code.



# Lesson Agenda

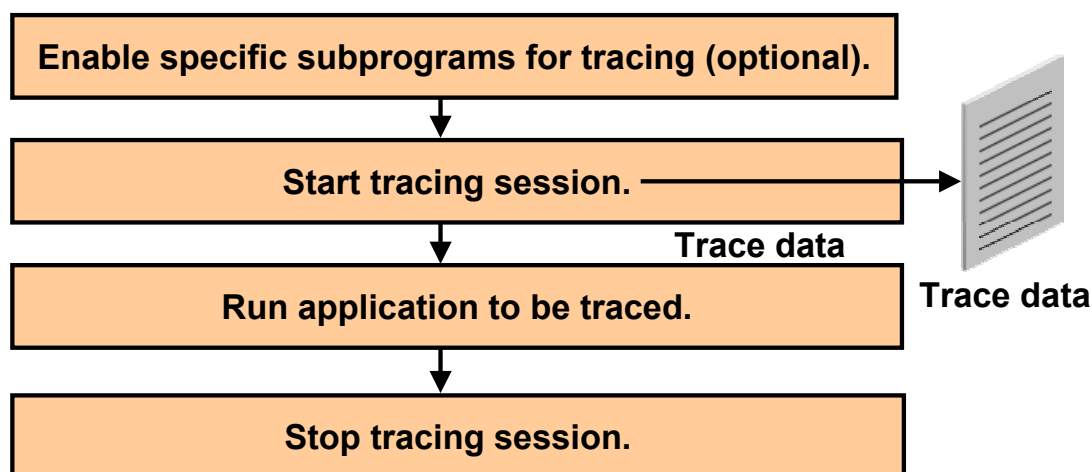
- Tracing PL/SQL program execution
- Profiling PL/SQL applications

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# Tracing PL/SQL Execution

Tracing PL/SQL execution provides you with a better understanding of the program execution path, and is possible by using the `dbms_trace` package.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Tracing PL/SQL Execution

In large and complex PL/SQL applications, it can sometimes become difficult to keep track of subprogram calls when a number of them call each other. By tracing your PL/SQL code, you can get a clearer idea of the paths and order in which your programs execute.

Though a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the `dbms_trace` package, to trace PL/SQL subprogram code.

**Note:** You cannot use PL/SQL tracing with the multithreaded server (MTS).

# Tracing PL/SQL Execution

The `dbms_trace` package contains:

- `set_plsql_trace (trace_level INTEGER)`
- `clear_plsql_trace`
- `plsql_trace_version`

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## The `dbms_trace` Programs

`dbms_trace` provides subprograms to start and stop PL/SQL tracing in a session. The trace data is collected as the program executes, and it is written out to data dictionary tables.

Procedure	Description
<code>set_plsql_trace</code>	Starts tracing data dumping in a session (You provide the trace level at which you want your PL/SQL code traced as an IN parameter.)
<code>clear_plsql_trace</code>	Stops trace data dumping in a session
<code>plsql_trace_version</code>	Returns the version number of the trace package as an OUT parameter

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session (`dbms_trace.set_plsql_trace`)
- Running the application that is to be traced
- Stopping the PL/SQL tracing session (`dbms_trace.clear_plsql_trace`)

# Tracing PL/SQL Execution

- Using `set_plsql_trace`, select a trace level to identify how to trace calls, exceptions, SQL, and lines of code.
- Trace-level constants:
  - `trace_all_calls`
  - `trace_enabled_calls`
  - `trace_all_sql`
  - `trace_enabled_sql`
  - `trace_all_exceptions`
  - `trace_enabled_exceptions`
  - `trace_enabled_lines`
  - `trace_all_lines`
  - `trace_stop`
  - `trace_pause`
  - `trace_resume`

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Specifying a Trace Level

During the trace session, there are two levels that you can specify to trace calls, exceptions, SQL, and lines of code.

### Trace Calls

- **Level 1:** Trace all calls. This corresponds to the constant `trace_all_calls`.
- **Level 2:** Trace calls only to enabled program units. This corresponds to the constant `trace_enabled_calls`.

### Trace Exceptions

- **Level 1:** Trace all exceptions. This corresponds to `trace_all_exceptions`.
- **Level 2:** Trace exceptions raised only in enabled program units. This corresponds to `trace_enabled_exceptions`.

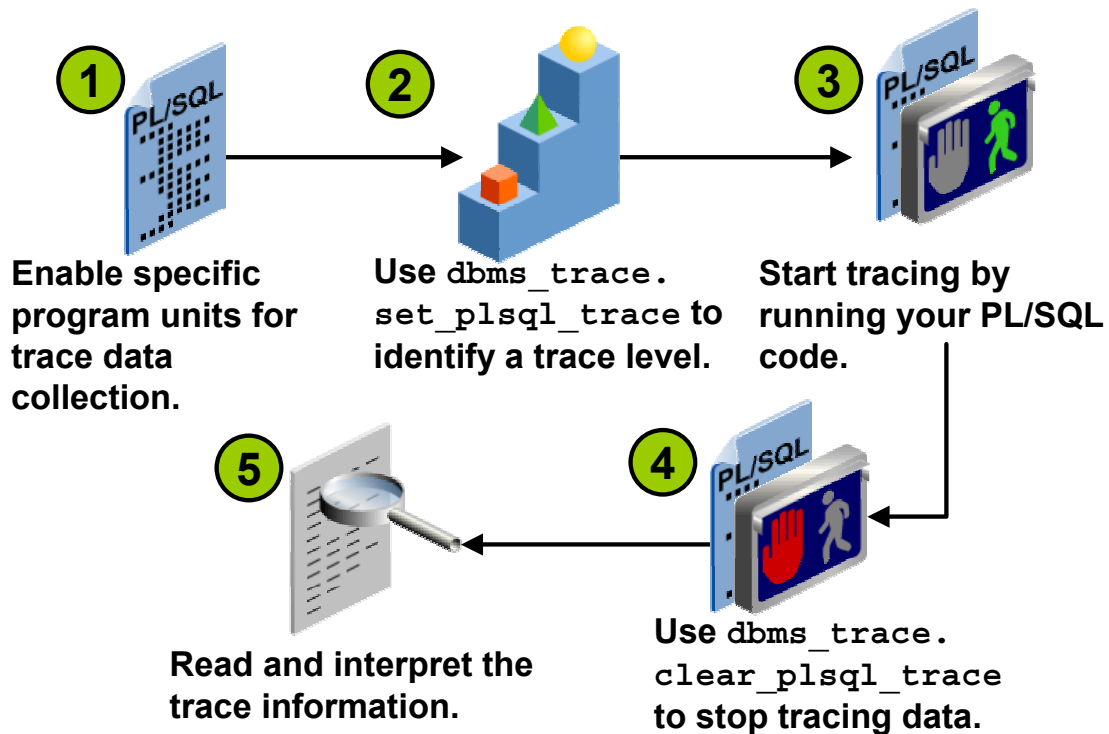
### Trace SQL

- **Level 1:** Trace all SQL. This corresponds to the constant `trace_all_sql`.
- **Level 2:** Trace SQL in only enabled program units. This corresponds to the constant `trace_enabled_sql`.

### Trace Lines

- **Level 1:** Trace all lines. This corresponds to the constant `trace_all_lines`.
- **Level 2:** Trace lines only in enabled program units. This corresponds to the constant `trace_enabled_lines`.

## Tracing PL/SQL: Steps



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Steps to Trace PL/SQL Code

There are five steps to trace PL/SQL code by using the `dbms_trace` package:

1. Enable specific program units for trace data collection.
2. Use `dbms_trace.set_plsql_trace` to identify a trace level.
3. Run your PL/SQL code.
4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

The next few pages demonstrate the steps to accomplish PL/SQL tracing.

## Step 1: Enable Specific Subprograms

Enable specific subprograms with one of the two methods:

- Enable a subprogram by compiling it with the debug option:

```
ALTER SESSION SET PLSQL_DEBUG=true;
```

```
CREATE OR REPLACE ....
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]  
subprogram-name COMPILE DEBUG [BODY];
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Step 1: Enable Specific Subprograms

Profiling large applications may produce a huge volume of data that can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling a specific subprogram for trace data collection. You can enable a subprogram by compiling it with the debug option. You can do this in one of two ways:

- Enable a subprogram by compiling it with the ALTER SESSION debug option, and then compile the program unit by using the CREATE OR REPLACE syntax:

```
ALTER SESSION SET PLSQL_DEBUG = true;  
CREATE OR REPLACE ...
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]  
    <subprogram-name> COMPILE DEBUG [BODY];  
ALTER PROCEDURE P5 COMPILE DEBUG;
```

**Note:** The second method cannot be used for anonymous blocks.

Enabling specific subprograms enables you to:

- Limit and control the amount of trace data, especially in large applications.
- Obtain additional trace information that is otherwise not available. For example, during the tracing session, if a subprogram calls another subprogram, the name of the called subprogram is included in the trace data if the calling subprogram was enabled by compiling it in debug mode.

## Steps 2 and 3: Identify a Trace Level and Start Tracing

- Specify the trace level by using `dbms_trace.set_plsql_trace`:

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE -  
        (tracelevel1 + tracelevel2 ...)
```

- Execute the code that is to be traced:

```
EXECUTE my_program
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Steps 2 and 3: Specify a Trace Level and Start Tracing

To trace PL/SQL code execution by using `dbms_trace`, perform these steps:

- Start the trace session by using the syntax shown in the slide. For example:  

```
EXECUTE -  
        DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```
- Execute the PL/SQL code. The trace data is written to the data dictionary views.

**Note:** To specify additional trace levels in the argument, use the “+” symbol between each trace level value.

## Step 4: Turn Off Tracing

Remember to turn tracing off by using the `dbms_trace.clear_plsql_trace` procedure:

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Step 4: Turn Off Tracing

After tracing the PL/SQL program unit, turn tracing off by executing `dbms_trace.clear_plsql_trace`. This stops any further writing to the trace tables.

To avoid the overhead of writing the trace information, it is recommended that you turn tracing off when you are not using it.



## Step 5: Examine the Trace Information

Examine the trace information:

- Call tracing writes out the program unit type, name, and stack depth.
- Exception tracing writes out the line number.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Step 5: Examine the Trace Information

- Lower trace levels supersede higher levels when tracing is activated for multiple tracing levels.
- If tracing is requested only for enabled subprograms and if the current subprogram is not enabled, no trace data is written.
- If the current subprogram is enabled, call tracing writes out the subprogram type, name, and stack depth.
- If the current subprogram is not enabled, call tracing writes out the subprogram type, line number, and stack depth.
- Exception tracing writes out the line number. Raising the exception shows information about whether the exception is user-defined or predefined and, in the case of predefined exceptions, the exception number.

**Note:** An enabled subprogram is compiled with the debug option.

## `plsql_trace_runs` and `plsql_trace_events`

- Trace information is written to the following dictionary views:
  - `plsql_trace_runs` dictionary view
  - `plsql_trace_events` dictionary view
- Run the `tracetab.sql` script to create the dictionary views.
- You need privileges to view the trace information in the dictionary views.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### The `plsql_trace_runs` and `plsql_trace_events` Dictionary Views

All trace information is written to the dictionary views `plsql_trace_runs` and `plsql_trace_events`. These views are created (typically by a DBA) by running the `tracetab.sql` script. The script is located in the `/u01/app/oracle/product/11.2.0/dbhome_1/rdbms/admin` folder. Run the script as SYS. After the script is run, you need the `SELECT` privilege to view information from these dictionary views.

```
--Execute as sys
@/u01/app/oracle/product/11.2.0/dbhome_1/rdbms/admin/tracetab
GRANT SELECT ON plsql_trace_runs TO OE;
GRANT SELECT ON plsql_trace_events TO OE;
```

## plsql\_trace\_runs and plsql\_trace\_events

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
ALTER PROCEDURE P5 COMPILE DEBUG;

EXECUTE DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
EXECUTE p5
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE

SELECT proc_name, proc_line,
       event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_proc_name = 'P5'
OR PROC_NAME = 'P5';
```

PROC_NAME	PROC_LINE	EVENT_PROC_NAME	EVENT_COMMENT
P5	1		Procedure Call
P4	1	P5	Procedure Call

2 rows selected.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Query the plsql\_trace\_runs and plsql\_trace\_events Views

Use the dictionary views `plsql_trace_runs` and `plsql_trace_events` to view the trace information generated by using the `dbms_trace` facility. `plsql_trace_runs` holds generic information about traced programs, such as the date, time, owner, and name of the traced stored program. `dbms_trace_events` holds more specific information about the traced subprograms.

# Lesson Agenda

- Tracing PL/SQL program execution
- Profiling PL/SQL applications

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# Hierarchical Profiling Concepts

- **Definition:**
  - Used to identify hotspots and performance tuning opportunities in PL/SQL applications
  - Reports the dynamic execution profile of a PL/SQL program organized by function calls
  - Reports SQL and PL/SQL execution times separately
  - Provides function level summaries
- **Benefits:**
  - Provides more information than a flat profiler
  - Can be used to understand the structure and control flow of complex programs

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Hierarchical Profiling Concepts

Starting in Oracle Database 11g, the hierarchical profiler is available to help you identify the hotspots and performance tuning opportunities in your PL/SQL applications. This feature enables you to view reports of how a PL/SQL program is executed, organized by function call, as well as SQL and PL/SQL execution times.

You can view function level summaries that include:

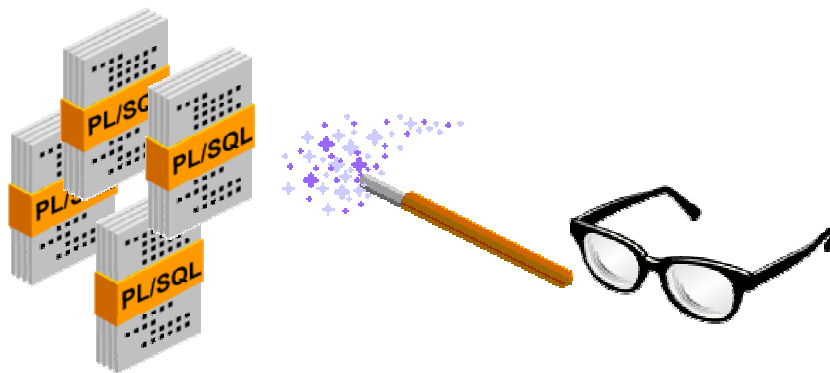
- Number of calls to a function
- Time spent in the function itself
- Time spent in the entire subtree under the function
- Detailed parent-children information for each function

You can use this information to tune your PL/SQL applications, and understand the structure, flow, and control of complex programs (especially those written by someone else).

# Hierarchical Profiling Concepts

The PL/SQL hierarchical profiler consists of the:

- Data collection component
- Analyzer component



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Hierarchical Profiling Concepts (continued)

The PL/SQL hierarchical profiler consists of two subcomponents:

- The data collection component is an intrinsic part of the PL/SQL Virtual Machine. The new DBMS\_HPROF package provides APIs to turn hierarchical profiling on and off. The raw profiler output is written to a file.
- The analyzer component, which is also exposed by the DBMS\_HPROF package, is used to process the raw profiler output and upload the results of the profiling into database tables that can then be queried. You can use the `plshprof` command-line utility to generate simple HTML reports directly from the raw profiler data.

## Using the PL/SQL Profiler

By using the PL/SQL profiler, you can find:

- The number of calls to a function
- The function time, not including descendants
- The subtree time, including descendants
- Parent-children information for each function
  - Who were the callers of a given function?
  - What functions were called from a particular function?
  - How much time was spent in function X when called from function Y?
  - How many calls to function X came from function Y?
  - How many times did X call Y?

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the PL/SQL Profiler

By using the hierarchical PL/SQL profiler, you can find both function-level execution summary information, as well as detailed parent-children information for each function.

When profiling is turned on, function entry and exit operations are logged to a file along with time information. Fine-grained elapsed information is collected.

You do not need to recompile PL/SQL modules to use the hierarchical profiler. You can analyze both interpreted and natively compiled PL/SQL modules.

# Using the PL/SQL Profiler

Sample data for profiling:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      ...
  END update_card_info;
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      ...
  END display_card_info;
END credit_card_pkg; -- package body
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Sample Data

The CREDIT\_CARD\_PKG that was created earlier is used to demonstrate hierarchical profiling. The full code is shown on the following page.



## Sample Data (continued)

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
                DBMS_OUTPUT.PUT('Card Type: ' || v_card_info(idx).card_type
                                || ' ');
                DBMS_OUTPUT.PUT_LINE('/ Card No: ' || v_card_info(idx).card_num );
            END LOOP;
        ELSE
            DBMS_OUTPUT.PUT_LINE('Customer has no credit cards. ');
        END IF;
    END display_card_info;
END credit_card_pkg; -- package body
/
```

# Using the PL/SQL Profiler

```
BEGIN
-- start profiling
  DBMS_HPROF.START_PROFILING('PROFILE_DATA', 'pd_cc_pkg.txt');
END;
```

1

```
DECLARE
  v_card_info typ_cr_card_nst;
BEGIN
-- run application
  credit_card_pkg.update_card_info
    (154, 'Discover', '123456789');
END;
```

2

```
BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;
```

3

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Steps for Using the PL/SQL Profiler

You use the new DBMS\_HPROF package to hierarchically profile PL/SQL code. You need to be granted the privilege to execute the routines in the DBMS\_HPROF package:

```
GRANT EXECUTE ON sys.dbms_hprof TO OE;
```

You also need to identify the location of the profiler files. Create a DIRECTORY object to identify this information:

```
CREATE DIRECTORY profile_data AS '/home/oracle/labs/labs';
```

1. The first step is to turn profiling on. Use the DBMS\_HPROF.START\_PROFILING procedure.

When calling this procedure, pass these two parameters:

- Directory object: An alias for a file system path name. You need to have WRITE privileges to this location.
- File name: The name of the file to which you want the output written.
- You can optionally pass a third parameter, max\_depth. When max\_depth value is NULL (the default), profile information is gathered for all functions irrespective of their call depth. When a non-NULL value is specified, the profiler collects data only for functions up to a call depth level of max\_depth.

2. The second step is to run the code that you want profiled.

3. The third step is to turn off profiling. Use the DBMS\_HPROF.STOP\_PROFILING procedure to stop the profiling.

```
EXECUTE DBMS_HPROF.STOP_PROFILING
```

# Understanding Raw Profiler Data

```
P#! PL/SQL Timer Started
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
P#X 1634
P#C PLSQL."OE"."CREDIT_CARD_PKG"::11."UPDATE_CARD_INFO"#71749359b90ac246
#24
P#X 7
P#C PLSQL."OE"."CREDIT_CARD_PKG"::11."CUST_CARD_INFO"#c2ad85321cb9b0ae
#4
P#X 11
P#C SQL."OE"."CREDIT_CARD_PKG"::11."__static_sql_exec_line10" #10
P#X 1502
P#R
...
P#C PLSQL."".""."__plsql_vm"
P#X 3
P#C PLSQL."".""."__anonymous_block"
P#X 15
P#C PLSQL."SYS"."DBMS_HPROF"::11."STOP_PROFILING"#980980e97e42f8ec #53
P#R
P#R
P#! PL/SQL Timer Stopped
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Understanding Raw Profiler Data

You can examine the contents of the generated raw profiler text file. However, it is easier to understand the data after using the analyzer component of hierarchical profiling. The text shown in the slide displays the contents of the `pd_cc_pkg.txt` file. Each line starts with a file indicator.

These are the meanings:

- P#V – PLSHPROF banner with version number
- P#C – Call to a subprogram (call event)
- P#R – Return from a subprogram (call event)
- P#X – Elapsed time between preceding and following events
- P#! – Comment

As you will see next, the `DBMS_HPROF.ANALYZE` function generates easier-to-decipher data and saves the data in tables.

## Using the Hierarchical Profiler Tables

- Upload the raw profiler data into the database tables.
- Run the `dbmshptab.sql` script that is located in the `ORACLE_HOME/rdbms/admin` folder to set up the profiler tables.

```
-- run this only once per schema
-- under the schema where you want the profiler tables located
@u01/app/oracle/product/11.2.0/dbhome_1/rdbms/admin/dbmshptab.sql
```

- Creates these tables:

Table	Description
DBMSHP_RUNS	Contains top-level information for each run command
DBMSHP_FUNCTION_INFO	Contains information on each function profiled
DBMSHP_PARENT_CHILD_INFO	Contains parent-child profiler information

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### The Hierarchical Profiler Tables

Before uploading the profiler data into the database tables, you need to create the hierarchical profiler database tables. Run the script named `dbmshptab.sql` under the schema where you want the profiling tables to create the hierarchical profiling tables. This script is located in your `/home/rdbms/admin/` folder.

The script creates three tables and other data structures that are required for persistently storing the profiler data.

**Note:** Running the script a second time drops any previously created hierarchical profiler tables.

## Using DBMS\_HPROF.ANALYZE

DBMS\_HPROF.ANALYZE:

- Analyzes the raw profiler data
- Generates hierarchical profiler information in the profiler database tables
- Definition:

```
DBMS_HPROF.ANALYZE(  
  location      IN VARCHAR2,  
  filename      IN VARCHAR2,  
  summary_mode  IN BOOLEAN      DEFAULT FALSE,  
  trace         IN VARCHAR2     DEFAULT NULL,  
  skip          IN PLS_INTEGER  DEFAULT 0,  
  collect       IN PLS_INTEGER  DEFAULT NULL,  
  run_comment   IN VARCHAR2     DEFAULT NULL)  
RETURN NUMBER;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using DBMS\_HPROF.ANALYZE

Use the DBMS\_HPROF.ANALYZE function to analyze the raw profiler output and produce the hierarchical profiler information in the database.

This function accepts the following parameters:

- Location: The name of the directory object that identifies the location from which to read.
- Filename: The file name of the raw profiler data to be analyzed.
- summary\_mode: A Boolean that identifies whether to generate only top-level summary information into the database tables (TRUE) or to provide detailed analysis (FALSE). The default is false.
- Trace: Identifies whether to analyze only the subtrees rooted at the specified trace entry or perform the analysis for the entire run. This parameter is specified in a special, qualified format within quotation marks. It includes the schema name, module name, and function name. For example "SCOTT"."PKG"."FOO".
- Skip: Analyzes only the subtrees rooted at the specified trace, but ignores the first "skip" invocations to trace. The default value for "skip" is 0. Used only when trace is specified.
- Collect: Analyzes "collect" member of invocations of traces (starting from "skip" + 1). By default, only one invocation is collected. Used only when trace is specified.
- run\_comment: A comment for your run.

## Using DBMS\_HPROF.ANALYZE to Write to Hierarchical Profiler Tables

- Use the DBMS\_HPROF.ANALYZE function to upload the raw profiler results into the database tables:

```
DECLARE
    v_runid NUMBER;
BEGIN
    v_runid := DBMS_HPROF.ANALYZE (LOCATION => 'PROFILE_DATA',
                                   FILENAME => 'pd_cc_pkg.txt');
    DBMS_OUTPUT.PUT_LINE('Run ID: ' || v_runid);
END;
```

- This function returns a unique run identifier for the run. You can use this identifier to look up results corresponding to this run from the hierarchical profiler tables.

```
RUN_ID
-----
      1
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.




### Using DBMS\_HPROF.ANALYZE (continued)

In the example shown in the slide, the DBMS\_HPROF.ANALYZE function is used to take the raw data from the text file named `pd_cc_pkg.txt` and place the resulting analyzed data in the profiler tables. The function returns a unique identifier for the run.

## Analyzer Output from the DBMSHP\_RUNS Table

Query the DBMSHP\_RUNS table to find top-level information for each run:

```
SELECT runid, run_timestamp, total_elapsed_time  
FROM dbmshp_runs  
WHERE runid = 2;
```

 RUNID	 RUN_TIMESTAMP	 TOTAL_ELAPSED_TIME
2	10-DEC-09 02.10.47.604825000 AM	120758

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Query the DBMSHP\_RUNS Table

The DBMS\_HPROF.ANALYZE PL/SQL API is used to analyze the raw profiler output and upload the results into the database tables. This enables you to query the data for custom report generation using tools, such as SQL Developer or other third-party tools.

In the example shown in the slide, the DBMSHP\_RUNS table is queried after the DBMS\_HPROF.ANALYZE is run. This table contains top-level information for each run of the DBMS\_HPROF.ANALYZE command. From the output, you can see the run ID, the timestamp of the run, and the total elapsed time in milliseconds. If you provided a comment for the run, you can retrieve that information from the RUN\_COMMENT column (not shown).

## Analyzer Output from the DBMSHP\_FUNCTION\_INFO Table

Query the DBMSHP\_FUNCTION\_INFO table to find information about each function profiled:

```
SELECT owner, module, type, function_line#, namespace,
       calls, function_elapsed_time
FROM   dbmshp_function_info
WHERE  runid = 1;
```

OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	FUNCTION_ELAPSED
(null)	(null)	(null)	__anonymous_block	PLSQL	3	
(null)	(null)	(null)	__pls_sql_vm	PLSQL	3	
OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	1	
SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1	
(null)	(null)	(null)	__dyn_sql_exec_line5	SQL	1	
OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line21	SQL	1	
OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line9	SQL	1	

**ORACLE**

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Query the DBMSHP\_FUNCTION\_INFO Table

You can query the DBMSHP\_FUNCTION\_INFO table to find summary information for each function profiled in a run of the analyzer.

In the query shown above, the following information is retrieved:

- **OWNER:** Module owner of the function
- **MODULE:** The module name
- **TYPE:** Module type, such as package, package body, or procedure
- **LINE#:** The line number in the module at which the function is defined. This line number helps identify the source location of the function in the module and can be used by the integrated development environment (IDE) tools to navigate to the appropriate location in the source where the function is defined. The line number can also be used to distinguish between overloaded routines.
- **NAMESPACE:** The language information. At this time, SQL and PL/SQL are supported.
- **CALLS:** The number of calls to the function
- **FUNCTION\_ELAPSED\_TIME:** The time in microseconds, not including the time spent in descendant functions



## plshprof: A Simple HTML Report Generator

- `plshprof` is a command-line utility.
- You can use it to generate simple HTML reports directly from the raw profiler data.
- The HTML reports can be browsed in any browser.
- The navigational capabilities combined with the links provide a means for you to analyze the performance of large applications.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Introducing `plshprof`

`plshprof` is a command-line utility that you can use to generate HTML reports based on the raw profiler data generated by the data collection component after running the `DBMS_HPROF.ANALYZE` PL/SQL API.

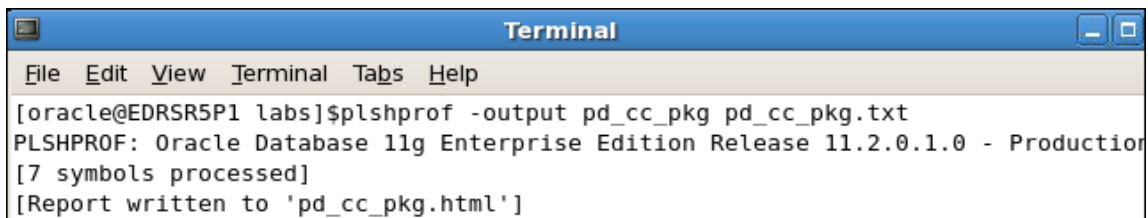
## Using plshprof

After generating the raw profiler output file:

1. Change to the directory where you want the HTML output placed.
2. Run `plshprof`.
  - Syntax:

```
plshprof [option...] output_filename_1 output_filename_2
```

- Example:



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR5P1 labs]$plshprof -output pd_cc_pkg pd_cc_pkg.txt
PLSHPROF: Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
[7 symbols processed]
[Report written to 'pd_cc_pkg.html']
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using plshprof

After you have the raw profiler output file, you can generate HTML reports for that run.

The `plshprof` utility has these options:

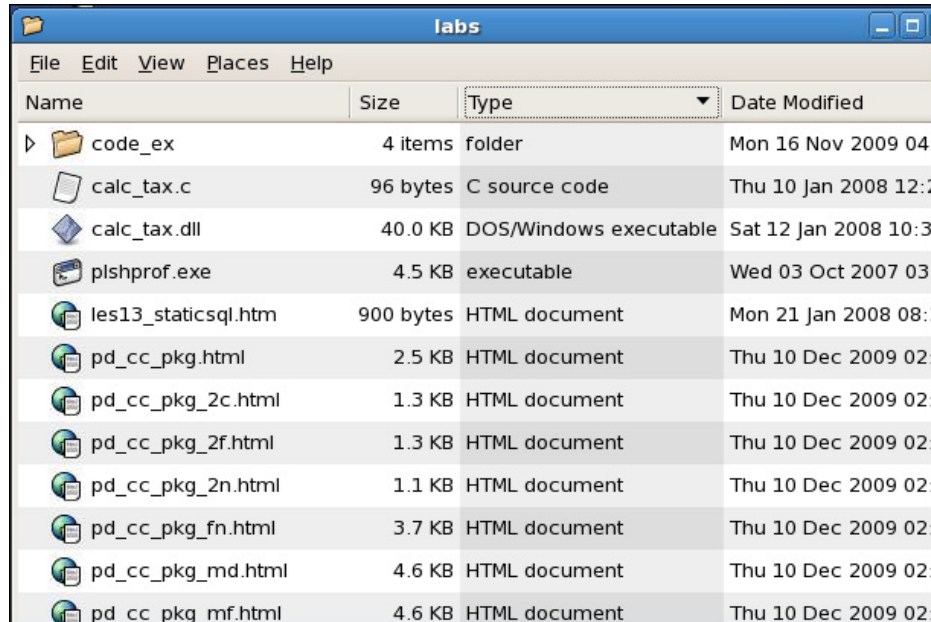
- `-skip count`: Skips the first count calls. Use only with the `-trace` symbol. The default is 0.
- `-collect count`: Collects information for count calls. Use only with `-trace` symbol. The default is 1.
- `-output filename`: Specifies the name of the output file `symbol.html` or `tracefile1.html`
- `-summary`: Prints only elapsed time. No default.
- `-trace symbol`: Specifies the function name of the tree root

Follow these steps to generate the HTML reports:

1. Change to the directory of the raw profiler output file. In the example shown in the slide, the raw profiler file is in the `/home/oracle/labs/labs` folder. Its name is `pd_dd_pkg.txt`.
2. Run the `plshprof` utility. This utility accepts line arguments. The slide's example identifies that the output file should be named `pd_cc_pkg.html` and the original raw profiler file is named `pd_dd_pkg.txt`.

# Using plshprof

Resulting generated files:



Name	Size	Type	Date Modified
code_ex	4 items	folder	Mon 16 Nov 2009 04
calc_tax.c	96 bytes	C source code	Thu 10 Jan 2008 12:2
calc_tax.dll	40.0 KB	DOS/Windows executable	Sat 12 Jan 2008 10:3
plshprof.exe	4.5 KB	executable	Wed 03 Oct 2007 03
les13_staticsql.htm	900 bytes	HTML document	Mon 21 Jan 2008 08:
pd_cc_pkg.html	2.5 KB	HTML document	Thu 10 Dec 2009 02:
pd_cc_pkg_2c.html	1.3 KB	HTML document	Thu 10 Dec 2009 02:
pd_cc_pkg_2f.html	1.3 KB	HTML document	Thu 10 Dec 2009 02:
pd_cc_pkg_2n.html	1.1 KB	HTML document	Thu 10 Dec 2009 02:
pd_cc_pkg_fn.html	3.7 KB	HTML document	Thu 10 Dec 2009 02:
pd_cc_pkg_md.html	4.6 KB	HTML document	Thu 10 Dec 2009 02:
pd_cc_pkg_mf.html	4.6 KB	HTML document	Thu 10 Dec 2009 02:

ORACLE

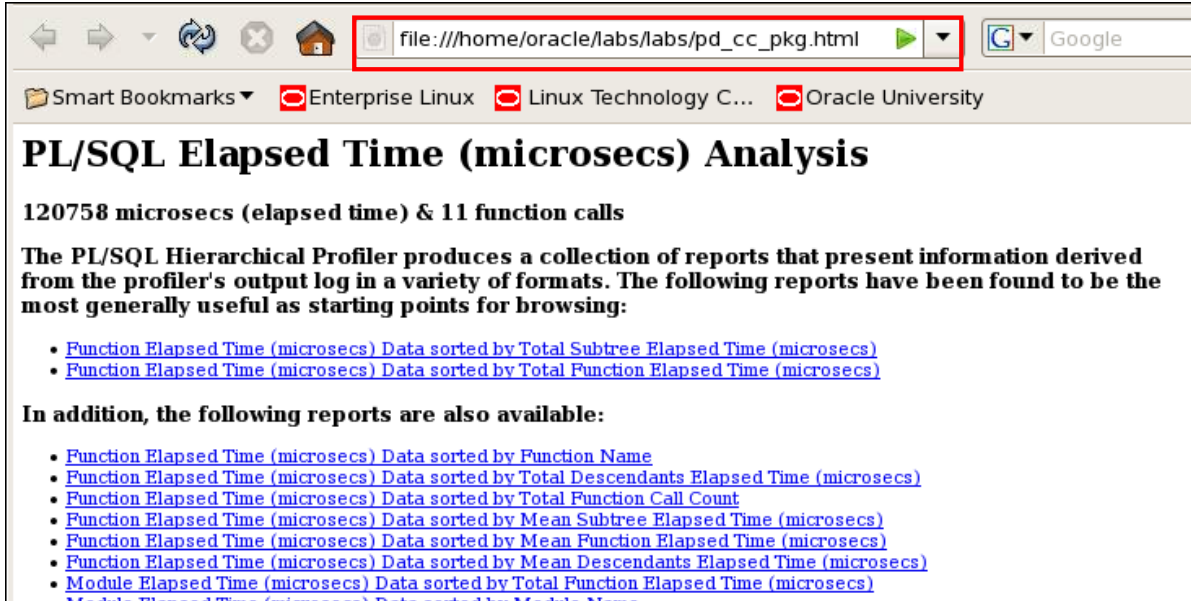
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using plshprof (continued)

Running the plshprof utility generates a set of HTML files. The *filename*.html is the root page where you start browsing. The other files with the same file name prefix are hyperlinks from the base *filename*.html file. In the slide's example, the root page is named pd\_cc\_pkg.html.

# Using plshprof

3. Open the filename.html in a browser:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using plshprof (continued)

3. You can open the start page file in any browser. In this example, the pd\_cc\_pkg.html file is opened. This is an example of a single-run report. It contains some overall summary information and hyperlinks to additional information.

# Using the HTML Reports

useful as starting points for browsing:

- [Function Elapsed Time \(microsecs\) Data sorted by Total Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)

## Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)

120758 microsecs (elapsed time) & 11 function calls

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120758	100%	17	0.0%	120741	100%	3	27.3%	<a href="#">plsql_vm</a>
120741	100%	1323	1.1%	119418	98.9%	3	27.3%	<a href="#">anonymous_block</a>
119292	98.8%	170	0.1%	119122	98.6%	1	9.1%	<a href="#">OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)</a>
92954	77.0%	92954	77.0%	0	0.0%	1	9.1%	<a href="#">OE.CREDIT_CARD_PKG._static_sql_exec_line9 (Line 9)</a>
26168	21.7%	26168	21.7%	0	0.0%	1	9.1%	<a href="#">OE.CREDIT_CARD_PKG._static_sql_exec_line21 (Line 21)</a>
126	0.1%	126	0.1%	0	0.0%	1	9.1%	<a href="#">dyn_sql_exec_line5 (Line 5)</a>
0	0.0%	0	0.0%	0	0.0%	1	9.1%	<a href="#">SYS.DBMS_HPROF.STOP_PROFILING (Line 59)</a>

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Viewing the Reports

The function-level summary reports provide you with a flat view of the profile information. The reports list the total number of calls, self time, subtree time, and descendants for each function. Each report is sorted on a particular attribute.

Note that in the slide's example, each function name is hyperlinked to its corresponding parents and children report. The column that is in bold identifies how the report is sorted.

If a subprogram is nested, the profiler reports display the fully qualified name, such as OE.P.A.B; that is, procedure B is nested within procedure A of package OE.P.

# Using the HTML Reports

useful as starting points for browsing:

- [Function Elapsed Time \(microsecs\) Data sorted by Total Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)

## Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

120758 microsecs (elapsed time) & 11 function calls

Subtree	Ind%	Function	Ind%	Cum%	Descendants	Ind%	Calls	Ind%	Function Name
92954	77.0%	92954	77.0%	77.0%	0	0.0%	1	9.1%	<a href="#">OE.CREDIT_CARD_PKG._static_sql_exec_line9 (Line 9)</a>
26168	21.7%	26168	21.7%	98.6%	0	0.0%	1	9.1%	<a href="#">OE.CREDIT_CARD_PKG._static_sql_exec_line21 (Line 21)</a>
120741	100%	1323	1.1%	99.7%	119418	98.9%	3	27.3%	<a href="#">_anonymous_block</a>
119292	98.8%	170	0.1%	99.9%	119122	98.6%	1	9.1%	<a href="#">OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)</a>
126	0.1%	126	0.1%	100%	0	0.0%	1	9.1%	<a href="#">_dyn_sql_exec_line5 (Line 5)</a>
120758	100%	17	0.0%	100%	120741	100%	3	27.3%	<a href="#">_plssql_vm</a>
0	0.0%	0	0.0%	100%	0	0.0%	1	9.1%	<a href="#">SYS.DBMS_HPROF.STOP_PROFILING (Line 59)</a>

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Viewing the Reports (continued)

You can use the module level summary report to find information about each PL/SQL module, including the total time spent in the module, the total number of calls to functions in the module, and the total number of calls to functions in the module. The total time spent in a module is calculated by adding the self times for all functions in that module.

# Using the HTML Reports

- [Namespace Elapsed Time \(microsecs\) Data sorted by Namespace](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Parents and Children Elapsed Time \(microsecs\) Data](#)

## Namespace Elapsed Time (microsecs) Data sorted by Namespace

120758 microsecs (elapsed time) & 11 function calls

Function	Ind%	Calls	Ind%	Namespace
1510	1.3%	8	72.7%	PLSQL
119248	98.7%	3	27.3%	SQL

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Viewing the Reports (continued)

Each function tracked by the profiler is associated with a namespace. PL/SQL functions, procedures, triggers, and packages fall under the category “PLSQL” namespace. Operations corresponding to SQL statement execution from PL/SQL, such as UPDATE, SELECT, FETCH, and EXECUTE IMMEDIATE, fall under the “SQL” namespace. The namespace level summary report provides information about the total time spent in that namespace and the total number of calls to functions in that namespace.

# Using the HTML Reports

- [Namespace Elapsed Time \(microsecs\) Data sorted by Namespace](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Parents and Children Elapsed Time \(microsecs\) Data](#)

**\_\_anonymous\_block**

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120741	100%	1323	1.1%	119418	98.9%	3	27.3%	<a href="#">__anonymous_block</a>
Parents:								
120741	100%	1323	100%	119418	100%	3	100%	<a href="#">__plsql_vm</a>
Children:								
119292	99.9%	170	100%	119122	100%	1	100%	<a href="#">OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)</a>
126	0.1%	126	100%	0	N/A	1	100%	<a href="#">__dyn_sql_exec_line5 (Line 5)</a>
0	0.0%	0	N/A	0	N/A	1	100%	<a href="#">SYS.DBMS_HPROF.STOP_PROFILING (Line 59)</a>

**\_\_plsql\_vm**

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120758	100%	17	0.0%	120741	100%	3	27.3%	<a href="#">__plsql_vm</a>
Parents:								
120758	100%	17	100%	120741	100%	3	100%	<a href="#">ORACLE.root</a>
Children:								
120741	100%	1323	100%	119418	100%	3	100%	<a href="#">__anonymous_block</a>

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Viewing the Reports (continued)

For each function tracked by the profiler, the parents and children report provides information about what functions call it (parents) and what functions it calls (children). It gives you the function's execution profile on a per-parent basis. It also provides the execution profile for each of the function's children when called from this function.



## Quiz

Select the correct order of the five steps to trace PL/SQL code using the `dbms_trace` package:

- A. Enable specific program units for trace data collection.
- B. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
- C. Run your PL/SQL code.
- D. Read and interpret the trace information.
- E. Use `dbms_trace.set_plsql_trace` to identify a trace level.

- a. A, E, C, B, D
- b. A, B, C, D, E
- c. A, C, E, B, D
- d. A, E, C, D, B

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## Quiz

You can view the hierarchical profiler function level summaries that include:

- a. Number of calls to a function
- b. Time spent in the function itself
- c. Time spent in the entire subtree under the function
- d. Detailed parent-children information for each function

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a, b, c, d**

## Quiz

Use the \_\_\_\_\_ to find information about each function profiled by the PL/SQL profiler:

- a. DBMS\_HPROF.ANALYZE table
- b. DBMSHP\_RUNS table
- c. DBMSHP\_FUNCTION\_INFO table
- d. plshprof command line utility

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a, d**

## Summary

In this lesson, you should have learned how to:

- Trace PL/SQL program execution
- Profile PL/SQL applications

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Summary

This lesson showed you how to trace PL/SQL program execution by using the DBMS\_TRACE package, how to profile your PL/SQL application, and, with profiling, how to set up the profiler tables, generate raw data, and use DBMS\_HPROF to analyze raw data in the profiler tables.

## Practice 10: Overview

This practice covers hierarchical profiling of PL/SQL code.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Practice 10: Overview

Using the OE application that you have created, write code to profile components in your application. Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”



# 11

## Implementing Fine-Grained Access Control for VPD

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Describe the process of fine-grained access control
- Implement and test fine-grained access control

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Objectives

In this lesson, you learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference*, *Oracle Label Security Administrator's Guide*, *Oracle Single Sign-On Application Developer's Guide*, and *Oracle Security Overview*.



# Lesson Agenda

- Describing the process of fine-grained access control
- Implementing and testing fine-grained access control

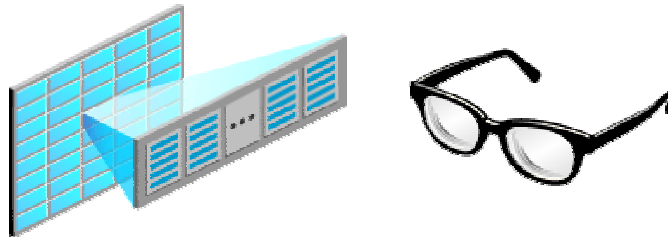
ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Fine-Grained Access Control: Overview

Fine-grained access control:

- Enables you to enforce security through a low level of granularity
- Restricts users to viewing only “their” information
- Is implemented through a security policy attached to tables
- Is implemented by highly privileged system DBAs, perhaps in coordination with developers
- Dynamically modifies user statements to fit the policy



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

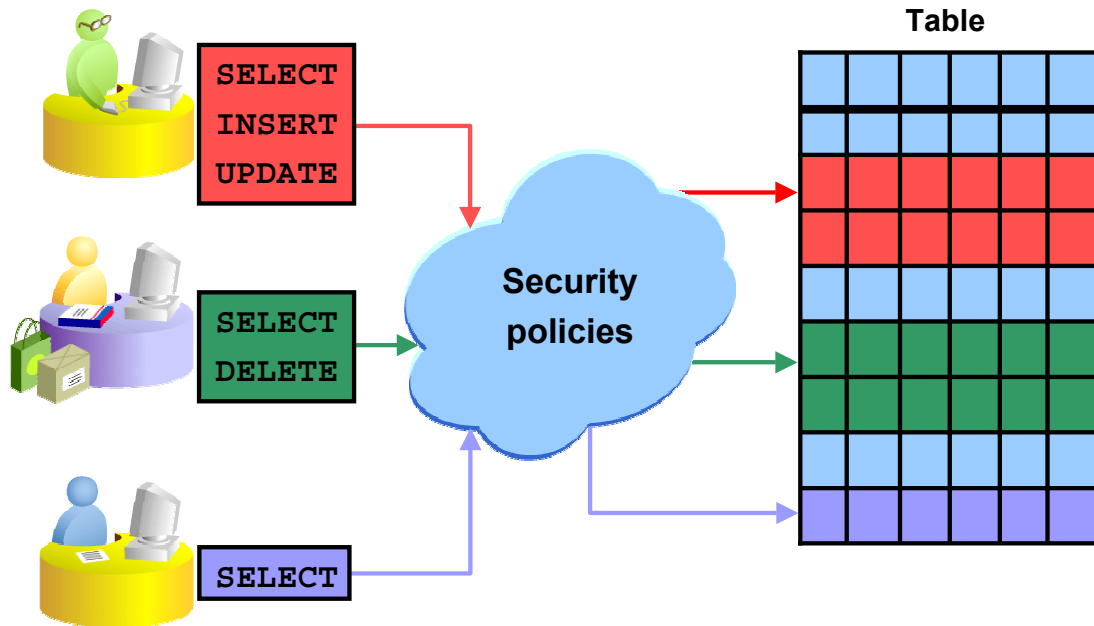
### Fine-Grained Access Control: Overview

Fine-grained access control enables you to build applications that enforce security rules (or policies) at a low level of granularity. For example, you can use it to restrict customers who access the Oracle server to see only their own account, physicians to see only the records of their own patients, or managers to see only the records of employees who work for them.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you based your application. When a user enters a data manipulation language (DML) statement on that object, the Oracle server dynamically modifies the user’s statement—transparently to the user—so that the statement implements the correct access control.

Fine-grained access is also known as a virtual private database (VPD), because it implements row-level security, essentially giving users access to their own private database. Fine-grained means at the individual row level.

## Identifying Fine-Grained Access Features



Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Features

You can use fine-grained access control to implement security rules, called “policies,” with functions, and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed.

A security policy is a collection of rules needed to enforce the appropriate privacy and security rules in the database itself, making it transparent to users of the data structure.

Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

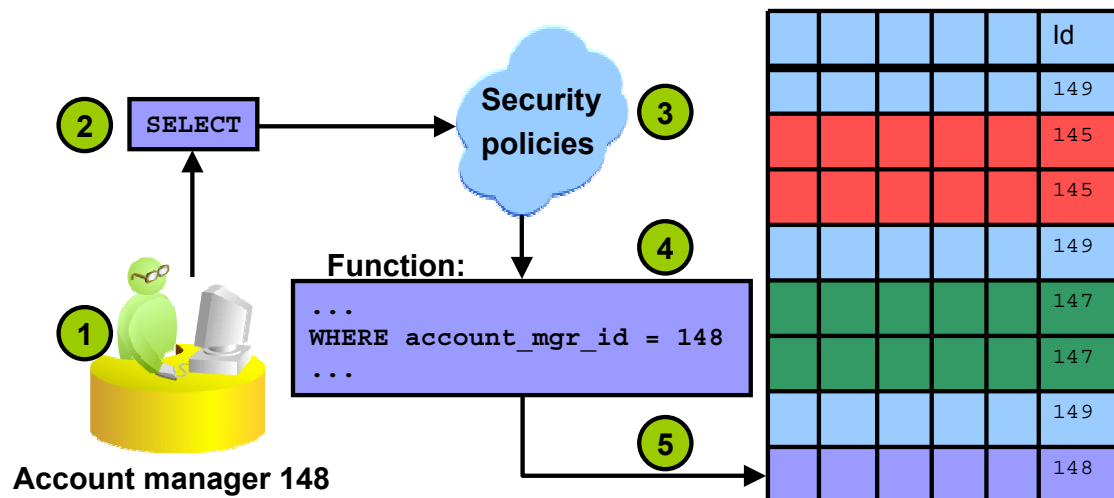
You can:

- Use different policies for SELECT, INSERT, UPDATE, and DELETE statements
- Use security policies only where you need them
- Use multiple policies for each table, including building on top of base policies in packaged applications
- Distinguish policies between different applications by using policy groups

## How Fine-Grained Access Works

Implement the policy on the CUSTOMERS table:

“Account managers can see only their own customers.”



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### How Fine-Grained Access Works

To implement a virtual private database so that account managers can see only their own customers, you must do the following:

1. Create a function to add a WHERE clause identifying a selection criterion to a user's SQL statement.
2. Have the user (the account manager) enter a SQL statement.
3. Implement the security policy through the function that you created. The Oracle server calls the function automatically.
4. Dynamically modify the user's statement through the function.
5. Execute the dynamically modified statement.

## How Fine-Grained Access Works

- You write a function to return the account manager ID:

```
account_mgr_id := (SELECT account_mgr_id
                    FROM    customers
                    WHERE    account_mgr_id =
                           SYS_CONTEXT ('userenv','session_user'));
```

- The account manager user enters a query:

```
SELECT customer_id, cust_last_name, cust_email
FROM    customers;
```

- The query is modified with the function results:

```
SELECT customer_id, cust_last_name, cust_email
FROM    orders
WHERE    account_mgr_id = (SELECT account_mgr_id
                           FROM    customers
                           WHERE    account_mgr_id =
                                   SYS_CONTEXT ('userenv','session_user'));
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.


