

Oracle Database 11g: Develop PL/SQL Program Units

Volume II • Student Guide

D49986GC20

Edition 2.0

September 2009

D63066

ORACLE®

Author

Lauran Serhal

**Technical Contributors
and Reviewers**

Anjulaponni Azhagulekshmi
Christian Bauwens
Christoph Burandt
Zarko Cesljas
Yanti Chang
Salome Clement
Laszlo Czinkoczki
Ingrid DelaHaye
Steve Friedberg
Laura Garza
Joel Goodman
Nancy Greenberg
Manish Pawar
Brian Pottle
Helen Robertson
Tulika Srivastava
Ted Witiuk

Editors

Arijit Ghosh
Raj Kumar

Publishers

Pavithran Adka
Sheryl Domingue

Copyright © 2009, Oracle. 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

I Introduction

Lesson Objectives	I-2
Lesson Agenda	I-3
Course Objectives	I-4
Suggested Course Agenda	I-5
Lesson Agenda	I-7
The Human Resources (HR) Schema That Is Used in This Course	I-8
Class Account Information	I-9
Appendices Used in This Course	I-10
PL/SQL Development Environments	I-11
What Is Oracle SQL Developer?	I-12
Coding PL/SQL in SQL*Plus	I-13
Coding PL/SQL in Oracle JDeveloper	I-14
Enabling Output of a PL/SQL Block	I-15
Lesson Agenda	I-16
Oracle 11g SQL and PL/SQL Documentation	I-17
Additional Resources	I-18
Summary	I-19
Practice I Overview: Getting Started	I-20

1 Creating Procedures

Objectives	1-2
Lesson Agenda	1-3
Creating a Modularized Subprogram Design	1-4
Creating a Layered Subprogram Design	1-5
Modularizing Development with PL/SQL Blocks	1-6
Anonymous Blocks: Overview	1-7
PL/SQL Run-time Architecture	1-8
What Are PL/SQL Subprograms?	1-9
The Benefits of Using PL/SQL Subprograms	1-10
Differences Between Anonymous Blocks and Subprograms	1-11
Lesson Agenda	1-12
What Are Procedures?	1-13
Creating Procedures: Overview	1-14
Creating Procedures with the SQL CREATE OR REPLACE Statement	1-15
Creating Procedures Using SQL Developer	1-16

Compiling Procedures and Displaying Compilation Errors in SQL Developer	1-17
Correcting Compilation Errors in SQL Developer	1-18
Naming Conventions of PL/SQL Structures Used in This Course	1-19
What Are Parameters and Parameter Modes?	1-20
Formal and Actual Parameters	1-21
Procedural Parameter Modes	1-22
Comparing the Parameter Modes	1-23
Using the <code>IN</code> Parameter Mode: Example	1-24
Using the <code>OUT</code> Parameter Mode: Example	1-25
Using the <code>IN OUT</code> Parameter Mode: Example	1-26
Viewing the <code>OUT</code> Parameters: Using the <code>DBMS_OUTPUT.PUT_LINE</code> Subroutine	1-27
Viewing <code>OUT</code> Parameters: Using SQL*Plus Host Variables	1-28
Available Notations for Passing Actual Parameters	1-29
Passing Actual Parameters: Creating the <code>add_dept</code> Procedure	1-30
Passing Actual Parameters: Examples	1-31
Using the <code>DEFAULT</code> Option for the Parameters	1-32
Calling Procedures	1-34
Calling Procedures Using SQL Developer	1-35
Lesson Agenda	1-36
Handled Exceptions	1-37
Handled Exceptions: Example	1-38
Exceptions Not Handled	1-39
Exceptions Not Handled: Example	1-40
Removing Procedures: Using the <code>DROP</code> SQL Statement or SQL Developer	1-41
Viewing Procedure Information Using the Data Dictionary Views	1-42
Viewing Procedures Information Using SQL Developer	1-43
Quiz	1-44
Summary	1-45
Practice 1 Overview: Creating, Compiling, and Calling Procedures	1-46

2 Creating Functions and Debugging Subprograms

Objectives	2-2
Lesson Agenda	2-3
Overview of Stored Functions	2-4
Creating Functions	2-5
The Difference Between Procedures and Functions	2-6
Creating and Running Functions: Overview	2-7
Creating and Invoking a Stored Function Using the <code>CREATE FUNCTION</code> Statement: Example	2-8

Using Different Methods for Executing Functions	2-9
Creating and Compiling Functions Using SQL Developer	2-11
Executing Functions Using SQL Developer	2-12
Advantages of User-Defined Functions in SQL Statements	2-13
Using a Function in a SQL Expression: Example	2-14
Calling User-Defined Functions in SQL Statements	2-15
Restrictions When Calling Functions from SQL Expressions	2-16
Controlling Side Effects When Calling Functions from SQL Expressions	2-17
Restrictions on Calling Functions from SQL: Example	2-18
Named and Mixed Notation from SQL	2-19
Named and Mixed Notation from SQL: Example	2-20
Removing Functions: Using the DROP SQL Statement or SQL Developer	2-21
Viewing Functions Using Data Dictionary Views	2-22
Viewing Functions Information Using SQL Developer	2-23
Quiz	2-24
Practice 2-1: Overview	2-25
Lesson Agenda	2-26
Debugging PL/SQL Subprograms Using the SQL Developer Debugger	2-27
Debugging a Subprogram: Overview	2-28
The Procedure or Function Code Editing Tab	2-29
The Procedure or Function Tab Toolbar	2-30
The Debugging – Log Tab Toolbar	2-31
Additional Tabs	2-33
Debugging a Procedure Example: Creating a New emp_list Procedure	2-34
Debugging a Procedure Example: Creating a New get_location Function	2-35
Setting Breakpoints and Compiling emp_list for Debug Mode	2-36
Compiling the get_location Function for Debug Mode	2-37
Debugging emp_list and Entering Values for the PMAXROWS Parameter	2-38
Debugging emp_list: Step Into (F7) the Code	2-39
Viewing the Data	2-41
Modifying the Variables While Debugging the Code	2-42
Debugging emp_list: Step Over the Code	2-43
Debugging emp_list: Step Out of the Code (Shift + F7)	2-44
Debugging emp_list: Run to Cursor (F4)	2-45
Debugging emp_list: Step to End of Method	2-46
Debugging a Subprogram Remotely: Overview	2-47
Practice 2-2 Overview: Introduction to the SQL Developer Debugger	2-48
Summary	2-49

3 Creating Packages

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

4 Working with Packages

- Objectives 4-2
- Lesson Agenda 4-3
- Overloading Subprograms in PL/SQL 4-4
- Overloading Procedures Example: Creating the Package Specification 4-6
- Overloading Procedures Example: Creating the Package Body 4-7
- Overloading and the STANDARD Package 4-8
- Illegal Procedure Reference 4-9
- Using Forward Declarations to Solve Illegal Procedure Reference 4-10
- Initializing Packages 4-11
- Using Package Functions in SQL 4-12
- Controlling Side Effects of PL/SQL Subprograms 4-13
- Package Function in SQL: Example 4-14

Lesson Agenda	4-15
Persistent State of Packages	4-16
Persistent State of Package Variables: Example	4-18
Persistent State of a Package Cursor: Example	4-19
Executing the CURS_PKG Package	4-21
Using Associative Arrays in Packages	4-22
Quiz	4-23
Summary	4-24
Practice 4: Overview	4-25
5 Using Oracle-Supplied Packages in Application Development	
Objectives	5-2
Lesson Agenda	5-3
Using Oracle-Supplied Packages	5-4
Examples of Some Oracle-Supplied Packages	5-5
Lesson Agenda	5-6
How the DBMS_OUTPUT Package Works	5-7
Using the UTL_FILE Package to Interact with Operating System Files	5-8
Some of the UTL_FILE Procedures and Functions	5-9
File Processing Using the UTL_FILE Package: Overview	5-10
Using the Available Declared Exceptions in the UTL_FILE Package	5-11
FOPEN and IS_OPEN Functions: Example	5-12
Using UTL_FILE: Example	5-14
What Is the UTL_MAIL Package?	5-16
Setting Up and Using the UTL_MAIL: Overview	5-17
Summary of UTL_MAIL Subprograms	5-18
Installing and Using UTL_MAIL	5-19
The SEND Procedure Syntax	5-20
The SEND_ATTACH_RAW Procedure	5-21
Sending Email with a Binary Attachment: Example	5-22
The SEND_ATTACH_VARCHAR2 Procedure	5-24
Sending Email with a Text Attachment: Example	5-25
Quiz	5-27
Summary	5-28
Practice 5: Overview	5-29
6 Using Dynamic SQL	
Objectives	6-2
Lesson Agenda	6-3
Execution Flow of SQL	6-4

Working with Dynamic SQL	6-5
Using Dynamic SQL	6-6
Native Dynamic SQL (NDS)	6-7
Using the EXECUTE IMMEDIATE Statement	6-8
Available Methods for Using NDS	6-9
Dynamic SQL with a DDL Statement: Examples	6-11
Dynamic SQL with DML Statements	6-12
Dynamic SQL with a Single-Row Query: Example	6-13
Executing a PL/SQL Anonymous Block Dynamically	6-14
Using Native Dynamic SQL to Compile PL/SQL Code	6-15
Lesson Agenda	6-16
Using the DBMS_SQL Package	6-17
Using the DBMS_SQL Package Subprograms	6-18
Using DBMS_SQL with a DML Statement: Deleting Rows	6-20
Using DBMS_SQL with a Parameterized DML Statement	6-21
Quiz	6-22
Summary	6-23
Practice 6 Overview: Using Native Dynamic SQL	6-24

7 Design Considerations for PL/SQL Code

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

Using the DETERMINISTIC Clause with Functions	7-24
Lesson Agenda	7-25
Using the RETURNING Clause	7-26
Using Bulk Binding	7-27
Bulk Binding: Syntax and Keywords	7-28
Bulk Binding FORALL: Example	7-30
Using BULK COLLECT INTO with Queries	7-32
Using BULK COLLECT INTO with Cursors	7-33
Using BULK COLLECT INTO with a RETURNING Clause	7-34
Using Bulk Binds in Sparse Collections	7-35
Using Bulk Bind with Index Array	7-38
Quiz	7-39
Summary	7-40
Practice 7: Overview	7-41

8 Creating Triggers

Objectives	8-2
What Are Triggers?	8-3
Defining Triggers	8-4
Trigger Event Types	8-5
Application and Database Triggers	8-6
Business Application Scenarios for Implementing Triggers	8-7
Available Trigger Types	8-8
Trigger Event Types and Body	8-9
Creating DML Triggers Using the CREATE TRIGGER Statement	8-10
Specifying the Trigger Firing (Timing)	8-11
Statement-Level Triggers Versus Row-Level Triggers	8-12
Creating DML Triggers Using SQL Developer	8-13
Trigger-Firing Sequence: Single-Row Manipulation	8-14
Trigger-Firing Sequence: Multirow Manipulation	8-15
Creating a DML Statement Trigger Example: SECURE_EMP	8-16
Testing Trigger SECURE_EMP	8-17
Using Conditional Predicates	8-18
Creating a DML Row Trigger	8-19
Using OLD and NEW Qualifiers	8-20
Using OLD and NEW Qualifiers: Example	8-21
Using OLD and NEW Qualifiers: Example the Using AUDIT_EMP Table	8-22
Using the WHEN Clause to Fire a Row Trigger Based on a Condition	8-23
Summary of the Trigger Execution Model	8-24
Implementing an Integrity Constraint with an After Trigger	8-25

INSTEAD OF Triggers	8-26
Creating an INSTEAD OF Trigger: Example	8-27
Creating an INSTEAD OF Trigger to Perform DML on Complex Views	8-28
The Status of a Trigger	8-30
Creating a Disabled Trigger	8-31
Managing Triggers Using the ALTER and DROP SQL Statements	8-32
Managing Triggers Using SQL Developer	8-33
Testing Triggers	8-34
Viewing Trigger Information	8-35
Using USER_TRIGGERS	8-36
Quiz	8-37
Summary	8-38
Practice 8 Overview: Creating Statement and Row Triggers	8-39

9 Creating Compound, DDL, and Event Database Triggers

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

10 Using the PL/SQL Compiler

- Objectives 10-2
- Lesson Agenda 10-3
- Initialization Parameters for PL/SQL Compilation 10-4
- Using the Initialization Parameters for PL/SQL Compilation 10-5
- The Compiler Settings 10-7
- Displaying the PL/SQL Initialization Parameters 10-8
- Displaying and Setting the PL/SQL Initialization Parameters 10-9
- Changing PL/SQL Initialization Parameters: Example 10-10
- Lesson Agenda 10-11
- Overview of PL/SQL Compile-Time Warnings for Subprograms 10-12
- Benefits of Compiler Warnings 10-14
- Categories of PL/SQL Compile-Time Warning Messages 10-15
- Setting the Warning Messages Levels 10-16
- Setting Compiler Warning Levels: Using `PLSQL_WARNINGS` 10-17
- Setting Compiler Warning Levels: Using `PLSQL_WARNINGS`, Examples 10-18
- Setting Compiler Warning Levels: Using `PLSQL_WARNINGS` in SQL
 - Developer 10-19
- Viewing the Current Setting of `PLSQL_WARNINGS` 10-20
- Viewing the Compiler Warnings: Using SQL Developer, SQL*Plus, or Data Dictionary
 - Views 10-21
- SQL*Plus Warning Messages: Example 10-22
- Guidelines for Using `PLSQL_WARNINGS` 10-23
- Lesson Agenda 10-24
- Setting Compiler Warning Levels: Using the `DBMS_WARNING` Package 10-25
- Using the `DBMS_WARNING` Package Subprograms 10-27
- The `DBMS_WARNING` Procedures: Syntax, Parameters, and Allowed Values 10-28
- The `DBMS_WARNING` Procedures: Example 10-29
- The `DBMS_WARNING` Functions: Syntax, Parameters, and Allowed Values 10-30
- The `DBMS_WARNING` Functions: Example 10-31
- Using `DBMS_WARNING`: Example 10-32
- Using the PLW 06009 Warning Message 10-34
- The PLW 06009 Warning: Example 10-35
- Quiz 10-36
- Summary 10-37
- Practice 10: Overview 10-38

11 Managing PL/SQL Code

- Objectives 11-2
- Lesson Agenda 11-3
- What Is Conditional Compilation? 11-4
- How Does Conditional Compilation Work? 11-5
- Using Selection Directives 11-6
- Using Predefined and User-Defined Inquiry Directives 11-7
- The `PLSQL_CCFLAGS` Parameter and the Inquiry Directive 11-8
- Displaying the `PLSQL_CCFLAGS` Initialization Parameter Setting 11-9
- The `PLSQL_CCFLAGS` Parameter and the Inquiry Directive: Example 11-10
- Using Conditional Compilation Error Directives to Raise User-Defined Errors 11-11
- Using Static Expressions with Conditional Compilation 11-12
- The `DBMS_DB_VERSION` Package: Boolean Constants 11-13
- The `DBMS_DB_VERSION` Package Constants 11-14
- Using Conditional Compilation with Database Versions: Example 11-15
- Using `DBMS_PREPROCESSOR` Procedures to Print or Retrieve Source Text 11-17
- Lesson Agenda 11-18
- What Is Obfuscation? 11-19
- Benefits of Obfuscating 11-20
- What's New in Dynamic Obfuscating Since Oracle 10g? 11-21
- Nonobfuscated PL/SQL Code: Example 11-22
- Obfuscated PL/SQL Code: Example 11-23
- Dynamic Obfuscation: Example 11-24
- The PL/SQL Wrapper Utility 11-25
- Running the Wrapper Utility 11-26
- Results of Wrapping 11-27
- Guidelines for Wrapping 11-28
- DBMS_DDL Package Versus the Wrap Utility 11-29
- Quiz 11-30
- Summary 11-31
- Practice 11: Overview 11-32

12 Managing Dependencies

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

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

Appendix A: Practices and Solutions

Appendix AP: Additional Practices and Solutions

Appendix B: Table Descriptions

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: Using JDeveloper

Objectives	E-2
Oracle JDeveloper	E-3
Database Navigator	E-4
Creating a Connection	E-5
Browsing Database Objects	E-6
Executing SQL Statements	E-7
Creating Program Units	E-8
Compiling	E-9
Running a Program Unit	E-10
Dropping a Program Unit	E-11
Structure Window	E-12
Editor Window	E-13
Application Navigator	E-14
Deploying Java Stored Procedures	E-15
Publishing Java to PL/SQL	E-16
How Can I Learn More About JDeveloper 11g ?	E-17
Summary	E-18

Appendix F: Review of PL/SQL

Objectives	F-2
Block Structure for Anonymous PL/SQL Blocks	F-3
Declaring PL/SQL Variables	F-4
Declaring Variables with the %TYPE Attribute: Examples	F-5
Creating a PL/SQL Record	F-6
%ROWTYPE Attribute: Examples	F-7
Creating a PL/SQL Table	F-8
SELECT Statements in PL/SQL: Example	F-9
Inserting Data: Example	F-10
Updating Data: Example	F-11
Deleting Data: Example	F-12
Controlling Transactions with the COMMIT and ROLLBACK Statements	F-13
IF, THEN, and ELSIF Statements: Example	F-14
Basic Loop: Example	F-15
FOR Loop: Example	F-16

WHILE Loop: Example	F-17
SQL Implicit Cursor Attributes	F-18
Controlling Explicit Cursors	F-19
Controlling Explicit Cursors: Declaring the Cursor	F-20
Controlling Explicit Cursors: Opening the Cursor	F-21
Controlling Explicit Cursors: Fetching Data from the Cursor	F-22
Controlling Explicit Cursors: Closing the Cursor	F-23
Explicit Cursor Attributes	F-24
Cursor FOR Loops: Example	F-25
FOR UPDATE Clause: Example	F-26
WHERE CURRENT OF Clause: Example	F-27
Trapping Predefined Oracle Server Errors	F-28
Trapping Predefined Oracle Server Errors: Example	F-29
Non-Predefined Error	F-30
User-Defined Exceptions: Example	F-31
RAISE_APPLICATION_ERROR Procedure	F-32
Summary	F-34

Appendix G: Studies for Implementing Triggers

Objectives	G-2
Controlling Security Within the Server	G-3
Controlling Security with a Database Trigger	G-4
Enforcing Data Integrity Within the Server	G-5
Protecting Data Integrity with a Trigger	G-6
Enforcing Referential Integrity Within the Server	G-7
Protecting Referential Integrity with a Trigger	G-8
Replicating a Table Within the Server	G-9
Replicating a Table with a Trigger	G-10
Computing Derived Data Within the Server	G-11
Computing Derived Values with a Trigger	G-12
Logging Events with a Trigger	G-13
Summary	G-15

Appendix H: Using the DBMS_SCHEDULER and HTP Packages

Objectives	H-2
Generating Web Pages with the HTP Package	H-3
Using the HTP Package Procedures	H-4
Creating an HTML File with SQL*Plus	H-5
The DBMS_SCHEDULER Package	H-6
Creating a Job	H-8

Creating a Job with Inline Parameters	H-9
Creating a Job Using a Program	H-10
Creating a Job for a Program with Arguments	H-11
Creating a Job Using a Schedule	H-12
Setting the Repeat Interval for a Job	H-13
Creating a Job Using a Named Program and Schedule	H-14
Managing Jobs	H-15
Data Dictionary Views	H-16
Summary	H-17

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a
non-transferable license to use this Student Guide.

Appendix A

Practices and Solutions

Table of Contents

Practices and Solutions for Lesson I	4
Practice I-1: Identifying the Available SQL Developer Resources	5
Practice I-2: Creating and Using a New SQL Developer Database Connection	6
Practice I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block	7
Practice I-4: Setting Some SQL Developer Preferences.....	8
Practice I-5: Accessing the Oracle Database 11g Release 2 Online Documentation Library.....	9
Practice Solutions I-1: Identifying the Available SQL Developer Resources	10
Practice Solutions I-2: Creating and Using a New SQL Developer Database Connection	12
Practice Solutions I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block	15
Practice Solutions I-4: Setting Some SQL Developer Preferences	19
Practice Solutions I-5: Accessing the Oracle Database 11g Release 2 Online Documentation Library	23
Practices and Solutions for Lesson 1	24
Practice 1-1: Creating, Compiling, and Calling Procedures	25
Practice Solutions 1-1: Creating, Compiling, and Calling Procedures	27
Practices and Solutions for Lesson 2	38
Practice 2-1: Creating Functions.....	39
Practice 2-2: Introduction to the SQL Developer Debugger	41
Practice Solutions 2-1: Creating Functions.....	42
Practice Solutions 2-2: Introduction to the SQL Developer Debugger	48
Practices and Solutions for Lesson 3	58
Practice 3-1: Creating and Using Packages	59
Practice Solutions 3-1: Creating and Using Packages	61
Practices and Solutions for Lesson 4	68
Practice 4-1: Working with Packages	69
Practice Solutions 4-1: Working with Packages	73
Practices and Solutions for Lesson 5	102
Practice 5-1: Using the UTL_FILE Package	103
Practice Solutions 5-1: Using the UTL_FILE Package	104
Practices and Solutions for Lesson 6	108
Practice 6-1: Using Native Dynamic SQL.....	109
Practice Solutions 6-1: Using Native Dynamic SQL.....	111
Practices and Solutions for Lesson 7	121
Practice 7-1: Using Bulk Binding and Autonomous Transactions	122
Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions	124
Practices and Solutions for Lesson 8	145
Practice 8-1: Creating Statement and Row Triggers.....	146
Practice Solutions 8-1: Creating Statement and Row Triggers	148
Practices and Solutions for Lesson 9	157
Practice 9-1: Managing Data Integrity Rules and Mutating Table Exceptions	158

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions	161
Practices and Solutions for Lesson 10	174
Practice 10-1: Using the PL/SQL Compiler Parameters and Warnings	175
Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings	176
Practices and Solutions for Lesson 11	185
Practice 11-1: Using Conditional Compilation.....	186
Practice Solutions 11-1: Using Conditional Compilation.....	188
Practices and Solutions for Lesson 12	195
Practice 12-1: Managing Dependencies in Your Schema.....	196
Practice Solutions 12-1: Managing Dependencies in Your Schema	197

Practices and Solutions for Lesson I

This is the first of many practices in this course. The solutions (if you require them) can be found in “Appendix A: Practices and Solutions.” Practices are intended to cover most of the topics that are presented in the corresponding lesson.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

In this practice, you review the available SQL Developer resources. You also learn about your user account that you will use in this course. You then start SQL Developer, create a new database connection, browse your schema tables, and create and execute a simple anonymous block. You also set some SQL Developer preferences, execute SQL statements, and execute an anonymous PL/SQL block using SQL Worksheet. Finally, you access and bookmark the Oracle Database 11g documentation and other useful Web sites that you can use in this course.

Practice I-1: Identifying the Available SQL Developer Resources

In this practice, you review the available SQL Developer resources.

- 1) Familiarize yourself with Oracle SQL Developer as needed using Appendix C: Using SQL Developer.
- 2) Access the SQL Developer Home Page available online at:
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 available online at:
<http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>
- 5) Preview and experiment with the available links and demos in the tutorial as needed, especially the “Creating a Database Connection” and “Accessing Data” links.

Practice I-2: Creating and Using a New SQL Developer Database Connection

In this practice, you start SQL Developer using your connection information and create a new database connection.

- 1) Start up SQL Developer using the user ID and password that are provided to you by the instructor such as ora61.
- 2) Create a database connection using the following information:
 - a) Connection Name: MyDBConnection
 - b) Username: ora61
 - c) Password: ora61
 - d) Hostname: Enter the host name for your PC
 - e) Port: 1521
 - f) SID: ORCL
- 3) Test the new connection. If the Status is Success, connect to the database using this new connection:
 - a) Double-click the MyDBConnection 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 I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block

In this practice, you browse your schema tables and create and execute a simple anonymous block.

- 1) Browse the structure of the EMPLOYEES table and display its data.
 - a) Expand the MyDBConnection connection by clicking the plus sign next to it.
 - b) Expand the Tables icon by clicking the plus sign next to it.
 - c) Display the structure of the EMPLOYEES table.
 - 2) Browse the EMPLOYEES table and display its data.
 - 3) 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 icon (F5) icons to execute the SELECT statement. Review the results of both methods of executing the SELECT statements in the appropriate tabs.
- Note:** Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides the description and data for all the tables in the HR schema that you will use in this course.
- 4) 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 (F5) icon to run the anonymous block.

Practice I-4: Setting Some SQL Developer Preferences

In this practice, you set some SQL Developer preferences.

- 1) In the SQL Developer menu, navigate to Tools > Preferences. The Preferences window is displayed.
- 2) Expand the Code Editor option, and then click the Display option to display the “Code Editor: Display” section. The “Code Editor: Display” section contains general options for the appearance and behavior of the code editor.
 - a) Enter 100 in the Right Margin Column text box in the Show Visible Right Margin section. This renders a right margin that you can set to control the length of lines of code.
 - b) Click the Line Gutter option. The Line Gutter option specifies options for the line gutter (left margin of the code editor). Select the Show Line Numbers check box to display the code line numbers.
- 3) Click the Worksheet Parameters option under the Database option. In the “Select default path to look for scripts” text box, specify the /home/oracle/labs/plpu folder. This folder contains the solutions scripts, code examples scripts, and any labs or demos used in this course.
- 4) Click OK to accept your changes and to exit the Preferences window.
- 5) Familiarize yourself with the /home/oracle/labs/plpu folder.
 - a) Click the **Files** tab (next to the **Connections** tab).
 - b) Navigate to the /home/oracle/labs/plpu folder. How many subfolders do you see in the labs folder?
 - c) Navigate through the folders, and open a script file without executing the code.
 - d) Clear the displayed code in the SQL Worksheet area.

Practice I-5: Accessing the Oracle Database 11g Release 2 Online Documentation Library

In this practice, you access and bookmark some of the Oracle Database 11g Release 2 documentation references that you will use in this course.

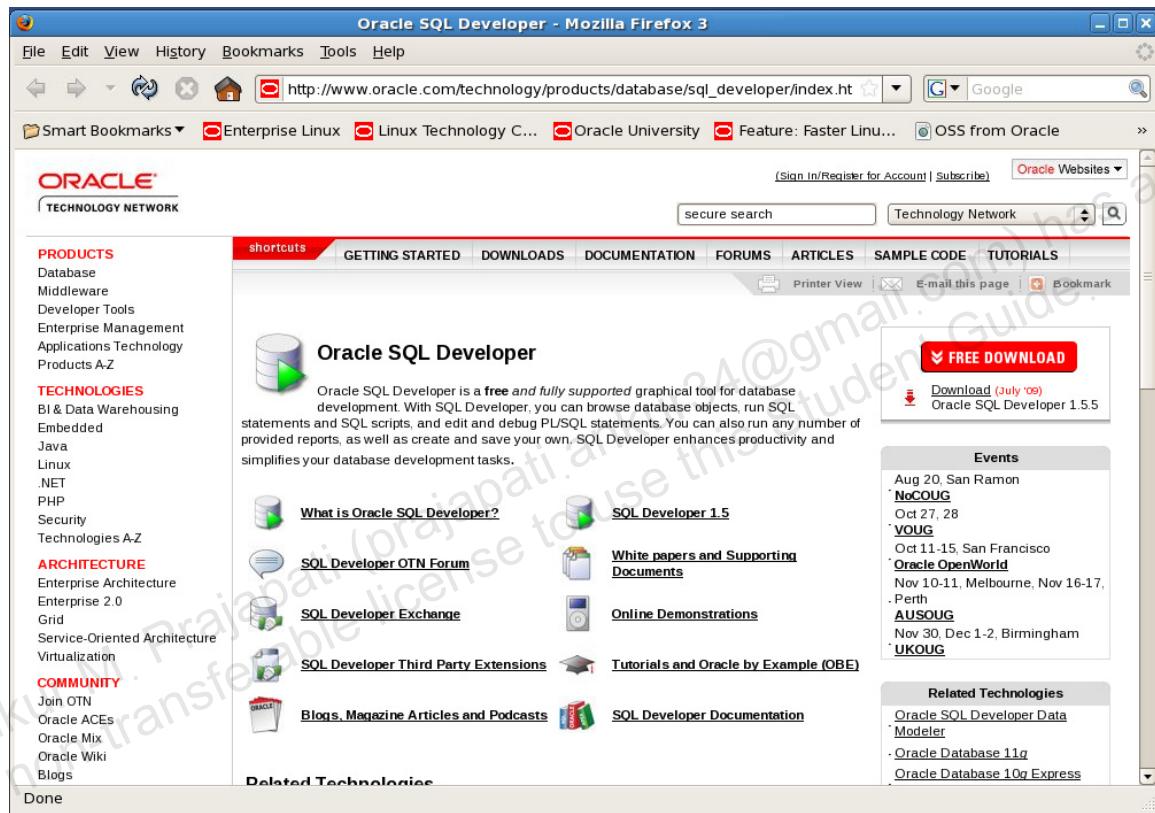
- 1) Access the Oracle Database 11g Release 2 documentation Web page at:
<http://www.oracle.com/pls/db111/homepage>.
- 2) Bookmark the page for easier future access.
- 3) Display the complete list of books available for Oracle Database 11g Release 2.
- 4) Make a note of the following documentation references that you will use in this course as needed:
 - a) *Advanced Application Developer's Guide*
 - b) *New Features Guide*
 - c) *PL/SQL Language Reference*
 - d) *Oracle Database Reference*
 - e) *Oracle Database Concepts*
 - f) *SQL Developer User's Guide*
 - g) *SQL Language Reference Guide*
 - h) *SQL*Plus User's Guide and Reference*

Practice Solutions I-1: Identifying the Available SQL Developer Resources

In this practice, you review the available SQL Developer resources.

- 1) Familiarize yourself with Oracle SQL Developer as needed using Appendix C: Using SQL Developer.
- 2) Access the online SQL Developer Home Page available online at:
http://www.oracle.com/technology/products/database/sql_developer/index.html

The SQL Developer Home page is displayed as follows:



- 3) Bookmark the page for easier future access.

No formal solution. The link is added to your Links toolbar as follows:

- 4) Access the SQL Developer tutorial available online at:
<http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>

Access the SQL Developer tutorial using the preceding URL. The following page is displayed:

Practice Solutions I-1: Identifying the Available SQL Developer Resources (continued)

The screenshot shows the Oracle SQL Developer Tutorial website in a browser window. The left sidebar contains a tree view of tutorial topics. A red box highlights the 'What to Do First' section, which includes 'Creating a Database Connection' and 'Accessing Data'. The main content area displays the 'Welcome to the Oracle SQL Developer Tutorial!' page, which includes a brief description of the tutorial's purpose, learning objectives, and a 'Start Tutorial' button.

- 5) Preview and experiment with the available links and demos 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.

Practice Solutions I-2: Creating and Using a New SQL Developer Database Connection

In this practice, you start SQL Developer using your connection information and create a new database connection.

- 1) Start up SQL Developer using the user ID and password that are provided to you by the instructor such as ora61.

Click the SQL Developer icon on your desktop.

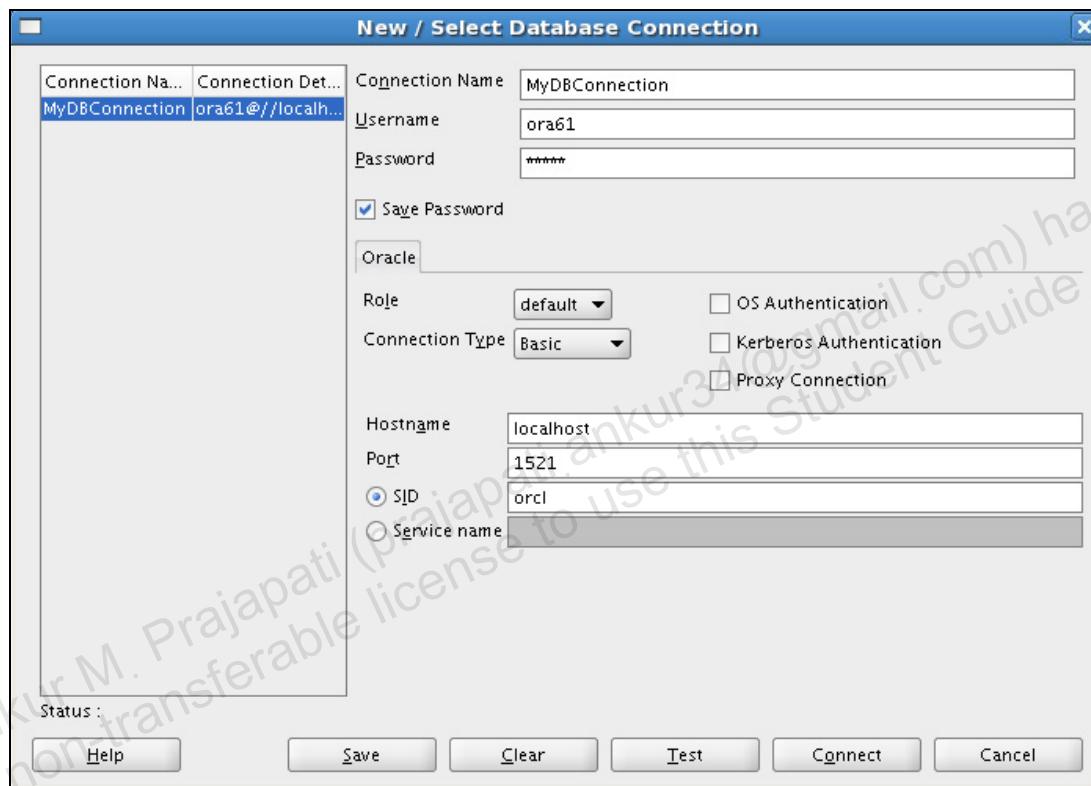


- 2) Create a database connection using the following information:
 - a) Connection Name: MyDBConnection
 - b) Username: ora61
 - c) Password: ora61
 - d) Hostname: Enter the host name for your PC
 - e) Port: 1521
 - f) SID: ORCL