### How Fine-Grained Access Works (continued)

Fine-grained access control is based on a dynamically modified statement. In the example in the slide, the user enters a broad query against the CUSTOMERS table that retrieves customer names and email names for a specific account manager. The Oracle server calls the function to implement the security policy. This modification is transparent to the user. It results in successfully restricting access to other customers' information, displaying only the information relevant to the account manager.

**Note:** The SYS\_CONTEXT function returns a value for an attribute, in this case, connection attributes. This is explained in detail in the following pages.

## Why Use Fine-Grained Access?

To implement the business rule “Account managers can see only their own customers,” you have three options:

Option	Comment
Modify all existing application code to include a predicate (a <code>WHERE</code> clause) for all SQL statements.	Does not ensure privacy enforcement outside the application. Also, all application code may need to be modified in the future as business rules change.
Create views with the necessary Predicates, and then create synonyms with the same name as the table names for these views.	This can be difficult to administer, especially if there are a large number of views to track and manage.
Create a VPD for each of the account managers by creating policy functions to generate dynamic predicates. These predicates can then be applied across all objects.	This option offers the best security without major administrative overheads and it also ensures complete privacy of information. 

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Why Use Fine-Grained Access?

There are other methods by which you can implement the business rule “Account managers can see only their own customers.” The options are listed above. However, by using fine-grained access, you implement security without major overheads.

# Lesson Agenda

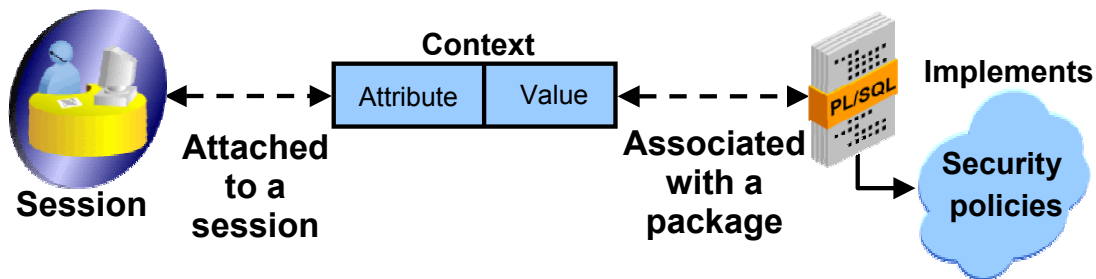
- Describing the process of fine-grained access control
- Implementing and testing fine-grained access control

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using an Application Context

- An application context is used to facilitate the implementation of fine-grained access control.
- It is a named set of attribute/value pairs associated with a PL/SQL package.
- Applications can have their own application-specific contexts.
- Users cannot change their application's context.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using an Application Context

An application context:

- Is a named set of attribute/value pairs associated with a PL/SQL package
- Is attached to a session
- Enables you to implement security policies with functions, and then associate them with applications

A context is a named set of attribute/value pairs that are global to your session. You can define an application context, name it, and associate a value with that context with a PL/SQL package. An application context enables you to write applications that draw upon certain aspects of a user's session information. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, you limit the customers' access to their own orders (ORDER\_ID) and customer number (CUSTOMER\_ID). Or, you may limit account managers (ACCOUNT\_MGR\_ID) to view only their own customers. These values can be used as security attributes. Your application can use a context to set values that are accessed within your code and used to generate WHERE clause predicates for fine-grained access control.

An application context is owned by SYS.



# Using an Application Context

System  
predefined

USERENV Context	
Attribute	Value
IP_ADDRESS	139.185.35.118
SESSION_USER	oe
CURRENT_SCHEMA	oe
DB_NAME	orcl

Application  
defined

YOUR_DEFINED Context	
Attribute	Value
customer_info	cus_1000
account_mgr	AM145

The function  
**SYS\_CONTEXT**  
returns a value  
of an attribute  
of a context.

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
FROM DUAL;
```

```
SYS_CONTEXT ('USERENV', 'SESSION_USER')
```

```
-----
OE
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using an Application Context (continued)

A predefined application context named USERENV has a predefined list of attributes. Predefined attributes can be very useful for access control. You find the values of the attributes in a context by using the SYS\_CONTEXT function. Although the predefined attributes in the USERENV application context are accessed with the SYS\_CONTEXT function, you cannot change them.

With the SYS\_CONTEXT function, you pass the context name and the attribute name. The attribute value is returned.

The following statement returns the name of the database that is being accessed:

```
SELECT SYS_CONTEXT ('USERENV', 'DB_NAME')
FROM DUAL;
```

```
SYS_CONTEXT ('USERENV', 'DB_NAME')
```

```
-----
ORCL
```

## Creating an Application Context

```
CREATE [OR REPLACE] CONTEXT namespace  
USING  [schema.]plsql_package
```

- Requires the CREATE ANY CONTEXT system privilege
- Parameters:
  - *namespace* is the name of the context.
  - *schema* is the name of the schema owning the PL/SQL package.
  - *plsql\_package* is the name of the package used to set or modify the attributes of the context. (It does not need to exist at the time of context creation.)

```
CREATE CONTEXT order_ctx USING oe.orders_app_pkg;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Creating an Application Context

For fine-grained access where you want account manager to view only their customers, customers to view only their information, and sales representatives to view only their orders, you can create a context called ORDER\_CTX and define for it the ACCOUNT\_MGR, CUST\_ID and SALE\_REP attributes.

Because a context is associated with a PL/SQL package, you need to name the package that you are associating with the context. This package does not need to exist at the time of context creation.

## Setting a Context

- Use the supplied package procedure `DBMS_SESSION.SET_CONTEXT` to set a value for an attribute within a context.

```
DBMS_SESSION.SET_CONTEXT('context_name',  
                          'attribute_name',  
                          'attribute_value')
```

- Set the attribute value in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg  
...  
BEGIN  
    DBMS_SESSION.SET_CONTEXT('ORDER_CTX',  
                             'ACCOUNT_MGR',  
                             v_user)  
...  
END;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Setting a Context

When a context is defined, you can use the `DBMS_SESSION.SET_CONTEXT` procedure to set a value for an attribute within a context. The attribute is set in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg  
IS  
    PROCEDURE set_app_context;  
END;  
/  
CREATE OR REPLACE PACKAGE BODY orders_app_pkg  
IS  
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';  
    PROCEDURE set_app_context  
    IS  
        v_user VARCHAR2(30);  
    BEGIN  
        SELECT user INTO v_user FROM dual;  
        DBMS_SESSION.SET_CONTEXT  
            (c_context, 'ACCOUNT_MGR', v_user);  
    END;  
END;  
/
```

## Setting a Context (continued)

In the example on the previous page, the ORDER\_CTX context has the ACCOUNT\_MGR attribute set to the current user logged (determined by the USER function).

For this example, assume that users AM145, AM147, AM148, and AM149 exist. As each user logs on and the DBMS\_SESSION.SET\_CONTEXT is invoked, the attribute value for that ACCOUNT\_MGR is set to the user ID.

```
GRANT EXECUTE ON oe.orders_app_pkg
  TO AM145, AM147, AM148, AM149;

CONNECT AM145/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
-----
AM145
```

If you switch the user ID, the attribute value is also changed to reflect the current user.

```
CONNECT AM147/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
-----
AM147
```

# Implementing a Policy

Follow these steps:

1. Set up a driving context.

```
CREATE OR REPLACE CONTEXT order_ctx  
  USING orders_app_pkg;
```

2. Create the package associated with the context that you defined in step 1. In the package:
  - a. Set the context.
  - b. Define the predicate.
3. Define the policy.
4. Set up a logon trigger to call the package at logon time and set the context.
5. Test the policy.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Implementing a Policy

In this example, assume that the users AM145, AM147, AM148, and AM149 exist. Next, create a context and a package associated with the context. The package will be owned by OE.

### Step 1: Set Up a Driving Context

Use the CREATE CONTEXT syntax to create a context.

```
CREATE CONTEXT order_ctx USING oe.orders_app_pkg;
```

## Step 2: Creating the Package

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
  PROCEDURE show_app_context;
  PROCEDURE set_app_context;
  FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END orders_app_pkg;      -- package spec
/
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Implementing a Policy (continued)

#### Step 2: Create a Package

In the OE schema, the ORDERS\_APP\_PKG is created. This package contains three routines:

- **show\_app\_context:** For learning and testing purposes, this procedure displays a context attribute and value.
- **set\_app\_context:** This procedure sets a context attribute to a specific value.
- **the\_predicate:** This function builds the predicate (the WHERE clause) that controls the rows visible in the CUSTOMERS table to a user. (Note that this function requires two input parameters. An error occurs when the policy is implemented if you exclude these two parameters.)

## Implementing a Policy (continued)

### Step 2: Create a Package (continued)

```
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    c_attrib  CONSTANT VARCHAR2(30) := 'ACCOUNT_MGR';

    PROCEDURE show_app_context
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Type: ' || c_attrib ||
            ' - ' || SYS_CONTEXT(c_context, c_attrib));
    END show_app_context;

    PROCEDURE set_app_context
    IS
        v_user VARCHAR2(30);
    BEGIN
        SELECT user INTO v_user FROM dual;
        DBMS_SESSION.SET_CONTEXT
            (c_context, c_attrib, v_user);
    END set_app_context;

    FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2
    IS
        v_context_value VARCHAR2(100) :=
            SYS_CONTEXT(c_context, c_attrib);
        v_restriction VARCHAR2(2000);
    BEGIN
        IF v_context_value LIKE 'AM%' THEN
            v_restriction :=
                'ACCOUNT_MGR_ID =
                SUBSTR('' ' || v_context_value || '', 3, 3)';
        ELSE
            v_restriction := null;
        END IF;
        RETURN v_restriction;
    END the_predicate;

END orders_app_pkg; -- package body
/
```

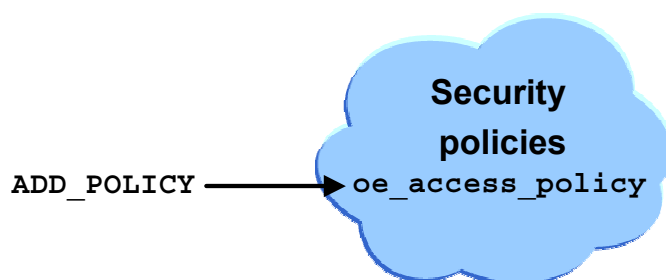
Note that the `THE_PREDICATE` function builds the WHERE clause and stores it in the `V_RESTRICTION` variable. If the `SYS_CONTEXT` function returns an attribute value that starts with AM, the WHERE clause is built with `ACCOUNT_MGR_ID = the last three characters of the attribute value`. If the user is AM145, the WHERE clause will be:

```
WHERE account_mgr_id = 145
```

## Step 3: Defining the Policy

Use the DBMS\_RLS package:

- It contains the fine-grained access administrative interface.
- It adds a fine-grained access control policy to a table or view.
- You use the ADD\_POLICY procedure to add a fine-grained access control policy to a table or view.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Implementing a Policy (continued)

The DBMS\_RLS package contains the fine-grained access control administrative interface. The package holds several procedures. But the package by itself does nothing until you add a policy. To add a policy, you use the ADD\_POLICY procedure within the DBMS\_RLS package.

**Note:** DBMS\_RLS is available only with the Enterprise Edition.

### Step 3: Define the Policy

The DBMS\_RLS.ADD\_POLICY procedure adds a fine-grained access control policy to a table or view. The procedure causes the current transaction, if any, to commit before the operation is carried out. However, this does not cause a commit first if it is inside a DDL event trigger. These are the parameters for the ADD\_POLICY procedure:

```
DBMS_RLS.ADD_POLICY (  
    object_schema    IN VARCHAR2 := NULL,  
    object_name      IN VARCHAR2,  
    policy_name      IN VARCHAR2,  
    function_schema  IN VARCHAR2 := NULL,  
    policy_function   IN VARCHAR2,  
    statement_types  IN VARCHAR2 := NULL,  
    update_check     IN BOOLEAN  := FALSE,  
    enable           IN BOOLEAN  := TRUE);
```



## Implementing a Policy (continued)

### Step 3: Define the Policy (continued)

Parameter	Description
OBJECT_SCHEMA	Schema containing the table or view (logon user, if NULL)
OBJECT_NAME	Name of the table or view to which the policy is added
POLICY_NAME	Name of the policy to be added. For any table or view, each POLICY_NAME must be unique.
FUNCTION_SCHEMA	Schema of the policy function (logon user, if NULL)
POLICY_FUNCTION	Name of the function that generates a predicate for the policy. If the function is defined within a package, the name of the package must be present.
STATEMENT_TYPES	Statement types that the policy will apply. It can be any combination of SELECT, INSERT, UPDATE, and DELETE. The default is to apply all these statement types to the policy.
UPDATE_CHECK	Optional argument for the INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after INSERT or UPDATE.
ENABLE	Indicates whether the policy is enabled when it is added. The default is TRUE.
SEC_RELEVANT_COLS SEC_RELEVANT_COLS_OPT	These allow for getting all rows from the table, but for the relevant columns everybody can see only their own data, in other columns they can see NULL values.

The following is a list of the procedures contained in the DBMS\_RLS package. For detailed information, refer to the *PL/SQL Packages and Types Reference 11g Release 2 (11.2)*.

Procedure	Description
ADD_POLICY	Adds a fine-grained access control policy to a table or view
DROP_POLICY	Drops a fine-grained access control policy from a table or view
REFRESH_POLICY	Causes all the cached statements associated with the policy to be reparsed
ENABLE_POLICY	Enables or disables a fine-grained access control policy
CREATE_POLICY_GROUP	Creates a policy group
ADD_GROUPED_POLICY	Adds a policy associated with a policy group
ADD_POLICY_CONTEXT	Adds the context for the active application
DELETE_POLICY_GROUP	Deletes a policy group
DROP_GROUPED_POLICY	Drops a policy associated with a policy group
DROP_POLICY_CONTEXT	Drops a driving context from the object so that it has one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
REFRESH_GROUPED_POLICY	Reparses the SQL statements associated with a refreshed policy

## Step 3: Defining the Policy

```
DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',      ← Object schema
    'CUSTOMERS', ← Table name
    'OE_ACCESS_POLICY', ← Policy name
    'OE',      ← Function schema
    'ORDERS_APP_PKG.THE_PREDICATE', ← Policy function
    'SELECT, UPDATE, DELETE', ← Statement types
    FALSE,     ← Update check
    TRUE);     ← Enabled
END;
/
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Implementing a Policy (continued)

#### Step 3: Define the Policy (continued)

The security policy `OE_ACCESS_POLICY` is created and added with the `DBMS_RLS.ADD_POLICY` procedure. The predicate function that defines how the policy is to be implemented is associated with the policy being added.

This example specifies that whenever a `SELECT`, `UPDATE`, or `DELETE` statement on the `OE.CUSTOMERS` table is executed, the predicate function return result is appended to the `WHERE` clause.

## Step 4: Setting Up a Logon Trigger

Create a database trigger that executes whenever anyone logs on to the database:

```
CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
    oe.orders_app_pkg.set_app_context;
END;
/
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Implementing a Policy (continued)

#### Step 4: Set Up a Logon Trigger

After the context is created, the security package is defined, the predicate is defined, and the policy is defined, you create a logon trigger to implement fine-grained access control. This trigger causes the context to be set as each user is logged on.

## Example Results

Data in the CUSTOMERS table:

```
SELECT  COUNT(*), account_mgr_id
FROM    customers
GROUP BY account_mgr_id;
```

COUNT(*)	ACCOUNT_MGR_ID
76	147
74	149
58	148
111	145

```
CONNECT AM148/oracle
SELECT  customer_id, cust_last_name
FROM    oe.customers;
```

CUSTOMER_ID	CUSTOMER_LAST_NAME
...	

58 rows selected.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Example Results

The AM148 user who logs on sees only those rows in the CUSTOMERS table that are defined by the predicate function. The user can issue SELECT, UPDATE, and DELETE statements against the CUSTOMERS table, but only the rows defined by the predicate function can be manipulated.

```
UPDATE oe.customers
SET credit_limit = credit_limit + 5000
WHERE customer_id = 101;
```

0 rows updated.

The AM148 user does not have access to customer ID 101. Customer ID 101 has the account manager of 145. Any updates, deletes, or selects attempted by user AM148 on customers that do not have him or her as an account manager are not performed. It is as though these customers do not exist.

# Data Dictionary Views

- USER\_POLICIES
- ALL\_POLICIES
- DBA\_POLICIES
- ALL\_CONTEXT
- DBA\_CONTEXT



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Data Dictionary Views

You can query the data dictionary views to find information about the policies available in your schema.

View	Description
USER_POLICIES	All policies owned by the current schema
ALL_POLICIES	All policies owned or accessible by the current schema
DBA_POLICIES	All policies in the database (its columns are the same as those in ALL_POLICIES)
ALL_CONTEXT	All active context namespaces defined in the session
DBA_CONTEXT	All context namespace information (active and inactive)

## Using the ALL\_CONTEXT Dictionary View

Use ALL\_CONTEXT to see the active context namespaces defined in your session:

```
CONNECT AS AM148
```

```
SELECT *  
FROM all_context;
```

NAMESPACE	SCHEMA	PACKAGE
-----	-----	-----
ORDER_CTX	OE	ORDERS_APP_PKG

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using the ALL\_CONTEXT Dictionary View

You can use the ALL\_CONTEXT dictionary view to view information about the contexts to which you have access. In the slide, the NAMESPACE column is equivalent to the context name.

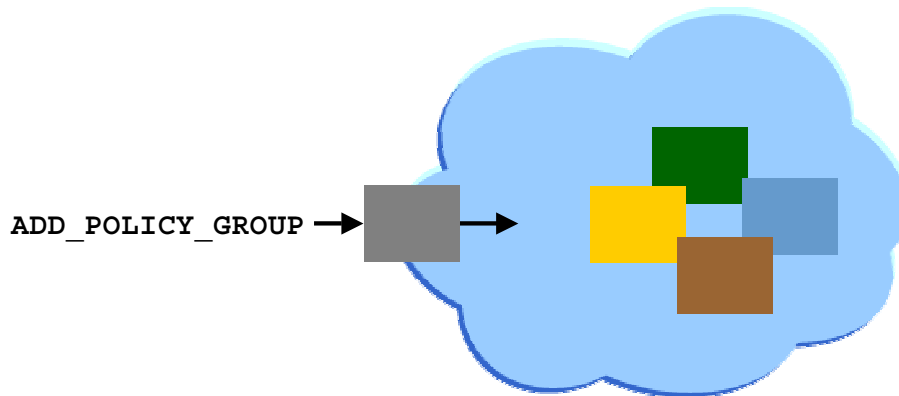
You can use the ALL\_POLICIES dictionary view to view information about the policies to which you have access. In the following example, information is shown about the OE\_ACCESS\_POLICY policy.

```
SELECT object_name, policy_name, pf_owner, package,  
       function, sel, ins, upd, del  
FROM all_policies;
```

OBJECT_NAME	POLICY_NAME	PF_OWNER	PACKAGE	FUNCTION	SEL	INS	UPD	DEL
CUSTOMERS	OE_ACCESS_POLICY	OE	ORDERS_APP_PKG	THE_PREDICATE	YES	NO	YES	YES

# Policy Groups

- Indicate a set of policies that belong to an application
- Are set up by a DBA through an application context called a driving context
- Use the `DBMS_RLS` package to administer the security policies



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Policy Groups

Policy groups were introduced in Oracle9i, release 1 (9.0.1). The DBA designates an application context, called a driving context, to indicate the policy group in effect. When tables or views are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all associated policies that belong to that policy group.

The PL/SQL `DBMS_RLS` package enables you to administer your security policies and groups. Using this package, you can add, drop, enable, disable, and refresh the policy groups that you create.

## More About Policies

- `SYS_DEFAULT` is the default policy group:
  - The `SYS_DEFAULT` group may or may not contain policies.
  - All policies belong to `SYS_DEFAULT` by default.
  - You cannot drop the `SYS_DEFAULT` policy group.
- Use `DBMS_RLS.CREATE_POLICY_GROUP` to create a new group.
- Use `DBMS_RLS.ADD_GROUPED_POLICY` to add a policy associated with a policy group.
- You can apply multiple driving contexts to the same table or view.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### More About Policies

A policy group is a set of security policies that belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. When the tables or views are accessed, the server looks up the driving context to determine the policy group in effect. It enforces all associated policies that belong to that policy group.

By default, all policies belong to the `SYS_DEFAULT` policy group. The policies defined in this group for a particular table or view are always executed along with the policy group specified by the driving context. The `SYS_DEFAULT` policy group may or may not contain policies. If you attempt to drop the `SYS_DEFAULT` policy group, an error is raised. If you add policies associated with two or more objects to the `SYS_DEFAULT` policy group, each such object has a separate `SYS_DEFAULT` policy group associated with it.

For example, the `CUSTOMERS` table in the `OE` schema has one `SYS_DEFAULT` policy group, and the `ORDERS` table in the `OE` schema has a different `SYS_DEFAULT` policy group associated with it.

```
SYS_DEFAULT
- policy1 (OE/CUSTOMERS)
- policy3 (OE/CUSTOMERS)
SYS_DEFAULT
- policy2 (OE/ORDERS)
```



## More About Policies (continued)

When adding a policy to a table or view, you can use the `DBMS_RLS.ADD_GROUPED_POLICY` interface to specify the group to which the policy belongs. To specify which policies are effective, you can add a driving context using the `DBMS_RLS.ADD_POLICY_CONTEXT` interface. If the driving context returns an unknown policy group, an error is returned.

If the driving context is not defined, all policies are executed. Likewise, if the driving context is `NULL`, the policies from all policy groups are enforced. Thus, an application that accesses the data cannot bypass the security setup module (that sets up the application context) to avoid applicable policies.

You can apply multiple driving contexts to the same table or view, and each of them are processed individually. Thus, you can configure multiple active sets of policies to be enforced.

You can create a new policy by using the `DBMS_RLS` package either from the command line or programmatically, or you can access the Oracle Policy Manager graphical user interface in Oracle Enterprise Manager.

## Quiz

Which of the following statements is *not* true about fine-grained access control?

- a. Fine-grained access control enables you to enforce security through a low level of granularity.
- b. Fine-grained access control restricts users to viewing only “their” information.
- c. Fine-grained access control is implemented through a security policy attached to tables.
- d. To implement fine-grained access control, hard code the security policy into the user code.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: d**

## Quiz

Application context for a user is fixed and does not change with change of session.

- a. True
- b. False

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: b**

## Quiz

Arrange the following steps used to implement a security policy:

- A. Define the policy.
- B. Create the package associated with the context.
- C. Set up a driving context.
- D. Set up a logon trigger to call the package at logon time and set the context.

- a. A, B, C D
- b. C, A, B, D
- c. B, A, C, D
- d. D, A, B, C

ORACLE

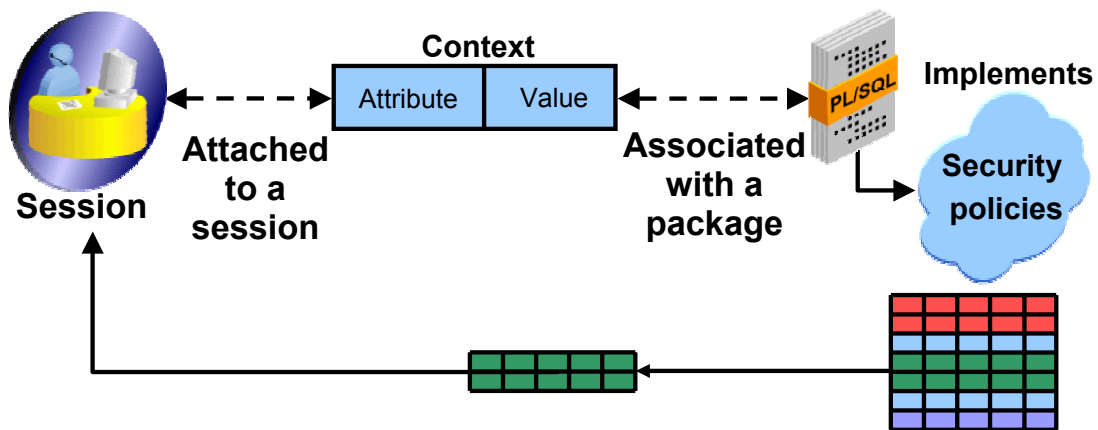
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: b**

# Summary

In this lesson, you should have learned how to:

- Describe the process of fine-grained access control
- Implement and test fine-grained access control



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Summary

In this lesson, you should have learned about fine-grained access control and the steps required to implement a virtual private database.

## Practice 11: Overview

This practice covers the following topics:

- Creating an application context
- Creating a policy
- Creating a logon trigger
- Implementing a virtual private database
- Testing the virtual private database

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Practice 11: Overview

In this practice, you implement and test fine-grained access control.

# 12

## **Safeguarding Your Code Against SQL Injection Attacks**

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Describe SQL injections
- Reduce attack surfaces
- Use `DBMS_ASSERT`
- Design immune code
- Test code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

In this lesson, you learn how to use techniques and tools to strengthen your code and applications against SQL injection attacks.



# Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

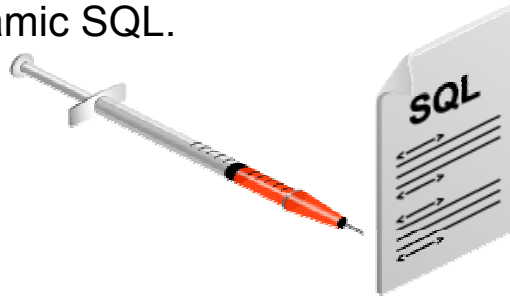
ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# Understanding SQL Injection

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

- Attackers trick the SQL engine into executing unintended commands.
- SQL injection techniques may differ, but they all exploit a single vulnerability in the application.
- To immunize your code against SQL injection attacks, use bind arguments or validate and sanitize all input concatenated to dynamic SQL.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Understanding SQL Injection

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

Attackers trick the SQL engine into executing unintended commands via supplying specially crafted string input, thereby gaining unauthorized access to a database in order to view or manipulate restricted data.

SQL injection techniques may differ, but they all exploit a single vulnerability in the application.

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement and interpreted as code by the SQL engine.

To immunize your code against SQL injection attacks, you must use bind arguments (either automatically with static SQL, or explicitly with dynamic SQL), or validate and sanitize all input concatenated to dynamic SQL.

Although a program or an application may be vulnerable to SQL injection, Web applications are at a higher risk, because an attacker can perpetrate SQL injection attacks without any database or application authentication.

# Identifying Types of SQL Injection Attacks

Category	Description
First-order attack	The attacker can simply enter a malicious string and cause the modified code to be executed immediately.
Second-order attack	The attacker injects into persistent storage (such as a table row), which is deemed a trusted source. An attack is subsequently executed by another activity.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Types of SQL Injection Attacks

The impact of SQL injection attacks may vary from gathering of sensitive data to manipulating database information, and from executing system-level commands to denial of service of the application. The impact also depends on the database on the target machine, and the roles and privileges that the SQL statement is running with.

Researchers generally divide injection attacks into two categories:

- First-order attack
- Second-order attack

SQL injection attacks do not have to return data directly to the user to be useful. “Blind” attacks (for example, that create a database user, but otherwise return no data) can still be very useful to an attacker.

In addition, hackers are known to use timing or other performance indicators, and even error messages in order to deduce the success or results of an attack.

# SQL Injection: Example

```
-- First order attack
CREATE OR REPLACE PROCEDURE GET_EMAIL
(p_last_name VARCHAR2 DEFAULT NULL)
AS
  TYPE      cv_custtyp IS REF CURSOR;
  cv        cv_custtyp;
  v_email   customers.cust_email%TYPE;
  v_stmt    VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT cust_email FROM customers
            WHERE cust last name = ''' || p_last_name || ''';
  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
    FETCH cv INTO v_email;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Email: ' || v_email);
  END LOOP;
  CLOSE cv;
EXCEPTION WHEN OTHERS THEN
  dbms_output.PUT_LINE(sqlerrm);
  dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END;
```

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement, and interpreted as code by the SQL engine.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## First-Order SQL Injection Attack: Example

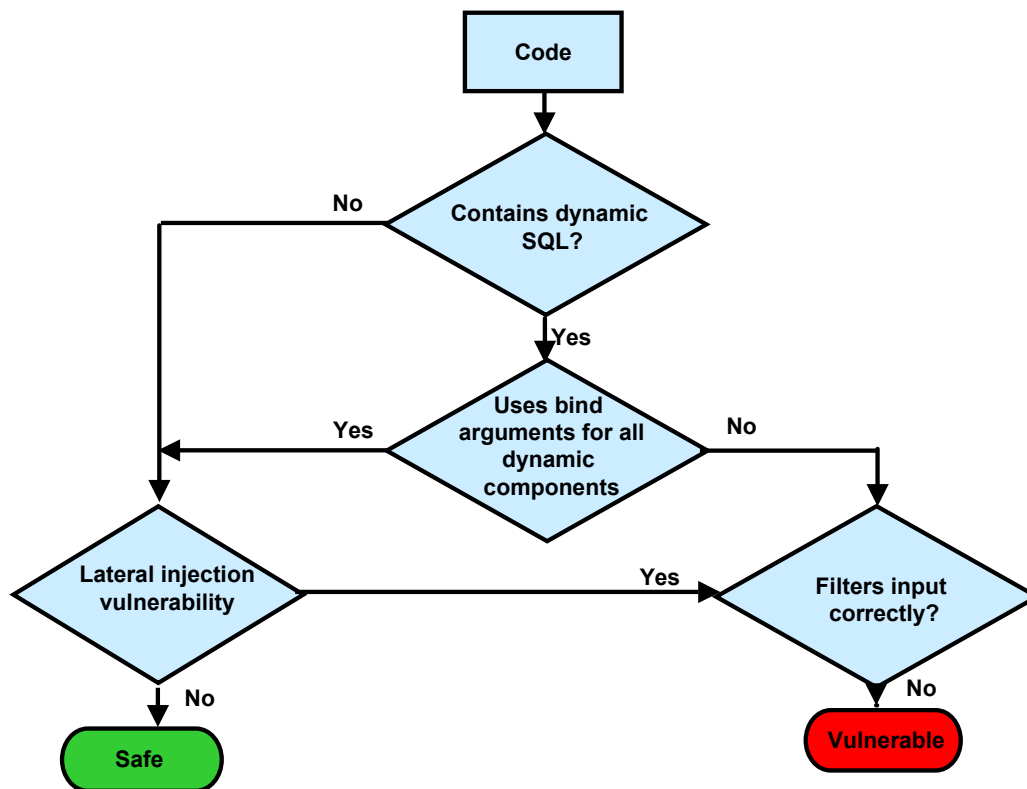
The example shown in the slide demonstrates a procedure with dynamic SQL constructed via concatenation of input value. This is vulnerable to SQL injection.

```
EXECUTE get_email('Andrews')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'Andrews'
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM
```

PL/SQL procedure successfully completed.

```
EXECUTE get_email('x' union select username from all_users where
'x'='x')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'x' union
select username from all_users where 'x'='x'
Email: ANONYMOUS
Email: APEX_PUBLIC_USER
Email: BI
Email: CTXSYS
...
```

## Assessing Vulnerability



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Assessing Vulnerability

You must analyze your code to determine SQL injection vulnerability. The flowchart shows you how to start assessing for vulnerability.

# Avoidance Strategies Against SQL Injection

Strategy	Description
Reduce the attack surface	Ensure that all excess database privileges are revoked and that only those routines that are intended for end-user access are exposed. Though this does not entirely eliminate SQL injection vulnerabilities, it does mitigate the impact of the attacks.
Avoid dynamic SQL with concatenated input	Dynamic SQL built with concatenated input values presents the easiest entry point for SQL injections. Avoid constructing dynamic SQL this way.
Use bind arguments	Parameterize queries using bind arguments. Not only do bind arguments eliminate the possibility of SQL injections, they also enhance performance.
Filter and sanitize input	<p>The Oracle-supplied <code>DBMS_ASSERT</code> package contains a number of functions that can be used to sanitize user input and to guard against SQL injection in applications that use dynamic SQL built with concatenated input values.</p> <p>If your filtering requirements cannot be satisfied by the <code>DBMS_ASSERT</code> package, create your own filter.</p>

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Avoidance Strategies Against SQL Injection

You can use several avoidance strategies to safeguard against, or mitigate the impact of, SQL injection attacks. Listed in the slide are high-level descriptions of each of the strategies that are examined in more detail on subsequent pages.

The available and best method(s) for eliminating SQL injection vulnerability may depend on the vulnerability itself. Not all methods are available for addressing every vulnerability.

### Methods

- Use static SQL:
  - If all Oracle identifiers (for example, column, table, view, trigger, program unit, or schema names) are known at code compilation time

**Note:** Static SQL automatically binds arguments. Data definition language (DDL) statements cannot be executed with static SQL.
- Use dynamic SQL with bind arguments:
  - If any `WHERE` clause values, `VALUES` clause values, or `SET` clause values are unknown, and any Oracle identifiers are unknown at code compilation time

**Note:** Use bind arguments for the values (the literals). Use string concatenation for the validated and sanitized Oracle identifiers.
- Validate and sanitize input:
  - If concatenating any strings and if bind arguments require additional filtering

# Protecting Against SQL Injection: Example

```
CREATE OR REPLACE PROCEDURE GET_EMAIL
(p_last_name VARCHAR2 DEFAULT NULL)
AS
BEGIN
  FOR i IN
    (SELECT cust_email
     FROM customers
     WHERE cust_last_name = p_last_name)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Email: ' || i.cust_email);
  END LOOP;
END;
```

This example avoids dynamic SQL with concatenated input values.

```
EXECUTE get_email('Andrews')
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM

PL/SQL procedure successfully completed.

EXECUTE get_email('x' union select username from all_users where
'x'='x')

PL/SQL procedure successfully completed.
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## First-Order SQL Injection Attack: Example

The example in the slide with static SQL is protected against SQL injection.

# Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.



## Reducing the Attack Surface

Use the following strategies to reduce attack surface:

- Expose the database only via a PL/SQL API.
- Use invoker's rights.
- Reduce arbitrary inputs.
- Strengthen database security.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Reducing the Attack Surface

If an interface is not available to an attacker, it is clearly not available to be abused. Thus the first, and arguably most important, line of defense is to reduce the exposed interfaces to only those absolutely required.

## Expose the Database Only Via PL/SQL API

- Expose the database to clients only via a PL/SQL API.
- When you design a PL/SQL package that accesses the database, use the following paradigm:
  - Establish a database user as the *only* one to which a client may connect. Hypothetically, let us call this user **myuser**.
  - **myuser** may own *only* synonyms and these synonyms may denote *only* PL/SQL units owned by other users.
  - Grant the Execute privilege on *only* the denoted PL/SQL units to **myuser**.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Expose the Database Only Via PL/SQL API

Expose the database to clients only via a PL/SQL API. Carefully control privileges so that the client has no direct access to the application's other kinds of objects, especially tables and views.

# Using Invoker's Rights

- Using invoker's rights:
  - Helps to limit the privileges
  - Helps to minimize the security exposure.
- The following example does not use invoker's rights:

```
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                          p_new_password VARCHAR2 DEFAULT NULL)
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'ALTER USER ' || p_username || ' IDENTIFIED BY '
                || p_new_password;
  EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

Note the use of dynamic SQL with concatenated input values.

```
GRANT EXECUTE ON change_password to OE, HR, SH;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using Invoker's Rights

Stored program units and SQL methods execute with a set of privileges. By default, the privileges are those of a schema owner, also known as the definer. The definer's rights not only dictate the privileges, they are also used to resolve object references. If a program unit does not need to be executed with the escalated privileges of the definer, you should specify that the program unit executes with the privileges of a caller, also known as the invoker.

1. The example shown in the slide uses definer's rights. Procedure `CHANGE_PASSWORD` is created under the `SYS` schema. It accepts two parameters and uses them in the `ALTER USER` statement.
2. `SYS` grants `OE`, `HR`, and `SH` the ability to execute the `CHANGE_PASSWORD` procedure.

## Using Invoker's Rights

- OE is successful at changing the SYS password, because, by default, CHANGE\_PASSWORD executes with SYS privileges:

```
EXECUTE sys.change_password ('SYS', 'mine')
```

- Add the AUTHID to change the privileges to the invokers:

```
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                          p_new_password VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER
IS
    v_sql_stmt VARCHAR2(500);
BEGIN
    v_sql_stmt := 'ALTER USER ' || p_username || ' IDENTIFIED BY '
                  || p_new_password;
    EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using Invoker's Rights (continued)

When OE executes the CHANGE\_PASSWORD procedure, it is executed under SYS privileges (the definer of the procedure), and with the code shown in the slide, OE can change the SYS password. Obviously, this is an unacceptable outcome.

To disallow another schema from changing a password that does not belong to the schema, redefine the procedure with the invoker's rights. This is done with the AUTHID CURRENT\_USER option.

```
EXECUTE change_password ('SYS', 'mine')
ERROR at line 1:
ORA-01031: Insufficient privileges
ORA-06512: at "SYS.CHANGE_PASSWORD", at line 1
ORA-06512: at line 1
```

Now OE can no longer change the SYS (or any other account) password.

Notice that the CHANGE\_PASSWORD procedure contains dynamic SQL with concatenated input values. This is a SQL injection vulnerability. Although using invoker's rights does not guarantee the elimination of SQL injection risks, it can help mitigate the exposure.

## Reducing Arbitrary Inputs

- Reduce the end-user interfaces to only those that are actually needed.
  - In a Web application, restrict users to accessing specified Web pages.
  - In a PL/SQL API, expose only those routines that are intended for customer use.
- Where user input is required, make use of language features to ensure that only data of the intended type can be specified.
  - Do not specify a `VARCHAR2` parameter when it will be used as a number.
  - Do not use numbers if you need only positive integers; use `natural` instead.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Reducing the Arbitrary Inputs

Because a SQL injection attack is possible only if user input is allowed, one key measure of prevention is to limit user input.

First, you must reduce the end-user interfaces to only those actually needed. For example:

- In a Web application, restrict users to accessing specified Web pages.
- In a PL/SQL API, expose only those routines that are intended for customer use.

Remove any debug, test, deprecated, or other unnecessary interfaces. They add nothing to product functionality, but do provide an attacker with more ways to target your application.

From Oracle Database 10.2 and later, you can use PL/SQL conditional compilation for managing self-tracing code, asserts, and so on.

Second, where user input is required, make use of language features to ensure that only data of the intended type can be specified. For example:

- Do not specify a `VARCHAR2` parameter when it will be used as a number.
- Do not use numbers if you need only positive integers; use `natural` instead.

Careful selection of parameter types to an API can considerably reduce the scope for attack, and make it a lot easier for customers to use.

## Strengthen Database Security

- Here is a list of some of the practices to observe when you secure the Oracle database:
  - Encrypt sensitive data so that it cannot be viewed.
  - Avoid the following:
    - PUBLIC privileges
    - EXECUTE ANY PROCEDURE privilege
    - Privileges WITH ADMIN option
  - Do not allow wide access to any standard Oracle packages that can operate on the operating system.
  - Lock the database default accounts and expire the default passwords.
  - Enforce password management.
  - Lock and expire the default user accounts and change the default user password.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Strengthen Database Security

Oracle Database contains inherent security features that help in protecting it from many types of attacks, including SQL injection. Here is a list of some of the practices to observe when you secure the Oracle database:

- Encrypt sensitive data so that it cannot be viewed.
- Evaluate all PUBLIC privileges and revoke them where possible.
- Do not widely grant EXECUTE ANY PROCEDURE.
- Avoid granting privileges WITH ADMIN option.
- Do not allow wide access to any standard Oracle packages that can operate on the operating system. These packages include UTL\_HTTP, UTL\_SMTP, UTL\_TCP, DBMS\_PIPE, UTL\_MAIL, and UTL\_FTP.
- Certain Oracle packages such as UTL\_FILE and DBMS\_LOB are governed by the privilege model of the Oracle DIRECTORY object. Protect Oracle DIRECTORY objects.
- Lock the database default accounts and expire the default passwords.
- Remove example scripts and programs from the Oracle directory.
- Run the database listener as a nonprivileged user.
- Apply basic password management rules, such as password length, history, and complexity, to all user passwords. Mandate that all the users change their passwords regularly.
- Lock and expire the default user accounts and change the default user password.

# Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- **Avoiding dynamic SQL**
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using Static SQL

- Eliminates SQL injection vulnerability
- Creates schema object dependencies upon successful compilation
- Can improve performance, when compared with `DBMS_SQL`

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using Static SQL

Because SQL injection is a feature of SQL statements that are dynamically constructed via user inputs, it follows that designing your application to be based on static SQL reduces the scope for attack.

There are two common situations where developers often use dynamic SQL, when static SQL would serve the purpose and be more secure:

- Handling a varying number of `IN`-list values in the query condition
- Handling the `LIKE` comparison operator in the query condition



# Using Static SQL

```
CREATE OR REPLACE PROCEDURE list_products_dynamic
(p_product_name VARCHAR2 DEFAULT NULL)
AS
  TYPE cv_prodtype IS REF CURSOR;
  cv cv_prodtype;
  v_prodname product_information.product_name%TYPE;
  v_minprice product_information.min_price%TYPE;
  v_listprice product_information.list_price%TYPE;
  v_stmt VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT product_name, min_price, list_price
            FROM product_information WHERE product_name LIKE
            ''%''||p_product_name||''%''';

  OPEN cv FOR v_stmt;
  dbms_output.put_line(v_stmt);
  LOOP
    FETCH cv INTO v_prodname, v_minprice, v_listprice;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Product Info: '||v_prodname||', '||
                        v_minprice||', '||v_listprice);
  END LOOP;
  CLOSE cv;
END;
```

You can convert this statement to static SQL.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Converting Dynamic SQL to Static SQL

By converting the dynamic SQL static shown in the slide, you can decrease your attack vulnerability. The code above uses dynamic SQL to handle the LIKE operator in the query condition. Notice that string concatenation is used to build the SQL statement.

Examine the execution of this injection:

```
EXECUTE list_products_dynamic('' and 1=0 union select
cast(username as varchar2(100)), null, null
from all_users --')
```

```
SELECT product_name, min_price, list_price FROM
product_information WHERE
product_name like '%' and 1=0 union select cast(username as
varchar2(100)),
null, null from all_users --%'
Product Info: ANONYMOUS, ,
Product Info: APEX_PUBLIC_USER, ,
Product Info: BI, ,
...
```

Notice that the injection succeeded through the concatenation of the UNION set operator to the dynamic SQL statement.

## Using Static SQL

- To use static SQL, accept the user input, and then concatenate the necessary string to a local variable.
- Pass the local variable to the static SQL statement.

```
CREATE OR REPLACE PROCEDURE list_products_static
(p_product_name VARCHAR2 DEFAULT NULL)
AS
  v_bind VARCHAR2(400);
BEGIN
  v_bind := '%' || p_product_name || '%';
  FOR i in
    (SELECT product_name, min_price, list_price
     FROM product_information
     WHERE product_name like v_bind)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Product Info: ' || i.product_name || ',
                          ' || i.min_price || ', ' || i.list_price);
  END LOOP;
END list_products_static;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Converting Dynamic SQL to Static SQL (continued)

In the example shown above, you can avoid SQL injection by concatenating the user input and placing it into a local variable, and then using the local variable within the static SQL.

Examine the results:

```
-- desired results - normal execution
EXECUTE list_products_static('Laptop')
Product Info: Laptop 128/12/56/v90/110, 2606, 3219
Product Info: Laptop 16/8/110, 800, 999
Product Info: Laptop 32/10/56, 1542, 1749
Product Info: Laptop 48/10/56/110, 2073, 2556
Product Info: Laptop 64/10/56/220, 2275, 2768
```

PL/SQL procedure successfully completed.

```
-- this example attempts injection
EXECUTE list_products_static('' and 1=0 union select
cast(username as varchar2(100)), null, null from all_users --')
```

PL/SQL procedure successfully completed.

## Using Dynamic SQL

- Dynamic SQL may be unavoidable in the following types of situations:
  - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure.
  - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
  - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using Dynamic SQL

- Dynamic SQL may be unavoidable in the following types of situations:
  - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure. For example, a `SELECT` statement that includes an identifier (such as table name) that is unknown at compile time or a `WHERE` clause in which the number of subclauses is unknown at compile time.
  - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
  - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.

If you cannot avoid input concatenation, you must validate the input values, and also consider constraining user input to a predefined list of values, preferably numeric values. Input filtering and sanitizing are covered in more detail later in this lesson.

# Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- **Using bind arguments**
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# Using Bind Arguments with Dynamic SQL

You can rewrite the following statement

```
v_stmt :=  
'SELECT ' || filter(p_column_list) || ' FROM customers ' ||  
'WHERE account_mgr_id = ' || p_sales_rep_id || ''';  
  
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic SQL with a placeholder (:1) by using a bind argument (p\_sales\_rep\_id):

```
v_stmt :=  
'SELECT ' || filter(p_column_list) || ' FROM customers ' ||  
'WHERE account_mgr_id = :1';  
  
EXECUTE IMMEDIATE v_stmt USING p_sales_rep_id;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using Bind Arguments

You can use bind arguments in the WHERE clause, the VALUES clause, or the SET clause of any SQL statement, as long as the bind arguments are not used as Oracle identifiers (such as column names or table names) or keywords.

Developers often use dynamic SQL to handle a varying number of IN-list values or LIKE comparison operators in the query condition.

## Using Bind Arguments with Dynamic PL/SQL

If you must use dynamic PL/SQL, try to use bind arguments. For example, you can rewrite the following dynamic PL/SQL with concatenated string values:

```
v_stmt :=  
'BEGIN  
  get_phone ('' || p_fname ||  
             ''',''' || p_lname || '''); END;'  
  
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic PL/SQL with placeholders (:1, :2) by using bind arguments (p\_fname, p\_lname):

```
v_stmt :=  
'BEGIN  
  get_phone (:1, :2); END;'  
  
EXECUTE IMMEDIATE v_stmt USING p_fname, p_lname;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using Bind Arguments with Dynamic PL/SQL

As with dynamic SQL, you should avoid constructing dynamic PL/SQL with string concatenation. The impact of SQL injection vulnerabilities in dynamic PL/SQL is more serious than in dynamic SQL, because, with dynamic PL/SQL, multiple statements (such as DELETE or DROP) can be batched together and injected.

## What If You Cannot Use Bind Arguments?

- Bind arguments cannot be used with:
  - DDL statements
  - Oracle identifiers
- If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### What If You Cannot Use Bind Arguments?

Although you should strive to use bind arguments with all dynamic SQL and PL/SQL statements, there are instances where bind arguments cannot be used:

- DDL statements (such as CREATE, DROP, and ALTER)
- Oracle identifiers (such as names of columns, tables, schemas, database links, packages, procedures, and functions)

If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement. In the following slides, you learn how to use the Oracle-supplied DBMS\_ASSERT package functions to filter and sanitize input values.

# Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- **Filtering input with DBMS\_ASSERT**
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.



# Understanding DBMS\_ASSERT

DBMS\_ASSERT functions:

Function	Description
ENQUOTE_LITERAL	Encloses string literal in single quotes
SIMPLE_SQL_NAME	Verifies that the string is a simple SQL name

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Understanding DBMS\_ASSERT

To guard against SQL injection in applications that do not use bind arguments with dynamic SQL, you must filter and sanitize concatenated strings. The primary use case for dynamic SQL with string concatenation is when an Oracle identifier (such as a table name) is unknown at code compilation time.

DBMS\_ASSERT is an Oracle-supplied PL/SQL package containing seven functions that can be used to filter and sanitize input strings, particularly those that are meant to be used as Oracle identifiers.

When using the DBMS\_ASSERT package, always specify the SYS schema rather than relying on a public synonym.

## Formatting Oracle Identifiers

- Example 1: The object name used as an identifier:

```
SELECT count(*) records FROM orders;
```

- Example 2: The object name used as a literal:

```
SELECT num_rows FROM user_tables  
WHERE table_name = 'ORDERS';
```

- Example 3: The object name used as a quoted (normal format) identifier:
  - The "orders" table referenced in example 3 is a different table compared to the orders table in examples 1 and 2.
  - It is vulnerable to SQL injection.