Practice Solutions I-2: Creating and Using a New SQL Developer Database Connection (continued)

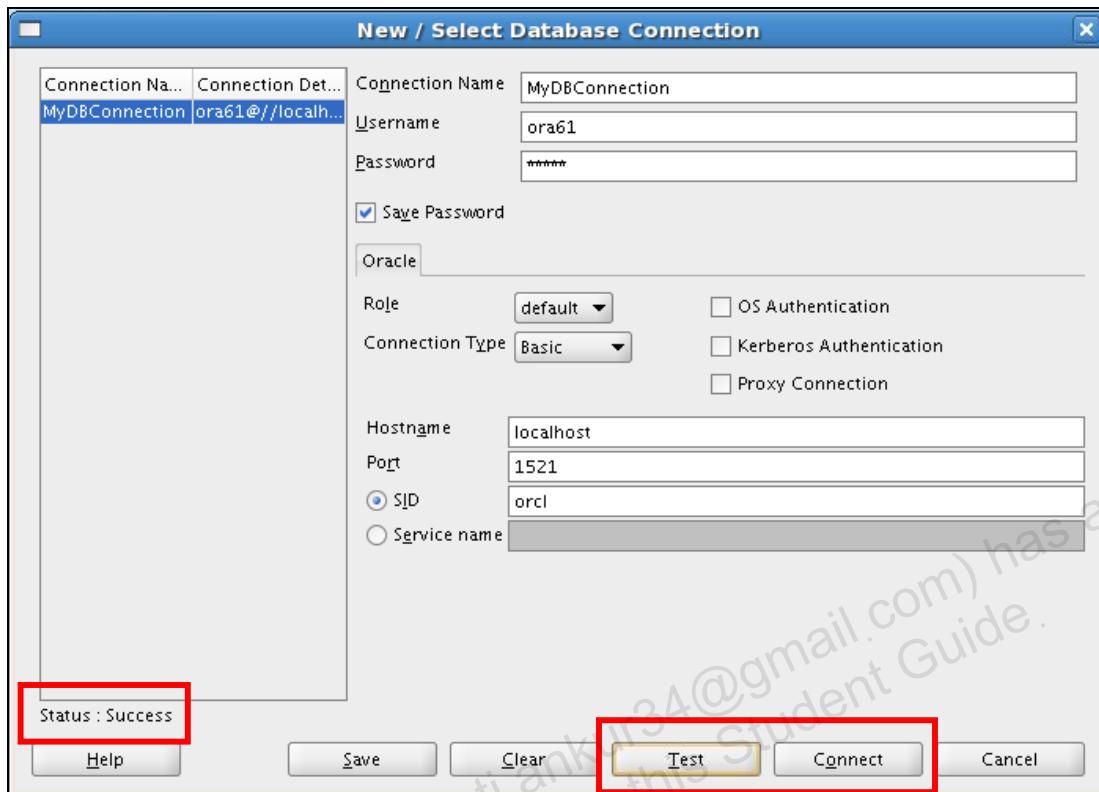
Right-click the Connections icon on the Connections tabbed page, and then select the New Connection option from the shortcut menu. The New>Select Database Connection window is displayed. Use the preceding information provided to create the new database connection.

Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Substitute the username, password, host name, and service name with the appropriate information as provided by your instructor. The following is a sample of the newly created database connection for student ora61:



- 3) Test the new connection. If the Status is Success, connect to the database using this new connection:
 - a) Double-click the MyDBConnection 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 I-2: Creating and Using a New SQL Developer Database Connection (continued)



Practice Solutions I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block

In this practice, you browse your schema tables and create and execute a simple anonymous block.

- 1) Browse the structure of the EMPLOYEES table and display its data.
 - a) Expand the MyDBConnection connection by clicking the plus sign next to it.
 - b) Expand the Tables icon by clicking the plus sign next to it.
 - c) Display the structure of the EMPLOYEES table.

Double-click the EMPLOYEES table. The Columns tab displays the columns in the EMPLOYEES table as follows:

The screenshot shows the Oracle SQL Developer interface. On the left, the Connections tree shows a connection named "MyDBConnection" expanded, revealing tables like COUNTRIES, DEPARTMENTS, and EMPLOYEES. The EMPLOYEES table is selected. The main workspace displays the "EMPLOYEES" table structure in the "Columns" tab. The columns are listed as follows:

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	Comments
EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	1	Primary key of employees table.
FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2		(null) First name of the employee. A not null column.
LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3		(null) Last name of the employee. A not null column.
EMAIL	VARCHAR2(25 BYTE)	No	(null)	4		(null) Email id of the employee
PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5		(null) Phone number of the employee; includes country code and area code.
HIRE_DATE	DATE	No	(null)	6		(null) Date when the employee started on this job. A not null column.
JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7		(null) Current job of the employee; foreign key to job_id column.
SALARY	NUMBER(8,2)	Yes	(null)	8		(null) Monthly salary of the employee. Must be greaterthan zero.
COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9		(null) Commission percentage of the employee. Only employees with commission_pct > 0 have non-zero values.
MANAGER_ID	NUMBER(6,0)	Yes	(null)	10		(null) Manager id of the employee; has same domain as manager_id column.
DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11		(null) Department id where employee works; foreign key to department_id column.

- 2) Browse the EMPLOYEES table and display its data.

To display the employees' data, click the Data tab. The EMPLOYEES table data is displayed as follows:

Practice Solutions I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block (continued)

The screenshot shows the Oracle SQL Developer interface with the title bar "Oracle SQL Developer : TABLE ORA61.EMPLOYEES@MyDBConnection". The left sidebar shows the connection "MyDBConnection" and its tables, including COUNTRIES, DEPARTMENTS, and EMPLOYEES. The EMPLOYEES table is selected, displaying 107 rows of employee data. The status bar at the bottom left indicates "All Rows Fetched: 107".

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
1	198 Donald	OConnell	DOCONNEL...	650.507.9833	21-JUN-99	SH_CLERK	2600	
2	199 Douglas	Grant	DGRANT...	650.507.9844	13-JAN-00	SH_CLERK	2600	
3	200 Jennifer	Whalen	MWHALEN...	515.123.4444	17-SEP-87	AD_ASST	4400	
4	201 Michael	Hartstein	MHARTSTEIN...	515.123.5555	17-FEB-96	MK_MAN	13000	
5	202 Pat	Fay	PFAY...	603.123.6666	17-AUG-97	MK_REP	6000	
6	203 Susan	Mavris	SMAVRIS...	515.123.7777	07-JUN-94	HR_REP	6500	
7	204 Hermann	Baer	HBAER...	515.123.8888	07-JUN-94	PR_REP	10000	
8	205 Shelley	Higgins	SHIGGINS...	515.123.8080	07-JUN-94	AC_MGR	12000	
9	206 William	Gietz	WGIETZ...	515.123.8181	07-JUN-94	AC_ACC...	8300	
10	100 Steven	King	SKING...	515.123.4567	17-JUN-87	AD_PRES	24000	
11	101 Neena	Kochhar	NKOCHHAR...	515.123.4568	21-SEP-89	AD_VP	17000	
12	102 Lex	De Haan	LDEHAAN...	515.123.4569	13-JAN-93	AD_VP	17000	
13	103 Alexander	Hunold	AHUNOLD...	590.423.4567	03-JAN-90	IT_PROG	9000	
14	104 Bruce	Ernst	BERNSTEIN...	590.423.4568	21-MAY-91	IT_PROG	6000	
15	105 David	Austin	DAUSTIN...	590.423.4569	25-JUN-97	IT_PROG	4800	
16	106 Valli	Pataballa	VPATABALLA...	590.423.4560	05-FEB-99	IT_PROG	4800	
17	107 Diana	Lorentz	DLORENTZ...	590.423.5567	07-FEB-99	IT_PROG	4200	
18	108 Nancy	Greenberg	NGREENBERG...	515.124.4569	17-AUG-94	FI_MGR	12000	
19	109 Daniel	Faviet	DFAVIET...	515.124.4169	16-AUG-94	FI_ACCOUNT...	9000	
20	110 John	Chen	JCHEN...	515.124.4269	28-SEP-97	FI_ACCOUNT...	8200	
21	111 Ismael	Scierra	ISCIRRA...	515.124.4369	30-SEP-97	FI_ACCOUNT...	7700	
22	112 Jose Manuel	Urman	JMURMAN...	515.124.4469	07-MAR-98	FI_ACCOUNT...	7800	
23	113 Luis	Popp	LPOPP...	515.124.4567	07-DEC-99	FI_ACCOUNT...	6900	
24	114 Den	Raphaely	DRAPHELY...	515.127.4561	07-DEC-94	PU_MAN	11000	
25	115 Alexander	Khoo	AKHOO...	515.127.4562	18-MAY-95	PU_CLERK	3100	
26	116 Shelli	Baida	SBAIDA...	515.127.4563	24-DEC-97	PU_CLERK	2900	

- 3) 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 icon (F5) icons to execute the SELECT statement. Review the results of both methods of executing the SELECT statements in the appropriate tabs.

Note: Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides the description and data for all the tables in the HR schema that you will use in this course.

Display the SQL Worksheet using one of the following two methods:

1. Select Tools > SQL Worksheet or click the Open SQL Worksheet icon. The Select Connection window is displayed.
2. Select the new MyDBConnection from the Connection drop-down list (if not already selected), and then click OK.

Open the sol_I_03.sql file in the /home/oracle/labs/plpu/solns folder as follows using one of the following two methods:

1. In the Files tab, select (or navigate to) the script file that you want to open.
2. Double-click the file name to open. The code of the script file is displayed in the SQL Worksheet area.

Practice Solutions I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block (continued)

3. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar.

Alternatively you can also:

1. Select Open from the File menu. 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. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar.

To run a single SELECT statement, click the Execute Statement (F9) icon (while making sure the cursor is on any of the SELECT statement lines) on the SQL Worksheet toolbar to execute the statement. The code and the result are displayed as follows:

```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE SALARY > 10000;
```

The screenshot shows the Oracle SQL Worksheet interface. At the top, there are tabs: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the tabs is a toolbar with icons for New, Open, Save, and Print. The main area displays a table with two columns: LAST_NAME and SALARY. The data consists of 15 rows, each showing a last name and its corresponding salary. The table has a dashed horizontal line separating the header from the data. At the bottom of the table, it says "15 rows selected".

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Greenberg	12000
Raphaely	11000
Russell	14000
Partners	13500
Errazuriz	12000
Cambrault	11000
Zlotkey	10500
Vishney	10500
Ozer	11500
Abel	11000
Hartstein	13000
Higgins	12000

15 rows selected

- 4) 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.

Enter the following command in the SQL Worksheet area, and then click the Run Script (F5) icon.

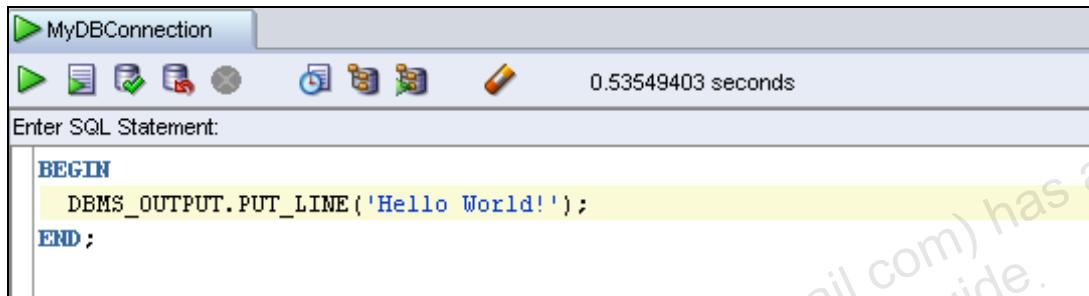
Practice Solutions I-3: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block (continued)

```
SET SERVEROUTPUT ON
```

- b) Use the SQL Worksheet area to enter the code for your anonymous block.

Enter the following code in the SQL Worksheet area as shown below.

Alternatively, open the `sol_I_04.sql` file in the
`/home/oracle/labs/plpu/solns` folder. The code is displayed as follows:



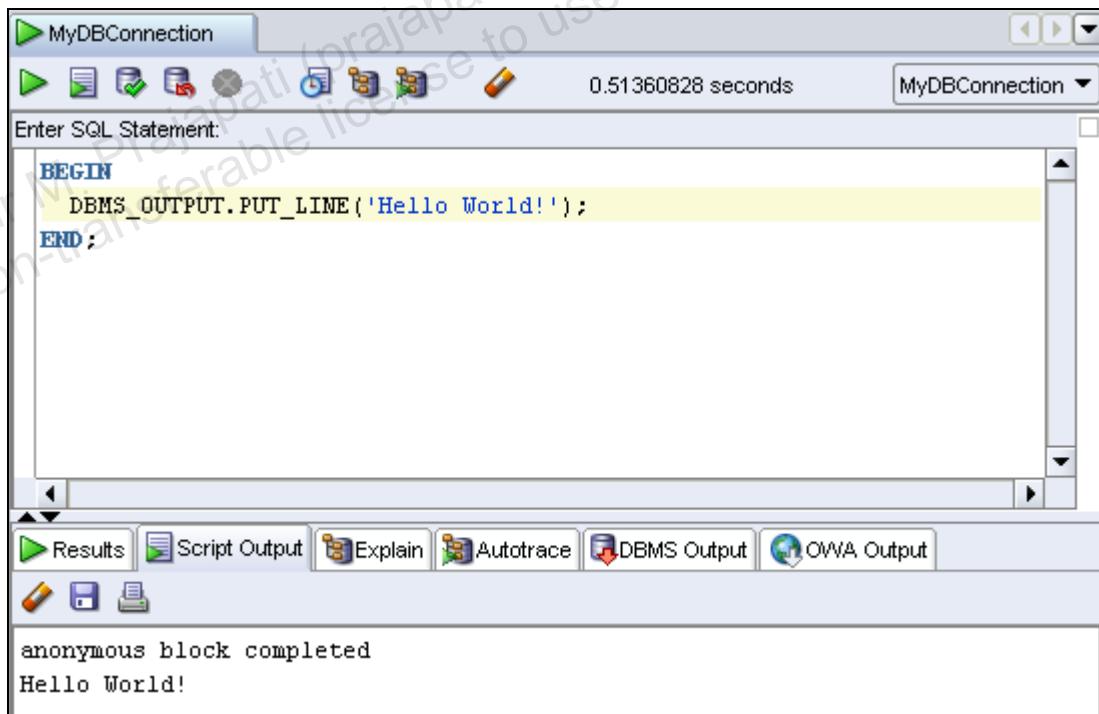
The screenshot shows the Oracle SQL Developer interface with a connection named "MyDBConnection". In the "SQL Worksheet" tab, there is a code editor window containing the following PL/SQL code:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
```

The code is highlighted in yellow. The status bar at the top right shows "0.53549403 seconds".

- c) Click the Run Script (F5) icon to run the anonymous block.

The Script Output tab displays the output of the anonymous block as follows:



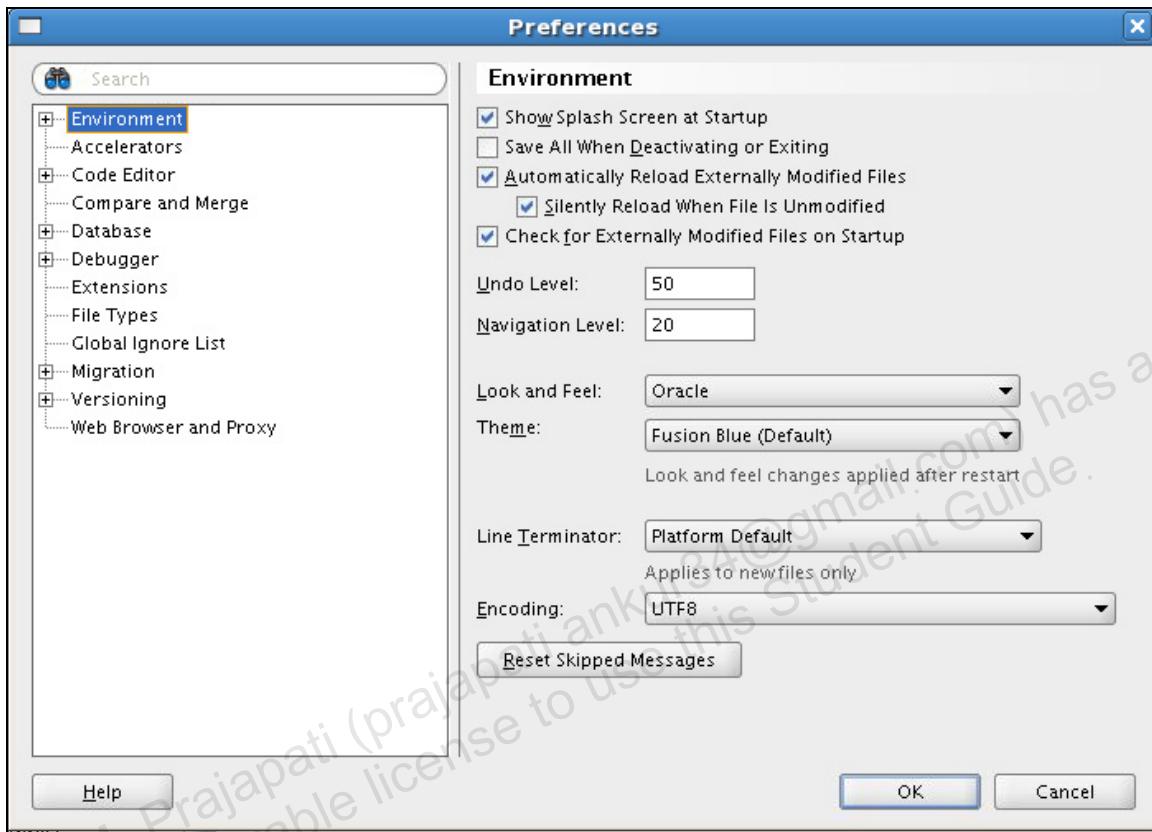
The screenshot shows the Oracle SQL Developer interface with a connection named "MyDBConnection". In the "Script Output" tab, there is a results window containing the following output:

```
anonymous block completed
Hello World!
```

Practice Solutions I-4: Setting Some SQL Developer Preferences

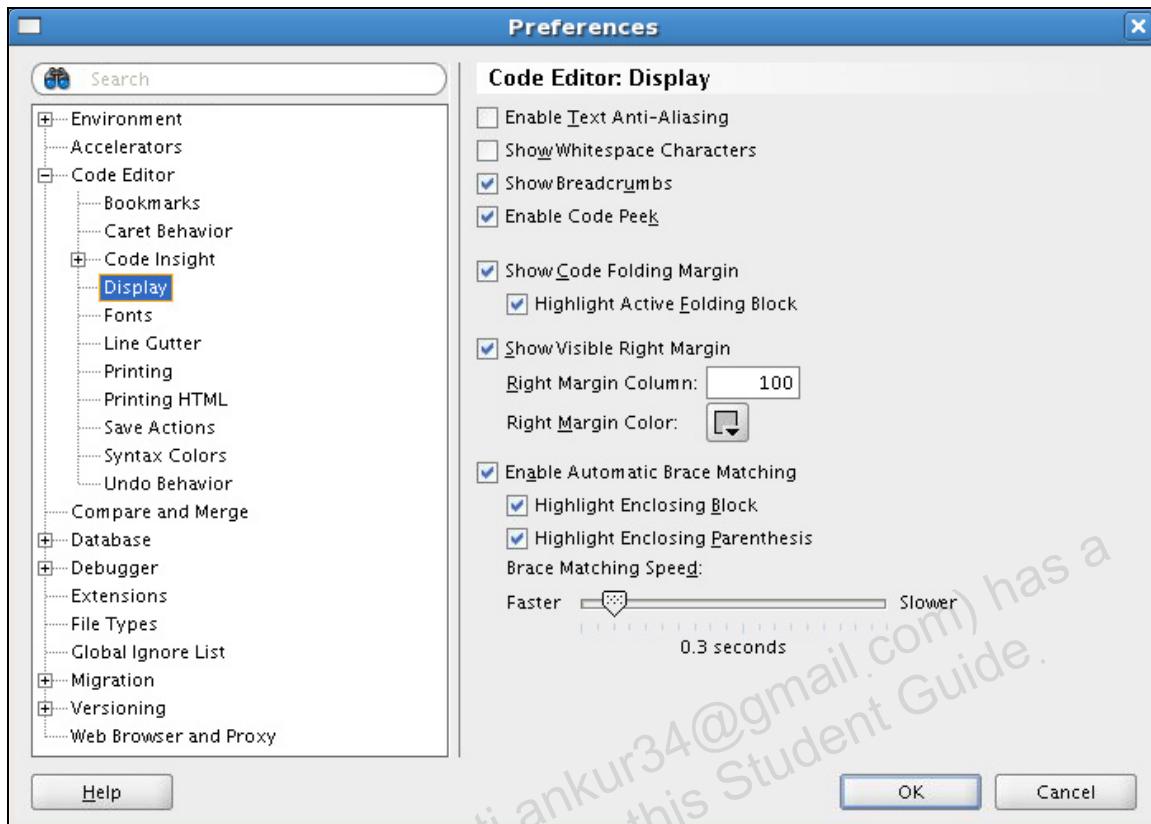
In this practice, you set some SQL Developer preferences.

- 1) In the SQL Developer menu, navigate to Tools > Preferences. The Preferences window is displayed.



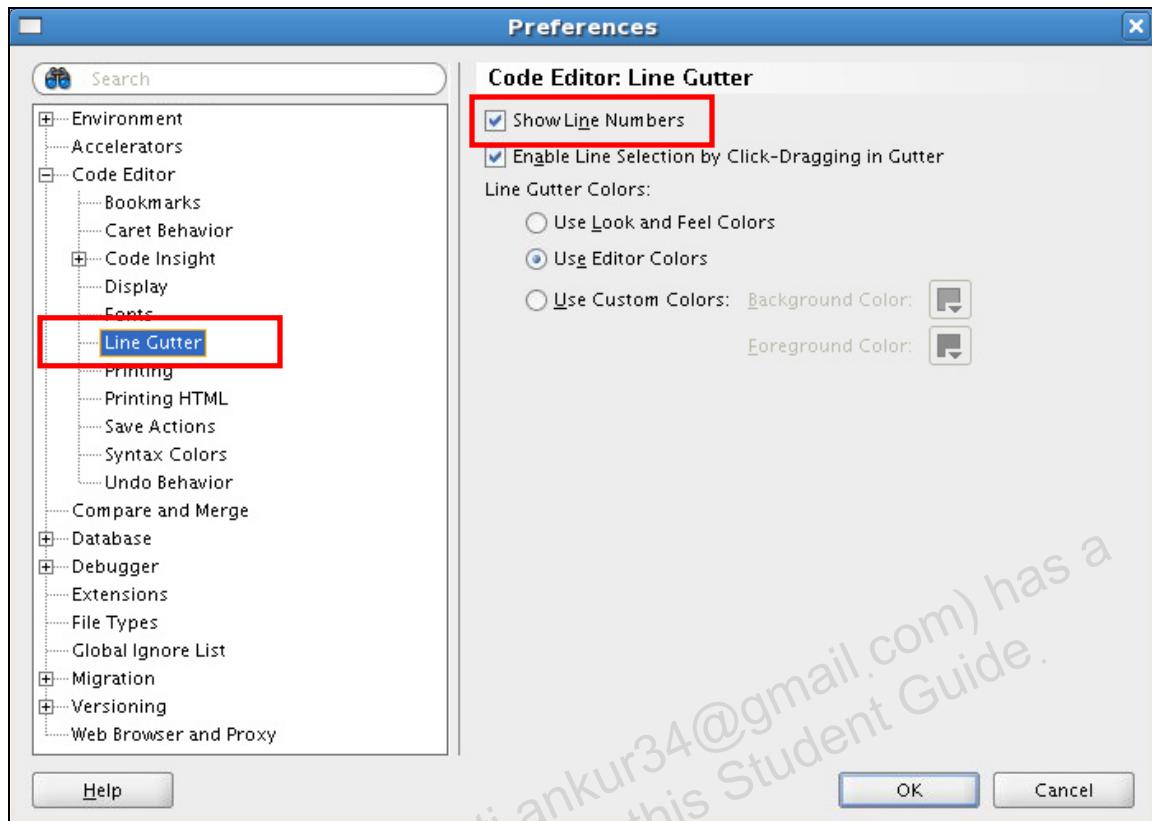
- 2) Expand the Code Editor option, and then click the Display option to display the "Code Editor: Display" section. The "Code Editor: Display" section contains general options for the appearance and behavior of the code editor.
 - a) Enter 100 in the Right Margin Column text box in the Show Visible Right Margin section. This renders a right margin that you can set to control the length of lines of code.

Practice Solutions I-4: Setting Some SQL Developer Preferences (continued)



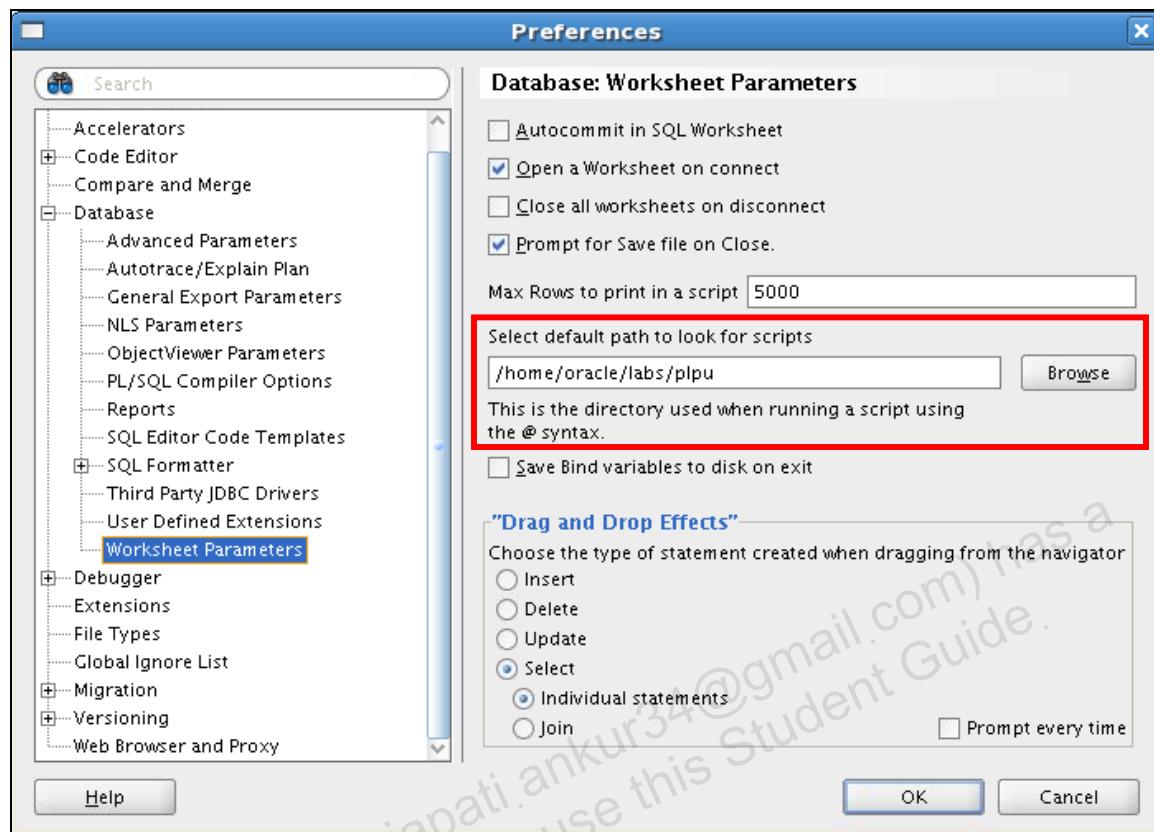
- b) Click the Line Gutter option. The Line Gutter option specifies options for the line gutter (left margin of the code editor). Select the Show Line Numbers check box to display the code line numbers.

Practice Solutions I-4: Setting Some SQL Developer Preferences (continued)



- 3) Click the Worksheet Parameters option under the Database option. In the “Select default path to look for scripts” text box, specify the /home/oracle/labs/plpu folder. This folder contains the solutions scripts, code examples scripts, and any labs or demos used in this course.

Practice Solutions I-4: Setting Some SQL Developer Preferences (continued)



- 4) Click OK to accept your changes and to exit the Preferences window.
- 5) Familiarize yourself with the labs folder on the /home/oracle/labs/plpu folder.
 - a) Click the **Files** tab (next to the **Connections** tab).
 - b) Navigate to the /home/oracle/labs/plpu folder.
 - c) How many subfolders do you see in the labs folder?
 - d) Navigate through the folders, and open a script file without executing the code.
 - e) Clear the displayed code in the SQL Worksheet area. In the SQL Developer menu, navigate to Tools > Preferences.

Practice Solutions I-5: Accessing the Oracle Database 11g Release 2 Online Documentation Library

In this practice, you access and bookmark some of the Oracle Database 11g Release 2 documentation references that you will use in this course.

- 1) Access the Oracle Database 11g Release 2 documentation Web page at:
<http://www.oracle.com/pls/db111/homepage>
- 2) Bookmark the page for easier future access.
- 3) Display the complete list of books available for Oracle Database 11g Release 2.
- 4) Make a note of the following documentation references that you will use in this course as needed:
 - a) *Advanced Application Developer's Guide*
 - b) *New Features Guide*
 - c) *PL/SQL Language Reference*
 - d) *Oracle Database Reference*
 - e) *Oracle Database Concepts*
 - f) *SQL Developer User's Guide*
 - g) *SQL Language Reference Guide*
 - h) *SQL*Plus User's Guide and Reference*

Practices and Solutions for Lesson 1

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

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 1-1: Creating, Compiling, and Calling Procedures

In this practice, you create and invoke the ADD_JOB procedure and review the results. You also create and invoke a procedure called UPD_JOB to modify a job in the JOBS table and create and invoke a procedure called DEL_JOB to delete a job from the JOBS table. Finally, you create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.

- 1) Create, compile, and invoke the ADD_JOB procedure and review the results.

- a) Create a procedure called ADD_JOB to insert a new job into the JOBS table. Provide the ID and job title using two parameters.

Note: You can create the procedure (and other objects) by entering the code in the SQL Worksheet area, and then click the Run Script (F5) icon. This creates and compiles the procedure. To find out whether or not the procedure has any errors, click the procedure name in the procedure node, and then select Compile from the pop-up menu.

- b) Invoke the procedure with IT_DBA as the job ID and Database Administrator as the job title. Query the JOBS table and view the results.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
<hr/>			
IT_DBA	Database Administrator		
1 rows selected			

- c) Invoke your procedure again, passing a job ID of ST_MAN and a job title of Stock Manager. What happens and why?
 - 2) Create a procedure called UPD_JOB to modify a job in the JOBS table.
- a) Create a procedure called UPD_JOB to update the job title. Provide the job ID and a new title using two parameters. Include the necessary exception handling if no update occurs.
 - b) Invoke the procedure to change the job title of the job ID IT_DBA to Data Administrator. Query the JOBS table and view the results.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
<hr/>			
IT_DBA	Data Administrator		
1 rows selected			

- c) Test the exception-handling section of the procedure by trying to update a job that does not exist. You can use the job ID IT_WEB and the job title Web Master.

```
Error starting at line 1 in command:
EXECUTE upd_job ('IT_WEB', 'Web Master')
Error report:
ORA-20202: No job updated.
ORA-06512: at "ORA80.UPD_JOB", line 9
ORA-06512: at line 1
```

Practice 1-1: Creating, Compiling, and Calling Procedures (continued)

- 3) Create a procedure called DEL_JOB to delete a job from the JOBS table.
- Create a procedure called DEL_JOB to delete a job. Include the necessary exception-handling code if no job is deleted.
 - Invoke the procedure using the job ID IT_DBA. Query the JOBS table and view the results.
- | JOB_ID | JOB_TITLE | MIN_SALARY | MAX_SALARY |
|-----------------|-----------|------------|------------|
| ----- | | | |
| 0 rows selected | | | |
- Test the exception-handling section of the procedure by trying to delete a job that does not exist. Use IT_WEB as the job ID. You should get the message that you included in the exception-handling section of the procedure as the output.