```
SELECT count(*) records FROM "orders";
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Understanding and Formatting Oracle Identifiers

To use DBMS\_ASSERT effectively, you must understand how Oracle identifiers can be specified and used.

In a SQL statement, you specify the name of an object with an unquoted or quoted identifier.

- An unquoted (or internal format) identifier is not surrounded by punctuation. It must begin with a letter, and may be followed by letters, numbers, or a small set of special characters. This is how identifiers are most often specified, and how object names are stored in data dictionary tables.
- A quoted (or normal format) identifier begins and ends with double quotation marks. The identifier can include almost any character within the double quotes. This format is user-supplied.

SQL injection attacks can use the quoted method to attempt to subvert code that is written to expect only the unquoted, more common, method.

## Working with Identifiers in Dynamic SQL

- For your identifiers, determine:
  1. Where will the input come from: user or data dictionary?
  2. What verification is required?
  3. How will the result be used, as an identifier or a literal value?
- These three factors affect:
  - What preprocessing is required (if any) prior to calling the verification functions
  - Which `DBMS_ASSERT` verification function is required
  - What post-processing is required before the identifier can actually be used

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Working with Identifiers in Dynamic SQL

When working with identifiers in dynamic SQL statements, you must first determine where the data is coming from, either user data or data dictionary data. For verification, you must determine whether the object must exist and the type of identifier. For types of identifiers, you have:

- **SQL Literal:** A SQL literal is a constant value, written at compile time and is read-only at run time. There are three kinds of SQL literal: text, datetime, and numeric.
- **Simple:** A simple SQL name is a string that conforms to the following basic characteristics:
  - The first character of the name is alphabetic.
  - The name contains only alphanumeric characters or the `_`, `$`, and `#` characters.
  - Quoted names must be enclosed within double quotation marks and may contain any characters, including quotation marks, provided they are represented by two quotation marks in a row.
- **Qualified SQL name:** A qualified SQL name is one or more simple SQL names that may be followed by a database link.

Lastly, determine how the result will be used, as an identifier or a literal value.

Answering these questions will impact your preprocessing, post-processing, and the use of `DBMS_ASSERT`.

## Choosing a Verification Route

Identifier Type	Verification
SQL literal	Verify whether the literal is a well-formed SQL literal by using <code>DBMS_ASSERT.ENQUOTE_LITERAL</code> .
Simple SQL name	Verify that the input string conforms to the basic characteristics of a simple SQL name by using <code>DBMS_ASSERT.SIMPLE_SQL_NAME</code> .
Qualified SQL name	Step 1: Decompose the qualified SQL name into its simple SQL names by using <code>DBMS_UTILITY.NAME_TOKENIZE()</code> . Step 2: Verify each of the simple SQL names using <code>DBMS_ASSERT.SIMPLE_SQL_NAME</code> .

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Choosing a Verification Route

After determining the type of identifier that you must verify, follow the guidelines in the slide's table to select the appropriate verification routine.

For example, you must write a procedure that allows a user to change his or her password. To choose the correct `DBMS_ASSERT` function to use, you must determine:

1. Where the input will come from, the user or the data dictionary
2. What verification is required (does the object need to exist and if so, what type of identifier is it?)
3. How the identifier will be used, as an identifier or as a literal

## Avoiding Injection by Using DBMS\_ASSERT.ENQUOTE\_LITERAL

```
CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as
Quote constant varchar2(1) := ''';
Quote_Quote constant varchar2(2) := Quote||Quote;
Safe_Literal varchar2(32767) :=
    Quote||replace(w,Quote,Quote_Quote)||Quote;
Stmt constant varchar2(32767) :=
    'SELECT count(*) FROM t WHERE a='||
    DBMS_ASSERT.ENQUOTE_LITERAL(Safe_Literal);
Row_Count number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/
```

Verify whether the literal  
is well-formed.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Avoiding Injection by Using DBMS\_ASSERT.ENQUOTE\_LITERAL

### The Complete Code

```
DROP USER testuser CASCADE;
DROP USER eviluser CASCADE;
--Create the testuser and the eviluser
GRANT Unlimited Tablespace, Create Session, Create Table,
    Create Procedure TO testuser identified by testuser;
GRANT Unlimited Tablespace, Create Session, Create Table,
    Create Procedure TO eviluser identified by eviluser;
--Connect to the testuser, create a table and insert some data
CONNECT testuser/testuser
SET SERVEROUTPUT ON
CREATE TABLE t(a varchar2(10));
BEGIN
    INSERT INTO t (a) values ('a');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('c'd');
    commit;
END;
```

## Avoiding Injection by Using DBMS\_ASSERT.ENQUOTE\_LITERAL (continued)

```
--Create a procedure that uses dynamic SQL to query table 't'
CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as
    -- Useful constant
    Quote constant varchar2(1) := ''';
    -- The statement we will execute
    Stmt constant varchar2(32767) :=
        'SELECT count(*) FROM t WHERE a='||
            Quote||w||Quote;
    -- The count of rows returned by the statement
    Row_Count number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE('The Statement is: '||Stmt);
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/
GRANT EXECUTE ON Count_Rows TO PUBLIC;

--Connect to the eviluser and exploit the vulnerable procedure
connect eviluser/eviluser
SET SERVEROUTPUT ON
CREATE OR REPLACE FUNCTION f RETURN varchar2 authid
current_user as pragma autonomous_transaction;
BEGIN
    -- Execute immediate because t is not granted to public
    EXECUTE IMMEDIATE 'DELETE FROM t';
    COMMIT;
    RETURN 'a';
END;
/
GRANT EXECUTE ON f TO PUBLIC;

--Injection successful
BEGIN testuser.Count_Rows('a' and eviluser.f='a'); END;
/

--Re-code testuser.Count_Rows with DBMS_ASSERT.ENQUOTE_LITERAL

connect testuser/testuser
SELECT * FROM t;
BEGIN
    INSERT INTO t (a) values ('a');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('c'd');
    commit;
END;
```

## Avoiding Injection by Using DBMS\_ASSERT.ENQUOTE\_LITERAL (continued)

```
CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as

    -- Useful constants
    Quote      constant varchar2(1) := '';
    Quote_Quote constant varchar2(2) :=
                                                Quote||Quote;
    Safe_Literal  varchar2(32767) :=
        Quote||replace(w,Quote,Quote_Quote)||Quote;

    -- The statement we will execute
    Stmt constant varchar2(32767) :=
        'SELECT count(*) FROM t WHERE a='||
        DBMS_ASSERT.ENQUOTE_LITERAL(Safe_Literal);

    -- The count of rows returned by the statement
    Row_Count number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE('The Statement is: '||Stmt);
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/

-- Now lets repeat the attack

connect eviluser/eviluser
SET SERVEROUTPUT ON
BEGIN testuser.Count_Rows('a' and eviluser.f=''a'); END;
/
BEGIN testuser.Count_Rows('b'); END;
/
--Injection averted

connect testuser/testuser

set serveroutput on

SELECT * FROM t;
```

## Avoiding Injection by Using DBMS\_ASSERT.SIMPLE\_SQL\_NAME

```
CREATE OR REPLACE PROCEDURE show_col2 (p_colname varchar2,  
p_tablename varchar2)  
AS  
type t is varray(200) of varchar2(25);  
Results t;
```

Verify that the input string conforms to the basic characteristics of a simple SQL name.

```
Stmt CONSTANT VARCHAR2(4000) :=  
    'SELECT ' || dbms_assert.simple_sql_name( p_colname ) || ' FROM  
' || dbms_assert.simple_sql_name( p_tablename ) ;
```

```
BEGIN  
    DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);  
    EXECUTE IMMEDIATE Stmt bulk collect into Results;  
    for j in 1..Results.Count() loop  
        DBMS_Output.Put_Line(Results(j));  
    end loop;  
END show_col2;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Avoiding Injection by Using DBMS\_ASSERT.SIMPLE\_SQL\_NAME

### The Complete Code

```
CONN hr/hr  
SET SERVEROUTPUT ON  
CREATE OR REPLACE  
PROCEDURE show_col (p_colname varchar2, p_tablename varchar2)  
AS  
type t is varray(200) of varchar2(25);  
Results t;  
    Stmt CONSTANT VARCHAR2(4000) :=  
        'SELECT ' || p_colname || ' FROM ' || p_tablename ;  
BEGIN  
    DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);  
    EXECUTE IMMEDIATE Stmt bulk collect into Results;  
    for j in 1..Results.Count() loop  
        DBMS_Output.Put_Line(Results(j));  
    end loop;  
    --EXCEPTION WHEN OTHERS THEN  
        --Raise_Application_Error(-20000, 'Wrong table name');  
END show_col;  
/
```



## Avoiding Injection by Using DBMS\_ASSERT.SIMPLE\_SQL\_NAME (continued)

```
execute show_col('Email','EMPLOYEES');
execute show_col('Email','EMP');
execute show_col('Email','EMPLOYEES where 1=2 union select
Username c1 from All_Users --');
```

```
CREATE OR REPLACE
PROCEDURE show_col2 (p_colname varchar2, p_tablename  varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
    'SELECT ' || dbms_assert.simple_sql_name( p_colname ) || '
FROM ' || dbms_assert.simple_sql_name( p_tablename ) ;

BEGIN
    DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
    EXECUTE IMMEDIATE Stmt bulk collect into Results;
    for j in 1..Results.Count() loop
        DBMS_Output.Put_Line(Results(j));
    end loop;
    --EXCEPTION WHEN OTHERS THEN
        --Raise_Application_Error(-20000, 'Wrong table name');
END show_col2;
/
```

```
execute show_col2('Email','EMPLOYEES');
execute show_col2('Email','EMP');
execute show_col2('Email','EMPLOYEES where 1=2 union select
Username c1 from All_Users --');
```

## DBMS\_ASSERT Guidelines

- Do not perform unnecessary uppercase conversions on identifiers.

```
--Bad:
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(UPPER(MY_SCHEMA));
--Good:
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA);
--Best:
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_NAME(
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_LITERAL(
    sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA));
```

- When using ENQUOTE\_LITERAL, do not add unnecessary double quotes around identifiers.

```
--Bad:
my_trace_routine(' | sys.dbms_assert.ENQUOTE_LITERAL(
my_procedure_name) | ');' | ...
--Good:
my_trace_routine(' | sys.dbms_assert.ENQUOTE_LITERAL(
replace(my_procedure_name, '''', ''''''') | ');' | ...
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## DBMS\_ASSERT Guidelines

Choosing the correct DBMS\_ASSERT verification routines is important, and using them correctly is just as important.

### Limitations

DBMS\_ASSERT is not a panacea for all sorts of PL/SQL evils. It is essentially a collection of pattern-matching routines that confirm whether the supplied string matches expected patterns. It can be used to protect against certain kinds of malicious input, but it cannot comprehensively defend against all such inputs.

Here are some instances where DBMS\_ASSERT may not help:

- It contains no routines to validate TNS connect strings, for example “((description =...”.
- It is not designed nor is it intended to be a defense against cross-site scripting attacks.
- It does not check for input string lengths, and therefore cannot be used as any kind of defense against a buffer overflow attack.
- It does not guarantee that a SQL name is, in fact, a parseable SQL name.
- It does not protect against parsing as the wrong user or other security risks due to inappropriate privilege management.

## DBMS\_ASSERT Guidelines

- Check and reject NULL or empty return results from DBMS\_ASSERT (test for NULL, '', and ' ' ).
- Prefix all calls to DBMS\_ASSERT with the owning schema, SYS.
- Protect all injectable parameters and code paths.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### DBMS\_ASSERT Guidelines (continued)

#### Protect all injectable parameters and code paths.

- **Bad**

```
FUNCTION name_elided
(LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
RETURN BOOLEAN IS
CRS INTEGER;
BEGIN
CRS := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(CRS, 'select ' || FIELD || ' from ' || OWNER || '.' ||
sys.dbms_assert.QUALIFIED_SQL_NAME(LAYER) || '_elided',
DBMS_SQL.NATIVE);
```

- **Good**

```
FUNCTION name_elided
(LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
RETURN BOOLEAN IS
CRS INTEGER;
BEGIN
CRS := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(CRS, 'select
' || sys.dbms_assert.SIMPLE_SQL_NAME(FIELD) || ' from
' || sys.dbms_assert.SIMPLE_SQL_NAME(OWNER) || '.' ||
sys.dbms_assert.SIMPLE_SQL_NAME(LAYER) || '_elided',
DBMS_SQL.NATIVE);
```

## DBMS\_ASSERT Guidelines

- If DBMS\_ASSERT exceptions are raised from a number of input strings, define and raise exceptions explicitly to ease debugging during application development.

```
-- Bad
CREATE OR REPLACE PROCEDURE change_password3
  (username VARCHAR2, password VARCHAR2)
AS
BEGIN
  ...
EXCEPTION WHEN OTHERS THEN
  RAISE;
END;
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### DBMS\_ASSERT Guidelines (continued)

You can use PL/SQL conditional compilation from Oracle Database 10.2 and later to manage self-tracing code. For production deployment, the debug messages can be turned off by setting the PLSQL\_CCFLAGS parameter for tracing to FALSE. You must ensure that error messages displayed in production deployment do not reveal information that is useful to hackers.

```
ALTER SESSION SET Plsql_CCFlags = 'Tracing:true';
```

```
CREATE OR REPLACE PROCEDURE change_password3
  (p_username VARCHAR2, p_password VARCHAR2) AS
BEGIN
  ...
EXCEPTION
  WHEN sys.dbms_assert.INVALID_SCHEMA_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid user.');
```

```
    $else dbms_output.put_line('Authentication failed.');
```

```
    $send
  WHEN sys.dbms_assert.INVALID_SQL_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid pw.');
```

```
    $else dbms_output.put_line('Authentication failed.');
```

```
    $send
  WHEN OTHERS THEN
    dbms_output.put_line('Something else went wrong');
```

```
END;
```

# Lesson Agenda

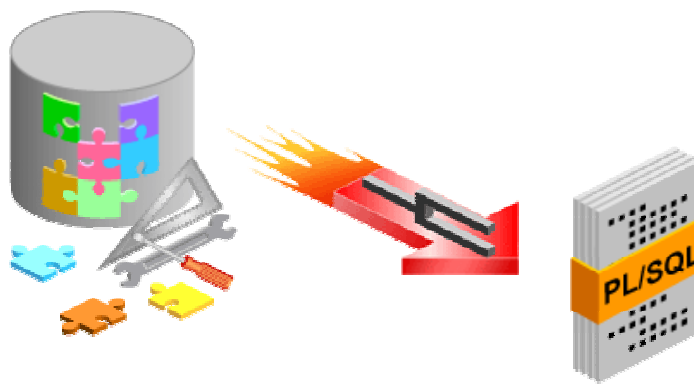
- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- **Designing code immune to SQL injections**
- Testing code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# Using Bind Arguments

- Most common vulnerability:
  - Dynamic SQL with string concatenation
- Your code design must:
  - Avoid input string concatenation in dynamic SQL
  - Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using Bind Arguments

Poor application design can lead to “designed in” vulnerabilities where there are no coding problems *per se* and everything works as intended.

However, you must design your code such that it is (ideally) entirely free of SQL injection vulnerabilities, or contains measures that mitigate the impact of a successful attack.

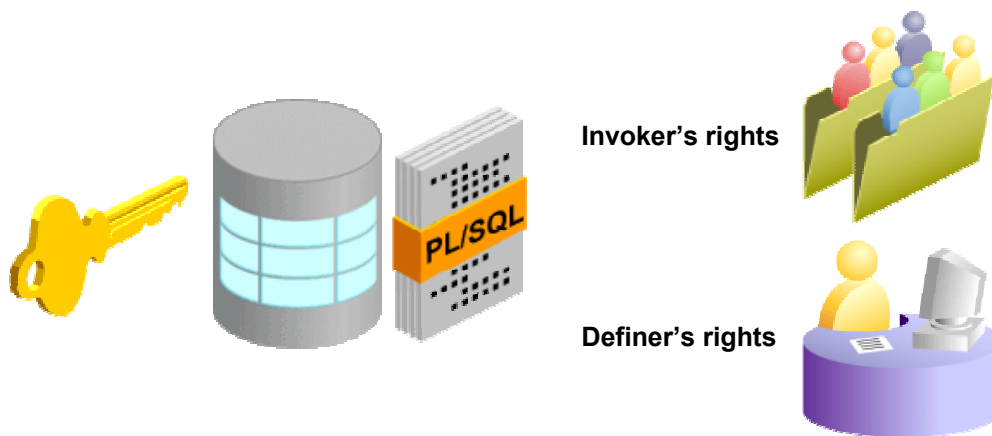
As you have seen with the examples presented thus far, the common flaw of all code vulnerable to SQL injection is the construction of dynamic SQL using string concatenation. Complete immunity from SQL injection attack can be achieved only through the elimination of input string concatenation in dynamic SQL.

- Avoid input string concatenation.
- Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements.

Design your code to use bind arguments wherever possible. The only exceptions should be when you must concatenate identifiers or keywords, because you have no other choice.

# Avoiding Privilege Escalation

- Give out privileges appropriately.
- Run code with invoker's rights when possible.
- Ensure that the database privilege model is upheld when using definer's rights.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Avoiding Privilege Escalation

Unless carefully designed, routines may effectively grant users more privileges than intended. Wherever possible, run code with invoker's rights to minimize the scope for privilege escalation attacks and to mitigate the impact of a successful SQL injection attack.

Where this is not possible, the routines that are run with definer's rights should be carefully reviewed to ensure that the database privilege model is upheld.

If none of the methods of execution (definer's rights, invoker's rights) appears suitable, consider implementing specific bypass checks for the duration of the call.

## Beware of Filter Parameters

- Filter parameter:
  - P\_WHERE\_CLAUSE is a filter.
  - It is difficult to protect against SQL injection.

```
stmt := 'SELECT session_id FROM sessions  
        WHERE' || p_where_clause;
```

- Prevention methods:
  - Do not specify APIs that allow arbitrary query parameters to be exposed.
  - Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Beware of Filter Parameters

Occasionally, routines may require a parameter that is used to form an expression in a query or other PL/SQL statements. Typically such parameters are referred to as “filters.”

How do you test that the provided expression is free from SQL injection? You cannot. In these cases, the only sure solution is prevention.

- Do not specify APIs that allow arbitrary query parameters to be exposed.
- Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.

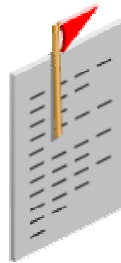
If you must make use of filter parameters, contact your security compliance team for approval.

Any such routines must be registered as possible sinks of dynamic SQL and recorded in static code analysis rulesets. This helps ensure that any users of these APIs are well behaved. Contact your security compliance team to register these routines.



# Trapping and Handling Exceptions

- Design your code to trap and handle exceptions appropriately.
- Before deploying your application:
  - Remove all code tracing
  - Remove all debug messages



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Trapping and Handling Exceptions

Normal SQL injection attacks depend, in large measure, on an attacker's reverse-engineering portions of the original SQL query by using information gained from error messages. Therefore, keep application error messages succinct and do not divulge metadata information (such as column names and table names).

**Note:** From Oracle Database 10.2 and later, you can use PL/SQL conditional compilation for managing self-tracing code, asserts, and so on.

# Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

# Coding Review and Testing Strategy

- Test:
  - Dynamic testing
  - Static testing
- Review:
  - Peer and self reviews
  - Analysis tools



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Coding Review and Testing Strategy

You can use a number of strategies to test for SQL injection vulnerability. Using a combination of these strategies should be regarded as a sensible minimum to get some degree of confidence in freedom from vulnerabilities.

Effective testing of complex products is essentially a process of investigation, and not merely a matter of creating and following routine procedure. Code reviews or walk-throughs are referred to as “static testing,” whereas actually running the program with a given set of test cases in a given development stage is often referred to as “dynamic testing.”

Testing for SQL injection flaws requires both static and dynamic testing. For static testing, you can begin with a peer (or self) code review and/or make use of a static code analysis tool. After finding and fixing the semantical SQL injection bugs, you must perform dynamic testing by using tools that generate random input (fuzzing), and also run through test cases that you define specifically for SQL injection detection within your code.

## Reviewing Code

Language	Look for...
PL/SQL	<code>EXECUTE IMMEDIATE</code> <code>OPEN cursor_var FOR ...</code> <code>DBMS_SQL</code> <code>DBMS_SYS_SQL</code>
C	String substitutions such as: <pre>static const oratext createsq[] = "CREATE SEQUENCE \"%.*s\".\".\"%.*s\" start with %.*s increment by %.*s";</pre> Followed by usage such as: <pre>DISCARD 1stprintf(sql_txt, createsq, owner1, owner, seqnam1, seqnam, sizeof(start), start, sinc by1, sincrement by);</pre>
Java	String concatenations such as: <pre>sqltext = "DROP VIEW " + this.username + "." + this.viewName;</pre>

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

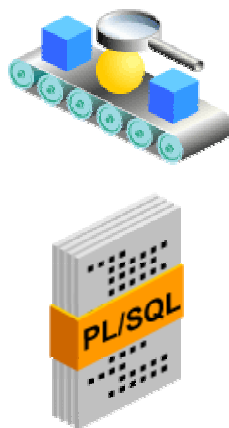
### Reviewing Code

When reviewing code, first identify all dynamic SQL statements. Depending on your programming language, some key indicators that dynamic SQL is in use are shown in the slide.

Next, check to make sure that bind arguments are used in all possible and practical instances. Where bind arguments cannot be used, make sure that the correct routines are used to filter and sanitize inputs.

# Running Static Code Analysis

- Generally performed by an automated tool
- Can be performed on some versions of the source code
- Can be performed on some forms of the object code
- Should be used as one of the initial steps of testing code



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Running Static Code Analysis

Static code analysis is the analysis of computer software that is performed without executing programs built from that software. In most cases, the analysis is performed on some version of the source code and in other cases, some form of the object code. The term is usually applied to the analysis performed by an automated tool.

Because SQL injections arise from dynamically generated SQL, static code analysis tools may have a hard job identifying all categories of SQL injection with some certainty, because this may require knowledge of object names and contents not available at analysis time. For example, a PL/SQL function might construct a string containing parts of a SQL statement that might then be combined with a malicious table name, thereby leading to an injection.

Static code analysis tools should not be used for any kind of security sign-off. Instead, it should be one of the initial steps in the code testing process.

## Testing with Fuzzing Tools

- Is a software testing technique that provides random data (“fuzz”) to the inputs of a program
- Can enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find
- Must not be used as a substitute for exhaustive testing or formal methods
- While tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Testing with Fuzzing Tools

Fuzz testing or fuzzing is a software testing technique that provides random data (“fuzz”) to the inputs of a program. If the program fails (for example, by crashing or by failing built-in code assertions), the defects can be noted. The great advantage of fuzz testing is that the test design is extremely simple and free of preconceptions about system behavior.

Fuzz testing is thought to enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find, and even careful human test designers would fail to create tests for.

However, fuzz testing is not a substitute for exhaustive testing or formal methods; it can only provide a random sample of the system’s behavior, and in many cases passing a fuzz test may only demonstrate that a piece of software handles exceptions without crashing, rather than behaving correctly. Thus, fuzz testing can be regarded only as a bug-finding tool rather than an assurance of quality.

While tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools.

## Generating Test Cases

- Test each input parameter individually.
- When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.
- Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.
- Always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.
- Certain object and schema names help you uncover SQL injection vulnerabilities.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Generating Test Cases

When designing SQL injection test cases, keep in mind that each input parameter must be tested individually. When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.

It can be tempting to delete everything that you are not working with to make things appear simpler, particularly with applications that have parameter lines going into many thousands of characters. Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.

So, when testing for SQL injection, always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.

Certain object and schema names help you uncover SQL injection vulnerabilities. On the next page is a list of names as input values in your tests.

## Generating Test Cases (continued)

Name	Test
AAAAAAAAAABBBB BBBBBBCCCCCCCC	Maximum size identifiers should be tested as both schema and object names. This checks that temporary object names can be created.
AAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAA AA & AAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAA A	Maximum and “nearly” maximum object names should be handled as separate objects. This checks that any name truncation operation does not result in non-unique names.
schema1.a schema2.a	Ensures that objects with the same name in different schemas are properly differentiated
AAA "aaa"	Ensures that objects with different case names are clearly differentiated
object@dblink	Ensures that link names (both valid and invalid) do not defeat security tests. Ensure that links are rejected unless specifically allowable.
Rename "ABC" to "XYZ"	Renames objects during tests to ensure that applications continue to function correctly
"quoted"	Ensures that quoted schema and object names can be handled
"AAA" " AAA" "AAA "	Ensures that quoted objects do not lead to second-order injections
"\$IF"	Refers to a variable and not a preprocessor directive
"BEGIN"	Refers to an object and not a reserved word
"A%TYPE"	Refers to an object and not an object attribute
"a/*X*/ <<A>>, %s<img>-- %00\0AAA ", "a' ' ' '   */ . @ \ ? : x & y { } ( ) ( > < ) [ ! = ; A"	Ensures that special characters in quoted variables do not have other side effects
"xxx"   chr (8)   "yy", "xxx"   chr (9)   "yy", "xxx"   chr (10)   "yy", "xxx"   chr (13)   "yy", "xxx"   chr (4)   "yy", "xxx"   chr (26)   "yy", "xxx"   chr (0)   "yy"	Ensures that object names with embedded control characters are handled correctly, and checks that all output correctly distinguishes these objects, especially any output written to files. Note that some object names may be (correctly) rejected as invalid syntax.
NULL	Null and empty strings may behave differently on different Oracle Database ports or versions.



## Quiz

Code that is most vulnerable to SQL Injection attack contains:

- a. Input parameters
- b. Dynamic SQL with bind arguments
- c. Dynamic SQL with concatenated input values
- d. Calls to external functions

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: c**

## Quiz

By default, a stored procedure executes with the privileges of its owner (definer's rights).

- a. True
- b. False

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## Quiz

If you must use dynamic SQL, avoid using input concatenation to build the dynamic SQL.

- a. True
- b. False

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## Quiz

In the statement, `SELECT total FROM orders WHERE ord_id=p_ord_id`, the table name `orders` is being used as which of the following ?

- a. A literal
- b. An identifier
- c. A placeholder
- d. An argument

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

**Answer: b**

## Summary

In this lesson, you should have learned how to:

- Detect SQL injection vulnerabilities
- Reduce attack surfaces
- Use `DBMS_ASSERT`
- Design immune code
- Test code for SQL injection flaws

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Summary

This lesson showed you techniques and tools to strengthen your code and applications against SQL injection attacks.

## Practice 12: Overview

This practice covers the following topics:

- Testing your knowledge of SQL injection
- Rewriting code to protect against SQL injection

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Practice 12: Overview

Using the OE, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

---

# **Appendix A**

## **Practices and Solutions**

---

# Table of Contents

Practices for Lesson 1 .....	3
Practice 1-1: Introduction .....	4
Practice Solutions 1-1: Introduction .....	7
Practices for Lesson 2 .....	19
Practice 2-1: PL/SQL Knowledge Quiz.....	20
Practice Solutions 2-1: PL/SQL Knowledge Quiz.....	22
Practices for Lesson 3 .....	27
Practice 3-1: Designing PL/SQL Code .....	28
Practice Solutions 3-1: Designing PL/SQL Code .....	29
Practices for Lesson 4 .....	32
Practice 4-1: Analyzing Collections .....	33
Practice Solutions 4-1: Analyzing Collections .....	37
Practices for Lesson 5 .....	45
Practice 5-1: Working with LOBs .....	46
Practice 5-2: Working with SecureFile LOBs .....	48
Practice Solutions 5-1: Working with LOBs .....	52
Practice Solutions 5-2: Working with SecureFile LOBs .....	57
Practices for Lesson 6 .....	65
Practice 6-1: Using Advanced Interface Methods .....	66
Practice Solutions 6-1: Using Advanced Interface Methods .....	68
Practices for Lesson 7 .....	72
Practice 7-1: Performance and Tuning .....	73
Practice Solutions 7-1: Performance and Tuning .....	79
Practices for Lesson 8 .....	89
Practice 8-1: Improving Performance with Caching .....	90
Practice Solutions 8-1: Improving Performance with Caching .....	92
Practices for Lesson 9 .....	96
Practice 9-1: Analyzing PL/SQL Code.....	97
Practice Solutions 9-1: Analyzing PL/SQL Code.....	99
Practices for Lesson 10 .....	106
Practice 10-1: Profiling and Tracing PL/SQL Code .....	107
Practice Solutions 10-1: Profiling and Tracing PL/SQL Code .....	108
Practices for Lesson 11 .....	112
Practice 11-1: Implementing Fine-Grained Access Control for VPD .....	113
Practice Solutions 11-1: Implementing Fine-Grained Access Control for VPD .....	116
Practices for Lesson 12 .....	121
Practice 12-1: Safeguarding Your Code Against SQL Injection Attacks .....	122
Practice Solutions 12-1: Safeguarding Your Code Against SQL Injection Attacks...	124



---

## Practices for Lesson 1

---

In this practice, you review the available SQL Developer resources. You also learn about the user account that you use in this course. You start SQL Developer, create a new database connection, and browse your SH, HR, and OE tables. You also execute SQL statements, and access and bookmark the Oracle Database 11g documentation and other useful Web sites that you can use in this course.

## **Practice 1-1: Introduction**

### **Identifying the Available SQL Developer Resources**

- 1) Familiarize yourself with Oracle SQL Developer, as needed, by referring to “Appendix C: Using SQL Developer.”
- 2) Access the SQL Developer Home page at [http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html)
- 3) Bookmark the page for easier future access.
- 4) Access the SQL Developer tutorial at <http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>.
- 5) Preview and experiment with the available links and demonstrations in the tutorial as needed, especially the “Creating a Database Connection” and Accessing Data links.

### **Creating and Using the New SQL Developer Database Connections**

- 6) Start SQL Developer.
- 7) Create a database connection to SH using the following information:
  - a) Connection Name: `sh_connection`
  - b) Username: `sh`
  - c) Password: `sh`
  - d) Hostname: `localhost`
  - e) Port: `1521`
  - f) SID: `orcl`
- 8) Test the new connection. If the Status is Success, use this new connection to connect to the database.
  - a) Double-click the `sh_connection` icon on the Connections tabbed page.
  - b) Click the Test button in the New/Select Database Connection window. If the status is Success, click the Connect button.
- 9) Create a new database connection named `hr_connection`.
  - a) Right-click the `sh_connection` connection in the Object Navigation tree, and select the Properties menu option.
  - b) Enter `hr_connection` as the connection name and `hr` as the username and password, and click Save. This creates the new connection.
  - c) Repeat step 8 to test the new `hr_connection` connection.
- 10) Repeat step 9 to create and test a new database connection named `oe_connection`. Enter `oe` as the database connection username and password.

## ***Practice 1-1: Introduction (continued)***

- 11) Repeat step 9 to create and test a new database connection named `sys_connection`. Enter `sys` as the database connection username, `oracle` as the password, and `SYSDBA` as the role.

### **Browsing the HR, SH, and OE Schema Tables**

- 12) Browse the structure of the `EMPLOYEES` table in the HR schema.
- a) Expand the `hr_connection` connection by clicking the plus symbol next to it.
  - b) Expand the Tables icon by clicking the plus symbol next to it.
  - c) Display the structure of the `EMPLOYEES` table.
- 13) Browse the `EMPLOYEES` table and display its data.
- 14) Use the SQL Worksheet to select the last names and salaries of all employees whose annual income is greater than \$10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements on the appropriate tabs.

**Note:** Take a few minutes to familiarize yourself with the data, or consult “Appendix B: Table Descriptions and Data,” which provides the description and data for all tables in the HR, SH, and OE schemas that you will use in this course.

- 15) Create and execute a simple anonymous block that outputs “Hello World.”
- a) Enable `SET SERVEROUTPUT ON` to display the output of the `DBMS_OUTPUT` package statements.
  - b) Use the SQL Worksheet area to enter the code for your anonymous block.
  - c) Click the Run Script icon (F5) to run the anonymous block.
- 16) Browse the structure of the `SALES` table in the SH schema and display its data.
- a) Double-click the `sh_connection` connection.
  - b) Expand the Tables icon by clicking the plus symbol next to it.
  - c) Display the structure of the `SALES` table.
  - d) Browse the `SALES` table and display its data.
- 17) Browse the structure of the `ORDERS` table in the OE schema and display its data.
- a) Double-click the `oe_connection` database connection.
  - b) Expand the Tables icon by clicking the plus symbol next to it.
  - c) Display the structure of the `ORDERS` table.
  - d) Browse the `ORDERS` table and display its data.

### **Accessing the Oracle Database 11g Release 2 Online Documentation Library**

- 18) Access the Oracle Database 11g Release documentation Web page at <http://www.oracle.com/pls/db111/homepage>

### ***Practice 1-1: Introduction (continued)***

- 19) Bookmark the page for easier future access.
- 20) Display the complete list of books available for Oracle Database 11g, Release 2.
- 21) Make a note of the following documentation references that you will use in this course:
  - *Advanced Application Developer's Guide*
  - *New Features Guide*
  - *PL/SQL Language Reference*
  - *Oracle Database Reference*
  - *Oracle Database Concepts*
  - *SQL Developer User's Guide*
  - *SQL Language Reference Guide*
  - *SQL\*Plus User's Guide and Reference*

## Practice Solutions 1-1: Introduction

- 1) Familiarize yourself with Oracle SQL Developer, as needed, by referring to “Appendix C: Using Oracle SQL Developer.”
- 2) Access the online Oracle SQL Developer Home page at [http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html)

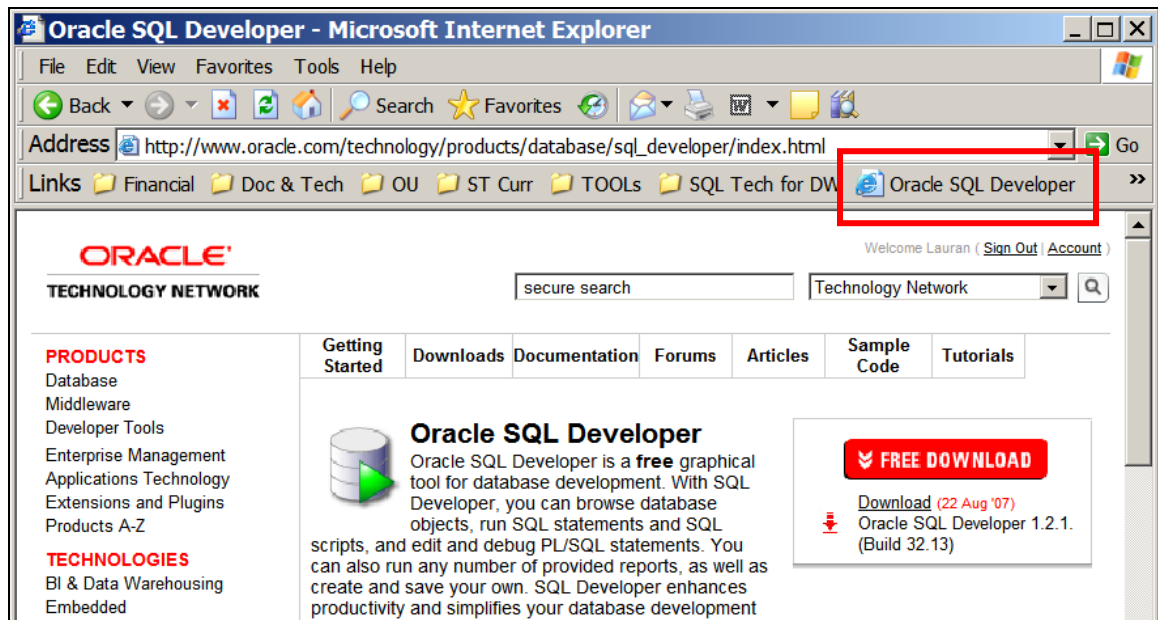
The Oracle SQL Developer Home page appears:



- 3) Bookmark the page for easier future access.

On the Windows Internet Explorer Address toolbar, click and drag the Explorer icon onto the Links toolbar. The link is added to your Links toolbar:

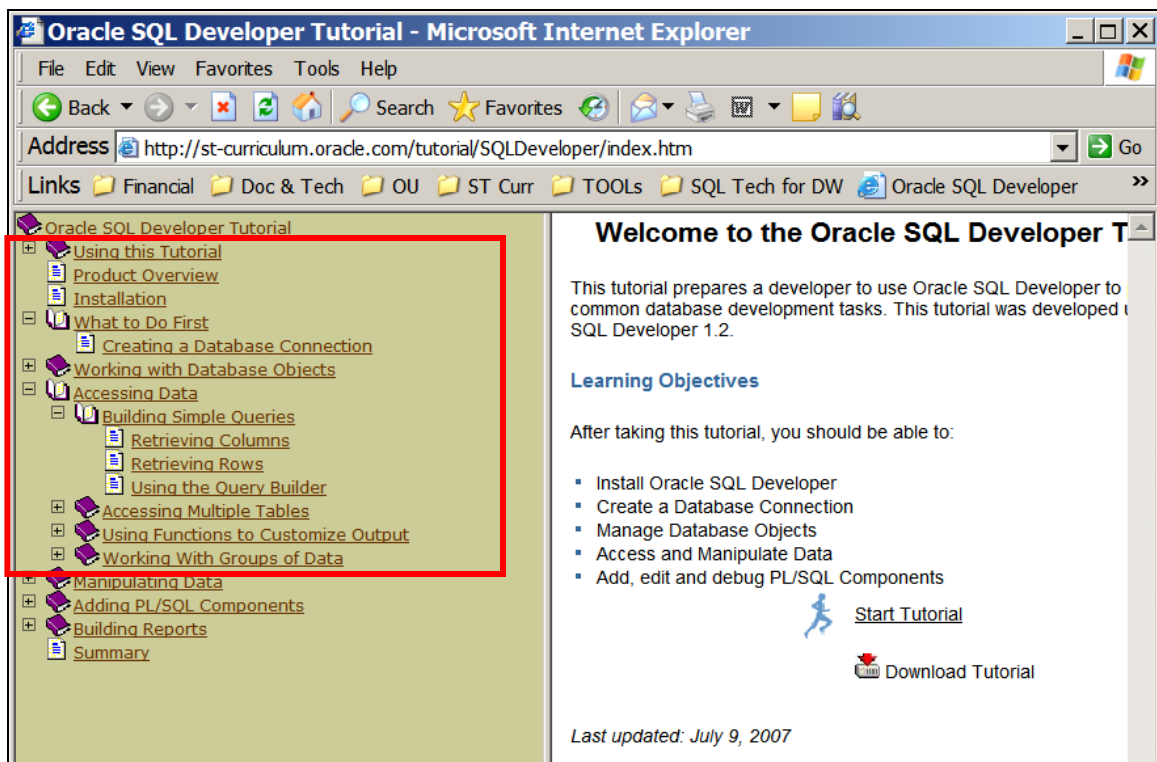
## Practice Solutions 1-1: Introduction (continued)



- 4) Access the Oracle SQL Developer tutorial at

<http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>

Access the Oracle SQL Developer tutorial by using the preceding URL. The following page appears:



## ***Practice Solutions 1-1: Introduction (continued)***

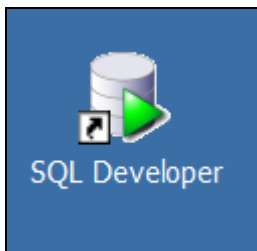
- 5) Preview and experiment with the available links and demonstrations in the tutorial as needed, especially the “Creating a Database Connection” and Accessing Data links.

**To review the section on creating a database connection, click the plus sign (+) next to the “What to Do First” link to display the “Creating a Database Connection” link. To review the “Creating a Database Connection” topic, click the topic’s link. To review the section on accessing data, click the plus sign (+) next to the Accessing Data link to display the list of available topics. To review any of the topics, click the topic’s link.**

### **Creating and Using a New Oracle SQL Developer Database Connection**

- 6) Start Oracle SQL Developer.

**Click the Oracle SQL Developer icon on your desktop.**



- 7) Create a database connection to SH schema using the following information:

- a) Connection Name: `sh_connection`
- b) Username: `sh`
- c) Password: `sh`
- d) Hostname: `localhost`
- e) Port: `1521`
- f) SID: `orcl`

**Right-click the Connections icon on the Connections tabbed page, and then select the New Connection option from the shortcut menu. The New/Select Database Connection window appears. Use the preceding information to create the new database connection.**

**Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Enter the username, password, host name, and service name with the appropriate information, as provided above. The following is a sample of the newly created database connection for the SH schema using a local connection:**

## Practice Solutions 1-1: Introduction (continued)

The screenshot shows the 'New / Select Database Connection' dialog box. On the left is a list of connections. The main area contains the following fields and options:

- Connection Name:** sh\_connection
- Username:** sh
- Password:** \*\*
- ☐ Save Password
- Database Type:** Oracle, Access (selected), MySQL, SQLServer
- Role:** default (dropdown)
- Connection Type:** ☒ Basic, ☐ TNS, ☐ Advanced
- Hostname:** localhost
- Port:** 1521
- ☒ SID: orcl
- ☐ Service name: (empty)
- Status:** (empty)

Buttons at the bottom: Help, Save, Clear, Test, Connect, Cancel.

- 8) Test the new connection. If the Status is Success, use this new connection to connect to the database.
  - a) Double-click the sh\_connection icon on the Connections tabbed page.
  - b) Click the Test button in the New/Select Database Connection window. If the status is Success, click the Connect button.



## Practice Solutions 1-1: Introduction (continued)

The screenshot shows the 'New / Select Database Connection' dialog box. The 'Connection Name' field is set to 'sh\_connection'. The 'Username' field is set to 'sh' and the 'Password' field is set to '\*\*'. The 'Save Password' checkbox is unchecked. The 'Database' tab is selected, showing 'Oracle', 'Access', 'MySQL', and 'SQLServer' buttons. The 'Role' dropdown is set to 'default'. The 'Connection Type' is set to 'Basic'. The 'Hostname' field is set to 'localhost', the 'Port' field is set to '1521', and the 'SID' radio button is selected. The 'Service name' radio button is also visible. The 'Status: Success' message is displayed in the bottom left. The 'Test' button is highlighted with a red box.

- 9) Create a new database connection named `hr_connection`.
  - a) Right-click the `sh_connection` connection in the Object Navigation tree, and select the Properties menu option.
  - b) Enter `hr_connection` as the connection name and `hr` as the username and password, and then click Save to create the new connection.
  - c) Repeat step 8 to test the new `hr_connection` connection.

## Practice Solutions 1-1: Introduction (continued)

The screenshot shows the 'New / Select Database Connection' dialog box. On the left, there is a list of existing connections with two visible: 'Connectio...' and 'sh\_conne...'. The main area contains the following fields and options:

- Connection Name:** hr\_connection
- Username:** hr
- Password:** \*\*
- ☐ Save Password
- Database Type:** Oracle, Access, MySQL, SQLServer (Access is selected)
- Role:** default
- Connection Type:** Basic (selected), TNS, Advanced
- Hostname:** localhost
- Port:** 1521
- ☒ SID: orcl
- ☐ Service name

At the bottom, the status is 'Status : Success'. There are buttons for Help, Save, Clear, Test, Connect, and Cancel.

- 10) Repeat step 9 to create and test a new database connection named `oe_connection`. Enter `oe` as the database connection username and password.

## Practice Solutions 1-1: Introduction (continued)

New / Select Database Connection

Connectio...	Connectio...
hr_conne...	hr@//local...
oe_conne...	oe@//loca...
sh_conne...	sh@//loca...

Connection Name: oe\_connection

Username: oe

Password: \*\*

☐ Save Password

Oracle Access MySQL SQLServer

Role: default

Connection Type: ☒ Basic ☐ TNS ☐ Advanced

Hostname: localhost

Port: 1521

☒ SID

☐ Service name

orcl

Status : Success

Help Save Clear Test Connect Cancel

- 11) Repeat step 9 to create and test a new database connection named `sys_connection`. Enter `sys` in the Username field, `oracle` in the Password field, and `SYSDBA` as the role.

## Practice Solutions 1-1: Introduction (continued)

The screenshot shows the 'New / Select Database Connection' dialog box. On the left, there is a list of existing connections: 'hr\_connection' (hr@Mocalho...) and 'oe\_connection' (oe@Mocalho...). The main form fields are as follows:

- Connection Name:** sys\_connection
- Username:** sys
- Password:** masked with asterisks
- ☐ Save Password
- Database Type:** Oracle (selected), Access, MySQL, SQLServer
- Role:** SYSDBA (highlighted with a red box)
- Connection Type:** Basic (selected), TNS, Advanced
- Hostname:** localhost
- Port:** 1521
- ☒ SID: orcl
- ☐ Service name: (empty)

At the bottom, the status is 'Status : Success'. There are buttons for Help, Save, Clear, Test (highlighted with a yellow border), Connect, and Cancel.

**From the Role drop-down list, select SYSDBA as the role.**

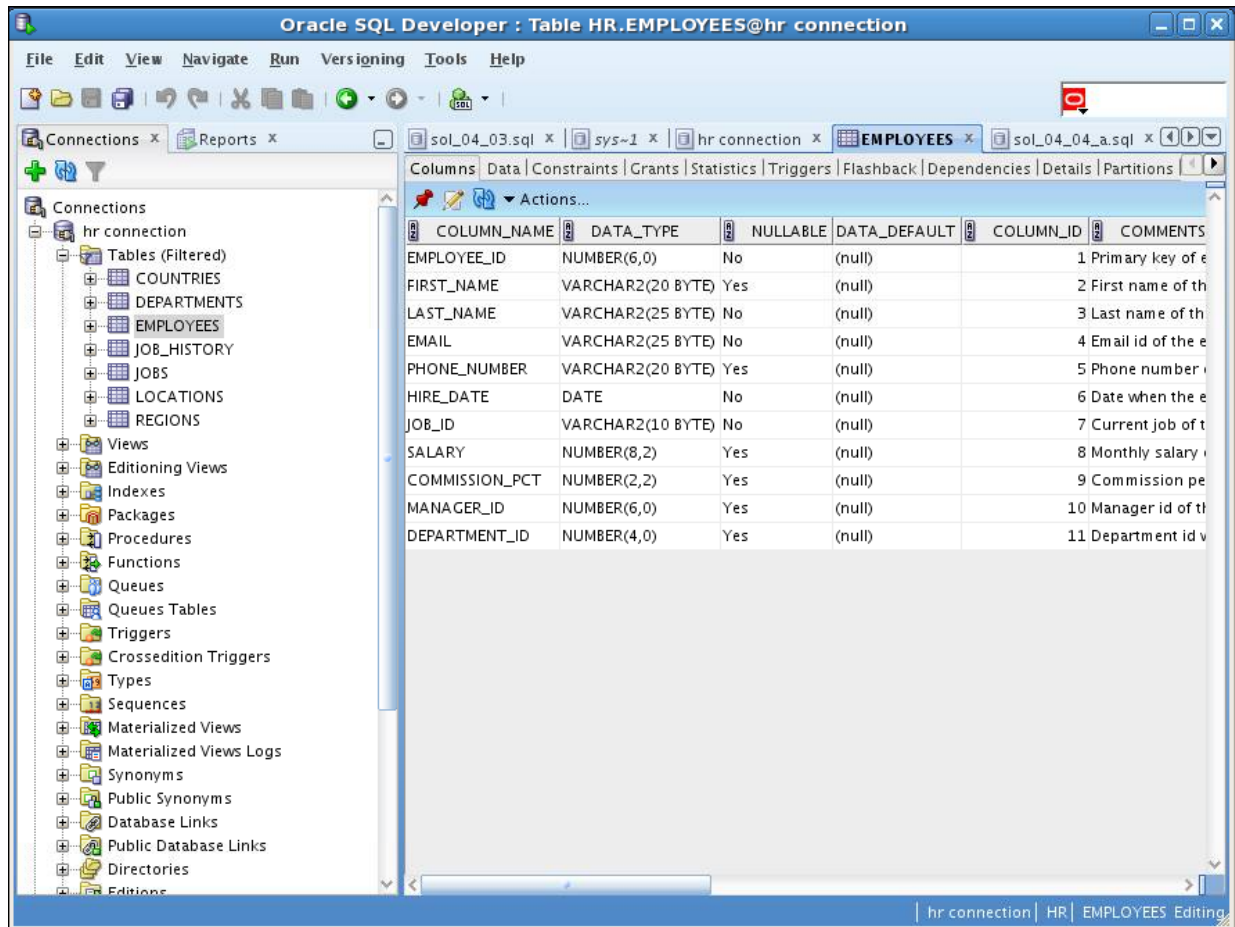
### Browsing the HR, SH, and OE Schema Tables

12) Browse the structure of the EMPLOYEES table in the HR schema.

- Expand the hr\_connection connection by clicking the plus sign next to it.
- Expand the Tables icon by clicking the plus sign next to it.
- Display the structure of the EMPLOYEES table.

**Double-click the EMPLOYEES table. The Columns tab displays the columns in the EMPLOYEES table:**

## Practice Solutions 1-1: Introduction (continued)



13) Browse the `EMPLOYEES` table and display its data.

**To display the employee data, click the Data tab. The `EMPLOYEES` table data is displayed:**

## Practice Solutions 1-1: Introduction (continued)

Oracle SQL Developer : Table HR.EMPLOYEES@hr connection

FileEditViewNavigateRunVersingToolsHelp

Connections x Reports x

hr connection

- Tables (Filtered)
  - COUNTRIES
  - DEPARTMENTS
  - EMPLOYEES
  - JOB\_HISTORY
  - JOBS
  - LOCATIONS
  - REGIONS
- Views
- Editing Views
- Indexes
- Packages
- Procedures
- Functions
- Queues
- Queues Tables
- Triggers
- Crossedition Triggers
- Types
- Sequences
- Materialized Views
- Materialized Views Logs
- Synonyms
- Public Synonyms
- Database Links
- Public Database Links
- Directories
- Editions

ColumnsDataConstraintsGrantsStatisticsTriggersFlashbackDependenciesDetailsPartitions

EMPLOYEE\_IDFIRST\_NAMELAST\_NAMEEMAILPHONE\_NUMBERHIRE\_DATE

1	198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-07
2	199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-08
3	200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-03
4	201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-04
5	202	Pat	Fay	PFAY	603.123.6666	17-AUG-05
6	203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-02
7	204	Hermann	Baer	HBAER	515.123.8888	07-JUN-02
8	205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-02
9	206	William	Gietz	WGIEZT	515.123.8181	07-JUN-02
10	100	Steven	King	SKING	515.123.4567	17-JUN-03
11	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-05
12	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-01
13	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-06
14	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-07
15	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-05
16	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-06
17	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-07
18	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-02
19	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-02
20	110	John	Chen	JCHEN	515.124.4269	28-SEP-05
21	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-05
22	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-06
23	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-07

- 14) Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than \$10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements on the appropriate tabs.

**Note:** Take a few minutes to familiarize yourself with the data, or consult “Appendix B: Table Descriptions and Data,” which provides the description and data for all tables in the HR, SH, and OE schemas that you will use in this course.

**Display the SQL Worksheet by using either of the following two methods: Select Tools > SQL Worksheet, or click the Open SQL Worksheet icon. The Select Connection window appears. Enter the following statement in the SQL Worksheet:**

```
SELECT *  
FROM employees  
WHERE SALARY > 10000;
```

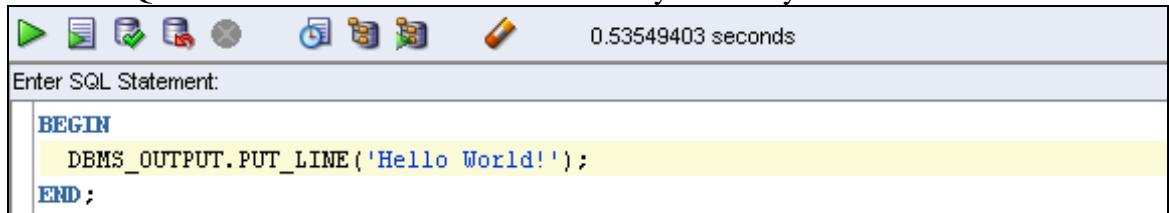
## Practice Solutions 1-1: Introduction (continued)

15) Create and execute a simple anonymous block that outputs “Hello World.”

- a) Enable SET SERVEROUTPUT ON to display the output of the DBMS\_OUTPUT package statements.

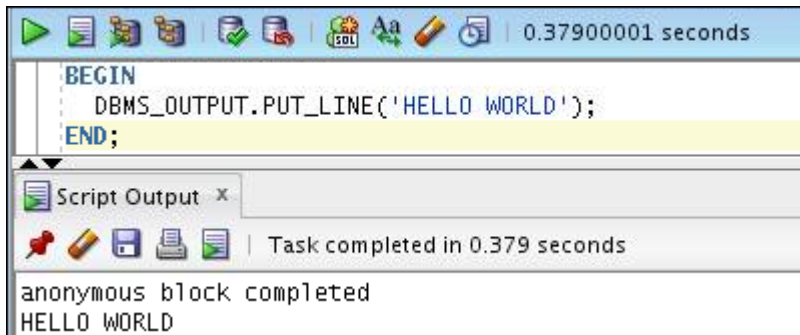


- b) Use the SQL Worksheet area to enter the code for your anonymous block.



- c) Click the Run Script icon (F5) to run the anonymous block.

**The Script Output tab displays the output of the anonymous block:**



16) Browse the structure of the SALES table in the SH schema and display its data.

- a) Double-click the sh\_connection database connection.
- b) Expand the Tables icon by clicking the plus sign next to it.
- c) Display the structure of the SALES table.
- d) Browse the SALES table and display its data.

**Double-click the SALES table. The Columns tab displays the columns in the SALES table. To display the sales data, click the Data tab. The SALES table data is displayed.**

17) Browse the structure of the ORDERS table in the OE schema and display its data.

- a) Double-click the oe\_connection database connection.
- b) Expand the Tables icon by clicking the plus sign next to it.
- c) Display the structure of the ORDERS table.
- d) Browse the ORDERS table and display its data.

## ***Practice Solutions 1-1: Introduction (continued)***

Double-click the **ORDERS** table. The **Columns** tab displays the columns in the **ORDERS** table. To display the order data, click the **Data** tab. The **ORDERS** table data is displayed.

### **Accessing the Oracle Database 11g, Release 2 Online Documentation Library**

18) Access the Oracle Database 11g Release documentation Web page at:

<http://www.oracle.com/pls/db111/homepage>

19) Bookmark the page for easier future access.

20) Display the complete list of books available for Oracle Database 11g, Release 2.

21) Make a note of the following documentation references that you will use in this course as needed:

- *Oracle Database 2 Day + Data Warehousing Guide 11g Release 2 (11.2)*
- *Oracle Database Data Warehousing Guide 11g Release 2 (11.2)*
- *Oracle Database SQL Developer User's Guide Release 1.2*
- *Oracle Database Reference 11g Release 2 (11.2)*
- *Oracle Database New Features Guide 11g Release 2 (11.2)*
- *Oracle Database SQL Language Reference 11g Release 2 (11.2)*
- *SQL\*Plus User's Guide and Reference Release 11.2*
- *Oracle Database SQLJ Developer's Guide and Reference 11g Release 2 (11.2)*
- *Oracle Database Concepts 11g Release 2 (11.2)*
- *Oracle Database Sample Schemas 11g Release 2 (11.2)*



---

## Practices for Lesson 2

---

In this practice, you test and review your PL/SQL knowledge. This knowledge is necessary as a baseline for the subsequent chapters to build upon.

## **Practice 2-1: PL/SQL Knowledge Quiz**

**Note:** If you do not know an answer, go on to the next question.

### **PL/SQL Basics**

- 1) Which are the four key areas of the basic PL/SQL block? What happens in each area?
- 2) What is a variable and where is it declared?
- 3) What is a constant and where is it declared?
- 4) What are the different modes for parameters and what does each mode do?
- 5) How does a function differ from a procedure?
- 6) Which are the two main components of a PL/SQL package?
  - a) In what order are they defined?
  - b) Are both required?
- 7) How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL\*Plus?
- 8) What is a record?
- 9) What is an index by table?
- 10) How are loops implemented in PL/SQL?
- 11) How is branching logic implemented in PL/SQL?

### **Cursor Basics**

- 12) What is an explicit cursor?
- 13) Where do you define an explicit cursor?
- 14) Name the five steps for using an explicit cursor.
- 15) What is the syntax used to declare a cursor?
- 16) What does the `FOR UPDATE` clause do within a cursor definition?
- 17) Which command opens an explicit cursor?
- 18) Which command closes an explicit cursor?
- 19) Name five implicit actions that a cursor `FOR` loop provides.
- 20) Describe what the following cursor attributes do:
  - `cursor_name%ISOPEN`
  - `cursor_name%FOUND`
  - `cursor_name%NOTFOUND`
  - `cursor_name%ROWCOUNT`

## ***Practice 2-1: PL/SQL Knowledge Quiz (continued)***

### **Exceptions**

- 21) An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?
- 22) An exception handler is mandatory within a PL/SQL subprogram. (True/False)
- 23) What syntax do you use in the exception handler area of a subprogram?
- 24) How do you code for a NO\_DATA\_FOUND error?
- 25) Name three types of exceptions.
- 26) To associate an exception identifier with an Oracle error code, what pragma do you use and where?
- 27) How do you explicitly raise an exception?
- 28) What types of exceptions are implicitly raised?
- 29) What does the RAISE\_APPLICATION\_ERROR procedure do?

### **Dependencies**

- 30) Which objects can a procedure or function directly reference?
- 31) Which are the two statuses that a schema object can have and where are they recorded?
- 32) The Oracle server automatically recompiles invalid procedures when they are called from the same \_\_\_\_\_. To avoid compile problems with remote database calls, you can use the \_\_\_\_\_ model instead of the timestamp model.
- 33) Which data dictionary contains information on direct dependencies?
- 34) What script would you run to create the deptree and ideptree views?
- 35) What does the deptree\_fill procedure do and what are the arguments that you must provide?

### **Oracle-Supplied Packages**

- 36) What does the dbms\_output package do?
- 37) How do you write “This procedure works.” from within a PL/SQL program by using dbms\_output?
- 38) What does dbms\_sql do and how does this compare with Native Dynamic SQL?

## **Practice Solutions 2-1: PL/SQL Knowledge Quiz**

**Note:** If you do not know an answer, go on to the next question.

### **PL/SQL Basics**

1) What are the four key areas of the basic PL/SQL block? What happens in each area?

- **Header section:** Names the program unit and identifies it as a procedure, function, or package; also identifies any parameters that the code may use
- **Declarative section:** Area used to define variables, constants, cursors, and exceptions; starts with the keyword **IS** or **AS**
- **Executable section:** Main processing area of the PL/SQL program; starts with the keyword **BEGIN**
- **Exception handler section:** Optional error handling section; starts with the keyword **EXCEPTION**

2) What is a variable and where is it declared?

**Variables are used to store data during PL/SQL block execution. You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the NOT NULL constraint on the variable.**

**Syntax:** `variable_name datatype[(size)][:= initial_value];`

3) What is a constant and where is it declared?

**Constants are variables that never change. Constants are declared and assigned a value in the declarative section, before the executable section.**

**Syntax:** `constant_name CONSTANT datatype[(size)] := initial_value;`

4) What are the different modes for parameters and what does each mode do?

**There are three parameter modes: IN, OUT, and IN OUT. IN is the default and it means that a value is passed into the subprogram. The OUT mode indicates that the subprogram is passing a value generated in the subprogram out to the calling environment. The IN OUT mode means that a value is passed into the subprogram. The subprogram may change the value and pass the value out to the calling environment.**

5) How does a function differ from a procedure?

**A function must execute a RETURN statement that returns a value. Functions are called differently than procedures. They are called as an expression embedded within another command. Procedures are called as statements.**

6) What are the two main components of a PL/SQL package?

**The package body and the package specification**

a) In what order are they defined?

## ***Practice Solutions 2-1: PL/SQL Knowledge Quiz (continued)***

First the package specification, and then the package body

b) Are both required?

**No, only a package specification is required. A specification can exist without a body, but a body cannot exist as valid without the specification.**

7) How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL\*Plus?

**The `INTO` clause is required with a `SELECT` statement that is in a PL/SQL subprogram.**

8) What is a record?

**A record is a composite type that has internal components, which can be manipulated individually. Use the `RECORD` data type to treat related but dissimilar data as a logical unit.**

9) What is an index-by table?

**Index-by tables are a data structure declared in a PL/SQL block. It is similar to an array and comprises two components—the index and the data field. The data field is a column of a scalar or record data type, which stores the `INDEX BY` table elements.**

10) How are loops implemented in PL/SQL?

**Looping constructs are used to repeat a statement or sequence of statements multiple times. PL/SQL has three looping constructs:**

- Basic loops that perform repetitive actions without overall conditions
- `FOR` loops that perform iterative control of actions based on a count
- `WHILE` loops that perform iterative control of actions based on a condition

11) How is branching logic implemented in PL/SQL?

**You can change the logical flow of statements within the PL/SQL block with a number of control structures. Branching logic is implemented within PL/SQL by using the conditional `IF` statement or `CASE` expressions.**

### **Cursor Basics**

12) What is an explicit cursor?

**The Oracle server uses work areas, called private SQL areas, to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. Use explicit cursors to individually process each row returned by a multiple-row `SELECT` statement.**

13) Where do you define an explicit cursor?

**A cursor is defined in the declarative section.**

14) Name the five steps for using an explicit cursor.

## ***Practice Solutions 2-1: PL/SQL Knowledge Quiz (continued)***

**Declare, Open, Fetch, Test for existing rows, and Close**

15) What is the syntax used to declare a cursor?

**CURSOR cursor\_name IS SELECT\_statement**

16) What does the FOR UPDATE clause do within a cursor definition?

**The FOR UPDATE clause locks the rows selected in the SELECT statement definition of the cursor.**

17) What command opens an explicit cursor?

**OPEN cursor\_name;**

18) What command closes an explicit cursor?

**CLOSE cursor\_name;**

19) Name five implicit actions that a cursor FOR loop provides.

**Declares a record structure to match the select list of the cursor; opens the cursor; fetches from the cursor; exits the loop when the fetch returns no row; and closes the cursor**

20) Describe what the following cursor attributes do:

- **%ISOPEN:** Returns a Boolean value indicating whether the cursor is open
- **%FOUND:** Returns a Boolean value indicating whether the last fetch returned a value
- **%NOTFOUND:** Returns a Boolean value indicating whether the last fetch did not return a value
- **%ROWCOUNT:** Returns an integer indicating the number of rows fetched so far

### **Exceptions**

21) An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?

**Control is passed to the exception handler. If the exception is handled in the inner block, processing continues to the outer block. If the exception is not handled in the inner block, an exception is raised in the outer block and control is passed to the exception handler of the outer block. If neither the inner nor the outer block traps the exception, the program ends unsuccessfully.**

22) An exception handler is mandatory within a PL/SQL subprogram. (True/False)

**False**

## ***Practice Solutions 2-1: PL/SQL Knowledge Quiz (continued)***

23) What syntax do you use in the exception handler area of a subprogram?

```
EXCEPTION
    WHEN named_exception THEN
        statement[s];
    WHEN others THEN
        statement[s];
END;
```

24) How do you code for a NO\_DATA\_FOUND error?

```
EXCEPTION
    WHEN no_data_found THEN
        statement[s];
END;
```

25) Name three types of exceptions.

**User-defined, Oracle server predefined, and Oracle server non-predefined**

26) To associate an exception identifier with an Oracle error code, what pragma do you use and where?

**Use the PRAGMA EXCEPTION\_INIT and place the PRAGMA EXCEPTION\_INIT in the declarative section.**

27) How do you explicitly raise an exception?

**Use the RAISE statement or the RAISE\_APPLICATION\_ERROR procedure.**

28) What types of exceptions are implicitly raised?

**All Oracle server exceptions (predefined and non-predefined) are automatically raised.**

29) What does the RAISE\_APPLICATION\_ERROR procedure do?

**It enables you to issue user-defined error messages from subprograms.**

### **Dependencies**

30) Which objects can a procedure or function directly reference?

**Table, view, sequence, procedure, function, package specification, object specification, and collection type**

31) What are the two statuses that a schema object can have and where are they recorded?

**The user\_objects dictionary view contains a column called status. Its values are VALID and INVALID.**

32) The Oracle server automatically recompiles invalid procedures when they are called from the same \_\_\_\_\_. To avoid compile problems with remote database calls, you can use the \_\_\_\_\_ model instead of the timestamp model.

## ***Practice Solutions 2-1: PL/SQL Knowledge Quiz (continued)***

**database**

**signature**

33) Which data dictionary contains information on direct dependencies?

**user\_dependencies**

34) Which script do you run to create the views `deptree` and `ideptree`?

**You use the `utldtree.sql` script.**

35) What does the `deptree_fill` procedure do and what are the arguments that you must provide?

**The `deptree_fill` procedure populates the `deptree` and `ideptree` views to display a tabular representation of all dependent objects, direct and indirect. You pass the object type, object owner, and object name to the `deptree_fill` procedure.**

### **Oracle-Supplied Packages**

36) What does the `dbms_output` package do?

**The `dbms_output` package enables you to send messages from stored procedures, packages, and triggers.**

37) How do you write “This procedure works.” from within a PL/SQL program by using `dbms_output`?

**`DBMS_OUTPUT.PUT_LINE('This procedure works.');`**

38) What does `dbms_sql` do and how does it compare with Native Dynamic SQL?

**`dbms_sql` enables you to embed dynamic data manipulation language (DML), data definition language (DDL), and data control language (DCL) statements within a PL/SQL program. Native dynamic SQL allows you to place dynamic SQL statements directly into PL/SQL blocks. Native dynamic SQL in PL/SQL is easier to use than `dbms_sql`, requires much less application code, and performs better.**



## Practices for Lesson 3

In this practice, you determine the output of a PL/SQL code snippet and modify the snippet to improve the performance. Next, you implement subtypes and use cursor variables to pass values to and from a package.

**Note:** Files used in the practices are found in the `/labs` folder. Additionally, solution scripts are provided for each question and are located in the `/soln` folder. Your instructor will provide you with the exact location of these files. Connect as `OE` to perform the steps.

### Practice 3-1: Designing PL/SQL Code

- 1) Determine the output of the following code snippet in the lab\_03\_01.sql file.

```
SET SERVEROUTPUT ON
BEGIN
  UPDATE orders SET order_status = order_status;
  FOR v_rec IN ( SELECT order_id FROM orders )
  LOOP
    IF SQL%ISOPEN THEN
      DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
    ELSE
      DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
    END IF;
  END LOOP;
END;
/
```

- 2) Modify the following code snippet in the lab\_03\_02.sql file to make better use of the FOR UPDATE clause and improve the performance of the program.

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
  LOOP
    IF v_rec IS NOT NULL
    THEN
      UPDATE customers
      SET credit_limit = credit_limit + 200
      WHERE customer_id = v_rec.customer_id;
    END IF;
  END LOOP;
END;
/
```

- 3) Create a package specification that defines a subtype that can be used for the warranty\_period field of the product\_information table. Name this package mytypes. The type holds the month and year for a warranty period.
- 4) Create a package named SHOW\_DETAILS that contains two subroutines. The first subroutine should show order details for the given order\_id. The second subroutine should show customer details for the given customer\_id, including the customer ID, first name, phone numbers, credit limit, and email address. Both subroutines should use the cursor variable to return the necessary details.

## Practice Solutions 3-1: Designing PL/SQL Code

- 1) Determine the output of the following code snippet in the lab\_03\_01.sql file.

**Connect as OE.**

```
SET SERVEROUTPUT ON
BEGIN
  UPDATE orders SET order_status = order_status;
  FOR v_rec IN ( SELECT order_id FROM orders )
  LOOP
    IF SQL%ISOPEN THEN
      DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
    ELSE
      DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
    END IF;
  END LOOP;
END;
/
```

**Execute the code from the lab\_03\_01.sql file. It will show FALSE - 105 for each row fetched.**

- 2) Modify the following code snippet in the lab\_03\_02.sql file to make better use of the FOR UPDATE clause and improve the performance of the program.

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
  LOOP
    IF v_rec IS NOT NULL THEN
      UPDATE customers
      SET credit_limit = credit_limit + 200
      WHERE customer_id = v_rec.customer_id;
    END IF;
  END LOOP;
END;
/
```

**Modify the lab\_03\_02.sql file as follows:**

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
  LOOP
    UPDATE customers
```

### ***Practice Solutions 3-1: Designing PL/SQL Code (continued)***

```
        SET credit_limit = credit_limit + 200
        WHERE CURRENT OF cur_update;
    END LOOP;
END;
/
```

**Alternatively, you can execute the code from the `sol_03_02.sql` file.**

- 3) Create a package specification that defines a subtype that can be used for the `warranty_period` field of the `product_information` table. Name this package `mytypes`. The type holds the month and year for a warranty period.

```
CREATE OR REPLACE PACKAGE mytypes
IS
    TYPE typ_warranty
        IS RECORD (month POSITIVE, year PLS_INTEGER);
    SUBTYPE warranty IS typ_warranty; -- based on RECORD type
END mytypes;
/
```

- 4) Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer ID, first name, phone numbers, credit limit, and email address. Both subroutines should use the cursor variable to return the necessary details.

```
CREATE OR REPLACE PACKAGE show_details AS

TYPE rt_order IS REF CURSOR RETURN orders%ROWTYPE;

TYPE typ_cust_rec IS RECORD
    (cust_id NUMBER(6), cust_name VARCHAR2(20),
     custphone customers.phone_numbers%TYPE,
     credit NUMBER(9,2), cust_email VARCHAR2(30));
TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;

PROCEDURE get_order(p_orderid IN NUMBER, p_cv_order IN OUT
rt_order);
```

```
PROCEDURE get_cust(p_custid IN NUMBER, p_cv_cust IN OUT
rt_cust);
END show_details;
/

CREATE OR REPLACE PACKAGE BODY show_details AS
PROCEDURE get_order
    (p_orderid IN NUMBER, p_cv_order IN OUT rt_order)
IS
BEGIN
    OPEN p_cv_order FOR
        SELECT * FROM orders
```

### ***Practice Solutions 3-1: Designing PL/SQL Code (continued)***

```
        WHERE order_id = p_orderid;
-- CLOSE p_cv_order
END get_order;

PROCEDURE get_cust
  (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)
IS
BEGIN
  OPEN p_cv_cust FOR
    SELECT customer_id, cust_first_name, phone_numbers,
           credit_limit,
           cust_email FROM customers
    WHERE customer_id = p_custid;
-- CLOSE p_cv_cust
END get_cust;
END;
/
```

**Alternatively, you can execute the code from the `sol_03_04.sql` file.**

---

## Practices for Lesson 4

---

In this practice, you analyze collections for common errors, create a collection, and then write a PL/SQL package to manipulate the collection.

Use the OE schema for this practice.

## Practice 4-1: Analyzing Collections

In this practice, you create a nested table collection and use PL/SQL code to manipulate the collection.

- 1) Examine the following definitions. Run the lab\_04\_01.sql script to create these objects.

```
CREATE TYPE typ_item AS OBJECT --create object
  (prodid  NUMBER(5),
   price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder ( -- create database table
  ordid    NUMBER(5),
  supplier NUMBER(5),
  requester    NUMBER(4),
  ordered DATE,
  items        typ_item_nst)
  NESTED TABLE items STORE AS item_stor_tab
/
```

- 2) The following code generates an error. Run the lab\_04\_02.sql script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO TABLE (SELECT items
                        FROM   pOrder
                        WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```

- a) Why does the error occur?
  - b) How can you fix the error?
- 3) Examine the following code, which produces an error. Which line causes the error, and how do you fix it?  
(**Note:** You can run the lab\_04\_03.sql script to view the error output).

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc    credit_card_typ := credit_card_typ();
  v_visa  credit_card_typ := credit_card_typ();
```

## Practice 4-1: Analyzing Collections (continued)

```
v_am    credit_card_typ;  
v_disc  credit_card_typ := credit_card_typ();  
v_dc    credit_card_typ := credit_card_typ();  
  
BEGIN  
    v_mc.EXTEND;  
    v_visa.EXTEND;  
    v_am.EXTEND;  
    v_disc.EXTEND;  
    v_dc.EXTEND;  
END;  
/
```

### Using Collections

Now you implement a nested table column in the CUSTOMERS table and write PL/SQL code to manipulate the nested table.

- 4) Create a nested table to hold credit card information.
  - a) Create an object type called `typ_cr_card`. It should have the following specification:

```
card_type  VARCHAR2(25)  
card_num   NUMBER
```
  - b) Create a nested table type called `typ_cr_card_nst` that is a table of `typ_cr_card`.
  - c) Add a column called `credit_cards` to the CUSTOMERS table. Make this column a nested table of type `typ_cr_card_nst`. You can use the following syntax:

```
ALTER TABLE customers ADD  
(credit_cards typ_cr_card_nst)  
    NESTED TABLE credit_cards STORE AS c_c_store_tab;
```

- 5) Create a PL/SQL package that manipulates the `credit_cards` column in the CUSTOMERS table.
  - a) Open the `lab_04_05.sql` file. It contains the package specification and part of the package body.
  - b) Complete the following code so that the package:
    - Inserts credit card information (the credit card name and number for a specific customer)
    - Displays credit card information in an unnested format

```
CREATE OR REPLACE PACKAGE credit_card_pkg  
IS  
    PROCEDURE update_card_info  
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no  
        VARCHAR2);  
    PROCEDURE display_card_info  
        (p_cust_id NUMBER);
```



### ***Practice 4-1: Analyzing Collections (continued)***

```
END credit_card_pkg;  -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            -- cards exist, add more

            -- fill in code here

        ELSE -- no cards for this customer, construct one

            -- fill in code here

        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;

        -- fill in code here to display the nested table
        -- contents

    END display_card_info;
END credit_card_pkg;  -- package body
/
```

- 6) Test your package with the following statements and compare the output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.
```

## Practice 4-1: Analyzing Collections (continued)

```
EXECUTE credit_card_pkg.update_card_info -
  (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
  (120, 'MC', 2323232323)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
  (120, 'DC', 44444444)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 44444444
PL/SQL procedure successfully completed
```

- 7) Write a SELECT statement against the credit\_cards column to unnest the data.  
Use the TABLE expression. Use SQL\*Plus.  
For example, if the SELECT statement returns:

```
SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;
```

```
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111),
TYP_CR_CARD('MC', 2323232323), TYP_CR_CARD('DC', 44444444))
```

...rewrite it using the TABLE expression so that the results look like this:

```
-- Use the table expression so that the result is:
CUSTOMER_ID CUST_LAST_NAME  CARD_TYPE      CARD_NUM
-----
120 Higgins      Visa           11111111
120 Higgins      MC             2323232323
120 Higgins      DC             44444444
```

## Practice Solutions 4-1: Analyzing Collections

In this practice, you create a nested table collection and use PL/SQL code to manipulate the collection.

- 1) Examine the following definitions. Run the lab\_04\_01.sql script to create these objects.

```
CREATE TYPE typ_item AS OBJECT --create object
  (prodid  NUMBER(5),
   price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder ( -- create database table
  ordid    NUMBER(5),
  supplier NUMBER(5),
  requester    NUMBER(4),
  ordered DATE,
  items    typ_item_nst)
  NESTED TABLE items STORE AS item_stor_tab
/
```

@lab\_04\_01.sql

- 2) The following code generates an error. Run the lab\_04\_02.sql script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO TABLE (SELECT items
                        FROM   pOrder
                        WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```

@lab\_04\_02.sql

- a) Why does the error occur?

**The error “ORA-22908: reference to NULL table value” results from setting the table columns to NULL.**

- b) How can you fix the error?

## Practice Solutions 4-1: Analyzing Collections (continued)

You should always use a nested table's default constructor to initialize it:

```
TRUNCATE TABLE pOrder;

-- A better approach is to avoid setting the table
-- column to NULL, and instead, use a nested table's
-- default constructor to initialize
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
  VALUES (1000, 12345, 9876, SYSDATE,
          typ_item_nst(typ_item(99, 129.00)));
END;
/

BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
  VALUES (1000, 12345, 9876, SYSDATE, null);
  -- Once the nested table is set to null, use the update
  -- update statement
  UPDATE pOrder
    SET items = typ_item_nst(typ_item(99, 129.00))
    WHERE ordid = 1000;
END;
/
```

- 3) Examine the following code. This code produces an error. Which line causes the error, and how do you fix it?

(Note: You can run the lab\_04\_03.sql script to view the error output.)

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ;
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```

## Practice Solutions 4-1: Analyzing Collections (continued)

This causes an **ORA-06531: Reference to uninitialized collection**. To fix it, initialize the **v\_am** variable by using the same technique as the others:

```
DECLARE
    TYPE credit_card_typ
    IS VARRAY(100) OF VARCHAR2(30);

    v_mc    credit_card_typ := credit_card_typ();
    v_visa  credit_card_typ := credit_card_typ();
    v_am    credit_card_typ := credit_card_typ();
    v_disc  credit_card_typ := credit_card_typ();
    v_dc    credit_card_typ := credit_card_typ();

BEGIN
    v_mc.EXTEND;
    v_visa.EXTEND;
    v_am.EXTEND;
    v_disc.EXTEND;
    v_dc.EXTEND;
END;
/
```

### Using Collections

Now you implement a nested table column in the **CUSTOMERS** table and write PL/SQL code to manipulate the nested table.

- 4) Create a nested table to hold credit card information.
  - a) Create an object type called **typ\_cr\_card**. It should have the following specification:  
card\_type VARCHAR2(25)  
card\_num NUMBER

```
CREATE TYPE typ_cr_card AS OBJECT --create object
(card_type VARCHAR2(25),
 card_num NUMBER);
/
```

- b) Create a nested table type called **typ\_cr\_card\_nst** that is a table of **typ\_cr\_card**.

```
CREATE TYPE typ_cr_card_nst -- define nested table type
AS TABLE OF typ_cr_card;
/
```

- c) Add a column called **credit\_cards** to the **CUSTOMERS** table. Make this column a nested table of type **typ\_cr\_card\_nst**. You can use the following syntax:

```
ALTER TABLE customers ADD
(credit_cards typ_cr_card_nst)
NESTED TABLE credit_cards STORE AS c_c_store_tab;
```

## ***Practice Solutions 4-1: Analyzing Collections (continued)***

- 5) Create a PL/SQL package that manipulates the `credit_cards` column in the `CUSTOMERS` table.
- a) Open the `lab_04_05` . file. It contains the package specification and part of the package body.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            -- cards exist, add more

            -- fill in code here
        ELSE -- no cards for this customer, construct one

            -- fill in code here
        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;

        -- fill in code here to display the nested table
```

## Practice Solutions 4-1: Analyzing Collections (continued)

```
-- contents

    END display_card_info;
END credit_card_pkg; -- package body
/
```

b) Complete the following code so that the package:

- Inserts the credit card information (the credit card name and number for a specific customer)
- Displays the credit card information in an unnested format

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;
```

## Practice Solutions 4-1: Analyzing Collections (continued)

```
PROCEDURE display_card_info
  (p_cust_id NUMBER)
IS
  v_card_info typ_cr_card_nst;
  i INTEGER;
BEGIN
  SELECT credit_cards
    INTO v_card_info
    FROM customers
    WHERE customer_id = p_cust_id;
  IF v_card_info.EXISTS(1) THEN
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
```

```
END IF;
END display_card_info;
END credit_card_pkg;  -- package body
/
```

6) Test your package with the following statements and compare the output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
  (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
  (120, 'MC', 2323232323)
PL/SQL procedure successfully completed.
```



## ***Practice Solutions 4-1: Analyzing Collections (continued)***

```
EXECUTE credit_card_pkg.update_card_info -
    (120, 'DC', 4444444)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
PL/SQL procedure successfully completed.
```

```
SET SERVEROUTPUT ON

EXECUTE credit_card_pkg.display_card_info(120)

EXECUTE credit_card_pkg.update_card_info(120, 'Visa',
11111111)

SELECT credit_cards
FROM    customers
WHERE   customer_id = 120;

EXECUTE credit_card_pkg.display_card_info(120)

EXECUTE credit_card_pkg.update_card_info(120, 'MC',
2323232323)

EXECUTE credit_card_pkg.update_card_info (120, 'DC', 4444444)

EXECUTE credit_card_pkg.display_card_info(120)
```

- 7) Write a SELECT statement against the credit\_cards column to unnest the data.  
Use the TABLE expression. Use SQL\*Plus.

For example, if the SELECT statement returns:

```
SELECT credit_cards
FROM    customers
WHERE   customer_id = 120;
```

```
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
```

```
-----
```

```
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111),
TYP_CR_CARD('MC', 2323232323), TYP_CR_CARD('DC', 4444444))
```

... rewrite it using the TABLE expression so that the results look like this:

```
-- Use the table expression so that the result is:
```

### ***Practice Solutions 4-1: Analyzing Collections (continued)***

CUSTOMER_ID	CUST_LAST_NAME	CARD_TYPE	CARD_NUM
120	Higgins	Visa	11111111
120	Higgins	MC	23232323
120	Higgins	DC	44444444

```
SELECT c1.customer_id, c1.cust_last_name, c2.*
FROM   customers c1, TABLE(c1.credit_cards) c2
WHERE  customer_id = 120;
```

---

## Practices for Lesson 5

---

In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS\_LOB package to populate the table and manipulate the data.

## Practice 5-1: Working with LOBs

- 1) Create a table called `PERSONNEL` by executing the `/home/oracle/labs/labs/lab_05_01.sql` script file. The table contains the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
LAST_NAME	VARCHAR2	35
REVIEW	CLOB	N/A
PICTURE	BLOB	N/A

- 2) Insert two rows into the `PERSONNEL` table, one each for employee 2034 (whose last name is Allen) and employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide `NULL` as the value for the BLOB.
- 3) Examine and execute the `/home/oracle/labs/labs/lab_05_03.sql` script. The script creates a table named `REVIEW_TABLE`. This table contains the annual review information for each employee. The script also contains two statements to insert review details about two employees.
- 4) Update the `PERSONNEL` table.

- a) Populate the CLOB for the first row by using this subquery in an UPDATE statement:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2034;
```

- b) Populate the CLOB for the second row by using PL/SQL and the `DBMS_LOB` package. Use the following SELECT statement to provide a value for the LOB locator:

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

- 5) Create a procedure that adds a locator to a binary file to the `PICTURE` column of the `PRODUCT_INFORMATION` table. The binary file is a picture of the product. The image files are named after the product IDs. You must load an image file locator into all rows in the Printers category (`CATEGORY_ID = 12`) in the `PRODUCT_INFORMATION` table.

- a) Create a `DIRECTORY` object called `PRODUCT_PIC` that references the location of the binary. These files are available in the `/home/oracle/Labs/DATA_FILES/PRODUCT_PIC` folder.

```
CREATE DIRECTORY product_pic AS
'/home/oracle/Labs/DATA_FILES/PRODUCT_PIC';
```

(Alternatively, use the `/home/oracle/labs/labs/lab_05_05a.sql` script.)

- b) Add the image column to the `PRODUCT_INFORMATION` table by using:

### ***Practice 5-1: Working with LOBs (continued)***

```
ALTER TABLE product_information ADD (picture BFILE);
```

(Alternatively, use the `/home/oracle/labs/labs/lab_05_05_b.sql` file.)

- c) Create a PL/SQL procedure called `load_product_image` that uses `DBMS_LOB.FILEEXISTS` to test whether the product picture file exists. If the file exists, set the `BFILE` locator for the file in the `PICTURE` column; otherwise, display a message that the file does not exist. Use the `DBMS_OUTPUT` package to report file size information about each image associated with the `PICTURE` column. (Alternatively, use the `/home/oracle/labs/labs/lab_05_05_c.sql` file.)
- d) Invoke the procedure by passing the name of the `PRODUCT_PIC` directory object as a string literal parameter value.
- e) Check the LOB space usage of the `PRODUCT_INFORMATION` table. Use the `/home/oracle/labs/labs/lab_05_05_e.sql` file to create the procedure and execute it.

## Practice 5-2: Working with SecureFile LOBs

1) In this lesson, you practice using the features of the SecureFile format LOBs.

Before you migrate a BasicFile format LOB to a SecureFile format LOB, you must set up several supporting structures:

- a) As the OE user, drop your existing PRODUCT\_DESCRIPTIONS table and create a new one:

```
DROP TABLE product_descriptions;  
CREATE TABLE product_descriptions  
  (product_id NUMBER);
```

- b) As the SYS user, create a tablespace to store the LOB information.

```
CREATE TABLESPACE lob_tbs2  
  DATAFILE 'lob_tbs2.dbf' SIZE 1500M REUSE  
  AUTOEXTEND ON NEXT 200M  
  MAXSIZE 3000M  
  SEGMENT SPACE MANAGEMENT AUTO;
```

- c) Create a directory object that identifies the location of your LOBs. In the Oracle classroom, the location is in the /home/oracle/labs/DATA\_FILES/PRODUCT\_PIC folder. Then grant read privileges on the directory to the OE user.

```
CREATE OR REPLACE DIRECTORY product_files  
  AS '/home/oracle/labs/DATA_FILES/PRODUCT_PIC';  
  
GRANT READ ON DIRECTORY product_files TO oe;
```

- d) As the OE user, alter the table and add a BLOB column of the BASICFILE storage type.

```
ALTER TABLE product_descriptions ADD  
  (detailed_product_info BLOB )  
  LOB (detailed_product_info) STORE AS BASICFILE (tablespace  
  lob_tbs2);
```

- e) Create the procedure to load the LOB data into the column. (You can run the /home/oracle/labs/labs/lab\_05\_01\_e.sql script.)

```
CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc (  
  p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2)  
IS  
  v_src_loc   BFILE := BFILENAME('PRODUCT_FILES', p_file_name);  
  v_amount    INTEGER := 4000;  
BEGIN  
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);  
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);  
  DBMS_LOB.LOADFROMFILE(p_dest_loc, v_src_loc, v_amount);  
  DBMS_LOB.CLOSE(v_src_loc);  
END loadLOBFromBFILE_proc;  
/
```

## Practice 5-2: Working with SecureFile LOBs (continued)

- f) As the OE user, create the procedure to write the LOB data. (You can run the /home/oracle/labs/labs /lab\_05\_01\_f.sql script.)

```
CREATE OR REPLACE PROCEDURE write_lob (p_file IN VARCHAR2)
IS
  i      NUMBER;    v_id NUMBER;  v_b  BLOB;
BEGIN
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');
  FOR i IN 1 .. 5 LOOP
    v_id:=SUBSTR(p_file, 1, 4);
    INSERT INTO product_descriptions
      VALUES (v_id, EMPTY_BLOB())
      RETURNING detailed_product_info INTO v_b;
    loadLOBFromBFILE_proc(v_b,p_file);
    DBMS_OUTPUT.PUT_LINE('Row ' || i || ' inserted.');
```

- g) As the OE user, execute the procedures to load the data. If you are using SQL\*Plus, you can set the timing on to observe the time. (You can run the /home/oracle/labs/labs /lab\_05\_01\_g.sql script.)
- Note:** If you are using SQL Developer, issue only the following EXECUTE statements. In SQL Developer, some of the SQL\*Plus commands are ignored. It is recommended that you use SQL\*Plus for this practice.

```
set serveroutput on
set verify on
set term on

timing start load_data
execute write_lob('1726_LCD.doc');
execute write_lob('1734_RS232.doc');
execute write_lob('1739_SDRAM.doc');
timing stop
```

- h) As the SYS user, check the segment type in the data dictionary.

```
SELECT segment_name, segment_type, segment_subtype
FROM   dba_segments
WHERE  tablespace_name = 'LOB_TBS2'
AND    segment_type = 'LOBSEGMENT';
```

- i) As the OE user, create an interim table.

```
CREATE TABLE product_descriptions_interim
(product_id NUMBER,
detailed_product_info BLOB)
LOB(detailed_product_info) STORE AS SECUREFILE
(TABLESPACE lob_tbs2);
```

## ***Practice 5-2: Working with SecureFile LOBs (continued)***

- j) Connect as the SYS user and run the redefinition script. (You can run the /home/oracle/labs/labs /lab\_05\_01\_j.sql script.)

```
DECLARE
  error_count PLS_INTEGER := 0;
BEGIN
  DBMS_REDEFINITION.START_REDEF_TABLE
    ('OE', 'product_descriptions',
    'product_descriptions_interim',
    'product_id product_id, detailed_product_info
    detailed_product_info',
    OPTIONS_FLAG => DBMS_REDEFINITION.CONST_USE_ROWID);
  DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
    ('OE', 'product_descriptions',
    'product_descriptions_interim',
    1, true,true,true,false, error_count);
  DBMS_OUTPUT.PUT_LINE('Errors := ' || TO_CHAR(error_count));
  DBMS_REDEFINITION.FINISH_REDEF_TABLE
    ('OE', 'product_descriptions',
    'product_descriptions_interim');
END;
/
```

- k) As the OE user, remove the interim table.

```
DROP TABLE product_descriptions_interim;
```

- l) As the SYS user, check the segment type in the data dictionary to make sure that it is now set to SECUREFILE.

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

- m) As the OE user, check the space of the table by executing the CHECK\_SPACE procedure. (You can run the /home/oracle/labs/labs /lab\_05\_01\_m.sql script.)

```
CREATE OR REPLACE PROCEDURE check_space
IS
  l_fs1_bytes NUMBER;
  l_fs2_bytes NUMBER;
  l_fs3_bytes NUMBER;
  l_fs4_bytes NUMBER;
  l_fs1_blocks NUMBER;
  l_fs2_blocks NUMBER;
  l_fs3_blocks NUMBER;
  l_fs4_blocks NUMBER;
  l_full_bytes NUMBER;
  l_full_blocks NUMBER;
  l_unformatted_bytes NUMBER;
  l_unformatted_blocks NUMBER;
```



## Practice 5-2: Working with SecureFile LOBs (continued)

```
BEGIN
  DBMS_SPACE.SPACE_USAGE(
    segment_owner      => 'OE',
    segment_name       => 'PRODUCT_DESCRIPTIONS',
    segment_type       => 'TABLE',
    fs1_bytes          => l_fs1_bytes,
    fs1_blocks         => l_fs1_blocks,
    fs2_bytes          => l_fs2_bytes,
    fs2_blocks         => l_fs2_blocks,
    fs3_bytes          => l_fs3_bytes,
    fs3_blocks         => l_fs3_blocks,
    fs4_bytes          => l_fs4_bytes,
    fs4_blocks         => l_fs4_blocks,
    full_bytes         => l_full_bytes,
    full_blocks        => l_full_blocks,
    unformatted_blocks => l_unformatted_blocks,
    unformatted_bytes  => l_unformatted_bytes
  );
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = ' || l_fs1_blocks || '
    Bytes = ' || l_fs1_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = ' || l_fs2_blocks || '
    Bytes = ' || l_fs2_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = ' || l_fs3_blocks || '
    Bytes = ' || l_fs3_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = ' || l_fs4_blocks || '
    Bytes = ' || l_fs4_bytes);
  DBMS_OUTPUT.PUT_LINE(' Full Blocks = ' || l_full_blocks || '
    Bytes = ' || l_full_bytes);
  DBMS_OUTPUT.PUT_LINE('=====
    =====');
  DBMS_OUTPUT.PUT_LINE('Total Blocks =
    ' || to_char(l_fs1_blocks + l_fs2_blocks +
    l_fs3_blocks + l_fs4_blocks + l_full_blocks) || ' ||
    Total Bytes = ' || to_char(l_fs1_bytes + l_fs2_bytes
    + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/

set serveroutput on
execute check_space;
```

## Practice Solutions 5-1: Working with LOBs

In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS\_LOB package to populate the table and manipulate the data.

Use your OE connection.

- 1) Create a table called PERSONNEL by executing the  
/home/oracle/labs/labs/lab\_05\_01.sql script file. The table contains the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
LAST_NAME	VARCHAR2	35
REVIEW	CLOB	N/A
PICTURE	BLOB	N/A

```
CREATE TABLE personnel
(id NUMBER(6) constraint personnel_id_pk PRIMARY KEY,
last_name VARCHAR2(35),
review CLOB,
picture BLOB);
```

```
@lab_05_01.sql
```

- 2) Insert two rows into the PERSONNEL table, one each for employee 2034 (whose last name is Allen) and employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide NULL as the value for the BLOB.

```
INSERT INTO personnel
VALUES (2034, 'Allen', empty_clob(), NULL);

INSERT INTO personnel
VALUES (2035, 'Bond', empty_clob(), NULL);
```

- 3) Examine and execute the /home/oracle/labs/labs/lab\_05\_03.sql script. The script creates a table named REVIEW\_TABLE. This table contains annual review information for each employee. The script also contains two statements to insert review details for two employees.

```
CREATE TABLE review_table (
employee_id number,
ann_review VARCHAR2(2000));

INSERT INTO review_table
VALUES (2034,
'Very good performance this year. '||
'Recommended to increase salary by $500');

INSERT INTO review_table
VALUES (2035,
```

## Practice Solutions 5-1: Working with LOBs (continued)

```
'Excellent performance this year. '||  
'Recommended to increase salary by $1000');  
  
COMMIT;
```

```
@/home/oracle/labs/labs /lab_05_03.sql
```

### 4) Update the PERSONNEL table.

- a) Populate the CLOB for the first row by using the following subquery in an UPDATE statement:

```
SELECT ann_review  
FROM   review_table  
WHERE  employee_id = 2034;
```

```
UPDATE personnel  
SET review = (SELECT ann_review  
              FROM   review_table  
              WHERE  employee_id = 2034)  
WHERE last_name = 'Allen';
```

- b) Populate the CLOB for the second row by using PL/SQL and the DBMS\_LOB package. Use the following SELECT statement to provide a value for the LOB locator:

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

```
UPDATE personnel  
SET review = (SELECT ann_review  
              FROM   review_table  
              WHERE  employee_id = 2035)  
WHERE last_name = 'Bond';
```

### 5) Create a procedure that adds a locator to a binary file to the PICTURE column of the PRODUCT\_INFORMATION table. The binary file is a picture of the product. The image files are named after the product IDs. You must load an image file locator into all rows in the Printers category (CATEGORY\_ID = 12) in the PRODUCT\_INFORMATION table.

- a) Create a DIRECTORY object called PRODUCT\_PIC that references the location of the binary file. These files are available in the /home/oracle/labs/DATA\_FILES/PRODUCT\_PIC folder.

```
CREATE DIRECTORY product_pic AS  
'/home/oracle/labs/DATA_FILES/PRODUCT_PIC';
```

## Practice Solutions 5-1: Working with LOBs (continued)

(Alternatively, use the /home/oracle/labs/labs/lab\_05\_05a.sql script.)

- b) Add the image column to the PRODUCT\_INFORMATION table by using:

```
ALTER TABLE product_information ADD (picture BFILE);
```

(Alternatively, use the /home/oracle/labs/labs/lab\_05\_05\_b.sql file.)