```
Error starting at line 1 in command:
EXECUTE del_job ('IT_WEB')
Error report:
ORA-20203: No jobs deleted.
ORA-06512: at "ORA80.DEL_JOB", line 6
ORA-06512: at line 1
```

- 4) Create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.
- Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Remove syntax errors, if any, and then recompile the code.
 - Execute the procedure using host variables for the two OUT parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

v_salary

8000
v_job

ST_MAN

- Invoke the procedure again, passing an EMPLOYEE_ID of 300. What happens and why?

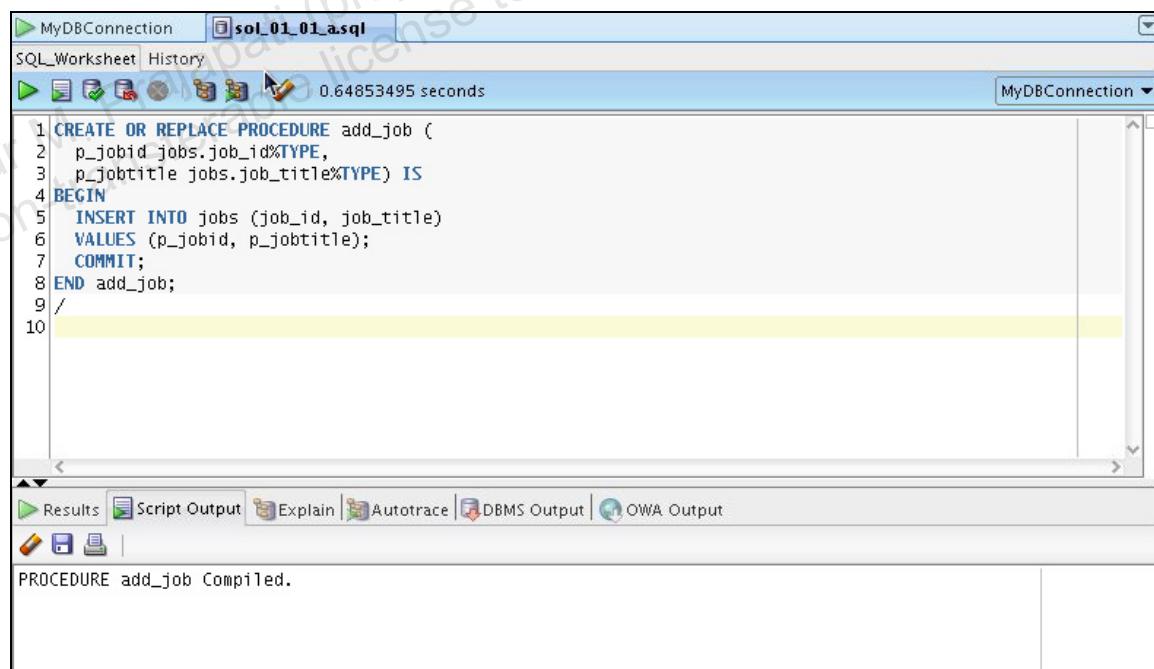
Practice Solutions 1-1: Creating, Compiling, and Calling Procedures

In this practice, you create and invoke the ADD_JOB procedure and review the results. You also create and invoke a procedure called UPD_JOB to modify a job in the JOBS table and create and invoke a procedure called DEL_JOB to delete a job from the JOBS table. Finally, you create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.

- 1) Create, compile, and invoke the ADD_JOB procedure and review the results.
 - a) Create a procedure called ADD_JOB to insert a new job into the JOBS table. Provide the ID and job title using two parameters.

Note: You can create the procedure (and other objects) by entering the code in the SQL Worksheet area, and then click the Run Script (F5) icon. This creates and compiles the procedure. If the procedure generates an error message when you create it, click the procedure name in the procedure node, edit the procedure, and then select Compile from the pop-up menu.

Open the sol_01_01_a.sql file in the /home/oracle/labs/plpu/solns folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:



The screenshot shows the Oracle SQL Worksheet interface. The top window title is "MyDBConnection" and the tab is "sol_01_01_a.sql". The code in the worksheet is:

```

1 CREATE OR REPLACE PROCEDURE add_job (
2   p_jobid jobs.job_id%TYPE,
3   p_jobtitle jobs.job_title%TYPE) IS
4 BEGIN
5   INSERT INTO jobs (job_id, job_title)
6   VALUES (p_jobid, p_jobtitle);
7   COMMIT;
8 END add_job;
9 /
10

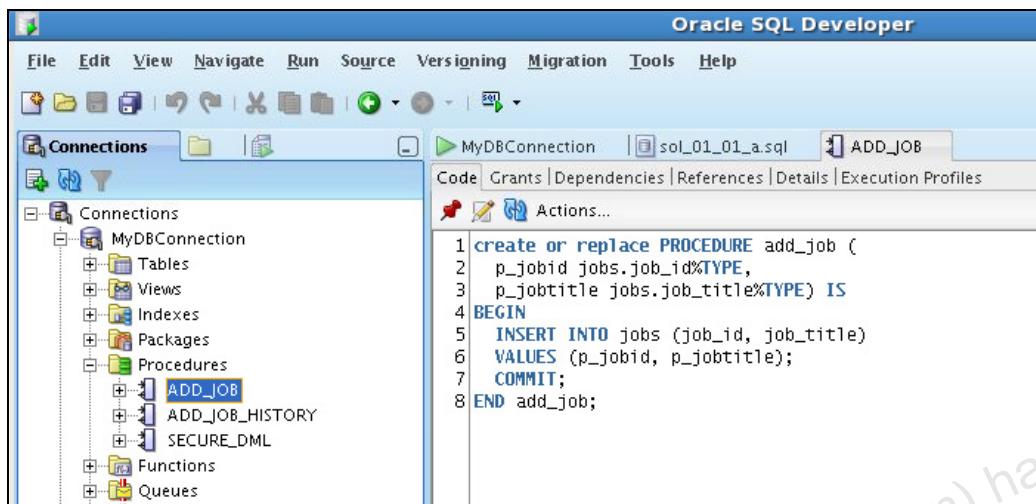
```

The status bar at the bottom indicates "0.64853495 seconds". Below the worksheet, the "Results" tab is selected, showing the message "PROCEDURE add_job Compiled."

To view the newly created procedure, click the Procedures node in the Object Navigator. If the newly created procedure is not displayed, right-click

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

the Procedures node, and then select Refresh from the shortcut menu. The new procedure is displayed as follows:



- b) Invoke the procedure with IT_DBA as the job ID and Database Administrator as the job title. Query the JOBS table and view the results.

Run the /home/oracle/labs/plpu/soln/sol_01_01_b.sql script.

The code and the result are displayed as follows:

The screenshot shows the Oracle SQL Developer SQL Worksheet window with the file sol_01_01_b.sql open. The code executed is:

```

1 EXECUTE add_job ('IT_DBA', 'Database Administrator')
2 SELECT * FROM jobs WHERE job_id = 'IT_DBA';
3
4
5
6

```

The Results tab shows the output:

```

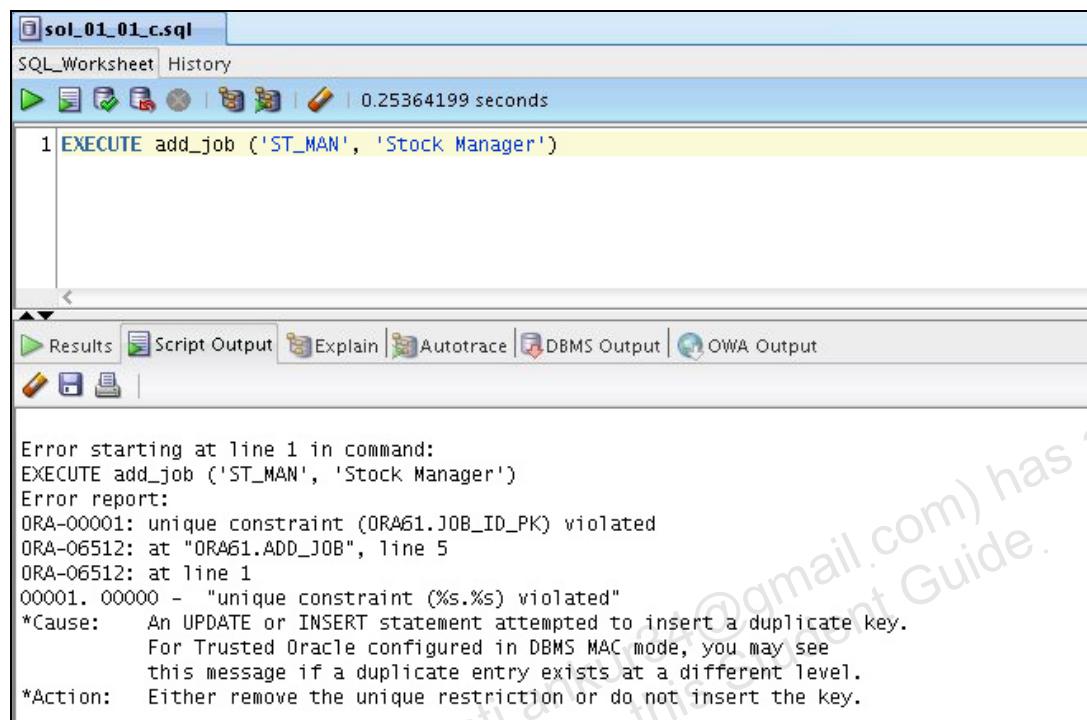
anonymous block completed
JOB_ID      JOB_TITLE              MIN_SALARY      MAX_SALARY
IT_DBA      Database Administrator
1 rows selected

```

- c) Invoke your procedure again, passing a job ID of ST_MAN and a job title of Stock Manager. What happens and why?

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

An exception occurs because there is a Unique key integrity constraint on the JOB_ID column.



The screenshot shows a SQL Worksheet window titled "sol_01_01_c.sql". The code entered is:

```
1 EXECUTE add_job ('ST_MAN', 'Stock Manager')
```

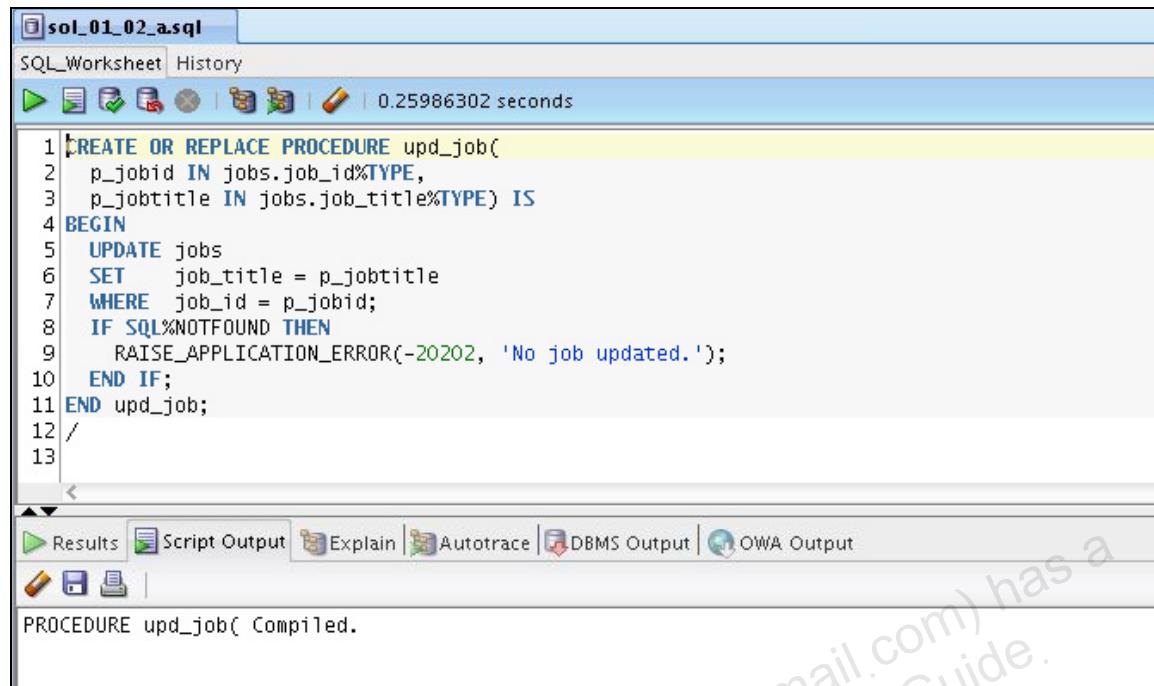
The results pane displays the following error message:

```
Error starting at line 1 in command:  
EXECUTE add_job ('ST_MAN', 'Stock Manager')  
Error report:  
ORA-00001: unique constraint (ORA61.JOB_ID_PK) violated  
ORA-06512: at "ORA61.ADD_JOB", line 5  
ORA-06512: at line 1  
00001. 00000 - "unique constraint (%s.%s) violated"  
*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.  
For Trusted Oracle configured in DBMS MAC mode, you may see  
this message if a duplicate entry exists at a different level.  
*Action: Either remove the unique restriction or do not insert the key.
```

- 2) Create a procedure called UPD_JOB to modify a job in the JOBS table.
 - a) Create a procedure called UPD_JOB to update the job title. Provide the job ID and a new title using two parameters. Include the necessary exception handling if no update occurs.

Run the /home/oracle/labs/plpu/soln/sol_01_02_a.sql script.
The code and the result are displayed as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)



The screenshot shows the SQL Worksheet interface with the following details:

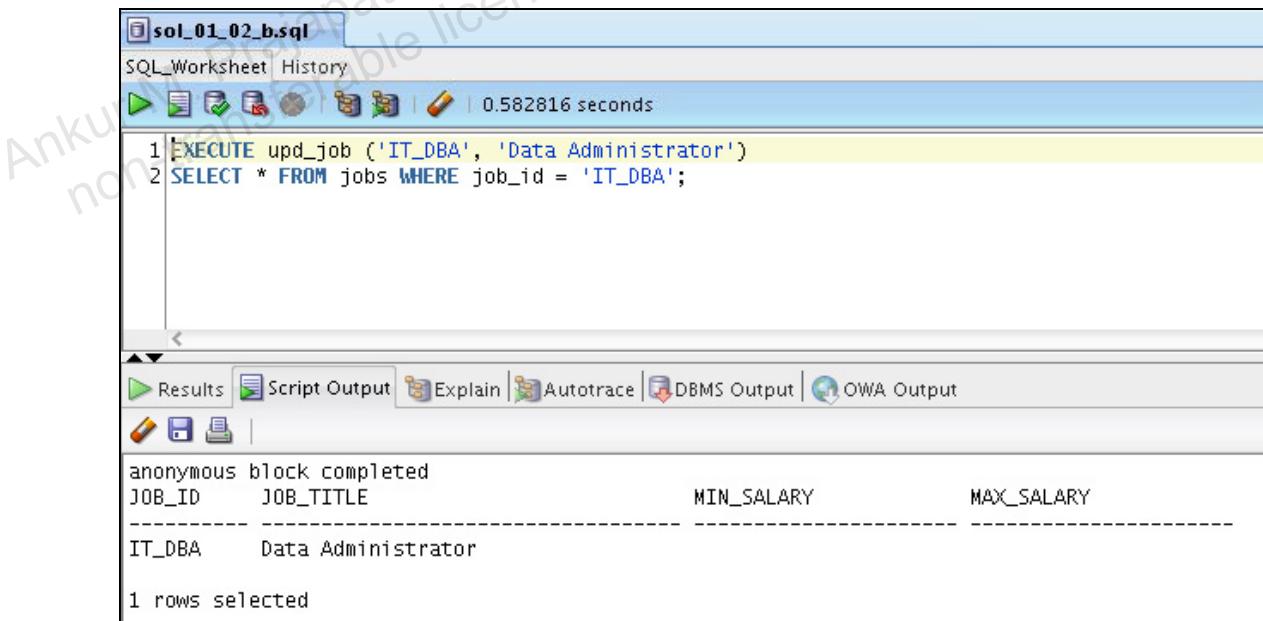
- Title Bar:** sol_01_02_a.sql
- Toolbar:** Includes icons for Run, Save, Undo, Redo, Copy, Paste, and others. A status bar at the bottom indicates "0.25986302 seconds".
- Code Area:**

```

1 CREATE OR REPLACE PROCEDURE upd_job(
2   p_jobid IN jobs.job_id%TYPE,
3   p_jobtitle IN jobs.job_title%TYPE) IS
4 BEGIN
5   UPDATE jobs
6     SET job_title = p_jobtitle
7   WHERE job_id = p_jobid;
8   IF SQL%NOTFOUND THEN
9     RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
10  END IF;
11 END upd_job;
12 /
13 
```
- Output Area:**
 - Results tab is selected.
 - Script Output, Explain, Autotrace, DBMS Output, and OWA Output tabs are also visible.
 - Message: "PROCEDURE upd_job(Compiled.)"

- b) Invoke the procedure to change the job title of the job ID IT_DBA to Data Administrator. Query the JOBS table and view the results.

**Run the /home/oracle/labs/plpu/soln/sol_01_02_b.sql script.
The code and the result are displayed as follows:**



The screenshot shows the SQL Worksheet interface with the following details:

- Title Bar:** sol_01_02_b.sql
- Toolbar:** Includes icons for Run, Save, Undo, Redo, Copy, Paste, and others. A status bar at the bottom indicates "0.582816 seconds".
- Code Area:**

```

1 EXECUTE upd_job ('IT_DBA', 'Data Administrator')
2 SELECT * FROM jobs WHERE job_id = 'IT_DBA';

```
- Output Area:**
 - Results tab is selected.
 - Script Output, Explain, Autotrace, DBMS Output, and OWA Output tabs are also visible.
 - Message: "anonymous block completed"
 - Query output for JOBS table:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		
 - Message: "1 rows selected"

- c) Test the exception-handling section of the procedure by trying to update a job that does not exist. You can use the job ID IT_WEB and the job title Web Master.

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

```

sol_01_02_C.sql
SQL Worksheet History
0.79948604 seconds

1 EXECUTE upd_job ('IT_WEB', 'Web Master')
2 SELECT * FROM jobs WHERE job_id = 'IT_WEB';

Error starting at line 1 in command:
EXECUTE upd_job ('IT_WEB', 'Web Master')
Error report:
ORA-20202: No job updated.
ORA-06512: at "ORA61.UPD_JOB", line 9
ORA-06512: at line 1