- c) Create a PL/SQL procedure called load\_product\_image that uses DBMS\_LOB.FILEEXISTS to test whether the product picture file exists. If the file exists, set the BFILE locator for the file in the PICTURE column; otherwise, display a message that the file does not exist. Use the DBMS\_OUTPUT package to report file size information for each image associated with the PICTURE column. (Alternatively, use the /home/oracle/labs/labs/lab\_05\_05\_c.sql file.)

```
CREATE OR REPLACE PROCEDURE load_product_image
(p_dir IN VARCHAR2)
IS
  v_file          BFILE;
  v_filename      VARCHAR2(40);
  v_rec_number    NUMBER;
  v_file_exists   BOOLEAN;
  CURSOR product_csr IS
    SELECT product_id
    FROM product_information
    WHERE category_id = 12
    FOR UPDATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('LOADING LOCATORS TO IMAGES...');
  FOR rec IN product_csr
  LOOP
    v_filename := rec.product_id || '.gif';
    v_file := BFILENAME(p_dir, v_filename);
    v_file_exists := (DBMS_LOB.FILEEXISTS(v_file) = 1);
    IF v_file_exists THEN
      DBMS_LOB.FILEOPEN(v_file);
      UPDATE product_information
      SET picture = v_file
      WHERE CURRENT OF product_csr;
      DBMS_OUTPUT.PUT_LINE('Set Locator to file: ' ||
v_filename
      || ' Size: ' || DBMS_LOB.GETLENGTH(v_file));
      DBMS_LOB.FILECLOSE(v_file);
      v_rec_number := product_csr%ROWCOUNT;
    ELSE
      DBMS_OUTPUT.PUT_LINE('File ' || v_filename ||
        ' does not exist');
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('TOTAL FILES UPDATED: ' ||
    v_rec_number);
EXCEPTION
```

## Practice Solutions 5-1: Working with LOBs (continued)

```
        WHEN OTHERS THEN
            DBMS_LOB.FILECLOSE(v_file);
            DBMS_OUTPUT.PUT_LINE('Error: ' || to_char(SQLCODE) ||
                                SQLERRM);
    END load_product_image;
/
```

- d) Invoke the procedure by passing the name of the PRODUCT\_PIC directory object as a string literal parameter value.

```
SET SERVEROUTPUT ON
EXECUTE load_product_image('PRODUCT_PIC');
```

```
LOADING LOCATORS TO IMAGES...
Set Locator to file: 1797.gif Size: 7888
Set Locator to file: 2459.gif Size: 9587
Set Locator to file: 3127.gif Size: 9587
Set Locator to file: 1782.gif Size: 7888
Set Locator to file: 2430.gif Size: 7462
Set Locator to file: 1792.gif Size: 7462
Set Locator to file: 1791.gif Size: 7462
Set Locator to file: 2302.gif Size: 7462
Set Locator to file: 2453.gif Size: 9587
TOTAL FILES UPDATED: 9
```

- e) Check the LOB space usage of the PRODUCT\_INFORMATION table. Use the /home/oracle/labs/labs/lab\_05\_05\_e.sql file to create the procedure and execute it.

```
CREATE OR REPLACE PROCEDURE check_space
IS
    l_fs1_bytes NUMBER;
    l_fs2_bytes NUMBER;
    l_fs3_bytes NUMBER;
    l_fs4_bytes NUMBER;
    l_fs1_blocks NUMBER;
    l_fs2_blocks NUMBER;
    l_fs3_blocks NUMBER;
    l_fs4_blocks NUMBER;
    l_full_bytes NUMBER;
    l_full_blocks NUMBER;
    l_unformatted_bytes NUMBER;
    l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name       => 'PRODUCT_INFORMATION',
        segment_type       => 'TABLE',
        fs1_bytes          => l_fs1_bytes,
        fs1_blocks         => l_fs1_blocks,
        fs2_bytes          => l_fs2_bytes,
        fs2_blocks         => l_fs2_blocks,
        fs3_bytes          => l_fs3_bytes,
        fs3_blocks         => l_fs3_blocks,
        fs4_bytes          => l_fs4_bytes,
```

## Practice Solutions 5-1: Working with LOBs (continued)

```
fs4_blocks      => l_fs4_blocks,
full_bytes      => l_full_bytes,
full_blocks     => l_full_blocks,
unformatted_blocks => l_unformatted_blocks,
unformatted_bytes => l_unformatted_bytes
);
DBMS_OUTPUT.ENABLE;
DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
  Bytes = '||l_fs1_bytes);
DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
  Bytes = '||l_fs2_bytes);
DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
  Bytes = '||l_fs3_bytes);
DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
  Bytes = '||l_fs4_bytes);
DBMS_OUTPUT.PUT_LINE(' Full Blocks = '||l_full_blocks||'
  Bytes = '||l_full_bytes);
DBMS_OUTPUT.PUT_LINE('=====');
DBMS_OUTPUT.PUT_LINE('Total Blocks =
  ||to_char(l_fs1_blocks + l_fs2_blocks +
  l_fs3_blocks + l_fs4_blocks + l_full_blocks)|| ' ||
  Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
  + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/
```

```
set serveroutput on
execute check_space;
```

```
FS1 Blocks = 0
  Bytes = 0
FS2 Blocks = 0
  Bytes = 0
FS3 Blocks = 0
  Bytes = 0
FS4 Blocks = 4
  Bytes = 32768
Full Blocks = 9
  Bytes = 73728
=====
Total Blocks =
13 ||
Total Bytes = 106496
```

## Practice Solutions 5-2: Working with SecureFile LOBs

1) In this lesson, you practice using the features of the SecureFile format LOBs.

Before you migrate a BasicFile format LOB to a SecureFile format LOB, you must set up several supporting structures:

- a) As the OE user, drop your existing PRODUCT\_DESCRIPTIONS table and create a new one:

```
DROP TABLE product_descriptions PURGE;
```

```
CREATE TABLE product_descriptions  
(product_id NUMBER);
```

```
DROP TABLE product_descriptions PURGE;
```

```
CREATE TABLE product_descriptions  
(product_id NUMBER);
```

- b) As the SYS user, create a tablespace to store the LOB information.

```
CREATE TABLESPACE lob_tbs2  
DATAFILE 'lob_tbs2.dbf' SIZE 1500M REUSE  
AUTOEXTEND ON NEXT 200M  
MAXSIZE 3000M  
SEGMENT SPACE MANAGEMENT AUTO;
```

```
CREATE TABLESPACE lob_tbs2  
DATAFILE 'lob_tbs2.dbf' SIZE 1500M REUSE  
AUTOEXTEND ON NEXT 200M  
MAXSIZE 3000M  
SEGMENT SPACE MANAGEMENT AUTO;
```

- c) Create a directory object that identifies the location of your LOBs. In the Oracle classroom, the location is in the Oracle /home/oracle/labs/DATA\_FILES/PRODUCT\_PIC folder. Then grant read privileges on the directory to the OE user.

```
CREATE OR REPLACE DIRECTORY product_files  
AS '/home/oracle/DATA_FILES/PRODUCT_PIC';
```

```
GRANT READ ON DIRECTORY product_files TO oe;
```

```
connect /as sysdba
```

```
CREATE OR REPLACE DIRECTORY product_files  
AS '/home/oracle/DATA_FILES/PRODUCT_PIC';
```

```
GRANT READ ON DIRECTORY product_files TO oe;
```

- d) As the OE user, alter the table and add a BLOB column of the BASICFILE storage type.

## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

```
ALTER TABLE product_descriptions ADD
  (detailed_product_info BLOB )
  LOB (detailed_product_info) STORE AS BASICFILE
  (tablespace lob_tbs2);
```

```
ALTER TABLE product_descriptions ADD
  (detailed_product_info BLOB )
  LOB (detailed_product_info)
  STORE AS BASICFILE (tablespace lob_tbs2);
```

- e) Create the procedure to load the LOB data into the column (You can run the /home/oracle/labs/lab\_05\_01\_e.sql script):

```
CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc (
  p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2)
IS
  v_src_loc BFILE := BFILENAME('PRODUCT_FILES',
  p_file_name);
  v_amount INTEGER := 4000;
BEGIN
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
  DBMS_LOB.LOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
  DBMS_LOB.CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
/
```

```
connect oe

CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc (
  p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2)
IS
  v_src_loc BFILE := BFILENAME('PRODUCT_FILES',
  p_file_name);
  v_amount INTEGER := 4000;
BEGIN
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
  DBMS_LOB.LOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
  DBMS_LOB.CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
/
```

- f) As the OE user, create the procedure to write the LOB data. (You can run the /home/oracle/labs/lab\_05\_01\_f.sql script.)

```
CREATE OR REPLACE PROCEDURE write_lob (p_file IN VARCHAR2)
IS
```



## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

```
        i      NUMBER;    v_id NUMBER;    v_b  BLOB;
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');
    FOR i IN 1 .. 5 LOOP
        v_id:=SUBSTR(p_file, 1, 4);
        INSERT INTO product_descriptions
            VALUES (v_id, EMPTY_BLOB())
            RETURNING detailed_product_info INTO v_b;
        loadLOBFromBFILE_proc(v_b,p_file);
        DBMS_OUTPUT.PUT_LINE('Row ' || i || ' inserted.');
```

```
connect oe

CREATE OR REPLACE PROCEDURE write_lob (p_file IN VARCHAR2)
IS
    i      NUMBER;    v_id NUMBER;    v_b  BLOB;
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');
    FOR i IN 1 .. 5 LOOP
        v_id:=SUBSTR(p_file, 1, 4);
        INSERT INTO product_descriptions
            VALUES (v_id, EMPTY_BLOB())
            RETURNING detailed_product_info INTO v_b;
        loadLOBFromBFILE_proc(v_b,p_file);
        DBMS_OUTPUT.PUT_LINE('Row ' || i || ' inserted.');
```

- g) As the OE user, execute the procedures to load the data. If you are using SQL\*Plus, you can set the timing on to observe the time. (You can run the /home/oracle/labs/lab\_05\_01\_g.sql script.)

**Note:** If you are using Oracle SQL Developer, issue only the following EXECUTE statements. In Oracle SQL Developer, some of the SQL\*Plus commands are ignored. It is recommended that you use SQL\*Plus for this practice.

```
set serveroutput on
set verify on
set term on
set lines 200
timing start load_data
execute write_lob('1726_LCD.doc');
```

## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

```
execute write_lob('1734_RS232.doc');
execute write_lob('1739_SDRAM.doc');
timing stop
```

```
connect oe
set serveroutput on
set verify on
set term on
set lines 200
timing start load_data
execute write_lob('1726_LCD.doc');
execute write_lob('1734_RS232.doc');
execute write_lob('1739_SDRAM.doc');
timing stop
```

```
SQL> set serveroutput on
SQL> set verify on
SQL> set term on
SQL> set lines 200
SQL>
SQL> timing start load_data
SQL> execute write_lob('1726_LCD.doc');
Begin inserting rows...
Row 1 inserted.
Row 2 inserted.
Row 3 inserted.
Row 4 inserted.
Row 5 inserted.

PL/SQL procedure successfully completed.

SQL> execute write_lob('1734_RS232.doc');
Begin inserting rows...
Row 1 inserted.
Row 2 inserted.
Row 3 inserted.
Row 4 inserted.
Row 5 inserted.

PL/SQL procedure successfully completed.

SQL> execute write_lob('1739_SDRAM.doc');
Begin inserting rows...
Row 1 inserted.
Row 2 inserted.
Row 3 inserted.
Row 4 inserted.
Row 5 inserted.

PL/SQL procedure successfully completed.

SQL> timing stop
timing for: load_data
Elapsed: 00:00:00.09
SQL>
```

- h) As the SYS user, check the segment type in the data dictionary.

```
SELECT segment_name, segment_type, segment_subtype
FROM   dba_segments
WHERE  tablespace_name = 'LOB_TBS2'
AND    segment_type = 'LOBSEGMENT';
```

## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

```
SELECT segment_name, segment_type, segment_subtype
FROM    dba_segments
WHERE   tablespace_name = 'LOB_TBS2'
AND     segment_type = 'LOBSEGMENT';
```

SEGMENT_NAME	SEGMENT_TYPE	SEGMENT_SU
SYS_LOB0000071571C00002\$\$	LOBSEGMENT	ASSM

- i) As the OE user, create an interim table.

```
CREATE TABLE product_descriptions_interim
(product_id NUMBER,
detailed_product_info BLOB)
LOB(detailed_product_info) STORE AS SECUREFILE
(TABLESPACE lob_tbs2);
```

```
CREATE TABLE product_descriptions_interim
(product_id NUMBER,
detailed_product_info BLOB)
LOB(detailed_product_info) STORE AS SECUREFILE
(TABLESPACE lob_tbs2);
```

- j) Connect as the SYS user and run the redefinition script. (You can run the /home/oracle/labs/lab\_05\_01\_j.sql script.)

```
DECLARE
error_count PLS_INTEGER := 0;
BEGIN
  DBMS_REDEFINITION.START_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     'product_id product_id, detailed_product_info
     detailed_product_info',
     OPTIONS_FLAG => DBMS_REDEFINITION.CONST_USE_ROWID);
  DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     1, true, true, true, false, error_count);
  DBMS_OUTPUT.PUT_LINE('Errors := ' ||
                       TO_CHAR(error_count));
  DBMS_REDEFINITION.FINISH_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim');
END;
/
```

## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

```
SET SERVEROUTPUT ON

DECLARE
  error_count PLS_INTEGER := 0;
BEGIN
  DBMS_REDEFINITION.START_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     'product_id product_id, detailed_product_info
      detailed_product_info',
     OPTIONS_FLAG => DBMS_REDEFINITION.CONST_USE_ROWID);
  DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     1, true, true, true, false, error_count);
  DBMS_OUTPUT.PUT_LINE('Errors := ' ||
                       TO_CHAR(error_count));
  DBMS_REDEFINITION.FINISH_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim');
END;
/
```

**Errors := 0**

k) As the OE user, remove the interim table.

```
DROP TABLE product_descriptions_interim;
```

```
DROP TABLE product_descriptions_interim;
```

l) As the SYS user, check the segment type in the data dictionary to make sure that it is now set to SECUREFILE.

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

SEGMENT_NAME	SEGMENT_TYPE	SEGMENT_SU
SYS_LOB0000071577C00002\$\$	LOBSEGMENT	SECUREFILE
SYS_LOB0000071571C00002\$\$	LOBSEGMENT	ASSM

- m) As the OE user, check the space of the table by executing the CHECK\_SPACE procedure. (You can run the /home/oracle/labs/lab\_05\_01\_m.sql script.)

```
CREATE OR REPLACE PROCEDURE check_space
IS
    l_fs1_bytes NUMBER;
    l_fs2_bytes NUMBER;
    l_fs3_bytes NUMBER;
    l_fs4_bytes NUMBER;
    l_fs1_blocks NUMBER;
    l_fs2_blocks NUMBER;
    l_fs3_blocks NUMBER;
    l_fs4_blocks NUMBER;
    l_full_bytes NUMBER;
    l_full_blocks NUMBER;
    l_unformatted_bytes NUMBER;
    l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name       => 'PRODUCT_DESCRIPTIONS',
        segment_type       => 'TABLE',
        fs1_bytes          => l_fs1_bytes,
        fs1_blocks         => l_fs1_blocks,
        fs2_bytes          => l_fs2_bytes,
        fs2_blocks         => l_fs2_blocks,
        fs3_bytes          => l_fs3_bytes,
        fs3_blocks         => l_fs3_blocks,
        fs4_bytes          => l_fs4_bytes,
        fs4_blocks         => l_fs4_blocks,
        full_bytes         => l_full_bytes,
        full_blocks        => l_full_blocks,
        unformatted_blocks => l_unformatted_blocks,
        unformatted_bytes  => l_unformatted_bytes
    );
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = ' || l_fs1_blocks || '
        Bytes = ' || l_fs1_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = ' || l_fs2_blocks || '
        Bytes = ' || l_fs2_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = ' || l_fs3_blocks || '
        Bytes = ' || l_fs3_bytes);
```

## Practice Solutions 5-2: Working with SecureFile LOBs (continued)

```

DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
  Bytes = '||l_fs4_bytes);
DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
  Bytes = '||l_full_bytes);
DBMS_OUTPUT.PUT_LINE('=====
  =====');
DBMS_OUTPUT.PUT_LINE('Total Blocks =
  '||to_char(l_fs1_blocks + l_fs2_blocks +
  l_fs3_blocks + l_fs4_blocks + l_full_blocks)|| ' ||
  Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
  + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/

set serveroutput on
execute check_space;

```

```

FS1 Blocks = 0
  Bytes = 0
FS2 Blocks = 0
  Bytes = 0
FS3 Blocks = 0
  Bytes = 0
FS4 Blocks = 0
  Bytes = 0
Full Blocks = 1
  Bytes = 8192
=====
  =====
Total Blocks =
  1
Total Bytes = 8192

```

## Practices for Lesson 6

In this practice, you write two PL/SQL programs: One program calls an external C routine, and the second program calls a Java routine.

## Practice 6-1: Using Advanced Interface Methods

An external C routine definition is created for you. The .c file is stored in the /home/oracle/labs/labs directory. This function returns the tax amount based on the total sales figure that is passed to the function as a parameter. The .c file is named calc\_tax.c. The function is defined as:

```
#include <ctype.h>
int calc_tax(int n)
{
    int tax;
    tax = (n*8)/100;
    return(tax);
}
```

- 1) A shared library file called calc\_tax.so was created for you. Copy the file from the /home/oracle/labs/labs directory into your /u01/app/oracle/product/11.2.0/dbhome\_1/bin directory.
- 2) Connect to the sys connection, create the alias library object. Name the library object c\_code and define its path as:

```
CREATE OR REPLACE LIBRARY c_code
AS '/u01/app/oracle/product/11.2.0/dbhome_1/bin/calc_tax.so';
/
```

- 3) Grant the execute privilege on the library to the OE user by executing the following command:

```
GRANT EXECUTE ON c_code TO OE;
```

- 4) Publish the external C routine. As the OE user, create a function named call\_c. This function has one numeric parameter and it returns a binary integer. Identify the AS LANGUAGE, LIBRARY, and NAME clauses of the function.
- 5) Create a procedure to call the call\_c function that was created in the previous step. Name this procedure C\_OUTPUT. It has one numeric parameter. Include a DBMS\_OUTPUT.PUT\_LINE statement so that you can view the results returned from your C function.
- 6) Set the SERVEROUTPUT ON and execute the C\_OUTPUT procedure.



## ***Practice 6-1: Using Advanced Interface Methods (continued)***

### **Calling Java from PL/SQL**

A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (4 digits followed by a space). The name of the .class file is `FormatCreditCardNo.class`. The method is defined as:

```
public class FormatCreditCardNo
{
    public static final void formatCard(String[] cardno)
    {
        int count=0, space=0;
        String oldcc=cardno[0];
        String[] newcc= {" "};
        while (count<16)
        {
            newcc[0]+= oldcc.charAt(count);
            space++;
            if (space ==4)
            { newcc[0]+=" "; space=0; }
            count++;
        }
        cardno[0]=newcc [0];
    }
}
```

- 7) Load the .java source file.
- 8) Publish the Java class method by defining a PL/SQL procedure named `CCFORMAT`. This procedure accepts one IN OUT parameter. Use the following definition for the NAME parameter:

```
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

- 9) Execute the Java class method. Define one SQL\*Plus or Oracle SQL Developer variable, initialize it, and use the `EXECUTE` command to execute the `CCFORMAT` procedure.

Your output should match the `PRINT` output as shown here:

```
VARIABLE x VARCHAR2(20)
EXECUTE :x := '1234567887654321'
EXECUTE ccformat(:x)
PRINT x
```

```
anonymous block completed
anonymous block completed
X
-----
1234 5678 8765 4321  1234 5678 1234 5678
```

## Practice Solutions 6-1: Using Advanced Interface Methods

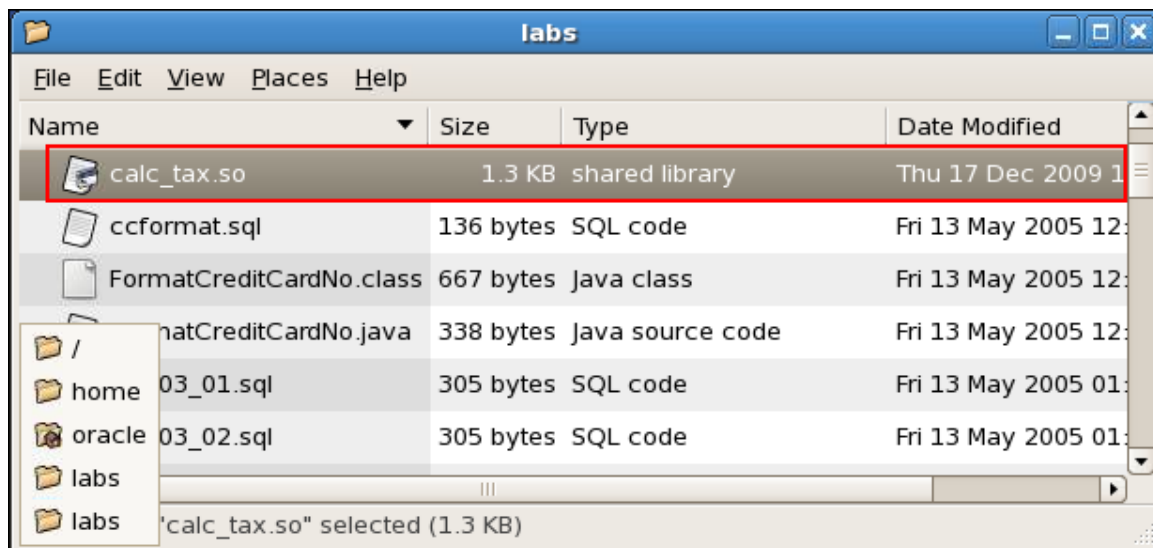
### Using External C Routines

An external C routine definition is created for you. The .c file is stored in the /home/oracle/labs/labs directory. This function returns the tax amount based on the total sales figure that is passed to the function as a parameter. The .c file is named calc\_tax.c. The function is defined as:

```
#include <ctype.h>
int calc_tax(int n)
{
    int tax;
    tax = (n*8)/100;
    return(tax);
}
```

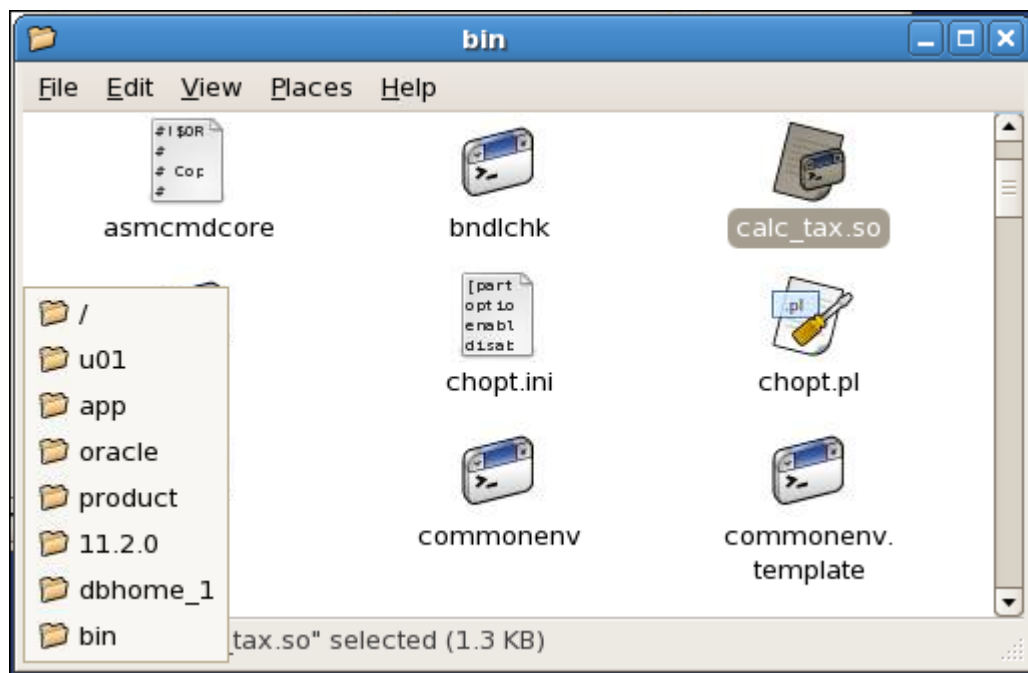
- 1) A shared library file called calc\_tax.so was created for you. Copy the file from the /home/oracle/labs/labs directory into your /u01/app/oracle/product/11.2.0/dbhome\_1/bin directory.

**Open the /home/oracle/labs/labs directory. Select calc\_tax.so . Select Edit > Copy.**



**Navigate to the /u01/app/oracle/product/11.2.0/dbhome\_1/bin folder. Right-click the BIN directory and select Paste from the submenu.**

## Practice Solutions 6-1: Using Advanced Interface Methods (continued)



- 2) Connect to the sys connection, create the alias library object. Name the library object `c_code` and define its path as:

```
CREATE OR REPLACE LIBRARY c_code
AS '/u01/app/oracle/product/11.2.0/dbhome_1/bin/calc_tax.so';
/
```

- 3) Grant execute privilege on the library to the OE user by executing the following command:

```
GRANT EXECUTE ON c_code TO OE;
```

- 4) Publish the external C routine. As the OE user, create a function named `call_c`. This function has one numeric parameter and it returns a binary integer. Identify the `AS LANGUAGE`, `LIBRARY`, and `NAME` clauses of the function.

```
CREATE OR REPLACE FUNCTION call_c
(x BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
LIBRARY sys.c_code
NAME "calc_tax";
/
```

## ***Practice Solutions 6-1: Using Advanced Interface Methods (continued)***

- 5) Create a procedure to call the `call_c` function created in the previous step. Name this procedure `C_OUTPUT`. It has one numeric parameter. Include a `DBMS_OUTPUT.PUT_LINE` statement so that you can view the results returned from your C function.

```
CREATE OR REPLACE PROCEDURE c_output
  (p_in IN BINARY_INTEGER)
IS
  i BINARY_INTEGER;
BEGIN
  i := call_c(p_in);
  DBMS_OUTPUT.PUT_LINE('The total tax is: ' || i);
END c_output;
/
```

- 6) Set `SERVEROUTPUT ON` and execute the `C_OUTPUT` procedure.

```
SET SERVEROUTPUT ON

EXECUTE c_output(1000000)
```

```
anonymous block completed
The total tax is: 80000
```

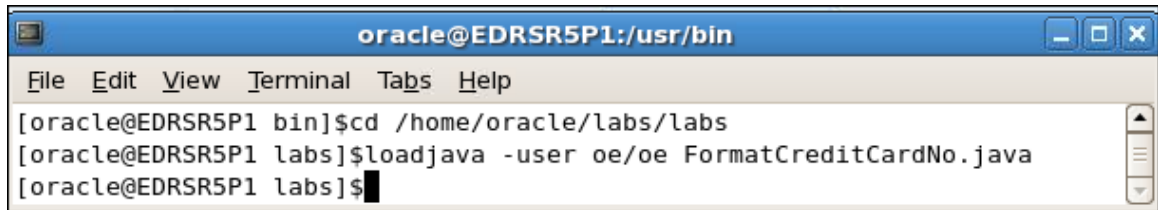
### **Calling Java from PL/SQL**

A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (four digits followed by a space). The name of the `.class` file is `FormatCreditCardNo.class`. The method is defined as:

```
public class FormatCreditCardNo
{
  public static final void formatCard(String[] cardno)
  {
    int count=0, space=0;
    String oldcc=cardno[0];
    String[] newcc= {" "};
    while (count<16)
    {
      newcc[0]+= oldcc.charAt(count);
      space++;
      if (space ==4)
      { newcc[0]+=" "; space=0; }
      count++;
    }
    cardno[0]=newcc [0];
  }
}
```

## Practice Solutions 6-1: Using Advanced Interface Methods (continued)

- 7) Load the .java source file.



```
oracle@EDRSR5P1:/usr/bin
File Edit View Terminal Tabs Help
[oracle@EDRSR5P1 bin]$cd /home/oracle/labs/labs
[oracle@EDRSR5P1 labs]$loadjava -user oe/oe FormatCreditCardNo.java
[oracle@EDRSR5P1 labs]$
```

- 8) Publish the Java class method by defining a PL/SQL procedure named CCFORMAT. This procedure accepts one IN OUT parameter. Use the following definition for the NAME parameter:

```
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';

CREATE OR REPLACE PROCEDURE ccformat
(x IN OUT VARCHAR2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
/
```

- 9) Execute the Java class method. Define one SQL\*Plus or Oracle SQL Developer variable, initialize it, and use the EXECUTE command to execute the CCFORMAT procedure. Your output should match the PRINT output shown here:

```
EXECUTE ccformat(:x);

X
-----
1234 5678 8765 4321
```

```
VARIABLE x VARCHAR2(20)
EXECUTE :x := '1234567887654321'

EXECUTE ccformat(:x)

PRINT x
```

## Practices for Lesson 7

In this practice, you measure and examine performance and tuning, and you tune some of the code that you created for the OE application.

- Break a previously built subroutine into smaller executable sections
- Pass collections into subroutines
- Add error handling for BULK INSERT

## Practice 7-1: Performance and Tuning

### Writing Better Code

- 1) Open the lab\_07\_01.sql file and examine the package. The package body is shown here:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;

        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type,
                                           p_card_no);

            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
```

## Practice 7-1: Performance and Tuning (continued)

```
v_card_info typ_cr_card_nst;  
i INTEGER;  
BEGIN  
  SELECT credit_cards  
    INTO v_card_info  
    FROM customers  
    WHERE customer_id = p_cust_id;  
  IF v_card_info.EXISTS(1) THEN  
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP  
      DBMS_OUTPUT.PUT('Card Type: ' ||  
        v_card_info(idx).card_type || ' ');  
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||  
        v_card_info(idx).card_num );  
    END LOOP;  
  ELSE  
    DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');  END IF;  
END display_card_info;  
END credit_card_pkg;  -- package body  
/
```

This code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.
- The same SELECT statement is run in the two procedures.
- The same IF v\_card\_info.EXISTS(1) THEN statement is in the two procedures.

### Using Efficient Data Types

2) To improve the code, make the following modifications:

- a) Change the local INTEGER variables to use a more efficient data type.
- b) Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg  
IS  
  FUNCTION cust_card_info  
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )  
    RETURN BOOLEAN;  
  PROCEDURE update_card_info  
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no  
    VARCHAR2);  
  PROCEDURE display_card_info  
    (p_cust_id NUMBER);  
END credit_card_pkg;  -- package spec  
/
```



### **Practice 7-1: Performance and Tuning (continued)**

- c) Have the function return TRUE if the customer has credit cards. The function should return FALSE if the customer does not have credit cards. Pass an uninitialized nested table into the function. The function places the credit card information into this uninitialized parameter.

- 3) Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info -
      (120, 'AM EX', 5555555555)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 44444444
Card Type: AM EX / Card No: 5555555555

PL/SQL procedure successfully completed.
```

**Note:** If you did not complete Practice 4, your results will be:

```
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: AM EX / Card No: 5555555555

PL/SQL procedure successfully completed.
```

- 4) You must modify the UPDATE\_CARD\_INFO procedure to return information (by using the RETURNING clause) about the credit cards being updated. Assume that this information will be used by another application developer in your team, who is writing a graphical reporting utility on customer credit cards.

- a) Open the lab\_07\_04\_a.sql file. It contains the code as modified in step 2.
- b) Modify the code to use the RETURNING clause to find information about the rows that are affected by the UPDATE statements.

You can test your modified code with the following procedure (contained in lab\_07\_04\_c.sql):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
  v_card_info typ_cr_card_nst;
BEGIN
  credit_card_pkg.update_card_info
    (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

- c) Test your code with the following statements that are set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
PL/SQL procedure successfully completed.
```

## Practice 7-1: Performance and Tuning (continued)

```
SELECT credit_cards FROM customers WHERE customer_id = 125;
```

```
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
```

```
-----
```

```
TYP_CR_CARD_NST(TYP_CR_CARD('AM EX', 123456789))
```

### Collecting Exception Information

5) Now you test exception handling with the SAVE EXCEPTIONS clause.

a) Run the lab\_07\_05\_a.sql file to create a test table:

```
CREATE TABLE card_table  
(accepted_cards VARCHAR2(50) NOT NULL);
```

b) Open the lab\_07\_05\_b.sql file and run the contents:

```
DECLARE  
  type typ_cards is table of VARCHAR2(50);  
  v_cards typ_cards := typ_cards  
  ( 'Citigroup Visa', 'Nationscard MasterCard',  
    'Federal American Express', 'Citizens Visa',  
    'International Discoverer', 'United Diners Club' );  
BEGIN  
  v_cards.Delete(3);  
  v_cards.DELETE(6);  
  FORALL j IN v_cards.first..v_cards.last  
    SAVE EXCEPTIONS  
    EXECUTE IMMEDIATE  
      'insert into card_table (accepted_cards) values (  
:the_card)'  
    USING v_cards(j);  
END;  
/
```

c) Note the output: \_\_\_\_\_

d) Open the lab\_07\_05\_d.sql file and run the contents:

```
DECLARE  
  type typ_cards is table of VARCHAR2(50);  
  v_cards typ_cards := typ_cards  
  ( 'Citigroup Visa', 'Nationscard MasterCard',  
    'Federal American Express', 'Citizens Visa',  
    'International Discoverer', 'United Diners Club' );  
  bulk_errors EXCEPTION;  
  PRAGMA exception_init (bulk_errors, -24381 );  
BEGIN  
  v_cards.Delete(3);  
  v_cards.DELETE(6);
```

## Practice 7-1: Performance and Tuning (continued)

```
FORALL j IN v_cards.first..v_cards.last
  SAVE EXCEPTIONS
  EXECUTE IMMEDIATE
    'insert into card_table (accepted_cards) values (
:the_card)'
  USING v_cards(j);
EXCEPTION
  WHEN bulk_errors THEN
    FOR j IN 1..sql%bulk_exceptions.count
    LOOP
      Dbms_Output.Put_Line (
        TO_CHAR( sql%bulk_exceptions(j).error_index ) || ' :
        ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );
    END LOOP;
END;
/
```

e) Note the output:\_\_\_\_\_

f) Why is the output different?

### Timing Performance of SIMPLE\_INTEGER and PLS\_INTEGER

- 6) Now you compare the performance between the PLS\_INTEGER and SIMPLE\_INTEGER data types with native compilation:
- a) Run the lab\_07\_06\_a.sql file to create a testing procedure that contains conditional compilation:

```
CREATE OR REPLACE PROCEDURE p
IS
  t0          NUMBER :=0;
  t1          NUMBER :=0;

  $IF $$Simple $THEN
    SUBTYPE My_Integer_t IS                SIMPLE_INTEGER;
    My_Integer_t_Name CONSTANT VARCHAR2(30) := 'SIMPLE_INTEGER';
  $ELSE
    SUBTYPE My_Integer_t IS                PLS_INTEGER;
    My_Integer_t_Name CONSTANT VARCHAR2(30) := 'PLS_INTEGER';
  $END

  v00 My_Integer_t := 0;      v01 My_Integer_t := 0;
  v02 My_Integer_t := 0;      v03 My_Integer_t := 0;
  v04 My_Integer_t := 0;      v05 My_Integer_t := 0;

  two          CONSTANT My_Integer_t := 2;
  lmt          CONSTANT My_Integer_t := 100000000;

BEGIN
  t0 := DBMS_UTILITY.GET_CPU_TIME();
  WHILE v01 < lmt LOOP
```

### ***Practice 7-1: Performance and Tuning (continued)***

```
v00 := v00 + Two;
v01 := v01 + Two;
v02 := v02 + Two;
v03 := v03 + Two;
v04 := v04 + Two;
v05 := v05 + Two;
END LOOP;

IF v01 <> lmt OR v01 IS NULL THEN
  RAISE Program_Error;
END IF;

t1 := DBMS_UTILITY.GET_CPU_TIME();
DBMS_OUTPUT.PUT_LINE(
  RPAD(LOWER($$PLSQL_Code_Type), 15)||
  RPAD(LOWER(My_Integer_t_Name), 15)||
  TO_CHAR((t1-t0), '9999')||' centiseconds');
END p;
```

b) Open the lab\_07\_06\_b.sql file and run the contents:

```
ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = NATIVE PLSQL_CCFlags = 'simple:true'
REUSE SETTINGS;

EXECUTE p()

ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = native PLSQL_CCFlags = 'simple:false'
REUSE SETTINGS;

EXECUTE p()
```

c) Note the output:\_\_\_\_\_

d) Explain the output.

## Practice Solutions 7-1: Performance and Tuning

### Writing Better Code

- 1) Open the lab\_07\_01.sql file and examine the package (the package body is as follows):

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
        p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type,
                                           p_card_no);

            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;
-- continued on next page.
```

## Practice Solutions 7-1: Performance and Tuning (continued)

```
PROCEDURE display_card_info
  (p_cust_id NUMBER)
IS
  v_card_info typ_cr_card_nst;
  i INTEGER;
BEGIN
  SELECT credit_cards
    INTO v_card_info
    FROM customers
    WHERE customer_id = p_cust_id;
  IF v_card_info.EXISTS(1) THEN
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' ||
        v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
        v_card_info(idx).card_num );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Customer has no credit
      cards. ');
  END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

This code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.
- The same SELECT statement is run in the two procedures.
- The same IF v\_card\_info.EXISTS(1) THEN statement is in the two procedures.

### Using Efficient Data Types

2) To improve the code, make the following modifications:

- a) Change the local INTEGER variables to use a more efficient data type.
- b) Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
      p_card_no VARCHAR2);
  PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/
```

## ***Practice Solutions 7-1: Performance and Tuning (continued)***

- c) Have the function return TRUE if the customer has credit cards. The function should return FALSE if the customer does not have credit cards. Pass an uninitialized nested table into the function. The function places the credit card information into this uninitialized parameter.

```
-- note: If you did not complete lesson 4 practice,
-- you will need to run solution files sol_04_04_a,
-- sol_04_04_b, and sol_04_04_c
-- in order to have the supporting structures in place.

CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN;
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN
    IS
        v_card_info_exists BOOLEAN;
    BEGIN
        SELECT credit_cards
            INTO p_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF p_card_info.EXISTS(1) THEN
            v_card_info_exists := TRUE;
        ELSE
            v_card_info_exists := FALSE;
        END IF;
        RETURN v_card_info_exists;
    END cust_card_info;

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i PLS_INTEGER;
    BEGIN
        IF cust_card_info(p_cust_id, v_card_info) THEN
```

## Practice Solutions 7-1: Performance and Tuning (continued)

```
-- cards exist, add more
  i := v_card_info.LAST;
  v_card_info.EXTEND(1);
  v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
  UPDATE customers
    SET credit_cards = v_card_info
    WHERE customer_id = p_cust_id;
ELSE -- no cards for this customer yet, construct one
  UPDATE customers
    SET credit_cards = typ_cr_card_nst
      (typ_cr_card(p_card_type, p_card_no))
    WHERE customer_id = p_cust_id;
END IF;
END update_card_info;

PROCEDURE display_card_info
(p_cust_id NUMBER)
IS
  v_card_info typ_cr_card_nst;
  i PLS_INTEGER;
BEGIN
  IF cust_card_info(p_cust_id, v_card_info) THEN
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' ||
        v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
        v_card_info(idx).card_num );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
```

3) Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info -
  (120, 'AM EX', 5555555555)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 44444444
Card Type: AM EX / Card No: 5555555555

PL/SQL procedure successfully completed.
```

**Note:** If you did not complete Practice 4, your results will be:

```
EXECUTE credit_card_pkg.display_card_info(120)
```



## Practice Solutions 7-1: Performance and Tuning (continued)

Card Type: AM EX / Card No: 5555555555

PL/SQL procedure successfully completed.

- 4) You must modify the UPDATE\_CARD\_INFO procedure to return information (by using the RETURNING clause) about the credit cards being updated. Assume that this information will be used by another application developer on your team, who is writing a graphical reporting utility on customer credit cards.
  - a) Open the lab\_07\_04\_a.sql file. It contains the code as modified in step 2.
  - b) Modify the code to use the RETURNING clause to find information about the rows that are affected by the UPDATE statements.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN;
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN
    IS
        v_card_info_exists BOOLEAN;
    BEGIN
        SELECT credit_cards
            INTO p_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF p_card_info.EXISTS(1) THEN
            v_card_info_exists := TRUE;
        ELSE
            v_card_info_exists := FALSE;
        END IF;
        RETURN v_card_info_exists;
    END cust_card_info;

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst)
    IS
        v_card_info typ_cr_card_nst;
        i PLS_INTEGER;
```

## Practice Solutions 7-1: Performance and Tuning (continued)

```
BEGIN

    IF cust_card_info(p_cust_id, v_card_info) THEN
        -- cards exist, add more
        i := v_card_info.LAST;
        v_card_info.EXTEND(1);
        v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
        UPDATE customers
            SET credit_cards = v_card_info
            WHERE customer_id = p_cust_id
            RETURNING credit_cards INTO o_card_info;
    ELSE -- no cards for this customer yet, construct one
        UPDATE customers
            SET credit_cards = typ_cr_card_nst
                (typ_cr_card(p_card_type, p_card_no))
            WHERE customer_id = p_cust_id
            RETURNING credit_cards INTO o_card_info;
    END IF;
END update_card_info;

PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
                v_card_info(idx).card_type || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
                v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
```

- c) You can test your modified code with the following procedure (contained in lab\_07\_04\_c.sql):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
    v_card_info typ_cr_card_nst;
BEGIN
    credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

## Practice Solutions 7-1: Performance and Tuning (continued)

```
-- execute this code:
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
    v_card_info typ_cr_card_nst;
BEGIN
    credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

anonymous block completed

d) Test your code with the following statements that are set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
PL/SQL procedure successfully completed.
```

```
SELECT credit_cards FROM customers WHERE customer_id = 125;
```

CREDIT_CARDS	
OE.TYP_CR_CARD_NST('OE.TYP_CR_CARD(AM EX,123456789)', 'OE.TYP_CR_CARD(AM EX,123456789)')	
1 rows selected	

### Collecting Exception Information

5) Now you test exception handling with the SAVE EXCEPTIONS clause.

a) Run the lab\_07\_05\_a.sql file to create a test table:

```
CREATE TABLE card_table
(accepted_cards VARCHAR2(50) NOT NULL);
```

b) Open the lab\_07\_05\_b.sql file and run the contents:

```
DECLARE
    type typ_cards is table of VARCHAR2(50);
    v_cards typ_cards := typ_cards
    ( 'Citigroup Visa', 'Nationscard MasterCard',
      'Federal American Express', 'Citizens Visa',
      'International Discoverer', 'United Diners Club' );
BEGIN
    v_cards.Delete(3);
    v_cards.DELETE(6);
    FORALL j IN v_cards.first..v_cards.last
        SAVE EXCEPTIONS
        EXECUTE IMMEDIATE
        'insert into card_table (accepted_cards) values
        ( :the_card)'
        USING v_cards(j);
/
```

## Practice Solutions 7-1: Performance and Tuning (continued)

```
END;  
/
```

c) Note the output:

```
Error report:  
ORA-24381: error(s) in array DML  
ORA-06512: at line 10  
24381. 00000 - "error(s) in array DML"  
*Cause:      One or more rows failed in the DML.
```

**This returns an “Error in Array DML (at line 10),” which is not very informative. The cause of this error: One or more rows failed in the DML.**

d) Open the lab\_07\_05\_d.sql file and run the contents:

```
DECLARE  
  type typ_cards is table of VARCHAR2(50);  
  v_cards typ_cards := typ_cards  
  ( 'Citigroup Visa', 'Nationscard MasterCard',  
    'Federal American Express', 'Citizens Visa',  
    'International Discoverer', 'United Diners Club' );  
  bulk_errors EXCEPTION;  
  PRAGMA exception_init (bulk_errors, -24381 );  
BEGIN  
  v_cards.Delete(3);  
  v_cards.DELETE(6);  
  FORALL j IN v_cards.first..v_cards.last  
    SAVE EXCEPTIONS  
    EXECUTE IMMEDIATE  
    'insert into card_table (accepted_cards) values (  
      :the_card)'  
    USING v_cards(j);  
EXCEPTION  
  WHEN bulk_errors THEN  
    FOR j IN 1..sql%bulk_exceptions.count  
    LOOP  
      Dbms_Output.Put_Line (   
        TO_CHAR( sql%bulk_exceptions(j).error_index ) || ':  
        ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );  
    END LOOP;  
END;  
/
```

## Practice Solutions 7-1: Performance and Tuning (continued)

e) Note the output:

```
B:
      ORA-22160: element at index 1 does not exist
```

f) Why is the output different?

**The PL/SQL block raises the exception 22160 when it encounters an array element that was deleted. The exception is handled and the block is completed successfully.**

### Timing Performance of SIMPLE\_INTEGER and PLS\_INTEGER

6) Now you compare the performance between the PLS\_INTEGER and SIMPLE\_INTEGER data types with native compilation:

a) Run the lab\_07\_06\_a.sql file to create a testing procedure that contains conditional compilation:

```
CREATE OR REPLACE PROCEDURE p
IS
    t0          NUMBER :=0;
    t1          NUMBER :=0;

    $IF $$Simple $THEN
        SUBTYPE My_Integer_t IS                SIMPLE_INTEGER;
        My_Integer_t_Name CONSTANT VARCHAR2(30) := 'SIMPLE_INTEGER';
    $ELSE
        SUBTYPE My_Integer_t IS                PLS_INTEGER;
        My_Integer_t_Name CONSTANT VARCHAR2(30) := 'PLS_INTEGER';
    $END

    v00 My_Integer_t := 0;      v01 My_Integer_t := 0;
    v02 My_Integer_t := 0;      v03 My_Integer_t := 0;
    v04 My_Integer_t := 0;      v05 My_Integer_t := 0;

    two          CONSTANT My_Integer_t := 2;
    lmt          CONSTANT My_Integer_t := 100000000;

BEGIN
    t0 := DBMS_UTILITY.GET_CPU_TIME();
    WHILE v01 < lmt LOOP
        v00 := v00 + Two;
        v01 := v01 + Two;
        v02 := v02 + Two;
        v03 := v03 + Two;
        v04 := v04 + Two;
        v05 := v05 + Two;
    END LOOP;

    IF v01 <> lmt OR v01 IS NULL THEN
        RAISE Program_Error;
```

## Practice Solutions 7-1: Performance and Tuning (continued)

```
END IF;

t1 := DBMS_UTILITY.GET_CPU_TIME();
DBMS_OUTPUT.PUT_LINE(
  RPAD(LOWER($$PLSQL_Code_Type), 15)||
  RPAD(LOWER(My_Integer_t_Name), 15)||
  TO_CHAR((t1-t0), '9999')||' centiseconds');
END p;
/
```

b) Open the lab\_07\_06\_b.sql file and run the contents:

```
ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = NATIVE PLSQL_CCFlags = 'simple:true'
REUSE SETTINGS;

EXECUTE p()

ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = native PLSQL_CCFlags = 'simple:false'
REUSE SETTINGS;

EXECUTE p()
```

c) Note the output:

First run:

native	simple_integer	1049 centiseconds
--------	----------------	-------------------

Second run:

native	pls_integer	1241 centiseconds
--------	-------------	-------------------

d) Explain the output.

The SIMPLE\_INTEGER runs much faster in this scenario. If you can use the SIMPLE\_INTEGER data type, it can improve performance.

## Practices for Lesson 8

In this practice, you implement SQL query result caching and PL/SQL result function caching. You run scripts to measure the cache memory values, manipulate queries and functions to turn caching on and off, and then examine cache statistics.

## Practice 8-1: Improving Performance with Caching

In this practice, you examine the Explain Plan for a query, add the `RESULT_CACHE` hint to the query, and reexamine the Explain Plan results. You also execute some code and modify the code so that PL/SQL result caching is turned on.

### Examining SQL and PL/SQL Result Caching

- 1) Use SQL Developer to connect to the OE schema. Examine the Explain Plan for the following query, which is found in the `lab_08_01.sql` lab file. To view the Explain Plan, click the Execute Explain Plan button on the toolbar in the Code Editor window.

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

- 2) Add the `RESULT_CACHE` hint to the query and reexamine the Explain Plan results.

```
SELECT /*+ result_cache */
       count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

Examine the Explain Plan results, compared to the previous results.

- 3) The following code is used to generate a list of warehouse names for pick lists in applications. The `WAREHOUSES` table is fairly stable and is not modified often.

Click the Run Script button to compile this code (You can use the `lab_08_03.sql` file):

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);
/

CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
    v_count BINARY_INTEGER;
    v_wh_names list_typ;
BEGIN
    SELECT count(*)
    INTO v_count
    FROM warehouses;
    FOR i in 1..v_count LOOP
        SELECT warehouse_name
        INTO v_wh_names(i)
        FROM warehouses;
```



### ***Practice 8-1: Improving Performance with Caching (continued)***

```
END LOOP;  
RETURN v_wh_names;  
END get_warehouse_names;
```

- 4) Because the function is called frequently, and because the content of the data returned does not change frequently, this code is a good candidate for PL/SQL result caching.

Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

## Practice Solutions 8-1: Improving Performance with Caching

In this practice, you examine the Explain Plan for a query, add the `RESULT_CACHE` hint to the query, and reexamine the Explain Plan results. You also execute some code and modify the code so that PL/SQL result caching is turned on.

### Examining SQL and PL/SQL Result Caching

- 1) Use SQL Developer to connect to the OE schema. Examine the Explain Plan for the following query, which is found in the `lab_08_01.sql` file. To view the Explain Plan, click the Execute Explain Plan button on the toolbar in the Code Editor window.

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

In Oracle SQL Developer, open the `lab_08_01.sql` file:

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

Select the OE connection

Click the Execute Explain Plan button on the toolbar and observe the results in the lower region:



**Results: SQL caching is not enabled and not visible in the Explain Plan.**

- 2) Add the `RESULT_CACHE` hint to the query and reexamine the Explain Plan results.

```
SELECT /*+ result_cache */
       count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

Examine the Explain Plan results, compared to the previous Results.

**Add the following code immediately after the `SELECT` command:**

```
/*+ result_cache */
```

## Practice Solutions 8-1: Improving Performance with Caching (continued)

Click the Execute Explain Plan button on the toolbar again, and compare the results in the lower region with the previous results:

Script Output Explain Plan		
0.006 seconds		
OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
RESULT CACHE	3anmja8pzpa8kc5f0pw7hdfjju	
HASH		GROUP BY
HASH JOIN		
Access Predicates		
INVENTORIES.PRODUCT_ID=		
TABLE ACCESS	PRODUCT_INFORMATION	FULL
TABLE ACCESS	INVENTORIES	FULL

**Results:** Note that result caching is used in the Explain Plan.

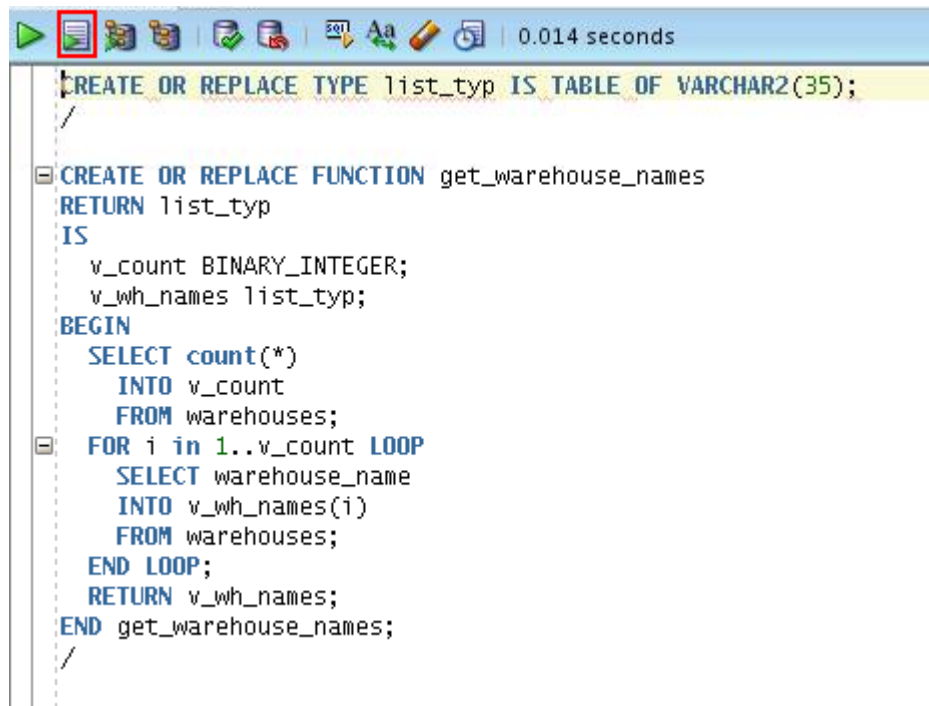
- 3) The following code is used to generate a list of warehouse names for pick lists in applications. The WAREHOUSES table is fairly stable and is not modified often.

Click the Run Script button to compile this code (You can use the lab\_08\_03.sql file.):

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);
/

CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
    v_count BINARY_INTEGER;
    v_wh_names list_typ;
BEGIN
    SELECT count(*)
    INTO v_count
    FROM warehouses;
    FOR i in 1..v_count LOOP
        SELECT warehouse_name
        INTO v_wh_names(i)
        FROM warehouses;
    END LOOP;
    RETURN v_wh_names;
END get_warehouse_names;
```

## Practice Solutions 8-1: Improving Performance with Caching (continued)



The screenshot shows a SQL IDE window with a toolbar at the top. The 'Run Script' button, represented by a green play icon, is highlighted with a red rectangle. The main editor area contains the following PL/SQL code:

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);  
  
CREATE OR REPLACE FUNCTION get_warehouse_names  
RETURN list_typ  
IS  
    v_count BINARY_INTEGER;  
    v_wh_names list_typ;  
BEGIN  
    SELECT count(*)  
    INTO v_count  
    FROM warehouses;  
    FOR i in 1..v_count LOOP  
        SELECT warehouse_name  
        INTO v_wh_names(i)  
        FROM warehouses;  
    END LOOP;  
    RETURN v_wh_names;  
END get_warehouse_names;
```

Open `lab_08_03.sql`. Click the Run Script button. You have compiled the function without PL/SQL result caching.

- 4) Because the function is called frequently, and because the content of the data returned does not frequently change, this code is a good candidate for PL/SQL result caching.

Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

Insert the following line after `RETURN list_typ`:

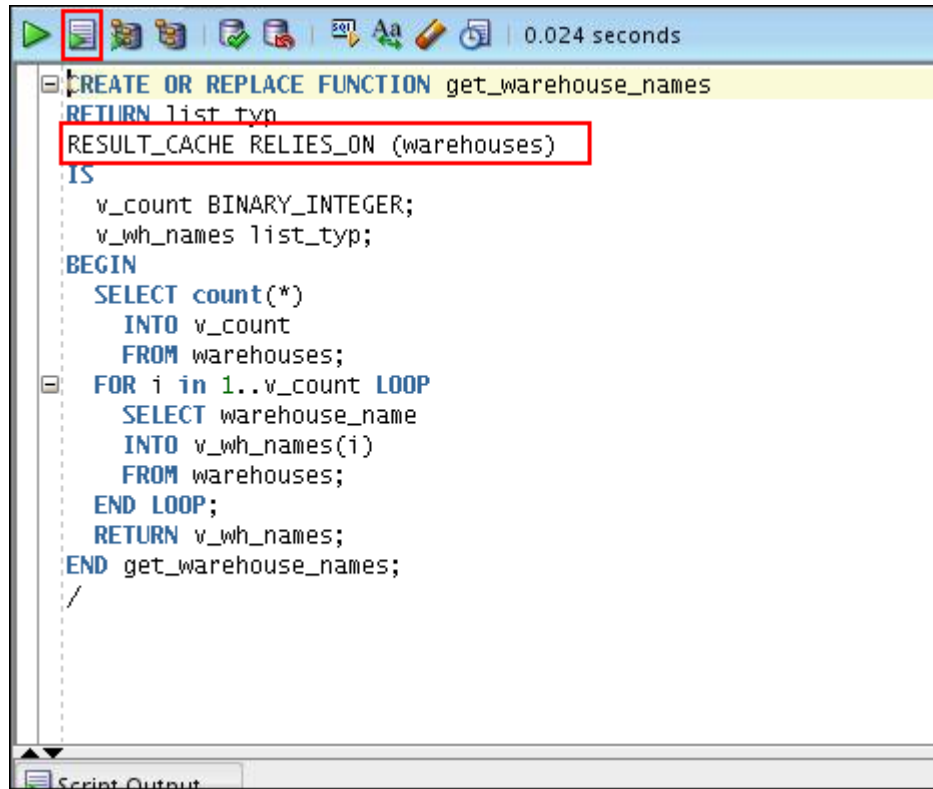
**RESULT\_CACHE RELIES\_ON (warehouses)**

```
CREATE OR REPLACE FUNCTION get_warehouse_names  
RETURN list_typ  
RESULT_CACHE RELIES_ON (warehouses)  
IS  
    v_count BINARY_INTEGER;  
    v_wh_names list_typ;  
BEGIN  
    SELECT count(*)  
    INTO v_count  
    FROM warehouses;  
    FOR i in 1..v_count LOOP  
        SELECT warehouse_name  
        INTO v_wh_names(i)  
        FROM warehouses;
```

## Practice Solutions 8-1: Improving Performance with Caching (continued)

```
END LOOP;  
RETURN v_wh_names;  
END get_warehouse_names;
```

Click the Run Script button to recompile the code.



## Practices for Lesson 9

In this practice you will perform the following:

- Find coding information
- Use PL/Scope
- Use DBMS\_METADATA

## Practice 9-1: Analyzing PL/SQL Code

In this practice, you use PL/SQL and Oracle SQL Developer to analyze your code.

Use your OE connection.

### Finding Coding Information

- 1) Create the QUERY\_CODE\_PKG package to search your source code.
  - a) Run the lab\_09\_01\_a.sql script to create the QUERY\_CODE\_PKG package.
  - b) Run the ENCAP\_COMPLIANCE procedure to see which of your programs reference tables or views. (**Note:** Your results might differ slightly.)
  - c) Run the FIND\_TEXT\_IN\_CODE procedure to find all references to 'ORDERS'. (**Note:** Your results might differ slightly.)
  - d) Use the SQL Developer Reports feature to find the same results for step C shown above.

### Using PL/Scope

- 2) In the following steps, you use PL/Scope.

- a) Enable your session to collect identifiers.

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
```

- b) Recompile your CREDIT\_CARD\_PKG code.

```
ALTER PACKAGE credit_card_pkg COMPILE;
```

- c) Verify that your PLSCOPE\_SETTING is set correctly by issuing the following statement:

```
SELECT PLSCOPE_SETTINGS
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';
```

- d) Execute the following statement to create a hierarchical report on the identifier information about the CREDIT\_CARD\_PKG code. You can run the lab\_09\_02\_d.sql script file.

```
WITH v AS
(
  SELECT      Line,
              Col,
              INITCAP(NAME) Name,
              LOWER(TYPE)   Type,
              LOWER(USAGE)  Usage,
              USAGE_ID, USAGE_CONTEXT_ID
  FROM USER_IDENTIFIERS
  WHERE Object_Name = 'CREDIT_CARD_PKG'
     AND Object_Type = 'PACKAGE BODY' )
  SELECT RPAD(LPAD(' ', 2*(Level-1)) ||
              Name, 20, '.') || ' ' ||
              RPAD(Type, 20) || RPAD(Usage, 20)
              IDENTIFIER_USAGE_CONTEXTS
  FROM v
```

### **Practice 9-1: Analyzing PL/SQL Code (continued)**

```
START WITH USAGE_CONTEXT_ID = 0
CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
ORDER SIBLINGS BY Line, Col;
```

3) Use DBMS\_METADATA to find the metadata for the ORDER\_ITEMS table.

a) Create the GET\_TABLE\_MD function. You can run the lab\_09\_03\_a.sql script.

```
CREATE FUNCTION get_table_md RETURN CLOB IS
  v_hdl  NUMBER; -- returned by 'OPEN'
  v_th   NUMBER; -- returned by 'ADD_TRANSFORM'
  v_doc  CLOB;
BEGIN
  -- specify the OBJECT TYPE
  v_hdl := DBMS_METADATA.OPEN('TABLE');
  -- use FILTERS to specify the objects desired
  DBMS_METADATA.SET_FILTER(v_hdl, 'SCHEMA', 'OE');
  DBMS_METADATA.SET_FILTER
    (v_hdl, 'NAME', 'ORDER_ITEMS');
  -- request to be TRANSFORMED into creation DDL
  v_th := DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL');
  -- FETCH the object
  v_doc := DBMS_METADATA.FETCH_CLOB(v_hdl);
  -- release resources
  DBMS_METADATA.CLOSE(v_hdl);
  RETURN v_doc;
END;
/
```

b) Issue the following statements to view the metadata generated from the GET\_TABLE\_MD function:

```
set pagesize 0
set long 1000000
SELECT get_table_md FROM dual;
```

c) Generate an XML representation of the ORDER\_ITEMS table by using the DBMS\_METADATA.GET\_XML function. Spool the output to a file named ORDER\_ITEMS\_XML.txt in the /home/oracle/labs/labs folder.

d) Verify that the ORDER\_ITEMS\_XML.txt file was created in the /home/oracle/labs/labs folder.



## Practice Solutions 9-1: Analyzing PL/SQL Code

In this practice, you use PL/SQL and Oracle SQL Developer to analyze your code.

Use your OE connection.

### Finding Coding Information

- 1) Create the QUERY\_CODE\_PKG package to search your source code.

- a) Run the lab\_09\_01\_a.sql script to create the QUERY\_CODE\_PKG package.

```
@/home/oracle/labs/labs/lab_09_01_a.sql
```

- b) Run the ENCAP\_COMPLIANCE procedure to see which of your programs reference tables or views. (**Note:** Your results might differ slightly.)

```
SET SERVEROUTPUT ON
EXECUTE query_code_pkg.encap_compliance
```

```
anonymous block completed
Programs that reference tables or views
OE.ADD_ORDER_ITEMS,OE.PORDER
OE.CHANGE_CREDIT,OE.CUSTOMERS
OE.CREDIT_CARD_PKG,OE.CUSTOMERS
OE.CUST_DATA,OE.CUSTOMERS
```

- c) Run the FIND\_TEXT\_IN\_CODE procedure to find all references to 'ORDERS'. (**Note:** Your results might differ slightly.)

```
SET SERVEROUTPUT ON
EXECUTE query_code_pkg.find_text_in_code('ORDERS')
```

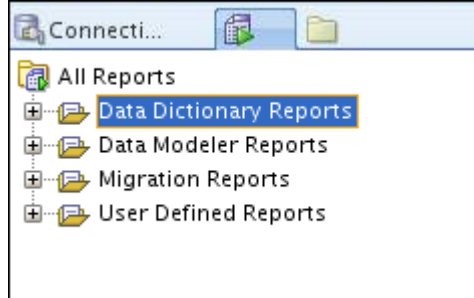
```
anonymous block completed
Checking for presence of ORDERS:
CUSTOMER_TYP-12,      , cust_orders      order_list_typ

GET_AVG_ORDER-8,SELECT customers.customer_id, customers.cust_last_name, AVG(orders.order_total)
GET_AVG_ORDER-10,FROM CUSTOMERS, ORDERS
GET_AVG_ORDER-11,WHERE customers.customer_id=orders.customer_id
```

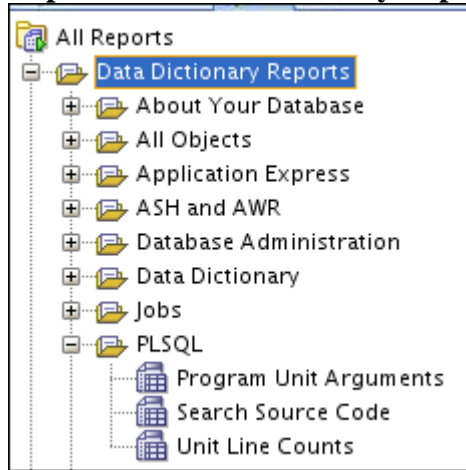
- d) Use the Oracle SQL Developer Reports feature to find the same results obtained in step c.

## Practice Solutions 9-1: Analyzing PL/SQL Code (continued)

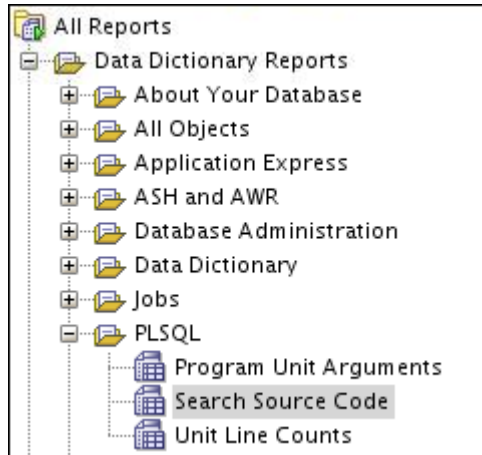
Navigate to the Reports tabbed page in Oracle SQL Developer.



Expand the Data Dictionary Reports node and expand the PL/SQL node.



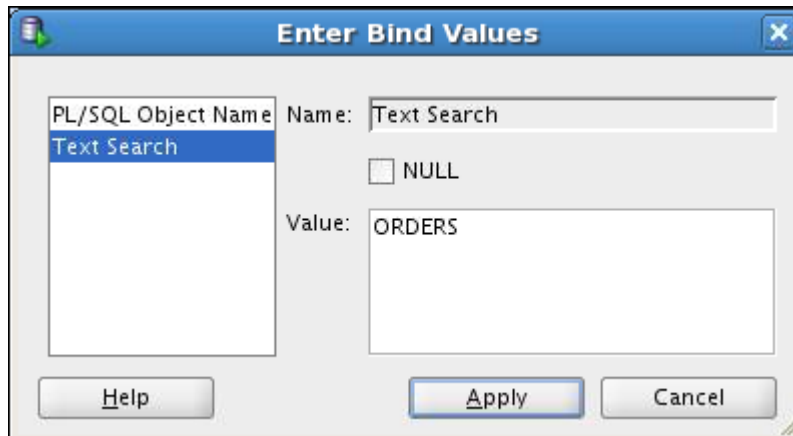
Select Search Source Code and then select your OE connection and click OK.



## Practice Solutions 9-1: Analyzing PL/SQL Code (continued)



Select Text search and enter ORDERS for the Value: field. Click the Apply button.



Owner	PL/SQL Object Name	Type	Line	Text
OE	CUSTOMER_TYP	TYPE	12	, cust_orders order_list_type
OE	GET_AVG_ORDER	PROCEDURE	8	SELECT customers.customer_id, customers.cust_last_
OE	GET_AVG_ORDER	PROCEDURE	10	FROM CUSTOMERS, ORDERS
OE	GET_AVG_ORDER	PROCEDURE	11	WHERE customers.customer_id=orders.customer_id
OE	GET_TABLE_MD	FUNCTION	11	(v_hdl,'NAME','ORDERS');

### Using PL/Scope

2) In the following steps, you use PL/Scope.

a) Enable your session to collect identifiers.

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
```

b) Recompile your CREDIT\_CARD\_PKG code.

```
ALTER PACKAGE credit_card_pkg COMPILE;
```

c) Verify that your PLSCOPE\_SETTING is set correctly by issuing the following statement:

```
SELECT PLSCOPE_SETTINGS
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';
```

## Practice Solutions 9-1: Analyzing PL/SQL Code (continued)

```
PLSCOPE_SETTINGS
-----
IDENTIFIERS:ALL
1 rows selected
```

- d) Execute the following statement to create a hierarchical report on the identifier information on the CREDIT\_CARD\_PKG code. You can run the lab\_09\_02\_d.sql script file.

```
WITH v AS
(
  SELECT      Line,
              Col,
              INITCAP(NAME) Name,
              LOWER(TYPE)  Type,
              LOWER(USAGE) Usage,
              USAGE_ID, USAGE_CONTEXT_ID
  FROM USER_IDENTIFIERS
  WHERE Object_Name = 'CREDIT_CARD_PKG'
        AND Object_Type = 'PACKAGE BODY' )
  SELECT RPAD(LPAD(' ', 2*(Level-1)) ||
              Name, 20, '.') || ' ' ||
              RPAD(Type, 20) || RPAD(Usage, 20)
          IDENTIFIER_USAGE_CONTEXTS
  FROM v
  START WITH USAGE_CONTEXT_ID = 0
  CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
  ORDER SIBLINGS BY Line, Col;
```

```
IDENTIFIER_USAGE_CONTEXTS
-----
Credit_Card_Pkg..... package          definition
  Cust_Card_Info.... function          definition
    P_Cust_Id..... formal in          declaration
      Number..... number datatype    reference
    P_Card_Info..... formal in out    declaration
      Typ_Cr_Card_Ns nested table      reference
    Boolean..... boolean datatype     reference
    V_Card_Info_Exis variable          declaration
      Boolean..... boolean datatype    reference
    P_Card_Info..... formal in out      assignment
    P_Cust_Id..... formal in          reference
    V_Card_Info_Exis variable          assignment
    V_Card_Info_Exis variable          reference
  Update_Card_Info.. procedure          definition
    P_Cust_Id..... formal in          declaration
      Number..... number datatype    reference
```

- 3) Use DBMS\_METADATA to find the metadata for the ORDER\_ITEMS table.
- a) Create the GET\_TABLE\_MD function. You can run the lab\_09\_03\_a.sql script.

## Practice Solutions 9-1: Analyzing PL/SQL Code (continued)

```
CREATE OR REPLACE FUNCTION get_table_md RETURN CLOB IS
  v_hdl  NUMBER; -- returned by 'OPEN'
  v_th   NUMBER; -- returned by 'ADD_TRANSFORM'
  v_doc  CLOB;
BEGIN
  -- specify the OBJECT TYPE
  v_hdl := DBMS_METADATA.OPEN('TABLE');
  -- use FILTERS to specify the objects desired
  DBMS_METADATA.SET_FILTER(v_hdl, 'SCHEMA', 'OE');
  DBMS_METADATA.SET_FILTER
    (v_hdl, 'NAME', 'ORDER_ITEMS');
  -- request to be TRANSFORMED into creation DDL
  v_th := DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL');
  -- FETCH the object
  v_doc := DBMS_METADATA.FETCH_CLOB(v_hdl);
  -- release resources
  DBMS_METADATA.CLOSE(v_hdl);
  RETURN v_doc;
END;
/
```

- b) Issue the following statements to view the metadata generated from the GET\_TABLE\_MD function:

```
set pagesize 0
set long 1000000

SELECT get_table_md FROM dual;
```

GET_TABLE_MD
<pre>CREATE TABLE "OE"."ORDER_ITEMS" ( "ORDER_ID" NUMBER(12,0),   "LINE_ITEM_ID" NUMBER(3,0) NOT NULL ENABLE,   "PRODUCT_ID" NUMBER(6,0) NOT NULL ENABLE,   "UNIT_PRICE" NUMBER(8,2),   "QUANTITY" NUMBER(8,0),   CONSTRAINT "ORDER_ITEMS_PK" PRIMARY KEY ("ORDER_ID", "LINE_ITEM_ID")   USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 NOLOGGING COMPUTE STATISTICS   STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645   PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)   TABLESPACE "EXAMPLE" ENABLE,   CONSTRAINT "ORDER_ITEMS_PRODUCT_ID_FK" FOREIGN KEY ("PRODUCT_ID")     REFERENCES "OE"."PRODUCT_INFORMATION" ("PRODUCT_ID") ENABLE,   CONSTRAINT "ORDER_ITEMS_ORDER_ID_FK" FOREIGN KEY ("ORDER_ID")     REFERENCES "OE"."ORDERS" ("ORDER_ID") ON DELETE CASCADE ENABLE NOVALIDATE ) SEGMENT CREATION IMMEDIATE PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESSION NOLOGGING STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645 PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT) TABLESPACE "EXAMPLE"</pre>

## Practice Solutions 9-1: Analyzing PL/SQL Code (continued)

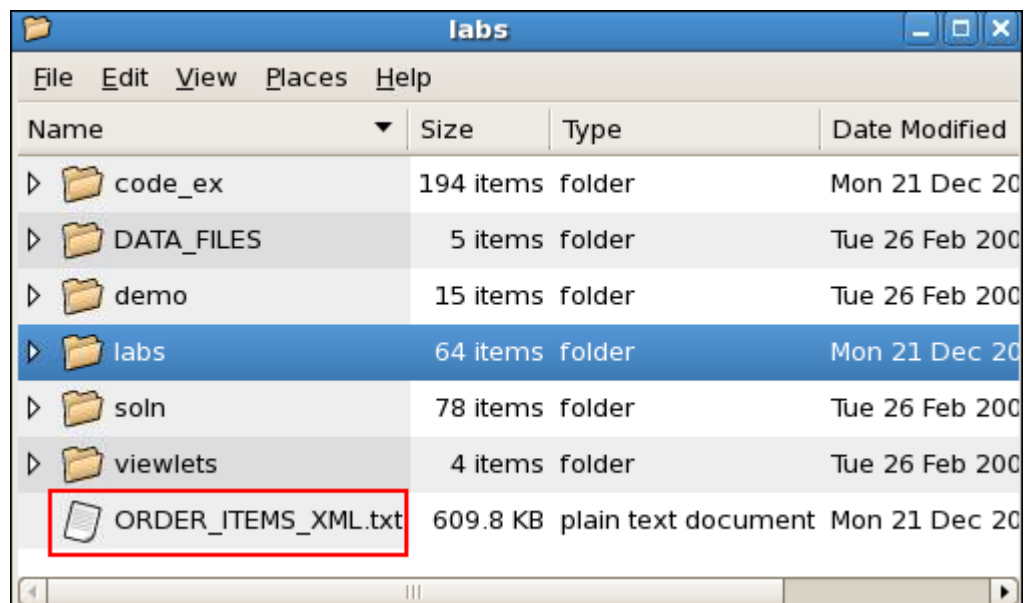
- c) Generate an XML representation of the ORDER\_ITEMS table by using the DBMS\_METADATA.GET\_XML function. Spool the output to a file named ORDER\_ITEMS\_XML.txt in the /home/oracle/labs/labs folder.

```
SPOOL /home/oracle/labs/labs/ORDER_ITEMS_XML.txt

SELECT DBMS_METADATA.GET_XML
       ('TABLE', 'ORDER_ITEMS', 'OE')
FROM   dual;

SPOOL OFF
```

- d) Verify that the ORDER\_ITEMS\_XML.txt file was created in the /home/oracle/labs/labs folder.



## Practice Solutions 9-1: Analyzing PL/SQL Code (continued)

Open the file to verify its contents:

```
> SELECT DBMS_METADATA.GET_XML
        ('TABLE', 'ORDER_ITEMS', 'OE')
FROM    dual

DBMS_METADATA.GET_XML
('TABLE', 'ORDER_ITEMS', 'OE')
-----
<?xml version="1.0"?
><ROWSET><ROW>

<TABLE_T>
  <VERS_MAJOR>1</
VERS_MAJOR>
  <VERS_MINOR>3 </
VERS_MINOR>
  <OBJ_NUM>73965</
OBJ_NUM>

<SCHEMA_OBJ>
  <OBJ_NUM>73965</
OBJ_NUM>
  <DATAOBJ_NUM>73339</
DATAOBJ_NUM>
  <OWNER_NUM>86</
OWNER_NUM>
  <OWNER_NAME>OE</
```

---

## Practices for Lesson 10

---

In this practice you write code to profile components in your application



## ***Practice 10-1: Profiling and Tracing PL/SQL Code***

In this practice, you generate profiler data and analyze it.

Use your OE connection.

- 1) Generate profiling data for your CREDIT\_CARD\_PKG.
  - a) Re-create CREDIT\_CARD\_PKG by running the  
`/home/oracle/labs/lab_10_01.sql` script.
  - b) You must identify the location of the profiler files. Create a DIRECTORY object to identify this information.  
  

```
CREATE DIRECTORY profile_data AS /home/oracle/labs/labs';
```
  - c) Use DBMS\_HPROF.START\_PROFILING to start the profiler for your session.
  - d) Run your CREDIT\_CARD\_PKG.UPDATE\_CARD\_INFO with the following data.  
  

```
credit_card_pkg.update_card_info  
  (154, 'Discover', '123456789');
```
  - e) Use DBMS\_HPROF.STOP\_PROFILING to stop the profiler.
- 2) Run the dbmshtab.sql script, located in the  
`/u01/app/oracle/product/11.2.0/dbhome_1/rdbms/admin` folder, to set up the profiler tables.
- 3) Use DBMS\_HPROF.ANALYZE to analyze the raw data and write the information to the profiler tables.
  - a) Get the RUN\_ID.
  - b) Query the DBMSHP\_RUNS table to find top-level information for the RUN\_ID that you retrieved.
  - c) Query the DBMSHP\_FUNCTION\_INFO table to find information about each function profiled.
- 4) Use the plshprof command-line utility to generate simple HTML reports directly from the raw profiler data.
  - a) Open a command window.
  - b) Change the working directory to `/home/oracle/labs/labs`
  - c) Run the plshprof utility.
- 5) Open the report in your browser and review the data.

## Practice Solutions 10-1: Profiling and Tracing PL/SQL Code

In this practice, you generate profiler data and analyze it.

Use your OE connection.

1) Generate profiling data for your CREDIT\_CARD\_PKG.

- a) Re-create CREDIT\_CARD\_PKG by running the  
/home/oracle/labs/labs/lab\_10\_01.sql script.
- b) You must identify the location of the profiler files. Create a DIRECTORY object to identify this information:

```
CREATE DIRECTORY profile_data AS '/home/oracle/labs/labs';
```

c) Use DBMS\_HPROF.START\_PROFILING to start the profiler for your session.

```
BEGIN
-- start profiling
  DBMS_HPROF.START_PROFILING('PROFILE_DATA', 'pd_cc_pkg.txt');
END;
/
```

d) Run your CREDIT\_CARD\_PKG.UPDATE\_CARD\_INFO with the following data.

```
credit_card_pkg.update_card_info
(154, 'Discover', '123456789');
```

```
DECLARE
  v_card_info typ_cr_card_nst;
BEGIN
-- run application
  credit_card_pkg.update_card_info
    (154, 'Discover', '123456789');
END;
/
```

e) Use DBMS\_HPROF.STOP\_PROFILING to stop the profiler.

```
BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;
/
```

2) Run the dbmshptab.sql script, located in the  
/u01/app/oracle/product/11.2.0/dbhome\_1/rdbms/admin folder. to set  
up the profiler tables.

```
@/u01/app/oracle/product/11.2.0/dbhome_1/rdbms/admin/dbmshptab.sql
```

3) Use DBMS\_HPROF.ANALYZE to analyze the raw data and write the information to the profiler tables.

a) Get the RUN\_ID.

```
SET SERVEROUTPUT ON
```

## Practice Solutions 10-1: Profiling and Tracing PL/SQL Code (continued)

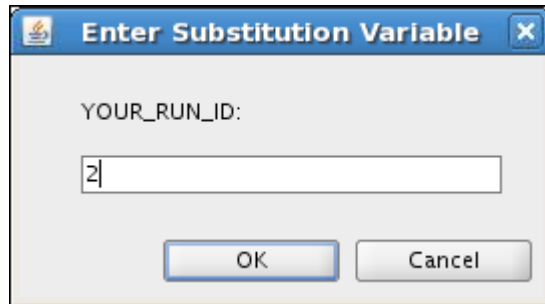
```
DECLARE
    v_runid NUMBER;
BEGIN
    v_runid := DBMS_HPROF.ANALYZE (LOCATION => 'PROFILE_DATA',
                                    FILENAME => 'pd_cc_pkg.txt');
    DBMS_OUTPUT.PUT_LINE('Run ID: ' || v_runid);
END;
/
```

Run ID: 2

- b) Query the DBMSHP\_RUNS table to find top-level information for the RUN\_ID that you retrieved.

```
SET VERIFY OFF

SELECT runid, run_timestamp, total_elapsed_time
FROM dbmshp_runs
WHERE runid = &your_run_id;
```



RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
2	21-DEC-09 05.34.37.484184000 AM	1621

1 rows selected


- c) Query the DBMSHP\_FUNCTION\_INFO table to find information on each function profiled.

```
SELECT owner, module, type, function_line#, namespace,
       calls, function_elapsed_time
FROM   dbmshp_function_info
WHERE  runid = 2;
```

## Practice Solutions 10-1: Profiling and Tracing PL/SQL Code (continued)

Script Output

Query R...

 All Rows Fetched: 9 in 0.003 seconds

	OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS
1	(null)	(null)	(null)	__anonymous_block	PLSQL	2
2	(null)	(null)	(null)	__anonymous_block@1	PLSQL	4
3	(null)	(null)	(null)	__plsql_vm	PLSQL	2
4	(null)	(null)	(null)	__plsql_vm@1	PLSQL	4
5	OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	1
6	OE	ORDERS_APP_PKG	PACKAGE BODY	THE_PREDICATE	PLSQL	4
7	SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1
8	OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line17	SQL	1
9	OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line9	SQL	1

- 4) Use the `plshprof` command-line utility to generate simple HTML reports directly from the raw profiler data.
  - a) Open a command window.
  - b) Change the working directory to `/home/oracle/labs/labs`.
  - c) Run the `plshprof` utility.

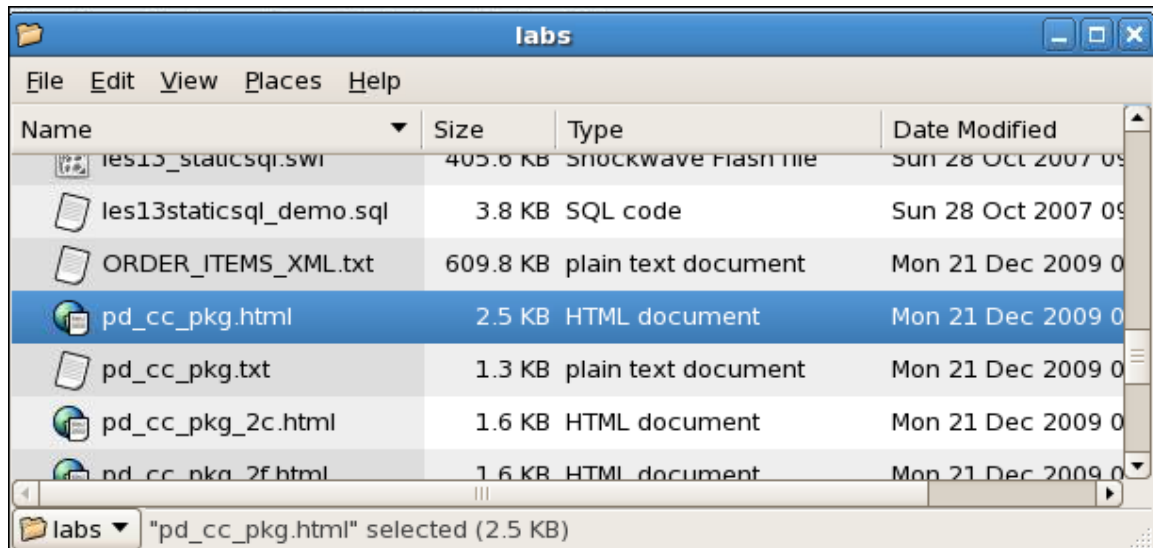
```
--at your command window, change your working directory to
/home/oracle/labs/labs
cd /home/oracle/labs/labs
plshprof -output pd_cc_pkg pd_cc_pkg.txt
```

```
oracle@EDRSR5P1:/usr/bin
File Edit View Terminal Tabs Help
[oracle@EDRSR5P1 admin]$cd /home/oracle/labs/labs
[oracle@EDRSR5P1 labs]$plshprof -output pd_cc_pkg pd_cc_pkg.txt
PLSHPROF: Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - P
duction
[9 symbols processed]
[Report written to 'pd_cc_pkg.html']
[oracle@EDRSR5P1 labs]$
```

- 5) Open the report in your browser and review the data.
 

**Navigate to the `/home/oracle/labs/labs` folder.**

## Practice Solutions 10-1: Profiling and Tracing PL/SQL Code (continued)



In this practice you:

- Create an application context
- Create a policy
- Create a logon trigger
- Implement a virtual private database
- Test the virtual private database

## Practice 11-1: Implementing Fine-Grained Access Control for VPD

In this practice, you define an application context and security policy to implement the policy: “Sales Representatives can see only their own order information in the `ORDERS` table.” You create sales representative IDs to test the success of your implementation. Examine the definition of the `ORDERS` table and the `ORDER` count for each sales representative:

```
DESCRIBE orders
```

Name	Null?	Type
ORDER_ID	NOT NULL	NUMBER(12)
ORDER_DATE	NOT NULL	TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE		VARCHAR2(8)
CUSTOMER_ID	NOT NULL	NUMBER(6)
ORDER_STATUS		NUMBER(2)
ORDER_TOTAL		NUMBER(8,2)
SALES_REP_ID		NUMBER(6)
PROMOTION_ID		NUMBER(6)

```
SELECT sales_rep_id, count(*)
FROM   orders
GROUP BY sales_rep_id;
```

SALES_REP_ID	COUNT(*)
153	5
154	10
155	5
156	5
158	7
159	7
160	6
161	13
163	12
	35

10 rows selected.

- 1) Use your `SYS` connection. Examine and then run the `lab_11_01.sql` script. This script creates the sales representative ID accounts with appropriate privileges to access the database.
- 2) Set up an application context:
  - a) Connect to the database as `SYS` before creating this context.
  - b) Create an application context named `sales_orders_ctx`.
  - c) Associate this context to the `oe.sales_orders_pkg`.
- 3) Connect as `OE`.

## Practice 11-1: Implementing Fine-Grained Access Control for VPD (continued)

a) Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
  PROCEDURE set_app_context;
  FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END sales_orders_pkg;      -- package spec
/
```

b) Create this package specification and the package body in the OE schema.

c) When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';
```

d) Use these constants in the SET\_APP\_CONTEXT procedure to set the application context to the current user.

4) Connect as SYS and define the policy.

a) Use DBMS\_RLS.ADD\_POLICY to define the policy.

b) Use these specifications for the parameter values:

```
object_schema    OE
object_name      ORDERS
policy_name      OE_ORDERS_ACCESS_POLICY
function_schema  OE
policy_function   SALES_ORDERS_PKG.THE_PREDICATE
statement_types  SELECT, INSERT, UPDATE, DELETE
update_check     FALSE,
enable           TRUE);
```

5) Connect as SYS and create a logon trigger to implement fine-grained access control. Name the trigger SET\_ID\_ON\_LOGON. This trigger causes the context to be set as each user is logged on.

6) Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID    COUNT(*)
-----
153              5
```



## ***Practice 11-1: Implementing Fine-Grained Access Control for VPD (continued)***

```
CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM    orders
GROUP BY sales_rep_id;

SALES_REP_ID    COUNT(*)
-----
154            10
```

### **Note:**

During debugging, you may need to disable or remove some of the objects created for this lesson.

If you need to disable the logon trigger, issue the following command:

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

If you need to remove the policy that you created, issue the following command:

```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
'OE_ORDERS_ACCESS_POLICY')
```

## ***Practice Solutions 11-1: Implementing Fine-Grained Access Control for VPD***

In this practice, you define an application context and security policy to implement the policy: “Sales representatives can see only their own order information in the ORDERS table.” You create sales representative IDs to test the success of your implementation. Examine the definition of the ORDERS table and the ORDER count for each sales representative:

- 1) Use your SYS connection. Examine and then run the lab\_11\_01.sql script. This script creates the sales representative ID accounts with appropriate privileges to access the database.

```
CREATE USER sr153 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr154 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr155 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr156 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr158 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr159 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr160 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

CREATE USER sr161 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;
```

## ***Practice Solutions 11-1: Implementing Fine-Grained Access Control for VPD (continued)***

```
CREATE USER sr163 IDENTIFIED BY oracle
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  QUOTA UNLIMITED ON USERS;

GRANT create session
  , alter session
TO sr153, sr154, sr155, sr156, sr158, sr159,
  sr160, sr161, sr163;

GRANT SELECT, INSERT, UPDATE, DELETE ON
  oe.orders TO sr153, sr154, sr155, sr156, sr158,
  sr159, sr160, sr161, sr163;

GRANT SELECT, INSERT, UPDATE, DELETE ON
  oe.order_items TO sr153, sr154, sr155, sr156, sr158,
  sr159, sr160, sr161, sr163;

CREATE PUBLIC SYNONYM orders FOR oe.orders;

CREATE PUBLIC SYNONYM order_items FOR oe.order_items;
```

```
@lab_11_01.sql
```

### 2) Set up an application context:

- a) Connect to the database as SYS before creating this context.
- b) Create an application context named sales\_orders\_ctx.
- c) Associate this context with the oe.sales\_orders\_pkg.

```
CREATE CONTEXT sales_orders_ctx
  USING oe.sales_orders_pkg;
```

### 3) Connect as OE.

- a) Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
  PROCEDURE set_app_context;
  FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
```

## Practice Solutions 11-1: Implementing Fine-Grained Access Control for VPD (continued)

```
END sales_orders_pkg;      -- package spec
/
```

- b) Create this package specification, and then the package body in the OE schema.
- c) When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';
```

- d) Use these constants in the SET\_APP\_CONTEXT procedure to set the application context to the current user.

```
CREATE OR REPLACE PACKAGE BODY sales_orders_pkg
IS
  c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
  c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';

  PROCEDURE set_app_context
  IS
    v_user VARCHAR2(30);
  BEGIN
    SELECT user INTO v_user FROM dual;
    DBMS_SESSION.SET_CONTEXT
      (c_context, c_attrib, v_user);
  END set_app_context;

  FUNCTION the_predicate
  (p_schema VARCHAR2, p_name VARCHAR2)
  RETURN VARCHAR2
  IS
    v_context_value VARCHAR2(100) :=
      SYS_CONTEXT(c_context, c_attrib);
    v_restriction VARCHAR2(2000);
  BEGIN
    IF v_context_value LIKE 'SR%' THEN
      v_restriction :=
        'SALES_REP_ID =
        SUBSTR('' || v_context_value || '', 3, 3)';
    ELSE
      v_restriction := null;
    END IF;
    RETURN v_restriction;
  END the_predicate;

END sales_orders_pkg; -- package body
/
```

- 4) Connect as SYS and define the policy.
  - a) Use DBMS\_RLS.ADD\_POLICY to define the policy.
  - b) Use the following specifications for the parameter values:

## Practice Solutions 11-1: Implementing Fine-Grained Access Control for VPD (continued)

```
object_schema    OE
object_name      ORDERS
policy_name      OE_ORDERS_ACCESS_POLICY
function_schema  OE
policy_function   SALES_ORDERS_PKG.THE_PREDICATE
statement_types  SELECT, INSERT, UPDATE, DELETE
update_check     FALSE,
enable           TRUE
```

```
DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',
    'ORDERS',
    'OE_ORDERS_ACCESS_POLICY',
    'OE',
    'SALES_ORDERS_PKG.THE_PREDICATE',
    'SELECT, INSERT, UPDATE, DELETE',
    FALSE,
    TRUE);
END;
/
```

- 5) Connect as SYS and create a logon trigger to implement fine-grained access control. Name the trigger SET\_ID\_ON\_LOGON. This trigger causes the context to be set as each user is logged on.

```
CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
  oe.sales_orders_pkg.set_app_context;
END;
/
```

- 6) Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match the following:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID    COUNT(*)
-----
153              5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
```

## ***Practice Solutions 11-1: Implementing Fine-Grained Access Control for VPD (continued)***

```
FROM orders
GROUP BY sales_rep_id;

SALES_REP_ID    COUNT(*)
-----
154            10
```

### **Note:**

During debugging, you may need to disable or remove some of the objects created for this lesson.

If you need to disable the logon trigger, issue the following command:

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

If you need to remove the policy that you created, issue the following command:

```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
'OE_ORDERS_ACCESS_POLICY')
```

## Practices for Lesson 12

In this practice you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

## Practice 12-1: Safeguarding Your Code Against SQL Injection Attacks

In this practice, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

- 1) Only code that is used in Web applications is vulnerable to SQL injection attack.  
True/False
- 2) Code that is most vulnerable to SQL injection attack contains (check all that apply):
  - ☐ \_\_\_ Input parameters
  - ☐ \_\_\_ Dynamic SQL with bind arguments
  - ☐ \_\_\_ Dynamic SQL with concatenated input values
  - ☐ \_\_\_ Calls to exterior functions
- 3) By default, a stored procedure or SQL method executes with the privileges of the owner (definer's rights).  
True/False
- 4) By using `AUTHID CURRENT_USER` in your code, you are (check all that apply):
  - ☐ \_\_\_ Specifying that the code executes with invoker's rights
  - ☐ \_\_\_ Specifying that the code executes with the highest privilege level
  - ☐ \_\_\_ Eliminating any possible SQL injection vulnerability
  - ☐ \_\_\_ Not eliminating all possible SQL injection vulnerabilities
- 5) Match each attack surface reduction technique with an example of the technique.

Technique	Example
Executes code with minimal privileges	Specify appropriate parameter types
Lock the database	Revoke privileges from PUBLIC
Reduce arbitrary input	Use invoker's rights

- 6) Examine the following code. Run the `lab_12_06.sql` script to create the procedure.

```
CREATE OR REPLACE PROCEDURE get_income_level (p_email VARCHAR2
DEFAULT NULL)
IS
```



## Practice 12-1: Safeguarding Your Code Against SQL Injection Attacks (continued)

```
TYPE      cv_custtyp IS REF CURSOR;
cv        cv_custtyp;
v_income  customers.income_level%TYPE;
v_stmt    VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT income_level FROM customers WHERE
            cust_email = '' || p_email || ''';

  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
    FETCH cv INTO v_income;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Income level is: ' || v_income);
  END LOOP;
  CLOSE cv;

EXCEPTION WHEN OTHERS THEN
  dbms_output.PUT_LINE(sqlerrm);
  dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END get_income_level;
/
```

a) Execute the following statements and note the results.

```
exec get_income_level('Sean.Olin@SCAUP.COM')

exec get_income_level('x' union select username from
all_users where 'x'='x')
```

b) Has SQL injection occurred?

7) Rewrite the code to protect against SQL injection. You can run the lab\_12\_07.sql script to re-create the procedure.

a) Execute the following statements and note the results:

```
exec get_income_level('Sean.Olin@SCAUP.COM')

exec get_income_level('x' union select username from
all_users where 'x'='x')
```

b) Has SQL injection occurred?

8) Play the les12\_staticsql viewlet located in the /home/oracle/labs/viewlets folder. This is an example of using static SQL to handle a varying number of IN-list values in a query condition.

To play a viewlet, double-click the les12\_staticsql.htm file located in the /home/oracle/labs/viewlets folder. When prompted, click Start.

## ***Practice Solutions 12-1: Safeguarding Your Code Against SQL Injection Attacks***

In this practice, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

### **Understanding SQL Injection**

- 1) Only code used in Web applications is vulnerable to SQL injection attack.  
True/**False**
  
- 2) Code that is most vulnerable to SQL injection attack contains (check all that apply):
  - ☐ Input parameters
  - ☐ Dynamic SQL with bind arguments
  - ⇒ Dynamic SQL with concatenated input values
  - ☐ Calls to exterior functions
  
- 3) By default, a stored procedure or SQL method executes with the privileges of the owner (definer's rights).  
**True**/False
  
- 4) By using `AUTHID CURRENT_USER` in your code, you are (check all that apply):
  - ⇒ Specifying that the code executes with invoker's rights
  - ☐ Specifying that the code executes with the highest privilege level
  - ☐ Eliminating any possible SQL injection vulnerability
  - ⇒ Not eliminating all possible SQL injection vulnerabilities
  
- 5) Match each attack surface reduction technique to an example of the technique.  
Technique: Example  
**Executes code with minimal privileges: Use invoker's rights**  
**Lock the database: Revoke privileges from PUBLIC**  
**Reduce arbitrary input: Specify appropriate parameter types**

### **Rewriting Code to Protect Against SQL Injection**

## Practice Solutions 12-1: Safeguarding Your Code Against SQL Injection Attacks (continued)

- 6) Examine this code. Run the lab\_12\_06.sql script to create the procedure.

```
CREATE OR REPLACE PROCEDURE get_income_level
  (p_email VARCHAR2 DEFAULT NULL)
IS
  TYPE      cv_custtyp IS REF CURSOR;
  cv        cv_custtyp;
  v_income  customers.income_level%TYPE;
  v_stmt    VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT income_level FROM customers WHERE
            cust_email = '' || p_email || ''';

  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
    FETCH cv INTO v_income;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Income level is: ' || v_income);
  END LOOP;
  CLOSE cv;

EXCEPTION WHEN OTHERS THEN
  dbms_output.PUT_LINE(sqlerrm);
  dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END get_income_level;
/
```

- a) Execute the following statements and note the results.

```
exec get_income_level('Sean.Olin@SCAUP.COM')
```

anonymous block completed SQL statement: SELECT income_level FROM customers WHERE cust_email = 'Sean.Olin@SCAUP.COM' Income level is: F: 110,000 - 129,999	
--	--

```
exec get_income_level('x' union select username from
all_users where ''x''=''x')
```

anonymous block completed SQL statement: SELECT income_level FROM customers WHERE cust_email = 'x' union select username Income level is: AM145 Income level is: AM147 Income level is: AM148 Income level is: AM149 Income level is: ANONYMOUS Income level is: APEX_030200	
---	--

- b) Has SQL injection occurred?  
**Yes, by using dynamic SQL constructed via concatenation of input values, you see all users in the database.**
- 7) Rewrite the code to protect against SQL injection. You can run the lab\_12\_07.sql script to recreate the procedure.

## Practice Solutions 12-1: Safeguarding Your Code Against SQL Injection Attacks (continued)

```
CREATE OR REPLACE
PROCEDURE get_income_level (p_email VARCHAR2 DEFAULT NULL)
AS
BEGIN
FOR i IN
  (SELECT income_level
   FROM customers
   WHERE cust_email = p_email)
LOOP
  DBMS_OUTPUT.PUT_LINE('Income level is:
    '||i.income_level);
END LOOP;
END get_income_level;
/
```

- a) Execute the following statements and note the results.