JOB_ID      JOB_TITLE          MIN_SALARY      MAX_SALARY
-----      -----
0 rows selected

```

- 3) Create a procedure called DEL_JOB to delete a job from the JOBS table.
- Create a procedure called DEL_JOB to delete a job. Include the necessary exception-handling code if no job is deleted.
- Run the /home/oracle/labs/plpu/soln/sol_01_03_a.sql script.
The code and the result are displayed as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "sol_01_03_a.sql". The main area contains the following PL/SQL code:

```
1 CREATE OR REPLACE PROCEDURE del_job (p_jobid jobs.job_id%TYPE) IS
2 BEGIN
3   DELETE FROM jobs
4   WHERE job_id = p_jobid;
5   IF SQL%NOTFOUND THEN
6     RAISE_APPLICATION_ERROR(-20203, 'No jobs deleted.');
7   END IF;
8 END del_job;
9 /
```

Below the code, the status bar shows "0.20295501 seconds". At the bottom, there are tabs for "Results", "Script Output", "Explain", "Autotrace", "DBMS Output", and "OWA Output". The "Script Output" tab is selected. It displays the message "PROCEDURE del_job Compiled."

- b) To invoke the procedure and then query the JOBS table, load the `sol_01_03_b.sql` file in the `/home/oracle/labs/plpu/solns` folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "sol_01_03_b.sql". The toolbar includes icons for Run Script (F5), Save, Print, and others. The status bar shows "0.212038 seconds". The code area contains two lines of PL/SQL:

```
1 EXECUTE del_job ('IT_DBA')
2 SELECT * FROM jobs WHERE job_id = 'IT_DBA';
```

The results pane shows the output of the query:

anonymous block completed	JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
0 rows selected				

- c) Test the exception-handling section of the procedure by trying to delete a job that does not exist. Use IT_WEB as the job ID. You should get the message that you included in the exception-handling section of the procedure as the output.

To invoke the procedure and then query the JOBS table, load the **sol_01_03_c.sql** file from the **/home/oracle/labs/plpu/solns** folder. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

The screenshot shows an Oracle SQL Worksheet window titled "sol_01_03_c.sql". The main pane contains the command: "1 EXECUTE del_job ('IT_WEB')". Below the command, an error message is displayed:

```
Error starting at line 1 in command:
EXECUTE del_job ('IT_WEB')
Error report:
ORA-20203: No jobs deleted.
ORA-06512: at "ORA61.DEL_JOB", line 6
ORA-06512: at line 1
```

The window also includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output, along with standard toolbar icons.

- 4) Create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.
 - a) Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Remove syntax errors, if any, and then recompile the code.

Open the /home/oracle/labs/plpu/solns/sol_01_04_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "sol_01_04_a.sql". The main area contains the following PL/SQL code:

```

1 CREATE OR REPLACE PROCEDURE get_employee
2   (p_empid IN employees.employee_id%TYPE,
3    p_sal    OUT employees.salary%TYPE,
4    p_job    OUT employees.job_id%TYPE) IS
5 BEGIN
6   SELECT salary, job_id
7   INTO   p_sal, p_job
8   FROM   employees
9   WHERE  employee_id = p_empid;
10 END get_employee;
11 /
12

```

Below the code, a message says "PROCEDURE get_employee Compiled." The bottom navigation bar includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output.

Note

If the newly created procedure is not displayed in the Object Navigator, right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the procedure's name in the Object Navigator, and then select Compile from the shortcut menu. The procedure is compiled.

The screenshot shows the Oracle SQL Worksheet interface with the "Messages - Log" tab selected. It displays the message "GET_EMPLOYEE Compiled".

- b) Execute the procedure using host variables for the two OUT parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

Open the /home/oracle/labs/plpu/solns/sol_01_04_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "sol_01_04_b.sql". The main area contains the following PL/SQL code:

```
1 VARIABLE v_salary NUMBER
2 VARIABLE v_job    VARCHAR2(15)
3 EXECUTE get_employee(120, :v_salary, :v_job)
4 PRINT v_salary v_job
5
```

Below the code, the results pane shows the output of the anonymous block:

```
anonymous block completed
v_salary
-----
8000
v_job
-----
ST_MAN
```

The results tab is selected at the bottom.

- c) Invoke the procedure again, passing an EMPLOYEE_ID of 300. What happens and why?

There is no employee in the EMPLOYEES table with an EMPLOYEE_ID of 300. The SELECT statement retrieved no data from the database, resulting in a fatal PL/SQL error: NO_DATA_FOUND as follows:

Practice Solutions 1-1: Creating, Compiling, and Calling Procedures (continued)

The screenshot shows the Oracle SQL Worksheet interface. The top menu bar includes 'SQL Worksheet' and 'History'. A toolbar with various icons is visible above the code editor. The connection is set to 'MyDBConnection'. The code in the editor is:

```
1 VARIABLE v_salary NUMBER
2 VARIABLE v_job    VARCHAR2(15)
3 EXECUTE get_employee(300, :v_salary, :v_job)
```

The results pane shows the output of the anonymous block:

```
anonymous block completed
v_salary
-----
8000

v_job
-----
ST_MAN

Error starting at line 1 in command:
EXECUTE get_employee(300, :v_salary, :v_job)
Error report:
ORA-01403: no data found
ORA-06512: at "ORA61.GET_EMPLOYEE", line 6
ORA-06512: at line 1
01403. 00000 -  "no data found"
*Cause:
*Action:
```

A large watermark in the background reads: Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 2

In practice 2-1, you create, compile, and use the following:

- A function called GET_JOB to return a job title.
- A function called GET_ANNUAL_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.
- A procedure called ADD_EMPLOYEE to insert a new employee into the EMPLOYEES table.

In practice 2-2, you are introduced to the basic functionality of the SQL Developer debugger:

- Create a procedure and a function.
- Insert breakpoints in the newly created procedure.
- Compile the procedure and function for debug mode.
- Debug the procedure and step into the code.
- Display and modify the subprograms' variables.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 2-1: Creating Functions

In this practice, you create, compile, and use stored functions and a procedure.

- 1) Create and invoke the GET_JOB function to return a job title.
 - a) Create and compile a function called GET_JOB to return a job title.
 - b) Create a VARCHAR2 host variable called b_title, allowing a length of 35 characters. Invoke the function with job ID SA REP to return the value in the host variable, and then print the host variable to view the result.

```

Results Script Output Explain Autotrace DBMS Output OWA Output
anonymous block completed
b_title
-----
Sales Representative

```

- 2) Create a function called GET_ANNUAL_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.
 - a) Create the GET_ANNUAL_COMP function, which accepts parameter values for the monthly salary and commission. Either or both values passed can be NULL, but the function should still return a non-NUL annual salary. Use the following basic formula to calculate the annual salary:

$$(\text{salary} * 12) + (\text{commission_pct} * \text{salary} * 12)$$
 - b) Use the function in a SELECT statement against the EMPLOYEES table for employees in department 30.

EMPLOYEE_ID	LAST_NAME	Annual Compensation
114	Raphaely	132000
115	Khoo	37200
116	Baida	34800
117	Tobias	33600
118	Himuro	31200
119	Colmenares	30000

6 rows selected

- 3) Create a procedure, ADD_EMPLOYEE, to insert a new employee into the EMPLOYEES table. The procedure should call a VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.

- a) Create a function called VALID_DEPTID to validate a specified department ID and return a BOOLEAN value of TRUE if the department exists.
- b) Create the ADD_EMPLOYEE procedure to add an employee to the EMPLOYEES table. The row should be added to the EMPLOYEES table if the VALID_DEPTID function returns TRUE; otherwise, alert the user with an appropriate message. Provide the following parameters:
- first_name
 - last_name
 - email
 - job: Use 'SA REP' as the default.
 - mgr: Use 145 as the default.
 - sal: Use 1000 as the default.
 - comm: Use 0 as the default.
 - deptid: Use 30 as the default.
 - Use the EMPLOYEES_SEQ sequence to set the employee_id column.
 - Set the hire_date column to TRUNC(SYSDATE).
- c) Call ADD_EMPLOYEE for the name 'Jane Harris' in department 15, leaving other parameters with their default values. What is the result?
- d) Add another employee named Joe Harris in department 80, leaving the remaining parameters with their default values. What is the result?

Practice 2-2: Introduction to the SQL Developer Debugger

In this practice, you experiment with the basic functionality of the SQL Developer debugger.

- 1) Enable SERVEROUTPUT.
- 2) Run the `sol_02_02_02.sql` script to create the `emp_list` procedure. Examine the code of the procedure and compile the procedure. Why do you get the compiler error?
- 3) Run the `sol_02_02_03.sql` script to create the `get_location` function. Examine the code of the function, compile the function, and then correct any errors, if any.
- 4) Re-compile the `emp_list` procedure. The procedure should compile successfully.
- 5) Edit the `emp_list` procedure and the `get_location` function.
- 6) Add four breakpoints to the `emp_list` procedure to the following lines of code:
 - a) `OPEN emp_cursor;`
 - b) `WHILE (emp_cursor%FOUND) AND (i <= pMaxRows) LOOP`
 - c) `v_city := get_location (emp_record.department_name);`
 - d) `CLOSE emp_cursor;`
- 7) Compile the `emp_list` procedure for debugging.
- 8) Debug the procedure.
- 9) Enter 100 as the value of the `PMAXROWS` parameter.
- 10) Examine the value of the variables in the Data tab. What are the values assigned to `REC_EMP` and `EMP_TAB`? Why?
- 11) Use the Step Into debug option to step into each line of code in `emp_list` and go through the while loop once only.
- 12) Examine the value of the variables in the Data tab. What are the values assigned to `REC_EMP`?
- 13) Continue pressing F7 until the `emp_tab(i) := rec_emp;` line is executed. Examine the value of the variables in the Data tab. What are the values assigned to `EMP_TAB`?
- 14) Use the Data tab to modify the value of the counter `i` to 98.
- 15) Continue pressing F7 until you observe the list of employees displayed in the Debugging – Log tab. How many employees are displayed?
- 16) If you use the Step Over debugger option to step through the code, do you step through the `get_location` function? Why or why not?

Practice Solutions 2-1: Creating Functions

In this practice, you create, compile, and use stored functions and a procedure.

- 1) Create and invoke the GET_JOB function to return a job title.

- a) Create and compile a function called GET_JOB to return a job title.

Open the /home/oracle/labs/plpu/solns/sol_02_01_01_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

```
CREATE OR REPLACE FUNCTION get_job (p_jobid IN
jobs.job_id%type)
  RETURN jobs.job_title%type IS
  v_title jobs.job_title%type;
BEGIN
  SELECT job_title
  INTO v_title
  FROM jobs
  WHERE job_id = p_jobid;
  RETURN v_title;
END get_job;
/
```



- b) Create a VARCHAR2 host variable called b_title, allowing a length of 35 characters. Invoke the function with job ID SA REP to return the value in the host variable, and then print the host variable to view the result.

Open the /home/oracle/labs/plpu/solns/sol_02_01_01_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

```
VARIABLE b_title VARCHAR2(35)
EXECUTE :b_title := get_job ('SA REP');
PRINT b_title
```

Practice Solutions 2-1: Creating Functions (continued)

```

Results Script Output Explain Autotrace DBMS Output OWA Output
anonymous block completed
b_title
-----
Sales Representative

```

- 2) Create a function called GET_ANNUAL_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.
- Create the GET_ANNUAL_COMP function, which accepts parameter values for the monthly salary and commission. Either or both values passed can be NULL, but the function should still return a non-NUL annual salary. Use the following basic formula to calculate the annual salary:

(salary*12) + (commission_pct*salary*12)

Open the /home/oracle/labs/plpu/solns/sol_02_01_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

```

CREATE OR REPLACE FUNCTION get_annual_comp(
    p_sal    IN employees.salary%TYPE,
    p_comm   IN employees.commission_pct%TYPE)
RETURN NUMBER IS
BEGIN
    RETURN (NVL(p_sal,0) * 12 + (NVL(p_comm,0) * nvl(p_sal,0)
    * 12));
END get_annual_comp;
/

```

```

Results Script Output Explain Autotrace DBMS Output OWA Output
FUNCTION get_annual_comp( Compiled.

```

- Use the function in a SELECT statement against the EMPLOYEES table for employees in department 30.

Open the /home/oracle/labs/plpu/solns/sol_02_01_02_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

Practice Solutions 2-1: Creating Functions (continued)

```
SELECT employee_id, last_name,
       get_annual_comp(salary,commission_pct) "Annual
Compensation"
  FROM   employees
 WHERE  department_id=30
 /
```

EMPLOYEE_ID	LAST_NAME	Annual Compensation
114	Raphaely	132000
115	Khoo	37200
116	Baida	34800
117	Tobias	33600
118	Himuro	31200
119	Colmenares	30000

6 rows selected

- 3) Create a procedure, ADD_EMPLOYEE, to insert a new employee into the EMPLOYEES table. The procedure should call a VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.
- a) Create a function called VALID_DEPTID to validate a specified department ID and return a BOOLEAN value of TRUE if the department exists.

Open the /home/oracle/labs/plpu/solns/sol_02_01_03_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

```
CREATE OR REPLACE FUNCTION valid_deptid(
  p_deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN IS
  v_dummy  PLS_INTEGER;

BEGIN
  SELECT 1
  INTO   v_dummy
  FROM   departments
  WHERE  department_id = p_deptid;
  RETURN TRUE;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
END valid_deptid;
/
```

Practice Solutions 2-1: Creating Functions (continued)

The screenshot shows a toolbar with tabs: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are two icons: a pencil and a square. The main area displays the following text:

```
FUNCTION valid_deptid( Compiled.
```

- b) Create the ADD_EMPLOYEE procedure to add an employee to the EMPLOYEES table. The row should be added to the EMPLOYEES table if the VALID_DEPTID function returns TRUE; otherwise, alert the user with an appropriate message. Provide the following parameters:

- first_name
- last_name
- email
- job: Use 'SA REP' as the default.
- mgr: Use 145 as the default.
- sal: Use 1000 as the default.
- comm: Use 0 as the default.
- deptid: Use 30 as the default.
- Use the EMPLOYEES_SEQ sequence to set the employee_id column.
- Set the hire_date column to TRUNC(SYSDATE).

Open the /home/oracle/labs/plpu/solns/sol_02_01_03_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:

```
CREATE OR REPLACE PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name  employees.last_name%TYPE,
    p_email      employees.email%TYPE,
    p_job        employees.job_id%TYPE          DEFAULT
'SA REP',
    p_mgr        employees.manager_id%TYPE       DEFAULT 145,
    p_sal        employees.salary%TYPE           DEFAULT 1000,
    p_comm       employees.commission_pct%TYPE   DEFAULT 0,
    p_deptid     employees.department_id%TYPE   DEFAULT 30)
IS
BEGIN
    IF valid_deptid(p_deptid) THEN
        INSERT INTO employees(employee_id, first_name,
last_name, email,
```

Practice Solutions 2-1: Creating Functions (continued)

```

        job_id, manager_id, hire_date, salary, commission_pct,
department_id)
    VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
ELSE
    RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
END IF;
END add_employee;
/

```

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are three icons: a pencil for edit, a floppy disk for save, and a printer. The main area displays the message: PROCEDURE add_employee(Compiled.

- c) Call ADD_EMPLOYEE for the name 'Jane Harris' in department 15, leaving other parameters with their default values. What is the result?
- Open the /home/oracle/labs/plpu/solns/sol_02_01_03_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:**

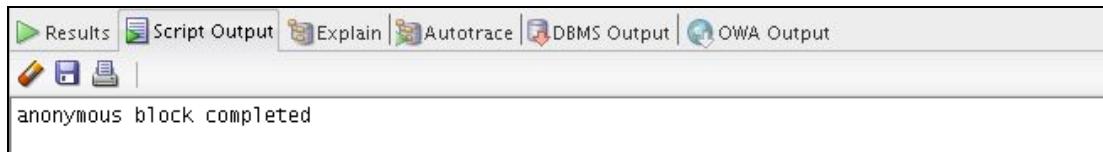
```
EXECUTE add_employee('Jane', 'Harris', 'JAHARRIS', p_deptid=> 15)
```

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are three icons: a pencil for edit, a floppy disk for save, and a printer. The main area displays the following error message:
Error starting at line 1 in command:
EXECUTE add_employee('Jane', 'Harris', 'JAHARRIS', p_deptid=> 15)
Error report:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA61.ADD_EMPLOYEE", line 17
ORA-06512: at line 1

- d) Add another employee named Joe Harris in department 80, leaving the remaining parameters with their default values. What is the result?
- Open the /home/oracle/labs/plpu/solns/sol_02_01_03_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:**

Practice Solutions 2-1: Creating Functions (continued)

```
EXECUTE add_employee('Joe', 'Harris', 'JAHARRIS',
p_deptid=> 80)
```



A screenshot of the Oracle SQL Developer interface. At the top, there is a toolbar with several tabs: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there is a toolbar with icons for edit, run, and save. The main area of the window contains the text "anonymous block completed".

Practice Solutions 2-2: Introduction to the SQL Developer Debugger

In this practice, you experiment with the basic functionality of the SQL Developer debugger.

- 1) Enable SERVEROUTPUT.

Enter the following command in the SQL Worksheet area, and then click the Run Script (F5) Click the icon on the SQL Worksheet toolbar.