```
exec get_income_level('Sean.Olin@SCAUP.COM')
```

```
anonymous block completed
Income level is: F: 110,000 - 129,999
```

```
exec get_income_level('x' union select username from
all_users where 'x'='x')
```

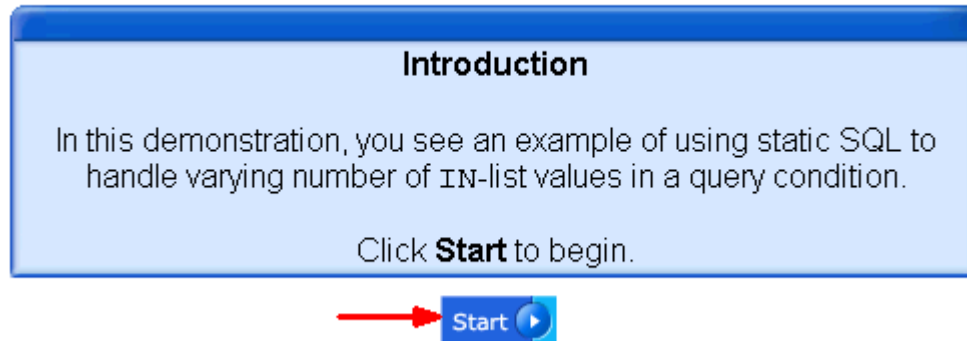
```
anonymous block completed
```

- b) Has SQL injection occurred?  
**No**

## ***Practice Solutions 12-1: Safeguarding Your Code Against SQL Injection Attacks (continued)***

- 8) Play the `les12_staticsql` viewlet located in the `/home/oracle/labs/viewlets` folder. This is an example of using static SQL to handle a varying number of `IN`-list values in a query condition.

To play a viewlet, double-click the `les13_staticsql.htm` file located in the `/home/oracle/labs/viewlets` folder. When prompted, click **Start**.





# **Table Descriptions and Data**



**ORACLE**

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Schema Descriptions

### Overall Description

The sample company portrayed by the Oracle Database Sample Schemas operates worldwide to fulfill orders for several different products. The company has several divisions:

- The Human Resources division tracks information about the employees and facilities of the company.
- The Order Entry division tracks product inventories and sales of the products of the company through various channels.
- The Sales History division tracks business statistics to facilitate business decisions.

Each division is represented by a schema. In this course, you have access to the objects in all these schemas. However, the emphasis of the examples, demonstrations, and practices utilizes the Order Entry (OE) schema.

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



## Schema Descriptions (continued)

### Order Entry (OE)

The company sells several categories of products, including computer hardware and software, music, clothing, and tools. The company maintains product information that includes product identification numbers, the category into which the product falls, the weight group (for shipping purposes), the warranty period if applicable, the supplier, the status of the product, a list price, a minimum price at which a product will be sold, and a URL for manufacturer information.

Inventory information is also recorded for all products, including the warehouse where the product is available and the quantity on hand. Because products are sold worldwide, the company maintains the names of the products and their descriptions in different languages.

The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

Customer information is tracked in some detail. Each customer is assigned an identification number. Customer records include name, street address, city or province, country, phone numbers (up to five phone numbers for each customer), and postal code. Some customers order through the Internet, so email addresses are also recorded. Because of language differences among customers, the company records the National Language Support (NLS) language and territory of each customer. The company places a credit limit on its customers to limit the amount for which they can purchase at one time. Some customers have account managers, whom the company monitors. It keeps track of a customer's phone number. These days, you never know how many phone numbers a customer might have, but you try to keep track of all of them. Because of the language differences of the customers, you identify the language and territory of each customer.

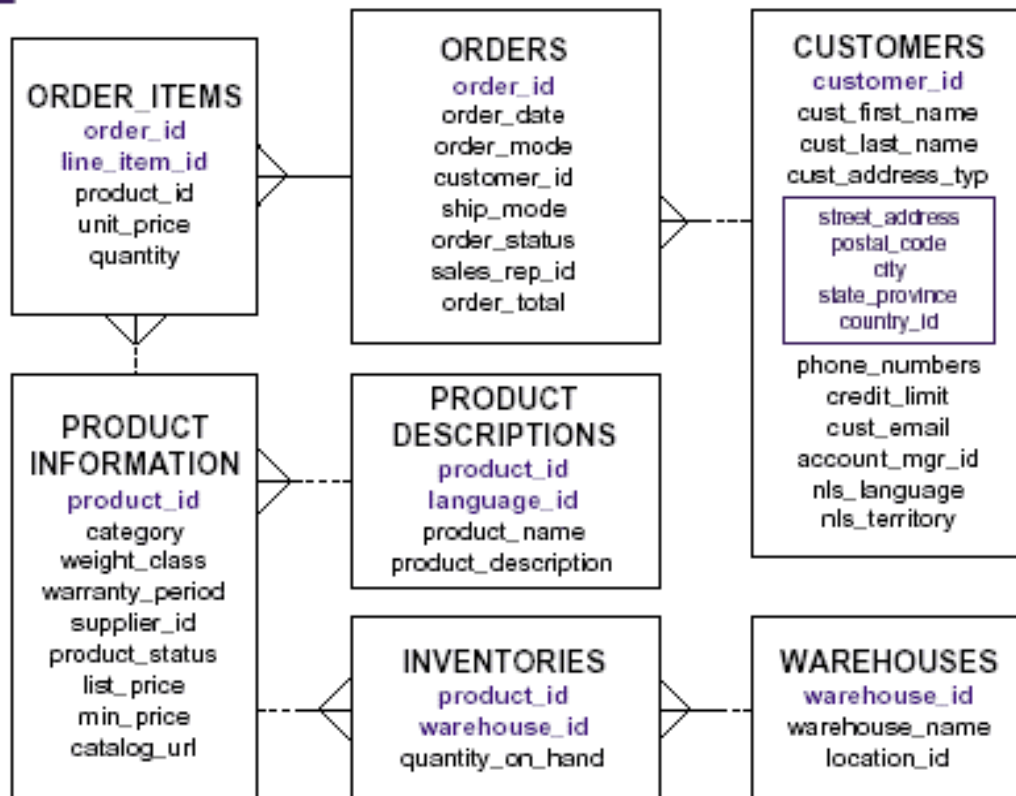
When a customer places an order, the company tracks the date of the order, the mode of the order, status, shipping mode, total amount of the order, and the sales representative who helped place the order. This may be the same individual as the account manager for a customer, it may be a different individual, or, in the case of an order over the Internet, the sales representative is not recorded. In addition to the order information, the company also tracks the number of items ordered, the unit price, and the products ordered.

For each country in which it does business, the company records the country name, currency symbol, currency name, and the region where the country resides geographically. This data is useful to interact with customers living in different geographic regions around the world.

## Schema Descriptions (continued)

### Order Entry (OE)

OE



## Schema Descriptions (continued)

### Order Entry (OE) Row Counts

```
SELECT COUNT(*) FROM customers;
COUNT(*)
-----
      319
```

```
SELECT COUNT(*) FROM inventories;
COUNT(*)
-----
     1112
```

```
SELECT COUNT(*) FROM orders;
COUNT(*)
-----
      105
```

```
SELECT COUNT(*) FROM order_items;
COUNT(*)
-----
     665
```

```
SELECT COUNT(*) FROM product_descriptions;
COUNT(*)
-----
     8640
```

```
SELECT COUNT(*) FROM product_information;
COUNT(*)
-----
      288
```

```
SELECT COUNT(*) FROM warehouses;
COUNT(*)
-----
        9
```

## Schema Descriptions (continued)

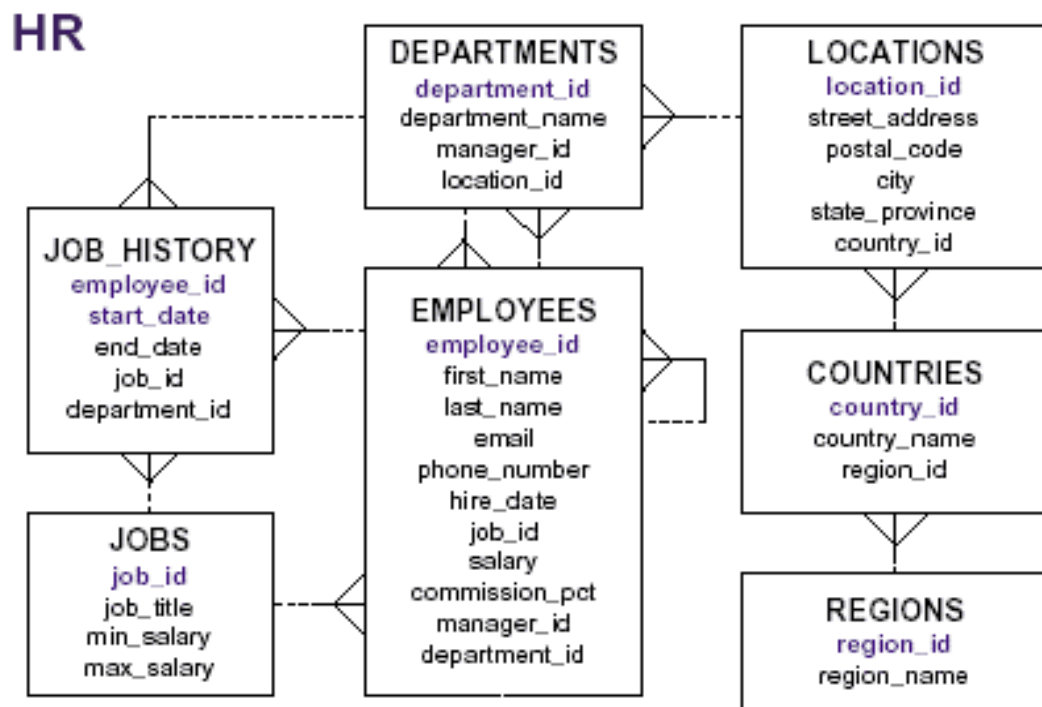
### Human Resources (HR)

In the human resource records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn a commission in addition to their salary.

The company also tracks information about the 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 switches jobs, the company records the start date and end date of the former job, the job identification number, and the department.

The sample company is regionally diverse, so it tracks the locations of not only its warehouses but also its departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. Each location has a full address that includes the street address, postal code, city, state or province, and country code.

For each location where it has facilities, the company records the country name, currency symbol, currency name, and the region where the country resides geographically.



## Schema Descriptions (continued)

### Human Resources (HR) Row Counts

```
SELECT COUNT(*) FROM employees;
COUNT(*)
-----
      107

SELECT COUNT(*) FROM departments;
COUNT(*)
-----
       27

SELECT COUNT(*) FROM locations;
COUNT(*)
-----
       23

SELECT COUNT(*) FROM countries;
COUNT(*)
-----
       25

SELECT COUNT(*) FROM regions;
COUNT(*)
-----
        4

SELECT COUNT(*) FROM jobs;
COUNT(*)
-----
       19

SELECT COUNT(*) FROM job_history;
COUNT(*)
-----
       10
```



# Using SQL Developer

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- List the key features of Oracle SQL Developer
- Identify 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

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

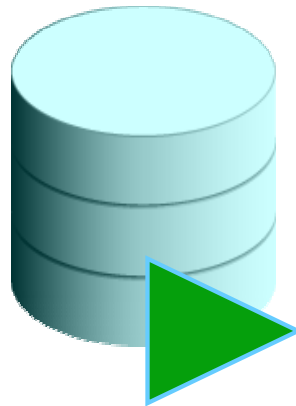
## Objectives

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 © 2010, Oracle and/or its affiliates. All rights reserved.

## What Is Oracle SQL Developer?

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, 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.

The SQL Developer 1.2 release tightly integrates with Developer Migration Workbench, which provides users with a single point to browse database objects and data in third-party databases, and to migrate from these databases to Oracle. You can also connect to schemas for selected third-party (non-Oracle) databases such as MySQL, Microsoft SQL Server, and Microsoft Access, and you can view metadata and data in these databases.

Additionally, SQL Developer includes support for Oracle Application Express 3.0.1 (Oracle APEX).

## Specifications of SQL Developer

- Shipped along with Oracle Database 11g Release 2
- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the Java Database Connectivity (JDBC) thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
  - [http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html)

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Specifications of SQL Developer

Oracle SQL Developer 1.5 is shipped along with Oracle Database 11g Release 2. 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.

Default connectivity to the database is through the 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

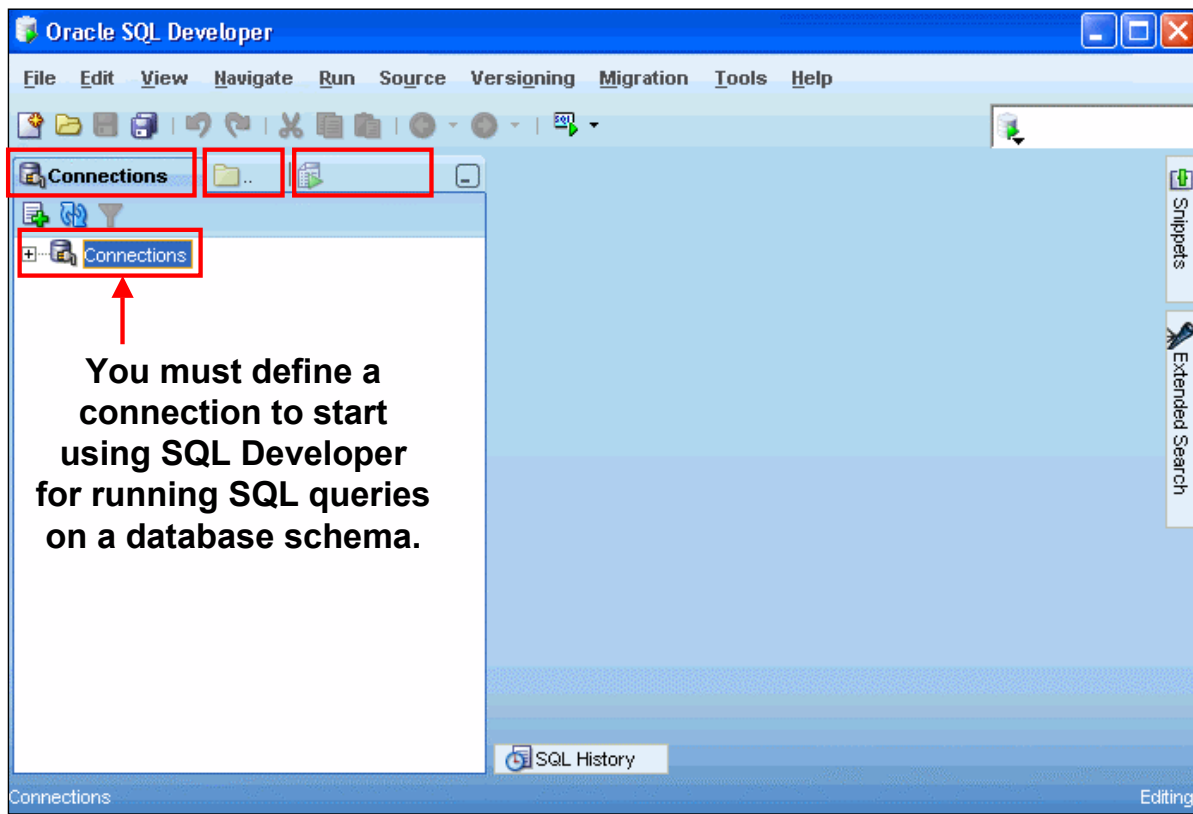
For Oracle Database versions earlier than Oracle Database 11g Release 2, you will have to download and install SQL Developer. SQL Developer 1.5 is freely downloadable from the following link:

[http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html).

For instructions on how to install SQL Developer, you can visit the following link:

[http://download.oracle.com/docs/cd/E12151\\_01/index.htm](http://download.oracle.com/docs/cd/E12151_01/index.htm)

# SQL Developer 1.5 Interface



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## SQL Developer 1.5 Interface

The SQL Developer 1.5 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.
- **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.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.

### 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.

## SQL Developer 1.5 Interface (continued)

### Menus

The following menus contain standard entries, plus entries for features 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 in the execution of subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **Source:** Contains options for use when you edit functions and procedures
- **Versioning:** Provides integrated support for the following versioning and source control systems: Concurrent Versions System (CVS) and Subversion
- **Migration:** Contains options related to migrating third-party databases to Oracle
- **Tools:** Invokes SQL Developer tools such as SQL\*Plus, Preferences, and SQL Worksheet

**Note:** The Run menu also contains options that are relevant when a function or procedure is selected for debugging. These are the same options that are found in the Debug menu in version 1.2.

## 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
  - 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.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Creating a Database Connection

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.

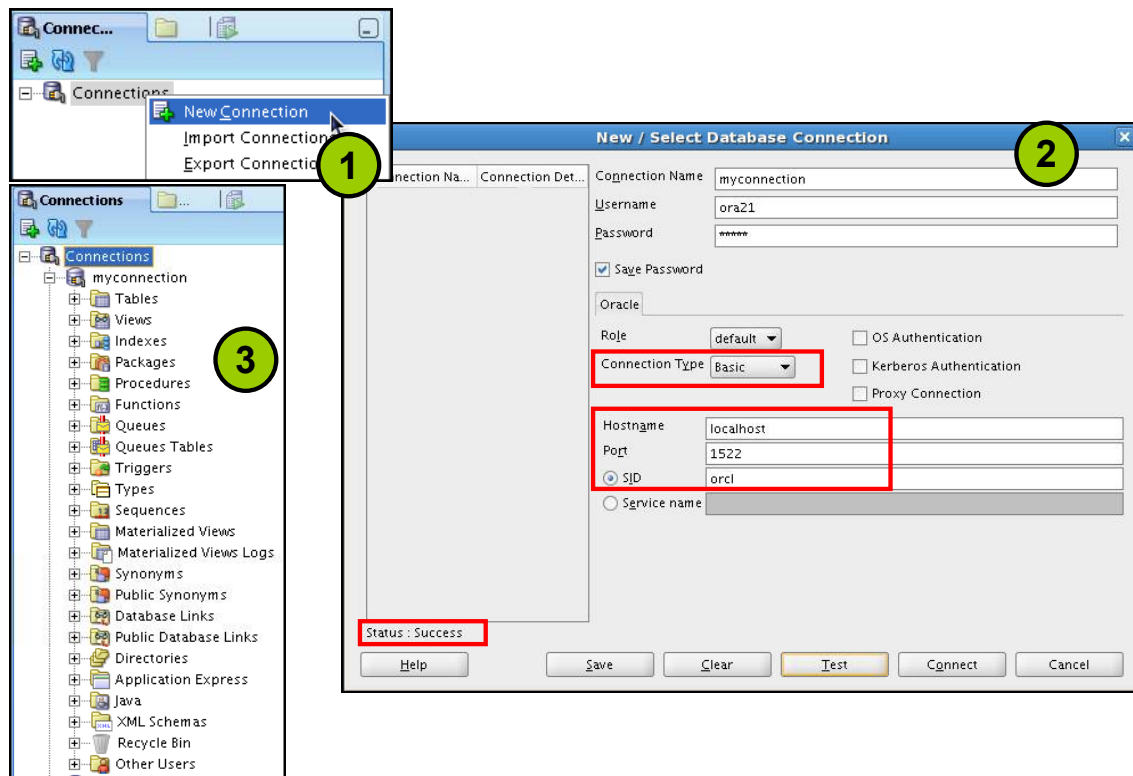
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically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

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

# Creating a Database Connection



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Creating a Database Connection (continued)

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 box, 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 hostname and SID for the database that you want to connect to. Port is already set to 1521. Or 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 JDBC URL to connect to the database.
  - c. Click Test to ensure that the connection has been set correctly.
  - d. Click Connect.

## Creating a Database Connection (continued)

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.

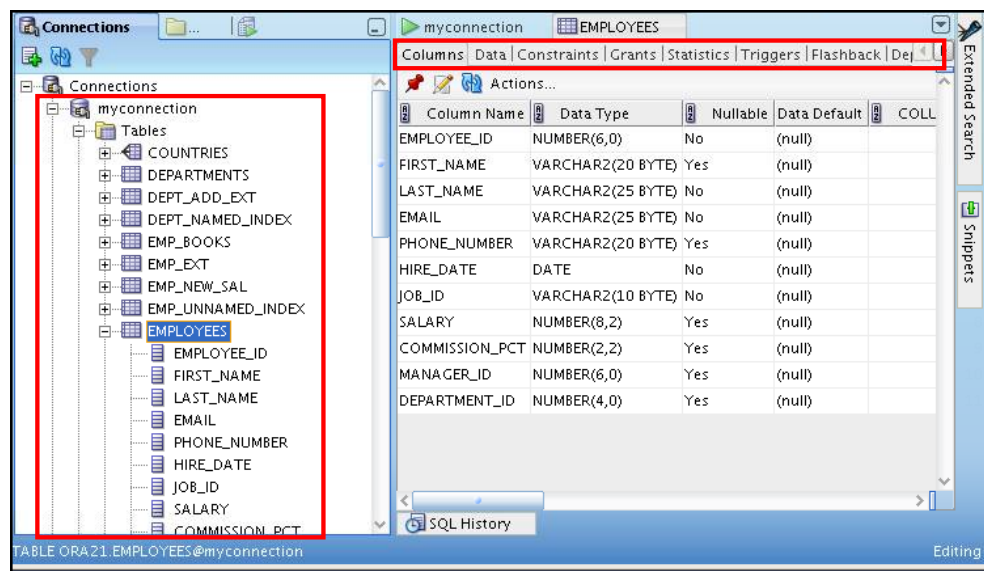
3. The connection is added to the Connections Navigator. You can expand the connection to view the database objects and view object definitions (for example, dependencies, details, and statistics).

**Note:** From the same New/Select Database Connection window, you can define connections to non-Oracle data sources by 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 © 2010, Oracle and/or its affiliates. All rights reserved.

## Browsing Database Objects

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.

Object definitions are displayed on various tabbed pages, based on information that is pulled from the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

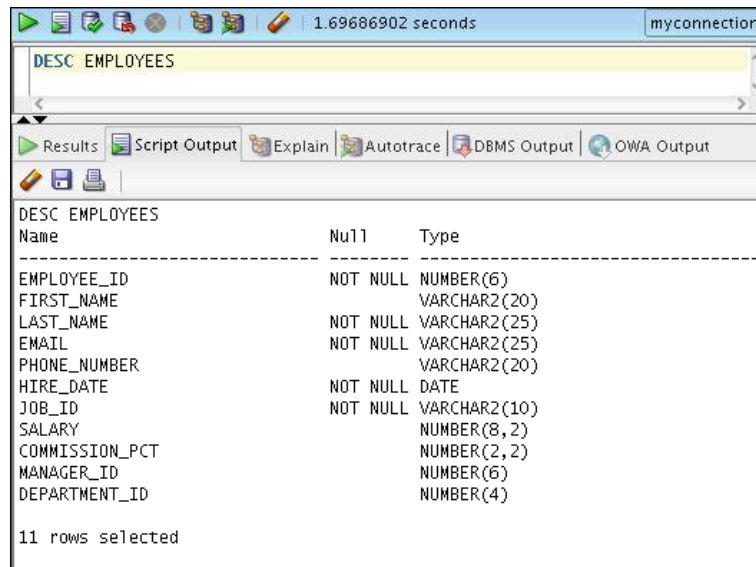
If you want to see the definition of the EMPLOYEES table as shown in the slide, perform the following steps:

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 SQL Developer interface with the command 'DESC EMPLOYEES' entered in the SQL window. The Results tab is active, displaying the table structure. The table has 11 rows selected. The columns are: EMPLOYEE\_ID (NUMBER(6), NOT NULL), FIRST\_NAME (VARCHAR2(20)), LAST\_NAME (VARCHAR2(25), NOT NULL), EMAIL (VARCHAR2(25), NOT NULL), PHONE\_NUMBER (VARCHAR2(20)), HIRE\_DATE (DATE, NOT NULL), JOB\_ID (VARCHAR2(10), NOT NULL), SALARY (NUMBER(8,2)), COMMISSION\_PCT (NUMBER(2,2)), MANAGER\_ID (NUMBER(6)), and DEPARTMENT\_ID (NUMBER(4)).

DESC EMPLOYEES	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)

11 rows selected

ORACLE

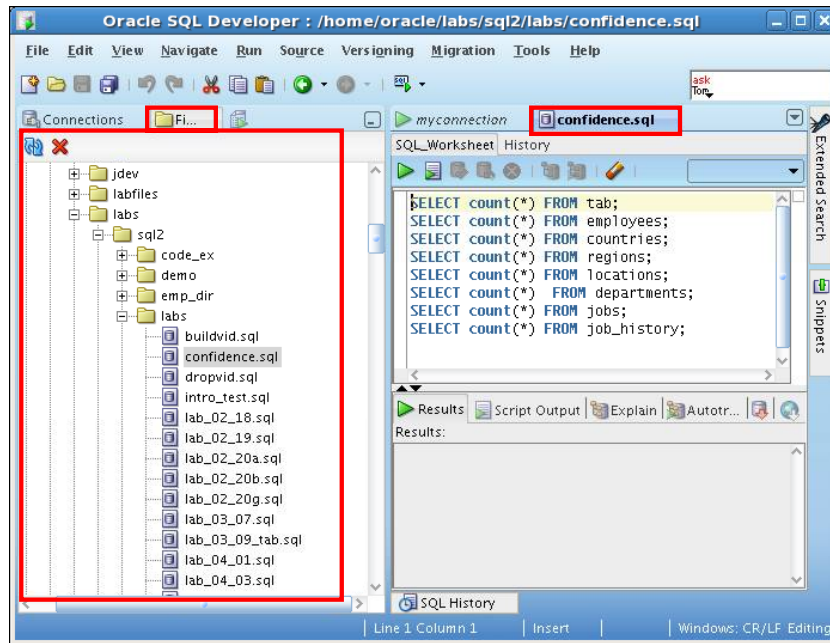
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Displaying the Table Structure

In SQL Developer, you can also display the structure of a table by 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.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

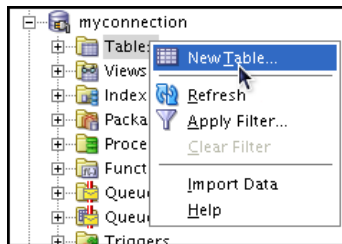
## Browsing Files

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

- To view the File Navigator, click the Files tab, or click 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 © 2010, Oracle and/or its affiliates. All rights reserved.

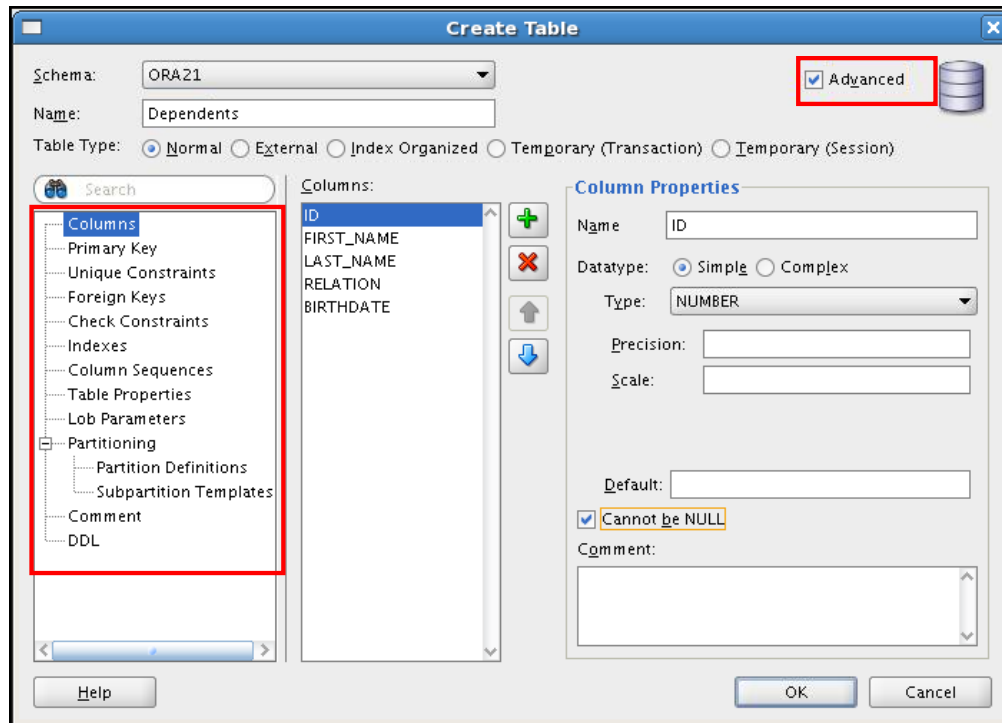
### Creating a Schema Object

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. You can edit the objects by 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 by 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 © 2010, Oracle and/or its affiliates. All rights reserved.

### Creating a New Table: Example

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.

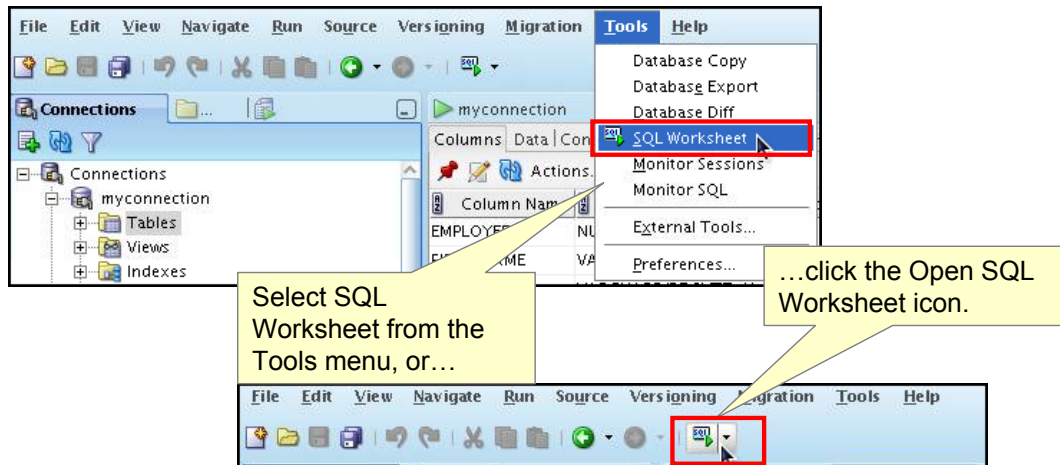
To create a new table, perform the following steps:

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

Although it is not required, you should also specify a primary key. Click Primary Key in the far left of 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.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the SQL Worksheet

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.

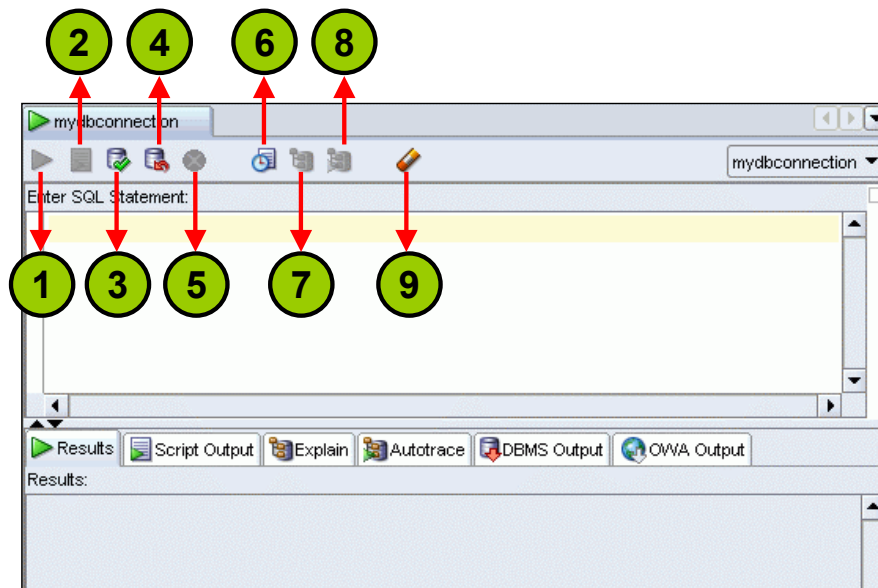
You can specify actions that can be processed by the database connection associated with the worksheet, such as:

- 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



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

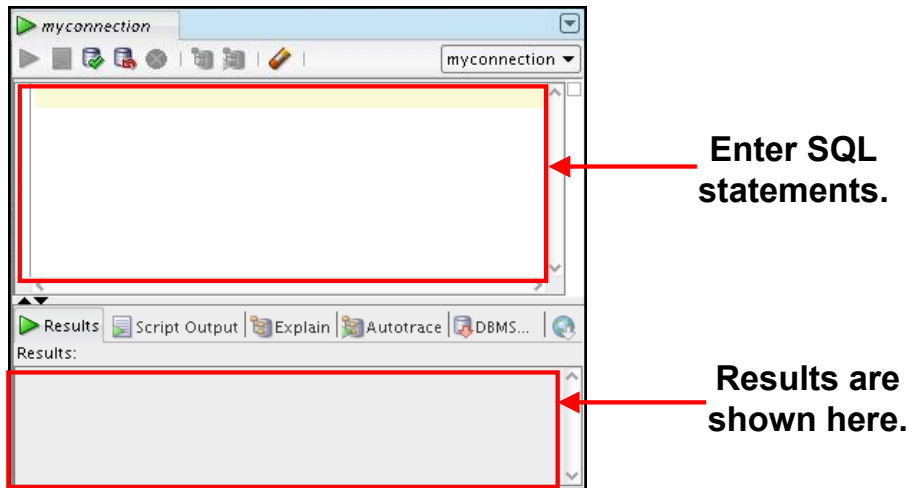
## Using the SQL Worksheet (continued)

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 SQL statements that you have executed. Use the buttons in the SQL Worksheet toolbar to perform the following tasks:

1. **Execute 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 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. **Commit:** Writes any changes to the database and ends the transaction
4. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction
5. **Cancel:** Stops the execution of any statements currently being executed
6. **SQL History:** Displays a dialog box with information about SQL statements that you have executed
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
8. **Autotrace:** Generates trace information for the statement
9. **Clear:** Erases the statement or statements in the Enter SQL Statement box

## 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.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the SQL Worksheet (continued)

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. SQL\*Plus commands used in the SQL Developer have to be interpreted by the SQL Worksheet before being passed to the database.

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

You can display a SQL Worksheet by using any of the following two 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.

The screenshot illustrates the Oracle SQL Developer interface with two windows. The top window, titled 'myconnection', has a text area for 'Enter SQL Statement' containing the query: `SELECT employee_id, last_name FROM employees;`. In this window, the 'Run Script' button (a green play icon) is highlighted with a red box and labeled 'F5', and the 'Execute Statement' button (a green play icon) is highlighted with a red box and labeled 'F9'. The bottom window shows two different result sets for the same query. The left window, titled 'Results', has its 'Results' button highlighted with a red box and labeled 'F9'. It displays a table with 5 rows and 2 columns: 

	EMPLOYEE_ID	LAST_NAME
1	100	King
2	101	Kochhar
3	102	De Haan
4	103	Hunold
5	104	Ernst

. The right window, titled 'Script Output', has its 'Script Output' button highlighted with a red box and labeled 'F5'. It displays a table with 6 rows and 2 columns: 

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
105	Austin

ORACLE

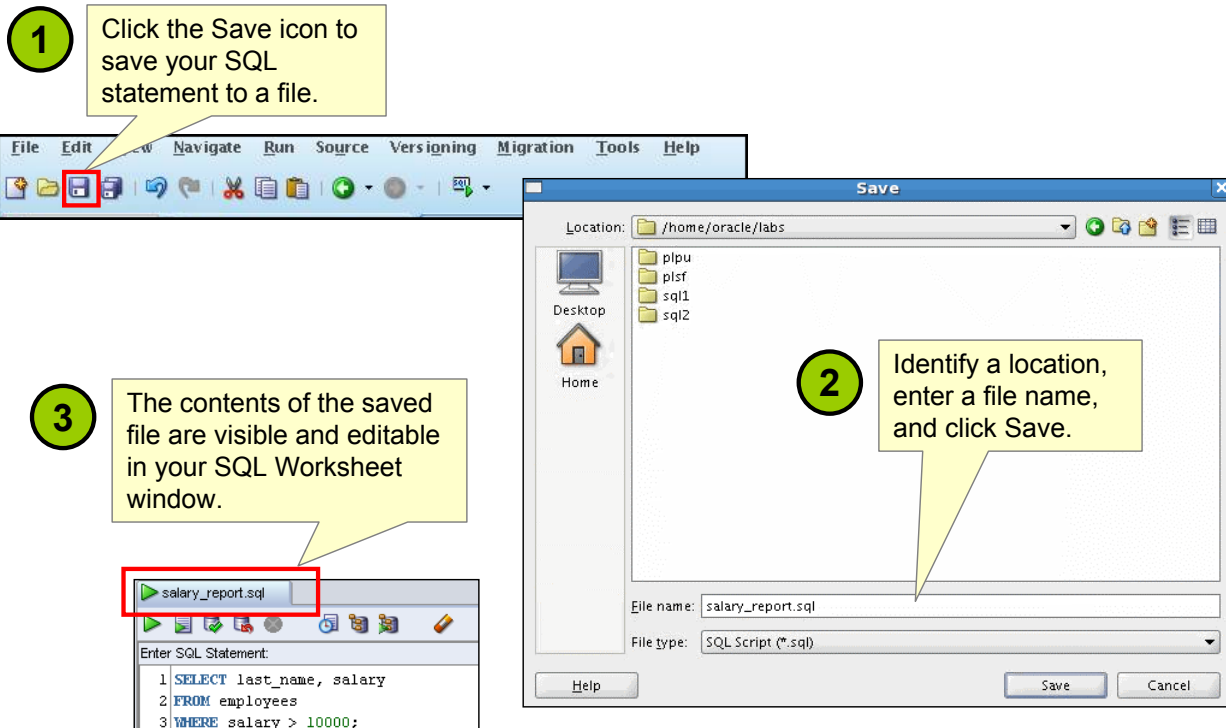
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Executing SQL Statements

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



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Saving SQL Scripts

You can save your SQL statements from the SQL Worksheet into a text file. To save the contents of the Enter SQL Statement box, follow these steps:

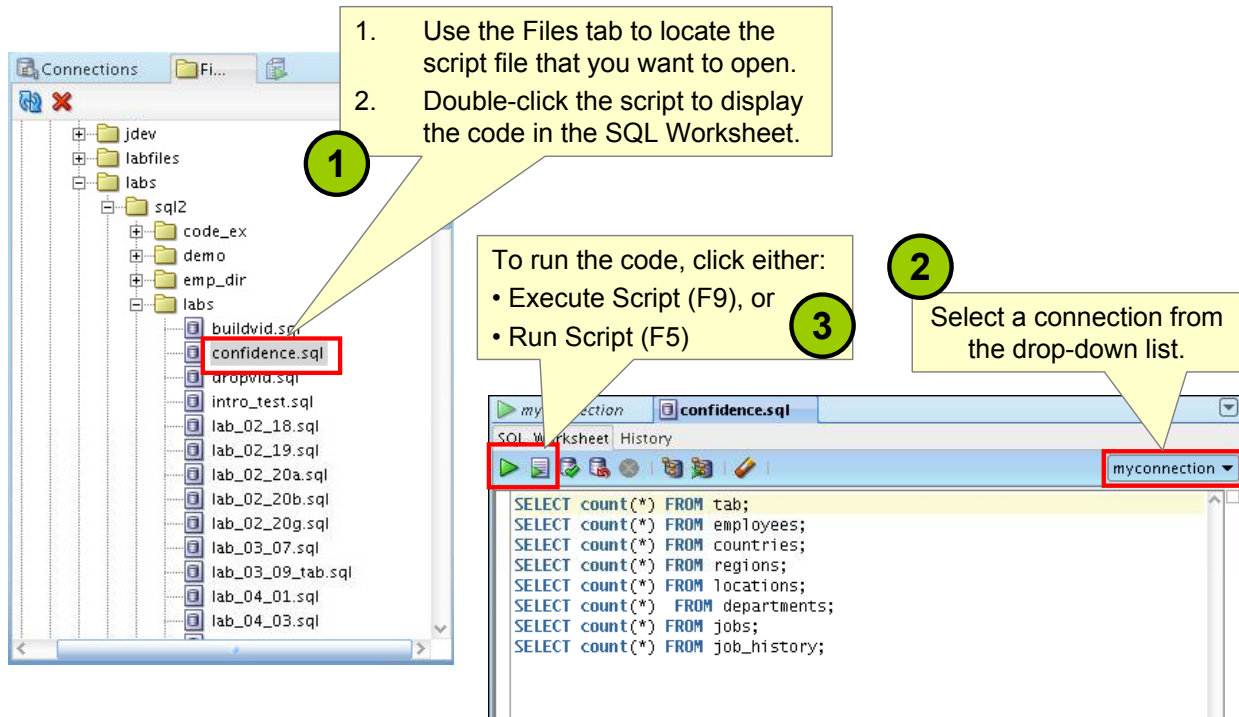
1. Click the Save icon or use the File > Save menu item.
2. In the Windows Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

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 © 2010, Oracle and/or its affiliates. All rights reserved.

## Executing Saved Script Files: Method 1

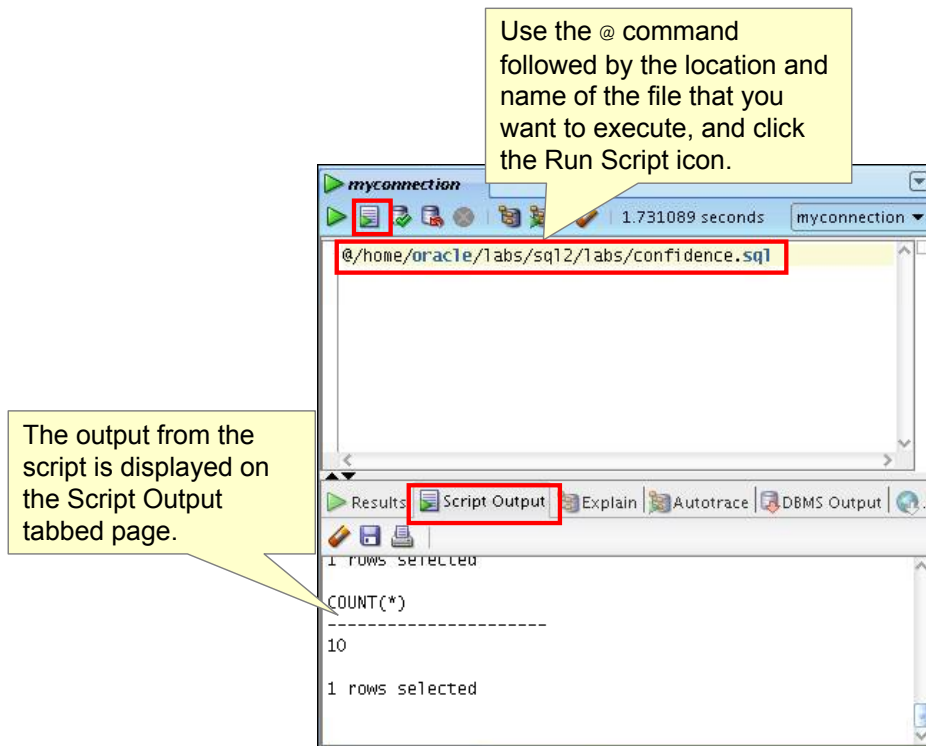
To open a script file and display the code in the SQL Worksheet area, perform the following:

1. In the files navigator select (or navigate to) the script file that you want to open.
2. Double-click to open the file. 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 you want to use for the script execution.

Alternatively, you can 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 © 2010, Oracle and/or its affiliates. All rights reserved.

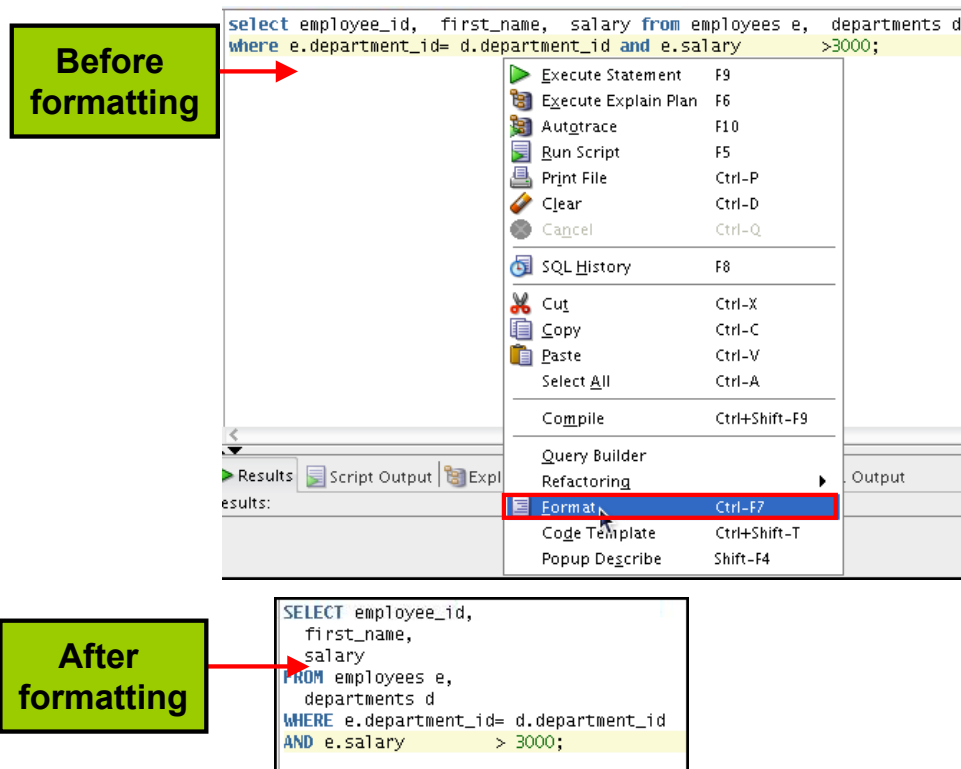
## Executing Saved Script Files: Method 2

To run a saved SQL script, perform the following:

1. Use the @ command, followed by the location, and 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 Windows Save dialog box appears and you can identify a name and location for your file.

## Formatting the SQL Code



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Formatting the SQL Code

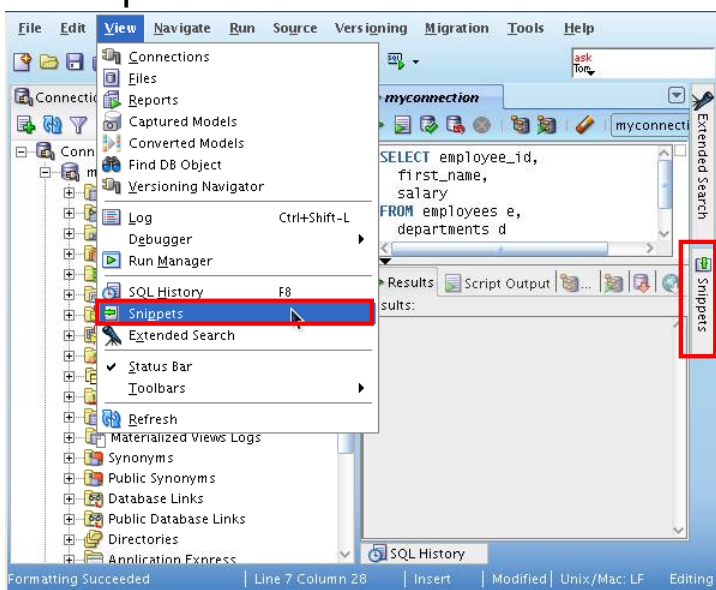
You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

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

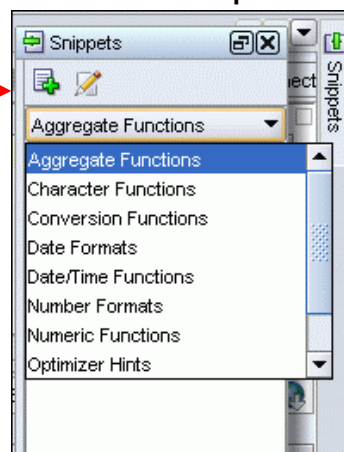
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

# Using Snippets

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



Place your cursor over the Snippets button to open the Snippets window, and then select the functions category that you want from the drop-down list.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using Snippets

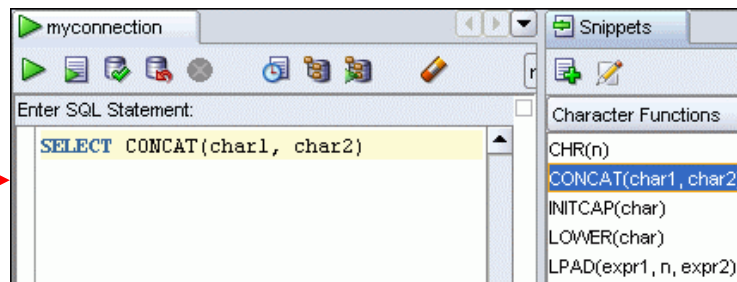
You may want to use certain code fragments when you use the SQL Worksheet or create or edit a PL/SQL function or procedure. SQL Developer has the feature called Snippets. Snippets are code fragments such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag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

To display Snippets, select View > Snippets.

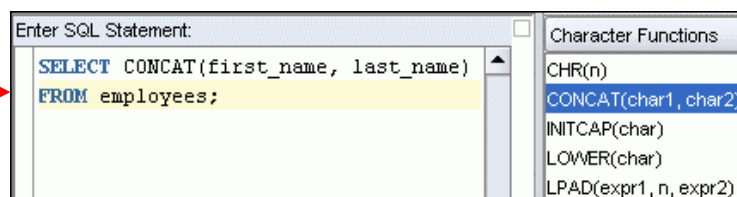
The Snippets window is displayed at the right side. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

## Using Snippets: Example

Inserting a snippet



Editing the snippet



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using Snippets: Example

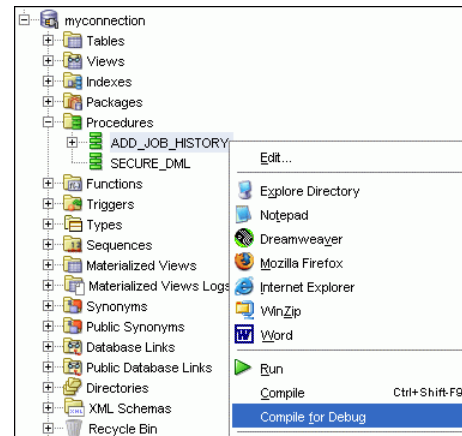
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function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

# 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 Debug menu options to set breakpoints, and to perform step into and step over tasks.



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Debugging Procedures and Functions

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

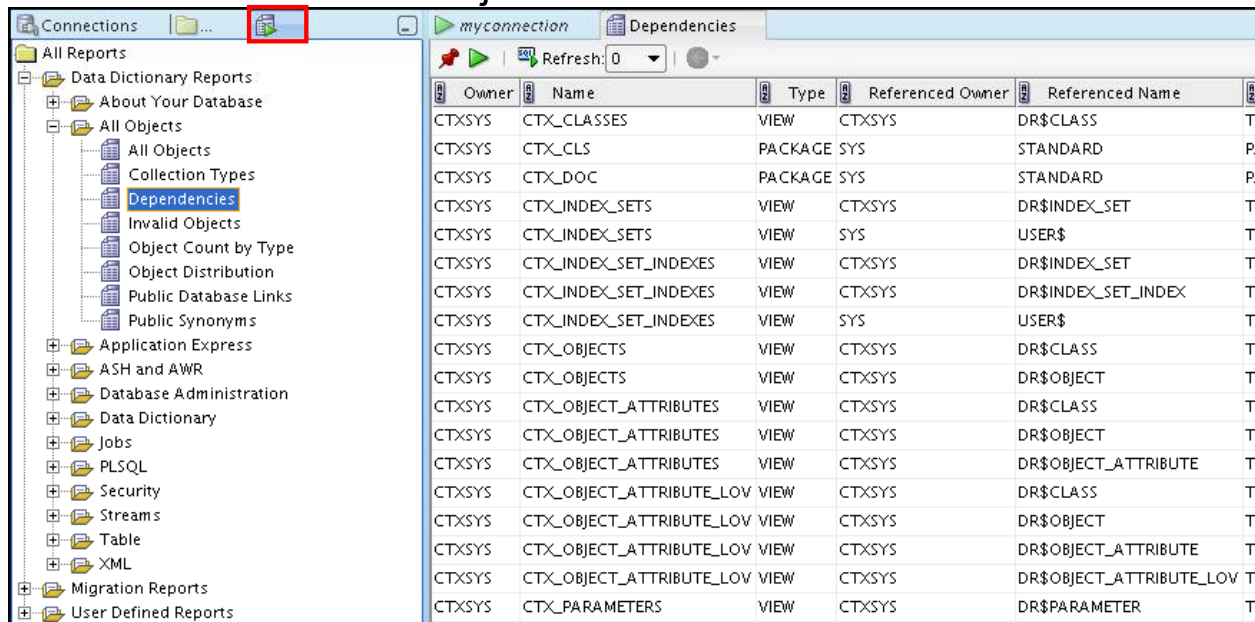
- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to 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 toolbar.



# Database Reporting

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



Owner	Name	Type	Referenced Owner	Referenced Name
CTXSYS	CTX_CLASSES	VIEW	CTXSYS	DR\$CLASS
CTXSYS	CTX_CLS	PACKAGE	SYS	STANDARD
CTXSYS	CTX_DOC	PACKAGE	SYS	STANDARD
CTXSYS	CTX_INDEX_SETS	VIEW	CTXSYS	DR\$INDEX_SET
CTXSYS	CTX_INDEX_SETS	VIEW	SYS	USER\$
CTXSYS	CTX_INDEX_SET_INDEXES	VIEW	CTXSYS	DR\$INDEX_SET
CTXSYS	CTX_INDEX_SET_INDEXES	VIEW	CTXSYS	DR\$INDEX_SET_INDEX
CTXSYS	CTX_INDEX_SET_INDEXES	VIEW	SYS	USER\$
CTXSYS	CTX_OBJECTS	VIEW	CTXSYS	DR\$CLASS
CTXSYS	CTX_OBJECTS	VIEW	CTXSYS	DR\$OBJECT
CTXSYS	CTX_OBJECT_ATTRIBUTES	VIEW	CTXSYS	DR\$CLASS
CTXSYS	CTX_OBJECT_ATTRIBUTES	VIEW	CTXSYS	DR\$OBJECT
CTXSYS	CTX_OBJECT_ATTRIBUTES	VIEW	CTXSYS	DR\$OBJECT_ATTRIBUTE
CTXSYS	CTX_OBJECT_ATTRIBUTE_LOV	VIEW	CTXSYS	DR\$CLASS
CTXSYS	CTX_OBJECT_ATTRIBUTE_LOV	VIEW	CTXSYS	DR\$OBJECT
CTXSYS	CTX_OBJECT_ATTRIBUTE_LOV	VIEW	CTXSYS	DR\$OBJECT_ATTRIBUTE
CTXSYS	CTX_OBJECT_ATTRIBUTE_LOV	VIEW	CTXSYS	DR\$OBJECT_ATTRIBUTE_LOV
CTXSYS	CTX_PARAMETERS	VIEW	CTXSYS	DR\$PARAMETER

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Database Reporting

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

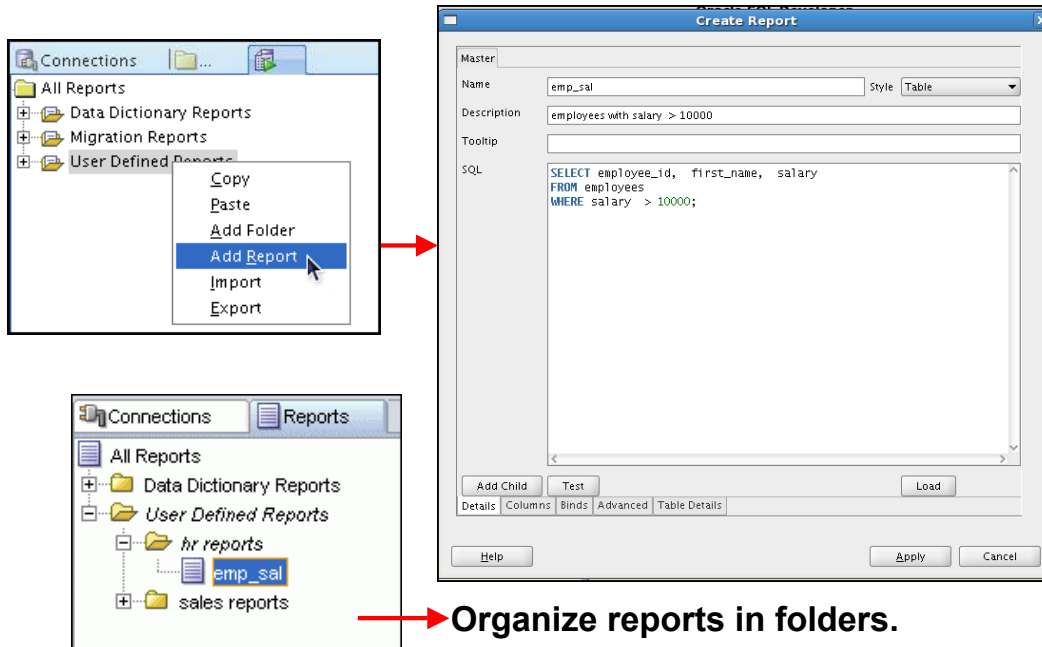
- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-Defined reports

To display reports, click the Reports tab at the left side of the window. Individual reports are displayed in tabbed panes at the right side 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.



Organize reports in folders.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Creating a User-Defined Report

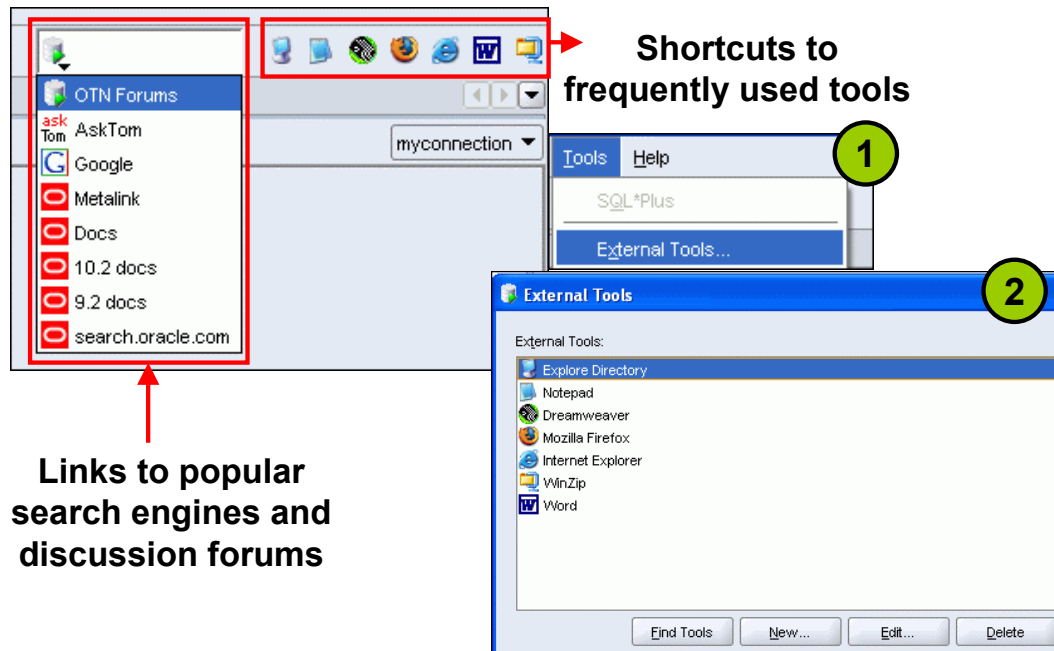
User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

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` under the directory for user-specific information.

# Search Engines and External Tools



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Search Engines and External Tools

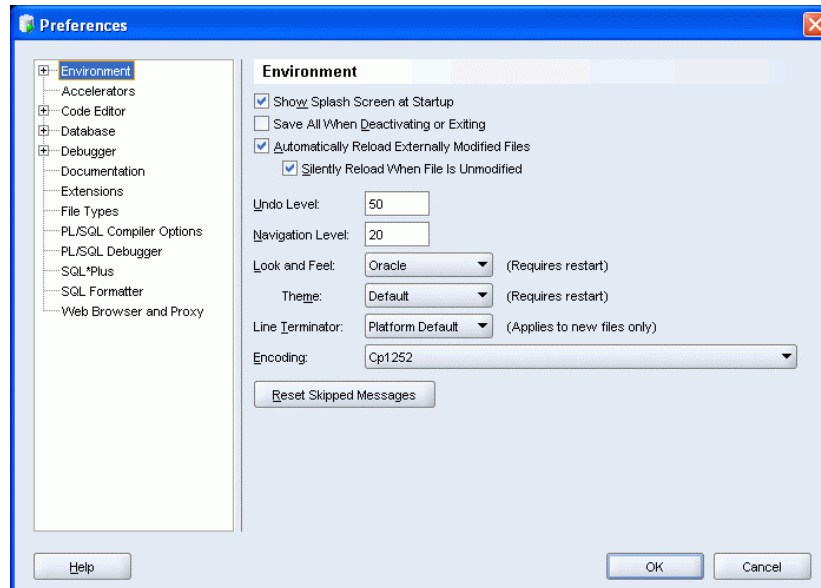
To enhance productivity of the SQL developers, SQL Developer has added quick links to popular search engines and discussion forums, such as AskTom and Google. Also, you have shortcut icons for some of the frequently used tools, such as Notepad, Microsoft Word, and Dreamweaver, that are available to you.

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

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 © 2010, Oracle and/or its affiliates. All rights reserved.

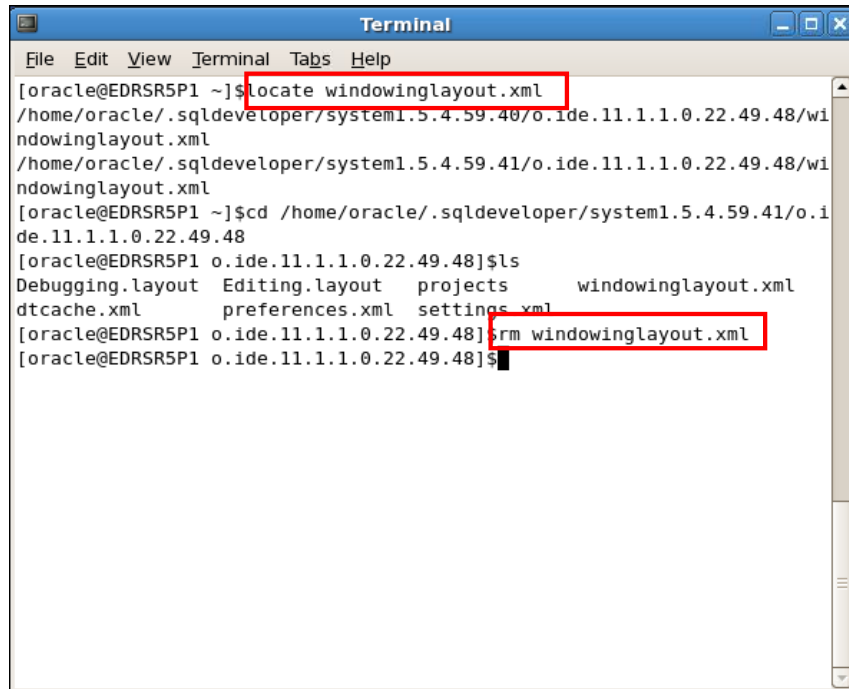
## Setting Preferences

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

The preferences are grouped into the following categories:

- Environment
- Accelerators (keyboard shortcuts)
- Code Editors
- Database
- Debugger
- Documentation
- Extensions
- File Types
- Migration
- PL/SQL Compilers
- PL/SQL Debugger, and so on

# Resetting the SQL Developer Layout



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR5P1 ~]$ locate windowinglayout.xml
/home/oracle/.sqldeveloper/system1.5.4.59.40/o.ide.11.1.1.0.22.49.48/wi
ndowinglayout.xml
/home/oracle/.sqldeveloper/system1.5.4.59.41/o.ide.11.1.1.0.22.49.48/wi
ndowinglayout.xml
[oracle@EDRSR5P1 ~]$ cd /home/oracle/.sqldeveloper/system1.5.4.59.41/o.i
de.11.1.1.0.22.49.48
[oracle@EDRSR5P1 o.ide.11.1.1.0.22.49.48]$ ls
Debugging.layout  Editing.layout  projects        windowinglayout.xml
dtcache.xml      preferences.xml  settings.xml
[oracle@EDRSR5P1 o.ide.11.1.1.0.22.49.48]$ rm windowinglayout.xml
[oracle@EDRSR5P1 o.ide.11.1.1.0.22.49.48]$
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Resetting the SQL Developer Layout

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.

## Summary

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

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Summary

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.





ORACLE

Copyright © 2010, 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 by using SQL\*Plus commands
- Interact with script files

ORACLE

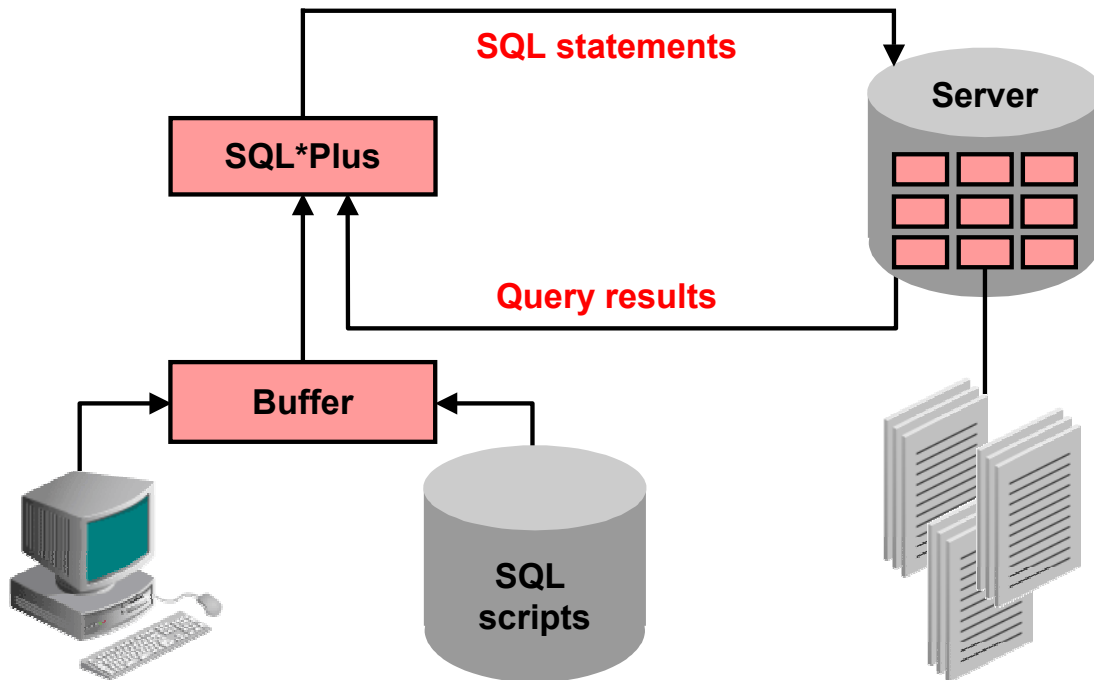
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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



# SQL and SQL\*Plus Interaction



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## SQL and SQL\*Plus

SQL is a command language 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

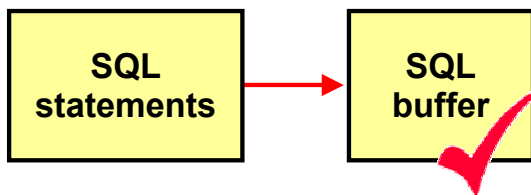
### Features of SQL\*Plus

- 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

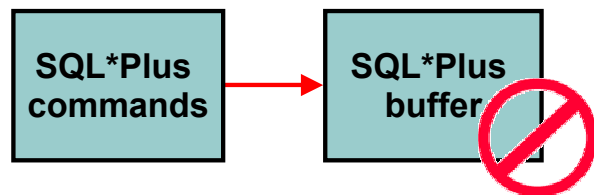
### SQL

- 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 © 2010, Oracle and/or its affiliates. All rights reserved.

### SQL and SQL\*Plus (continued)

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 to be edited from the file to the buffer.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## SQL\*Plus

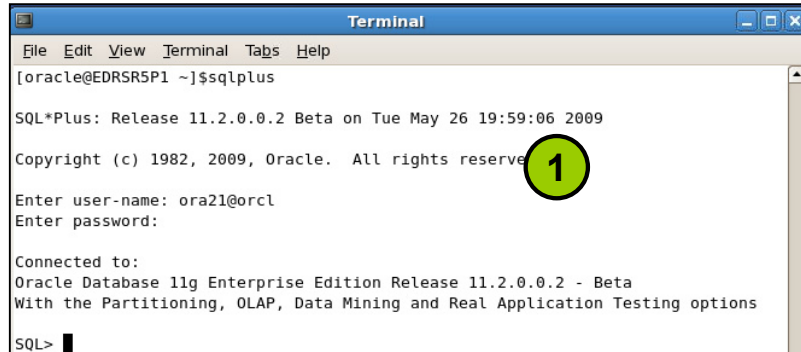
SQL\*Plus is an environment in which you can:

- 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



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR5P1 ~]$sqlplus

SQL*Plus: Release 11.2.0.0.2 Beta on Tue May 26 19:59:06 2009

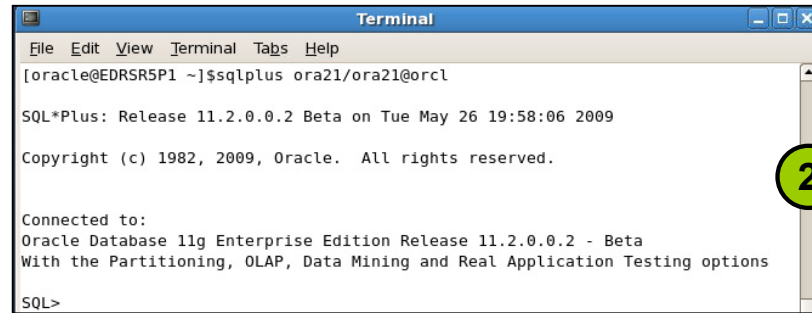
Copyright (c) 1982, 2009, Oracle. All rights reserved.

Enter user-name: ora21@orcl
Enter password:

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.0.2 - Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

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



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR5P1 ~]$sqlplus ora21/oracle@orcl

SQL*Plus: Release 11.2.0.0.2 Beta on Tue May 26 19:58:06 2009

Copyright (c) 1982, 2009, Oracle. All rights reserved.

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.0.2 - Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Logging In to SQL\*Plus

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

To log in from a Linux environment:

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:

*username* Your database username  
*password* Your database password (Your password is visible if you enter it here.)  
*@database* 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
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Displaying the Table Structure

In SQL\*Plus, you can display the structure of a table by 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.

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 this 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)

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Displaying the Table Structure (continued)

The example code in the slide displays the 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`

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### SQL\*Plus Editing Commands

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

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

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### SQL\*Plus Editing Commands (continued)

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
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using LIST, n, and APPEND

- 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
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the CHANGE Command

- 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`

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### SQL\*Plus File Commands

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> <code>[REP[LACE]APP[END]]</code>	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 <code>afiedt.buf</code>
<code>ED[IT] [filename [.ext]]</code>	Invokes the editor to edit the contents of a saved file
<code>SPO[OL] [filename [.ext]]</code> <code>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 and 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.
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the SAVE and START Commands

#### SAVE

Use the SAVE command to store the current contents of the buffer in a file. In this way, 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
```

## SERVEROUTPUT 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_WRAPPED] | TRU[NCATED]}]
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### SERVEROUTPUT Command

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 you require 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 the *Oracle Database PL/SQL User's Guide and Reference 11g*.

## 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

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Using the SQL\*Plus SPOOL Command

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 only use SPOOL to create (and replace) a file. REPLACE is the default.

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

You must use quotation marks around file names containing 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 data manipulation statements (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]]  
[STAT[ISTICS]]
```

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

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Using the AUTOTRACE Command

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, see the *Oracle Database PL/SQL Packages and Types Reference 11g* guide.
- For additional information about the EXPLAIN PLAN, see *Oracle Database SQL Reference 11g*.
- For additional information about execution plans and their statistics, see the *Oracle Database Performance Tuning Guide 11g*.

## 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

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Summary

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.



# PL/SQL Server Pages

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Embed PL/SQL code in Web pages (PL/SQL server pages)
- Explain the format of a PL/SQL server page
- Write the code and content for the PL/SQL server page
- Load the PL/SQL server page into the database as a stored procedure
- Run a PL/SQL server page via a URL
- Debug PL/SQL server page problems

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Objectives

In this lesson, you learn about the powerful features of PL/SQL Server Pages (PSP). Using PSP, you can embed PL/SQL in an HTML Web page.

## PSP Uses and Features

- Uses:
  - If you have a large body of HTML, and want to include dynamic content or make it the front end of a database application
  - If most work is done using HTML authoring tools
- Features:
  - You can include JavaScript or other client-side script code in a PL/SQL server page.
  - PSP uses the same script tag syntax as JavaServer Pages (JSP), to make it easy to switch back and forth.
  - Processing is done on the server.
  - The browser receives a plain HTML page with no special script tags.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### PSP Uses and Features

You can produce HTML pages with dynamic content in several ways. One method is to create PSP. This is useful when you have a large body of HTML, and want to include dynamic content or make it the front end of a database application. If most of the work is done through an HTML authoring tool, PSP is more efficient.

You can also use the PL/SQL Web Toolkit to generate PSPs. This toolkit provides packages such as `OWA`, `http`, and `htf` that are designed for generating Web pages. For more information, take the *Oracle AS 10g: Develop Web Pages with PL/SQL* course. This is useful when there is a large body of PL/SQL code that produces formatted output. If you use authoring tools that produce PL/SQL code for you, such as the page-building wizards in Oracle Application Server Portal, then it might be less convenient to use PSP.

## Format of the PSP File

- The file must have a `.psp` extension.
- The `.psp` file can contain text, tags, PSP directives, declarations, and scriptlets.
- Typically, HTML provides the static portion of the page, and PL/SQL provides the dynamic content.



# Test.psp

ORACLE®

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Format of the PSP File

It is easier to maintain the PSP file if you keep all your directives and declarations together near the beginning of a PL/SQL server page. To share procedures, constants, and types across different PL/SQL server pages, compile them into a separate package in the database by using a plain PL/SQL source file. Although you can reference packaged procedures, constants, and types from PSP scripts, the PSP scripts can only produce stand-alone procedures, not packages.

## Page Directive

Specifies characteristics of the PL/SQL server page:

- What scripting language it uses
- What type of information (MIME type) it produces
- What code to run to handle all uncaught exceptions. This might be an HTML file with a friendly message, renamed to a .psp file.

**Syntax:**

```
<%@ page [language="PL/SQL"]
contentType="content type string" [errorPage="file.psp"] %>
```

## Procedure Directive

Specifies the name of the stored procedure produced by the PSP file. By default, the name is the file name without the .psp extension.

### Syntax:

```
<%@ plsql procedure="procedure name" %>
```

## Format of the PSP File (continued)

### Parameter Directive

Specifies the name, and optionally the type and default, for each parameter expected by the PSP stored procedure.

#### Syntax:

```
<%@ plsql parameter="parameter name"
    [type="PL/SQL type"] [default="value"] %>
```

If the parameter data type is CHARACTER, put single quotation marks around the default value, with double quotation marks surrounding the entire default value.

### Include Directive

Specifies the name of a file to be included at a specific point in the PSP file. The file must have an extension other than .psp. It can contain HTML, PSP script elements, or a combination of both. The name resolution and file inclusion happens when the PSP file is loaded into the database as a stored procedure, so any changes to the file after that are not reflected when the stored procedure is run.

#### Syntax:

```
<%@ include file="path name" %>
```

### Declaration Block

Declares a set of PL/SQL variables that are visible throughout the page, not just within the next BEGIN/END block. This element typically spans multiple lines, with individual PL/SQL variable declarations ended by semicolons.

#### Syntax:

```
<%! PL/SQL declaration; [ PL/SQL declaration; ] ... %>
```

### Code Block (Scriptlets)

Executes a set of PL/SQL statements when the stored procedure is run. This element typically spans multiple lines, with individual PL/SQL statements ended by semicolons. The statements can include complete blocks, or can be the bracketing parts of IF/THEN/ELSE or BEGIN/END blocks. When a code block is split into multiple scriptlets, you can put HTML or other directives in the middle, and those pieces are conditionally executed when the stored procedure is run.

#### Syntax:

```
<% PL/SQL statement; [ PL/SQL statement; ] ... %>
```

### Expression Block

Specifies a single PL/SQL expression, such as a string, an arithmetic expression, a function call, or a combination of those things. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. You do not need to end the PL/SQL expression with a semicolon.

#### Syntax:

```
<%= PL/SQL expression %>
```

**Note:** To identify a file as a PL/SQL server page, include a `<%@ page language="PL/SQL" %>` directive somewhere in the file. This directive is for compatibility with other scripting environments.

## Development Steps for PSP

1. Create the PSP.
2. Load the PSP into the database as a stored procedure.

```
loadpsp [ -replace ]  
-user username/password[@connect_string]  
[ include_file_name ... ] [ error_file_name ]  
psp_file_name ...
```

3. Run the PSP through a URL.

```
http://sitename/schemaname/pspname?parmname1=  
value1&parmname2=value2
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Steps to Create a PSP

#### Step 1

Create an HTML page, embedding the PL/SQL code in the HTML page.

# Development Steps for PSP

## Creating the PSP:

```
<%@ page language="PL/SQL" %>      ← Page directive
<%@ plsql procedure="show_table" %> ← Procedure directive
<% -- Inventories Table Contents -- %> ← Comment
<HTML>
<HEAD><TITLE>Show Contents of Inventories </TITLE></HEAD>
<BODY>
<p><b><font face="Arial, Helvetica, Tahoma"
  size="4">INVENTORIES TABLE: </font></b></p>
<p><%
declare                                ← Scriptlet
dummy boolean;
begin
dummy := owa_util.tableprint('INVENTORIES','border');
end;
%> </p>
<p>&nbsp;</p><p>&nbsp;</p><p>&nbsp;</p></BODY>
</HTML>
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Creating the PSP

First, create an HTML page, embedding the PL/SQL code in the HTML page. In this example, the contents of the INVENTORIES table are displayed in a Web page.

The page directive identifies the scripting language. The procedure directive identifies that a procedure named `show_table` will be created and stored in the database to represent this HTML page. The scriptlet executes a set of PL/SQL statements when the stored procedure is run. The result is substituted as a string at that spot in the HTML page that is produced by the stored procedure. The `owa_util.tableprint` procedure prints out the contents of a database table that is identified to the procedure through the first parameter.

**Note:** `owa_util.tableprint` is part of the PL/SQL Web Toolkit and is installed in the SYS schema.

### Include Comments

To put a comment in the HTML portion of a PL/SQL server page, for the benefit of people reading the PSP source code, use the following syntax:

#### Syntax:

```
<%-- Comment text --%>
```

These comments do not appear in the HTML output from the PSP.

To create a comment that is visible in the HTML output, place the comment in the HTML portion and use the regular HTML comment syntax:

#### Syntax:

```
<!-- Comment text -->
```

## Development Steps for PSP

- Loading the PSP into the database from the operating system:

```
>loadpsp -replace -user oe/oe show_table.psp
"show_table.psp" : procedure "show_table" created.
>
```

- Optionally include other file names and the error file name:

```
>loadpsp -replace -user oe/oe
banner.inc error.psp show_table.psp
"banner.inc": uploaded.
"error.psp": procedure "error" created.
"show_table.psp" : procedure "show_table" created.
>
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Loading the PSP

#### Step 2

In the second step, you load one or more PSP files into the database as stored procedures. Each .psp file corresponds to one stored procedure. To perform a “CREATE OR REPLACE” on the stored procedures, include the `-replace` flag.

The loader logs on to the database using the specified username, password, and connect string. The stored procedures are created in the corresponding schema.

In the first example:

- The stored procedure is created in the database. The database is accessed as user `oe` with password `oe`, both when the stored procedure is created and when it is executed.
- `show_table.psp` contains the main code and text for the Web page.



## Loading the PSP (continued)

In the second example:

- The stored procedure is created in the database. The database is accessed as user `oe` with password `oe`, both to create the stored procedure and when the stored procedure is executed.
- `banner.inc` is a file containing boilerplate text and script code, that is included by the `.psp` file. The inclusion happens when the PSP is loaded into the database, not when the stored procedure is executed.
- `error.psp` is a file containing code or text that is processed when an unhandled exception occurs, to present a friendly page rather than an internal error message.
- `show_table.psp` contains the main code and text for the Web page.

Include the names of all the include files (whose names do not have the `.psp` extension) before the names of the PL/SQL server pages (whose names have the `.psp` extension). Also include the name of the file specified in the `errorPage` attribute of the `page` directive.

These file names on the `loadpsp` command line must exactly match the names specified within the PSP `include` and `page` directives, including any relative pathname such as `../include/`.

## Development Steps for PSP

The `show_table` procedure is stored in the data dictionary views.

```
SQL> SELECT text
  2 FROM   user_source
  3 WHERE  name = 'SHOW_TABLE';

TEXT
-----
PROCEDURE show_table AS
BEGIN NULL;
...
declare
dummy boolean;
begin
dummy := owa_util.tableprint('INVENTORIES','border');
end;
...
23 rows selected.
```

ORACLE

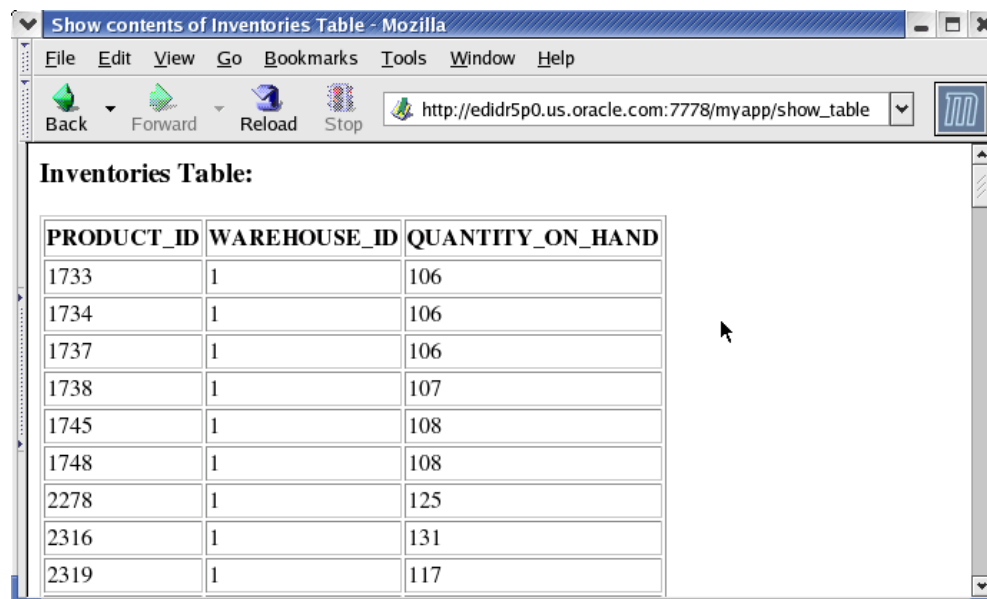
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Loading the PSP (continued)

After the `loadpsp` utility is run, the procedure is created and stored in the database.

# Development Steps for PSP

Running the PSP through a URL:



ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Running the PSP

### Step 3

For the third step, run the PSP in a browser. Identify the HTTP URL through a Web browser or some other Internet-aware client program. The virtual path in the URL depends on the way the Web gateway is configured. The name of the stored procedure is placed at the end of the virtual path.

Using METHOD=GET, the URL may look like this:

`http://sitename/DAD/pspname?parmname1=value1&parmname2=value2`

Using METHOD=POST, the URL does not show the parameters:

`http://sitename/DAD/pspname`

The METHOD=GET format is more convenient for debugging and allows visitors to pass exactly the same parameters when they return to the page through a bookmark.

The METHOD=POST format allows a larger volume of parameter data, and is suitable for passing sensitive information that should not be displayed in the URL.

## Printing the Table Using a Loop

- To print the results of a multirow query, use a loop:

```
<% FOR item IN (SELECT * FROM some_table) LOOP %>
  <TR>
    <TD><%= item.col1 %></TD>
    <TD><%= item.col2 %></TD>
  </TR>
<% END LOOP; %>
```

- Alternatively, use OWA\_UTIL.TABLEPRINT or OWA\_UTIL.CELLSPRINT procedures from the PL/SQL Web Toolkit.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Printing the Content of a Table

You can iterate through each row of the result set, printing the appropriate columns by using HTML list or table tags. Following is an example of a list:

```
<@@ page language="PL/SQL" %>
<@@ plsql procedure="show_customers" %>
<HTML>
<HEAD><TITLE>Show Contents of Customers (using a loop)
</TITLE></HEAD>
<BODY>
<UL>
  <% for item in (select customer_id, cust_first_name,
                    credit_limit, cust_email
                    from customers order by credit_limit) loop %>

    <LI>
      ID = <%= item.customer_id %><BR>
      Name = <%= item.cust_first_name %><BR>
      Credit = <%= item.credit_limit %><BR>
      Email = <I><%= item.cust_email %></I><BR>
    <% end loop; %>
  </UL>
</BODY>
</HTML>
```

## Specifying a Parameter

- Include the parameter directive in the .psp file.

– Syntax:

```
<%@ plsql parameter="parameter name"  
[type="PL/SQL type"] [default="value"] %>
```

– Example:

```
<%@ plsql parameter="mincredit" type="NUMBER"  
default="3000" %>
```

- Assign the parameter a value through the URL call:

```
http://edidr5p0.us.oracle.com/DAD  
/show_customers_hc?mincredit=3000
```

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Specifying a Parameter

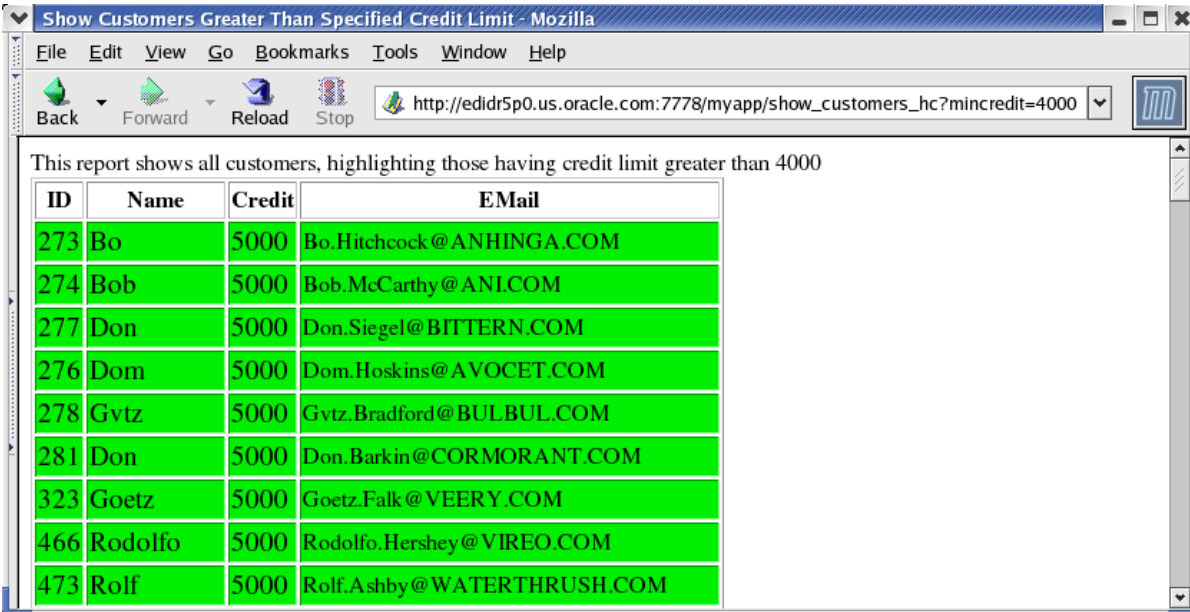
You can pass parameters to the PSP by identifying the parameter name and value in the URL call.

## Specifying a Parameter (continued)

The following example creates a parameter named mincredit. There is also some conditional processing to highlight values that are greater than a specified price.

```
<%@ page language="PL/SQL" %>
<%@ plsql procedure="show_customers_hc" %>
<%@ plsql parameter="mincredit" type="NUMBER"
default="3000" %>
<%! color varchar2(7); %>
<HTML>
<HEAD><TITLE>Show Customers Greater Than Specified Credit
Limit</TITLE></HEAD>
<BODY>
<P>This report shows all customers, highlighting those
having credit limit is greater than <%= mincredit %>.
<TABLE BORDER>
<TR>
<TH>ID</TH>
<TH>Name</TH>
<TH>Credit</TH>
<TH>Email </TH>
</TR>
<%
for item in (select * from customers
              order by credit_limit desc) loop
  if item.credit_limit > mincredit then
    color := '#white';
  else
    color := '#green';
  end if;
%>
<TR BGCOLOR="<%= color %>">
<TD><BIG><%= item.customer_id %></BIG></TD>
<TD><BIG><%= item.cust_first_name %></BIG></TD>
<TD><BIG><%= item.credit_limit %></BIG></TD>
<TD><%= item.cust_email %></TD>
</TR>
<% end loop; %>
</TABLE>
</BODY>
</HTML>
```

# Specifying a Parameter



This report shows all customers, highlighting those having credit limit greater than 4000

ID	Name	Credit	EMail
273	Bo	5000	Bo.Hitchcock@ANHINGA.COM
274	Bob	5000	Bob.McCarthy@ANL.COM
277	Don	5000	Don.Siegel@BITTERN.COM
276	Dom	5000	Dom.Hoskins@AVOCET.COM
278	Gvtz	5000	Gvtz.Bradford@BULBUL.COM
281	Don	5000	Don.Barkin@CORMORANT.COM
323	Goetz	5000	Goetz.Falk@VEERY.COM
466	Rodolfo	5000	Rodolfo.Hershey@VIREO.COM
473	Rolf	5000	Rolf.Ashby@WATERTHRUSH.COM

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

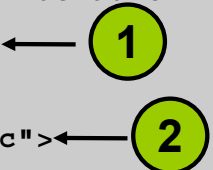
## Specifying a Parameter (continued)

You passed `mincredit=4000` as the parameter along with the URL. The output shows all the records and highlights those having a credit limit greater than 4,000.

## Using an HTML Form to Call a PSP

1. Create an HTML form.
2. Call the PSP from the form.

```
<@@ page language="PL/SQL" %>
<@@ plsql procedure="show_customer_call" %>
<@@ plsql parameter="mincredit" type="NUMBER" default=
"3000" %>
<html>
<body>
<form method="POST" action="show_customers_hc">
<p>Enter the credit limit:
<input type="text" name="mincredit">
<input type="submit" value="Submit">
</form>
</body>
</html>
```



ORACLE

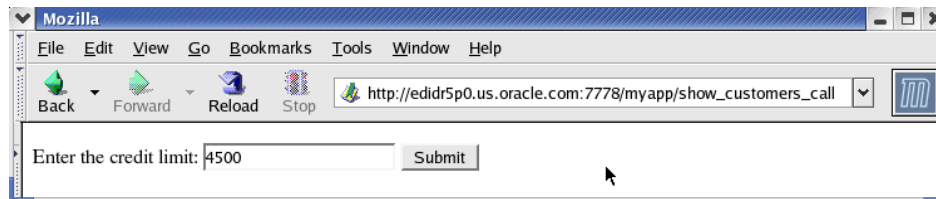
Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Calling a PSP from an HTML Form

Create an HTML form that calls the PSP. To avoid coding the entire URL of the stored procedure in the ACTION= attribute of the form, make the form a PSP file so that it goes in the same directory as the PSP file it calls.



## Using an HTML Form to Call a PSP



This report shows all customers, highlighting those having credit limit greater than 4500

ID	Name	Credit	E-Mail
273	Bo	5000	Bo.Hitchcock@ANHINGA.COM
274	Bob	5000	Bob.McCarthy@ANL.COM
277	Don	5000	Don.Siegel@BITTERN.COM
276	Dom	5000	Dom.Hoskins@AVOCET.COM
278	Gvtz	5000	Gvtz.Bradford@BULBUL.COM
281	Don	5000	Don.Barkin@CORMORANT.COM
323	Goetz	5000	Goetz.Falk@VEERY.COM

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Calling a PSP from an HTML Form (continued)

Initially, you are calling the HTML form that accepts the credit limit from the user. After submitting the HTML form, call the PSP, which shows all the records and highlight all the records having a credit limit greater than the value submitted by the user.

## Debugging PSP Problems

- Code the PL/SQL syntax and PSP directive syntax correctly. It will not compile with syntax errors.
- Run the PSP file by requesting its URL in a Web browser. An error might indicate that the file is not found.
- When the PSP script is run, and the results come back to the browser, use standard debugging techniques to check for and correct wrong output.
- Use `http.p('string')` to print information to the screen.

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

### Debugging PSP Problems

The first step is to code PL/SQL syntax and PSP directive syntax correctly. It will not compile with syntax errors.

- Use semicolons to terminate lines if required.
- If required, enclose a value with quotation marks. You may need to enclose a value that is within single quotation marks (needed by PL/SQL) inside double quotation marks (needed by PSP).
- Mistakes in the PSP directives are usually reported through PL/SQL syntax messages. Check that your directives use the right syntax, that directives are closed properly, and that you are using the right element (declaration, expression, or code block) depending on what goes inside it.
- PSP attribute names are case-sensitive. Most are specified in all lowercase; `contentType` and `errorPage` must be specified as mixed case.

Run the PSP file by requesting its URL in a Web browser.

- Request the right virtual path, depending on the way the Web gateway is configured. Typically, the path includes the host name, optionally a port number, the schema name, and the name of the stored procedure (with no `.psp` extension).
- If you use the `-replace` option when compiling the file, the old version of the stored procedure is erased. You may want to test new scripts in a separate schema until they are ready, and then load them into the production schema.
- If you copied the file from another file, remember to change any procedure name directives in the source to match the new file name.

## Debugging PSP Problems (continued)

- If you receive one file-not-found error, make sure to request the latest version of the page the next time. The error page may be cached by the browser. You may need to press and hold Shift and click Reload in the browser to bypass its cache.

When the PSP script is run, and the results come back to the browser, use standard debugging techniques to check for and correct wrong output. The tricky part is to set up the interface between different HTML forms, scripts, and CGI programs so that all the right values are passed into your page. The page may return an error because of a parameter mismatch.

- To see exactly what is being passed to your page, use METHOD=GET in the calling form so that the parameters are visible in the URL.
- Make sure that the form or CGI program that calls your page passes the correct number of parameters, and that the names specified by the NAME=attributes on the form match the parameter names in the PSP file. If the form includes any hidden input fields, or uses the NAME= attribute on the Submit or Reset buttons, then the PSP file must declare equivalent parameters.
- Make sure that the parameters can be cast from string into the correct PL/SQL types. For example, do not include alphabetic characters if the parameter in the PSP file is declared as a NUMBER.
- Make sure that the query string of the URL consists of name-value pairs, separated by equal signs, especially if you are passing parameters by constructing a hard-coded link to the page.
- If you are passing a lot of parameter data, such as large strings, you may exceed the volume that can be passed with METHOD=GET. You can switch to METHOD=POST in the calling form without changing your PSP file.
- You can display text or variables by putting the following in your code:

```
http.p(' My Var: ' || my_var);
```

When you run the program, the information is displayed on the screen.

## Summary

In this lesson, you should have learned how to:

- Define PL/SQL server pages
- Explain the format of a PL/SQL server page
- Write the code and content for the PL/SQL server page
- Load the PL/SQL server page into the database as a stored procedure
- Run a PL/SQL server page via a URL
- Debug PL/SQL server page problems

ORACLE

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

## Summary

You can use PL/SQL embedded in HTML and store the code as a PL/SQL server page (PSP) in the database. The three steps for creating a PSP are:

1. Create the PSP.
2. Load the PSP into the database as a stored procedure.
3. Run the PSP in a browser.