```
SET SERVEROUTPUT ON
```

- 2) Run the `sol_02_02_02.sql` script to create the `emp_list` procedure. Examine the code of the procedure and compile the procedure. Why do you get the compiler error?

Open the `/home/oracle/labs/plpu/sols/sol_02_02_02.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure. The codex and the result are displayed as follows:

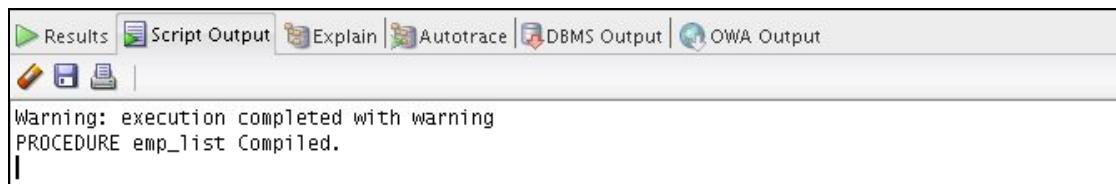
```
CREATE OR REPLACE PROCEDURE emp_list
(p_maxrows IN NUMBER)
IS
CURSOR cur_emp IS
    SELECT d.department_name, e.employee_id, e.last_name,
           e.salary, e.commission_pct
    FROM departments d, employees e
    WHERE d.department_id = e.department_id;
    rec_emp cur_emp%ROWTYPE;
    TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY
BINARY_INTEGER;
    emp_tab emp_tab_type;
    i NUMBER := 1;
    v_city VARCHAR2(30);
BEGIN
    OPEN cur_emp;
    FETCH cur_emp INTO rec_emp;
    emp_tab(i) := rec_emp;
    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
        i := i + 1;
        FETCH cur_emp INTO rec_emp;
        emp_tab(i) := rec_emp;
        v_city := get_location (rec_emp.department_name);
        dbms_output.put_line('Employee ' || rec_emp.last_name ||
                           ' works in ' || v_city );
    END LOOP;
    CLOSE cur_emp;
    FOR j IN REVERSE 1..i LOOP
```

Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)

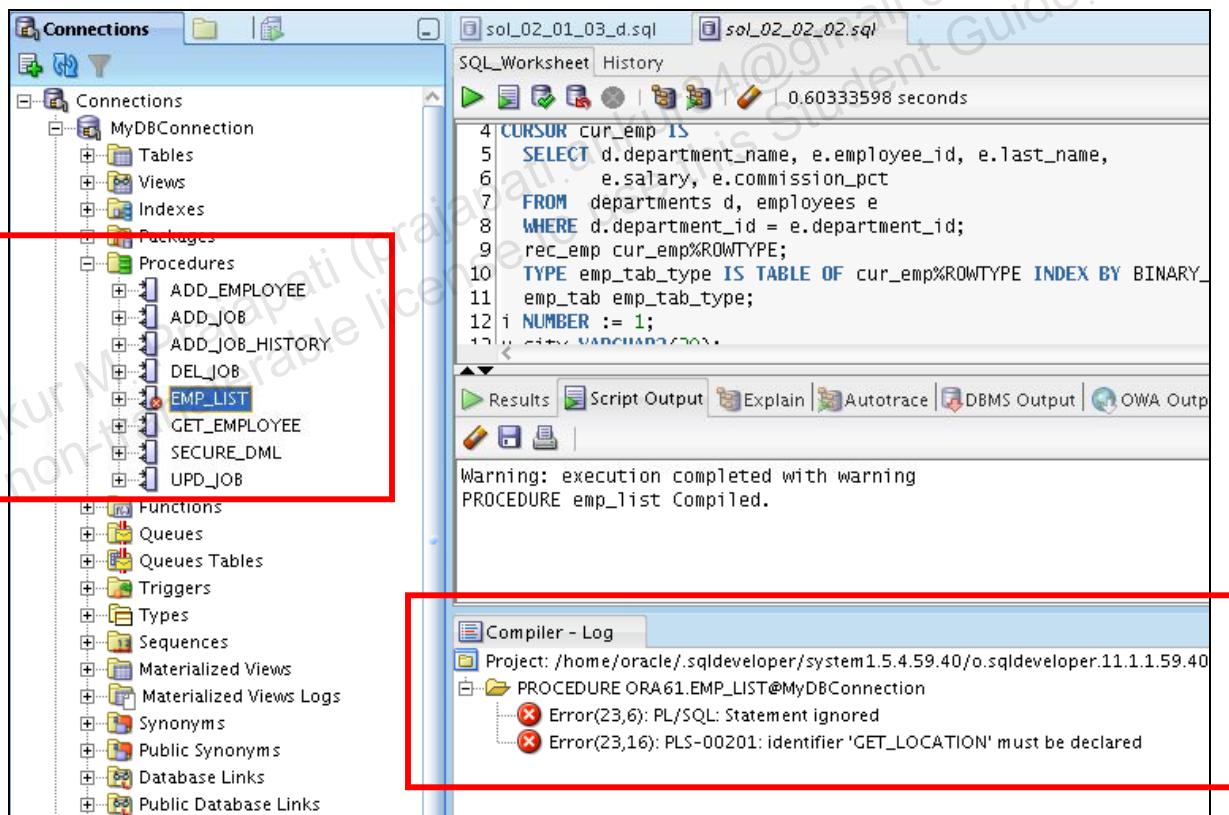
```

    DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
END LOOP;
END emp_list;
/

```



The compilation warning is because the `get_location` function is not yet declared. To display the compile error in more detail, right-click the `EMP_LIST` procedure in the Procedures node (you might need to refresh the procedures list in order to view the newly created `EMP_LIST` procedure), and then select Compile from the pop-up menu. The detailed warning message is displayed in the Compiler-Log tab as follows:



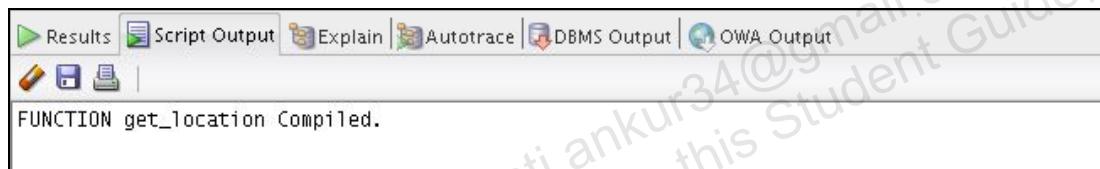
- 3) Run the `sol_02_02_03.sql` script to create the `get_location` function. Examine the code of the function, compile the function, and then correct any errors, if any.

Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)

Open the `/home/oracle/labs/plpu/solns/sol_02_02_03.sql` script.

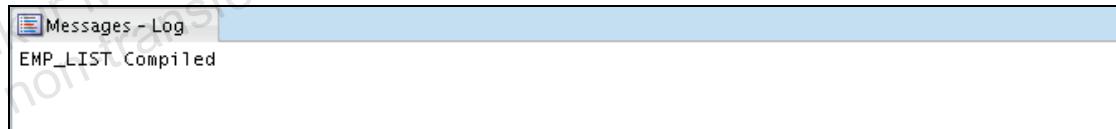
Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:

```
CREATE OR REPLACE FUNCTION get_location
( p_deptname IN VARCHAR2 ) RETURN VARCHAR2
AS
  v_loc_id NUMBER;
  v_city    VARCHAR2(30);
BEGIN
  SELECT d.location_id, l.city INTO v_loc_id, v_city
  FROM departments d, locations l
  WHERE upper(department_name) = upper(p_deptname)
  and d.location_id = l.location_id;
  RETURN v_city;
END GET_LOCATION;
/
```



- 4) Recompile the `emp_list` procedure. The procedure should compile successfully.

To recompile the procedure, right-click the procedure's name, and then select Compile from the shortcut menu.



- 5) Edit the `emp_list` procedure and the `get_location` function.

Right-click the `emp_list` procedure name in the Object Navigator, and then select Edit. The `emp_list` procedure is opened in edit mode. If the procedure is already displayed in the SQL Worksheet area, but is in read-only mode, click the Edit icon (pencil icon) in the Code tab.

Right-click the `get_location` function name in the Object Navigator, and then select Edit. The `get_location` function is opened in edit mode. If the function is already displayed in the SQL Worksheet area, but is in read-only mode, click the Edit icon (pencil icon) in the Code tab.

- 6) Add four breakpoints to the `emp_list` procedure to the following lines of code:

Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)

- a) OPEN emp_cursor;
- b) WHILE (emp_cursor%FOUND) AND (i <= pMaxRows) LOOP
- c) v_city := get_location (emp_record.department_name);
- d) CLOSE emp_cursor;

To add a breakpoint, click the line gutter next to each of the lines listed above as shown below:

```

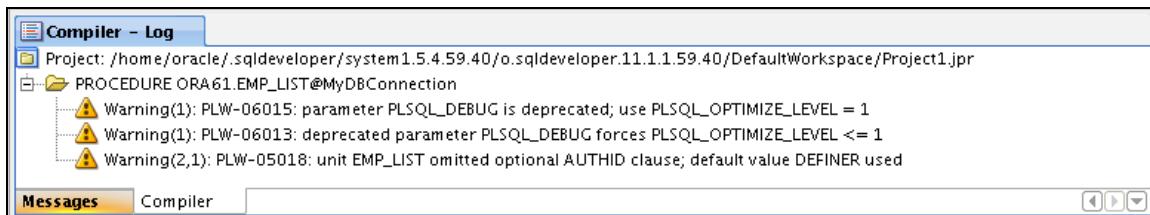
create or replace
PROCEDURE emp_list
(p_maxrows IN NUMBER)
IS
CURSOR cur_emp IS
    SELECT d.department_name, e.employee_id, e.last_name,
           e.salary, e.commission_pct
    FROM departments d, employees e
    WHERE d.department_id = e.department_id;
    rec_emp cur_emp%ROWTYPE;
    TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
    emp_tab emp_tab_type;
    i NUMBER := 1;
    v_city VARCHAR2(30);
BEGIN
    OPEN cur_emp;
    FETCH cur_emp INTO rec_emp;
    emp_tab(i) := rec_emp;
    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
        i := i + 1;
        FETCH cur_emp INTO rec_emp;
        emp_tab(i) := rec_emp;
        v_city := get_location (rec_emp.department_name);
        dbms_output.put_line('Employee ' || rec_emp.last_name ||
                           ' works in ' || v_city );
    END LOOP;
    CLOSE cur_emp;
    FOR j IN REVERSE 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
    END LOOP;
END emp_list;

```

- 7) Compile the emp_list procedure for debugging.

Click the “Compile for Debug” icon on the procedure’s toolbar as shown below:

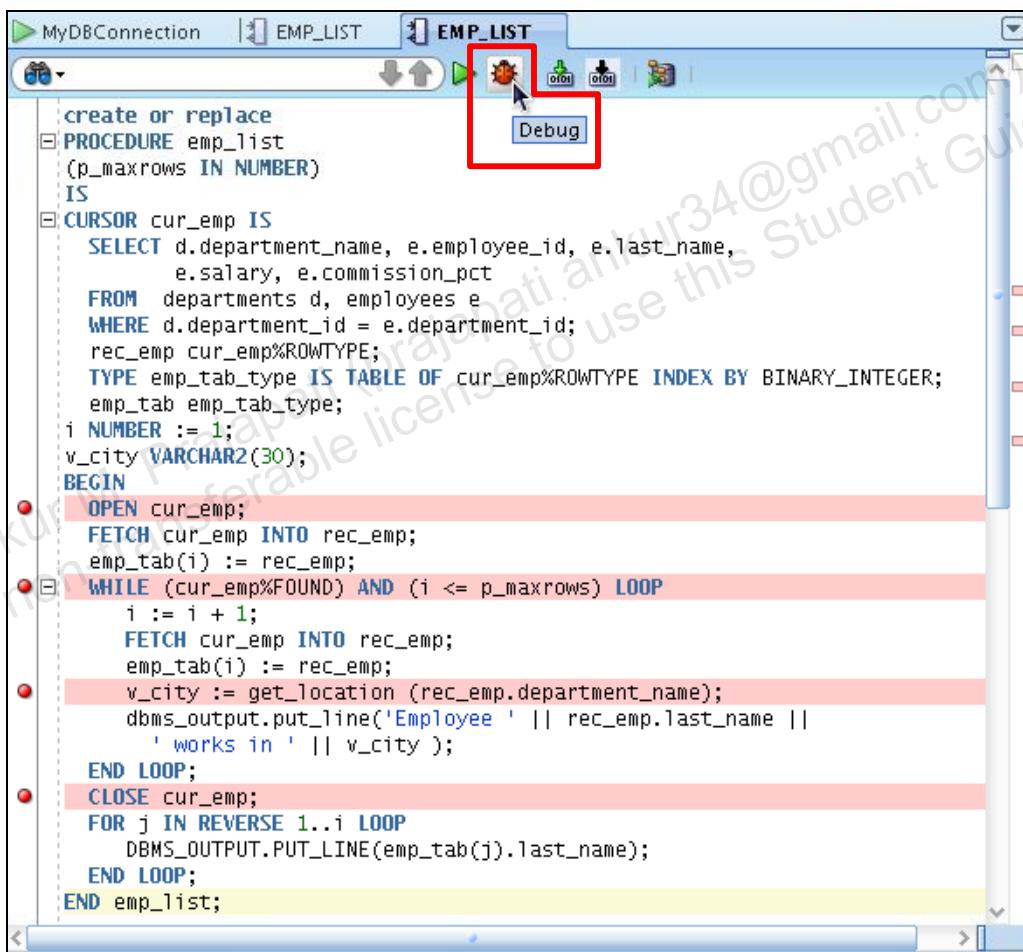
Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)



Note: If you get the above warnings, it is expected. The first two warnings are because the PLSQL_DEBUG parameter was deprecated in Oracle Database 11g, while SQL Developer is still using that parameter. The last warning is in regards to using the AUTHID clause with a procedure. This clause will be discussed in a later lesson.

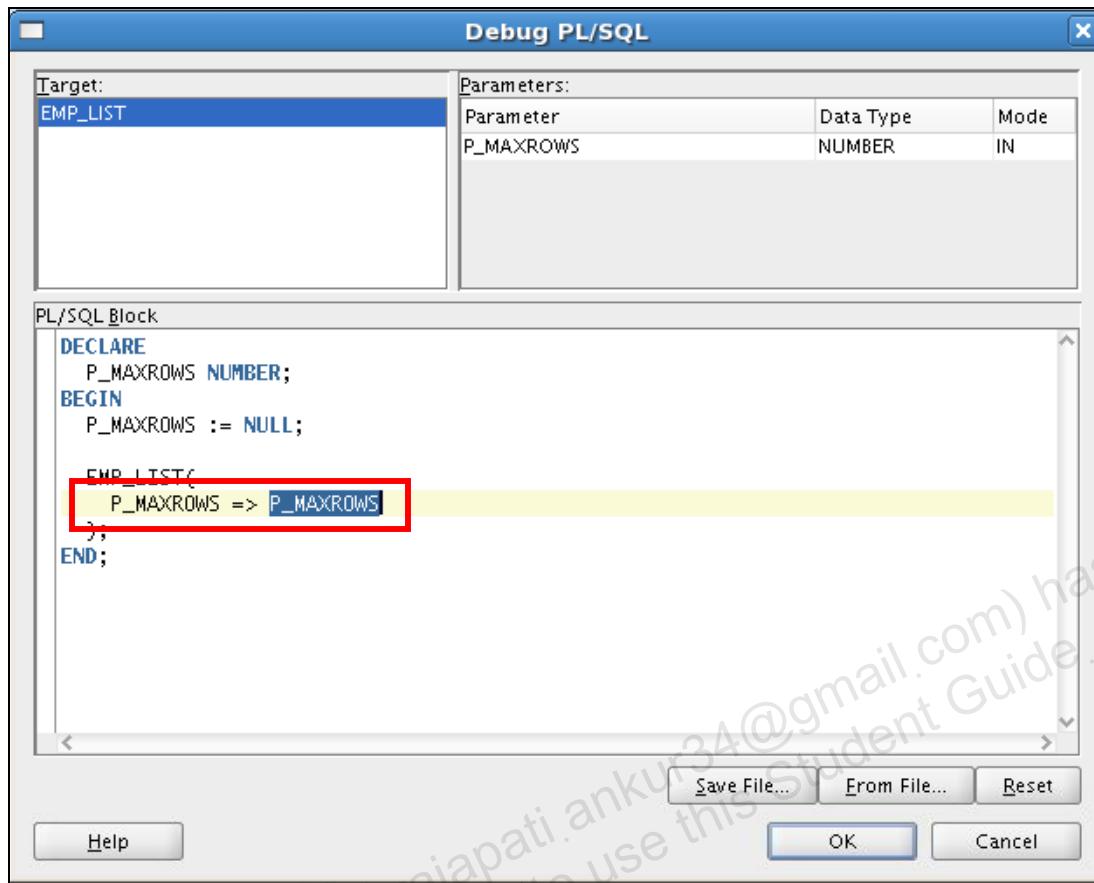
- 8) Debug the procedure.

Click the Debug icon on the procedure's toolbar as shown below:



The Debug PL/SQL window is displayed as follows:

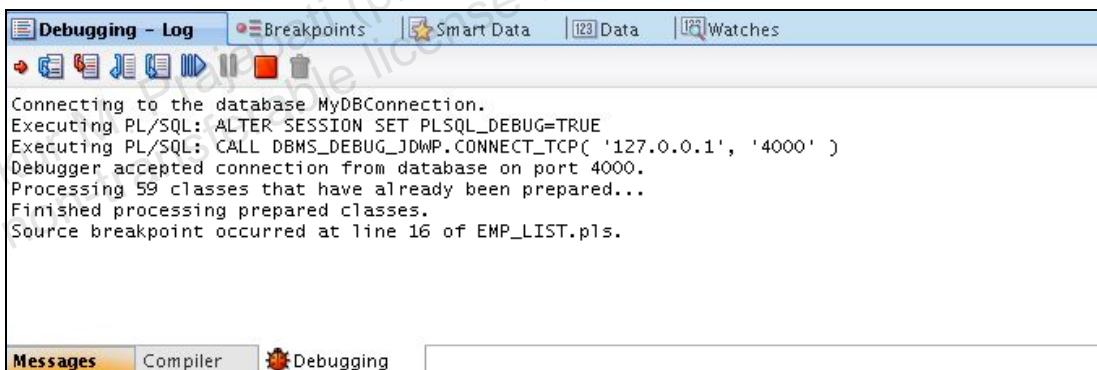
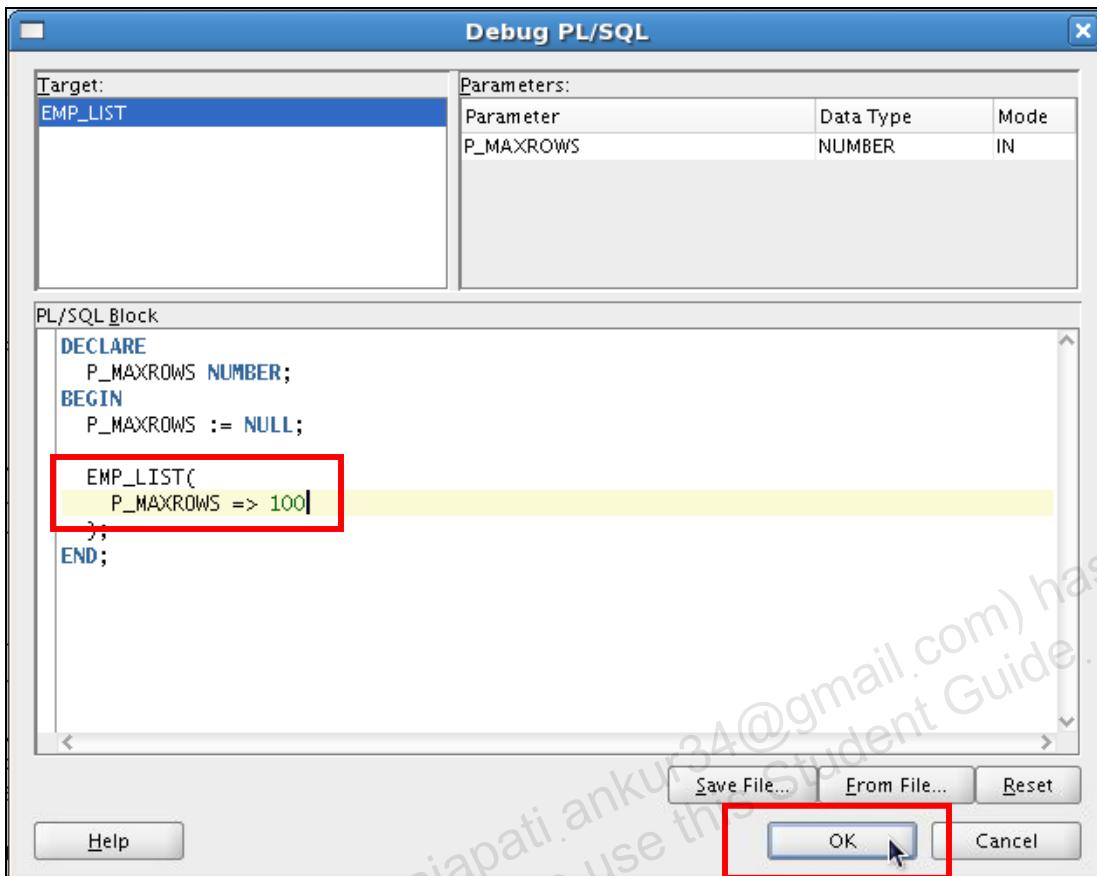
Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)



- 9) Enter 100 as the value of the PMAXROWS parameter.

Replace the second P_MAXROWS with 100, and then click OK. Notice how the program control stops at the first breakpoint in the procedure as indicated by the blue highlight color and the red arrow pointing to that line of code. The additional debugging tabs are displayed at the bottom of the page.

Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)



- 10) Examine the value of the variables in the Data tab. What are the values assigned to REC_EMP and EMP_TAB? Why?

Both are set to NULL because the data is not yet fetched into the cursor.

Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
DEPARTMENT_NAME	NULL	VARCHAR2(30)
EMPLOYEE_ID	NULL	NUMBER(6,0)
LAST_NAME	NULL	VARCHAR2(25)
SALARY	NULL	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB	indexed table	EMP_TAB_TYPE
_values		EMP_TAB_TYPE element[0]
I	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

- 11) Use the Step Into debug option to step in to each line of code in `emp_list` and go through the while loop only once.

Press [F7] to step into the code only once.

- 12) Examine the value of the variables in the Data tab. What are the values assigned to `REC_EMP` and `EMP_TAB`?

Note that when the line `FETCH cur_emp INTO rec_emp;` is executed, `rec_emp` is initialized as shown below:

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
DEPARTMENT_NAME	'Administration'	VARCHAR2(30)
EMPLOYEE_ID	200	NUMBER(6,0)
LAST_NAME	"Whalen"	VARCHAR2(25)
SALARY	4400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB	indexed table	EMP_TAB_TYPE
_values		EMP_TAB_TYPE element[0]
I	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

- 13) Continue pressing F7 until the `emp_tab(i) := rec_emp;` line is executed. Examine the value of the variables in the Data tab. What are the values assigned to `EMP_TAB`?

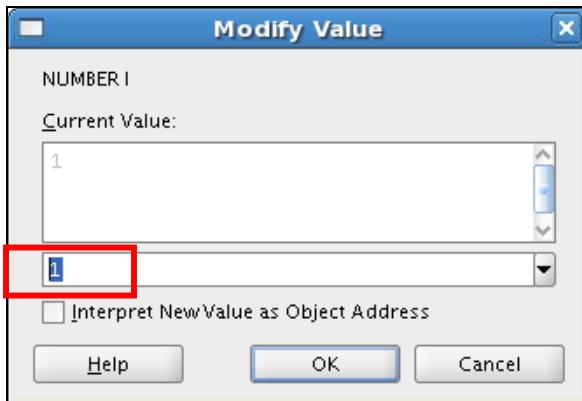
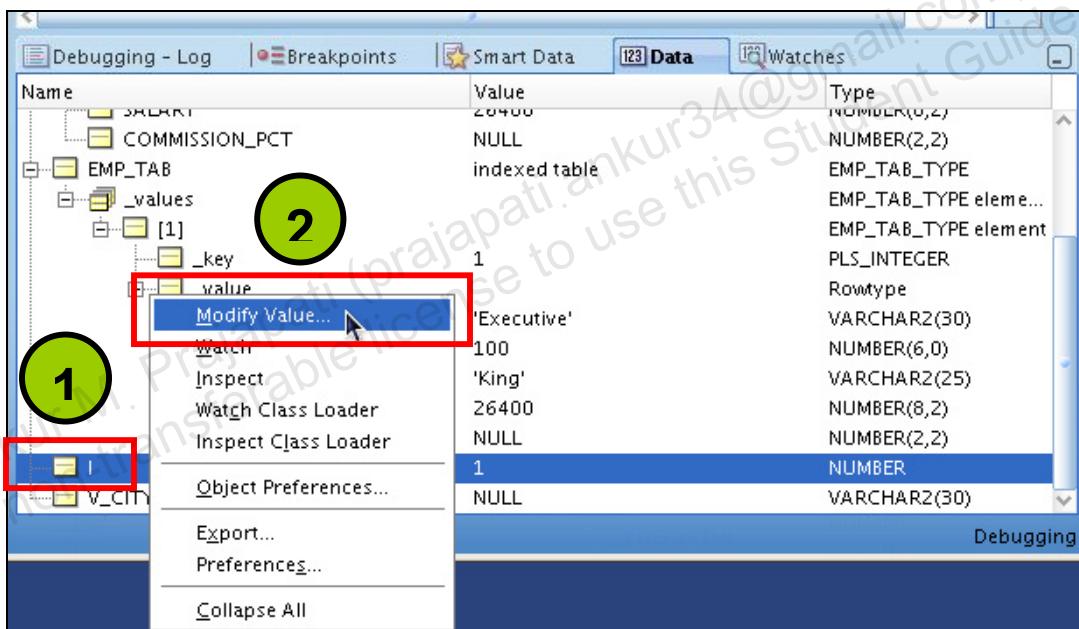
When the line `emp_tab(i) := rec_emp;` is executed, `emp_tab` is initialized to `rec_emp` as shown below:

Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)

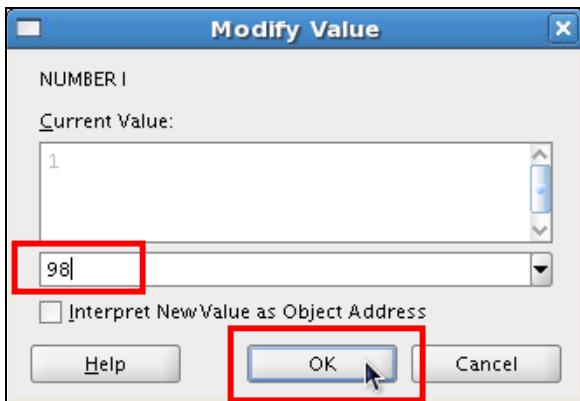
Name	Value	Type
SALARY	4400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB	indexed table	EMP_TAB_TYPE
_values		EMP_TAB_TYPE element[1]
[1]	1	EMP_TAB_TYPE element
_key		PLS_INTEGER
_value		Rowtype
DEPARTMENT_NAME	'Administration'	VARCHAR2(30)
EMPLOYEE_ID	200	NUMBER(6,0)
LAST_NAME	'Whalen'	VARCHAR2(25)
SALARY	4400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)

- 14) Use the Data tab to modify the value of the counter i to 98.

In the Data tab, right-click I, and then select Modify Value from the shortcut menu. The Modify Value window is displayed. Replace the value 1 with 98 in the text box, and then click OK as shown below:



Practice Solutions 2-2: Introduction to the SQL Developer Debugger (continued)



- 15) Continue pressing F7 until you observe the list of employees displayed in the Debugging – Log tab. How many employees are displayed?

The output at the end of the debugging session is shown below where it displays three employees:

```

Debugging - Log
Exception breakpoint occurred at line 29 of EMP_LIST.pls.
$Oracle.EXCEPTION_ORA_1403:
ORA-01403: no data found
ORA-06512: at "ORA61.EMP_LIST", line 28
ORA-06512: at line 6
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.DISCONNECT()
Employee Fay works in Toronto
Employee Hartstein works in Toronto
Employee Colmenares works in Seattle
Colmenares
Hartstein
Fay
Process exited.
Disconnecting from the database MyDBConnection.
Debugger disconnected from database.

Messages Compiler Debugging

```

- 16) If you use the Step Over debugger option to step through the code, do you step through the get_location function? Why or why not?

Although the line of code where the third breakpoint is set contains a call to the get_location function, the Step Over (F8) executes the line of code and retrieves the returned value of the function (same as [F7]); however, control is not transferred to the get_location function.

Practices and Solutions for Lesson 3

In this practice, you create a package specification and body called JOB_PKG, containing a copy of your ADD_JOB, UPD_JOB, and DEL_JOB procedures as well as your GET_JOB function. You also create and invoke a package that contains private and public constructs by using sample data.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 3-1: Creating and Using Packages

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

- 1) Create a package specification and body called JOB_PKG, containing a copy of your ADD_JOB, UPD_JOB, and DEL_JOB procedures as well as your GET_JOB function.

Note: Use the code from your previously saved procedures and functions when creating the package. You can copy the code in a procedure or function, and then paste the code into the appropriate section of the package.

- a) Create the package specification including the procedures and function headings as public constructs.
- b) Create the package body with the implementations for each of the subprograms.
- c) Delete the following stand-alone procedures and function you just packaged using the Procedures and Functions nodes in the Object Navigation tree:
 - i) The ADD_JOB, UPD_JOB, and DEL_JOB procedures
 - ii) The GET_JOB function
- d) Invoke your ADD_JOB package procedure by passing the values IT_SYSAN and SYSTEMS ANALYST as parameters.
- e) Query the JOBS table to see the result.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
<hr/>			
IT_SYSAN	Systems Analyst		
<hr/>			

1 rows selected

- 2) Create and invoke a package that contains private and public constructs.
 - a) Create a package specification and a package body called EMP_PKG that contains the following procedures and function that you created earlier:
 - i) ADD_EMPLOYEE procedure as a public construct
 - ii) GET_EMPLOYEE procedure as a public construct
 - iii) VALID_DEPTID function as a private construct
 - b) Invoke the EMP_PKG.ADD_EMPLOYEE procedure, using department ID 15 for employee Jane Harris with the email ID JAHARRIS. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.

Practice 3-1: Creating and Using Packages (continued)

- c) Invoke the ADD_EMPLOYEE package procedure by using department ID 80 for employee David Smith with the email ID DASMITH.
- d) Query the EMPLOYEES table to verify that the new employee was added.

Practice Solutions 3-1: Creating and Using Packages

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

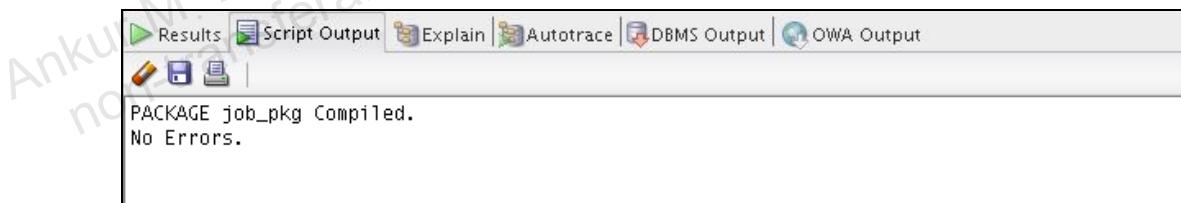
- 1) Create a package specification and body called JOB_PKG, containing a copy of your ADD_JOB, UPD_JOB, and DEL_JOB procedures as well as your GET_JOB function.

Note: Use the code from your previously saved procedures and functions when creating the package. You can copy the code in a procedure or function, and then paste the code into the appropriate section of the package.

- a) Create the package specification including the procedures and function headings as public constructs.

Open the /home/oracle/labs/plpu/solns/sol_03_01_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package specification. The code and the result are displayed as follows:

```
CREATE OR REPLACE PACKAGE job_pkg IS
    PROCEDURE add_job (p_jobid jobs.job_id%TYPE, p_jobtitle
jobs.job_title%TYPE);
    PROCEDURE del_job (p_jobid jobs.job_id%TYPE);
    FUNCTION get_job (p_jobid IN jobs.job_id%type) RETURN
jobs.job_title%type;
    PROCEDURE upd_job(p_jobid IN jobs.job_id%TYPE, p_jobtitle
IN jobs.job_title%TYPE);
END job_pkg;
/
SHOW ERRORS
```



- b) Create the package body with the implementations for each of the subprograms.

Open the /home/oracle/labs/plpu/solns/sol_03_01_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package body. The code and the result are displayed as follows:

```
CREATE OR REPLACE PACKAGE BODY job_pkg IS
    PROCEDURE add_job (
        p_jobid jobs.job_id%TYPE,
        p_jobtitle jobs.job_title%TYPE) IS
    BEGIN
```

Practice Solutions 3-1: Creating and Using Packages (continued)

```

        INSERT INTO jobs (job_id, job_title)
        VALUES (p_jobid, p_jobtitle);
        COMMIT;
    END add_job;

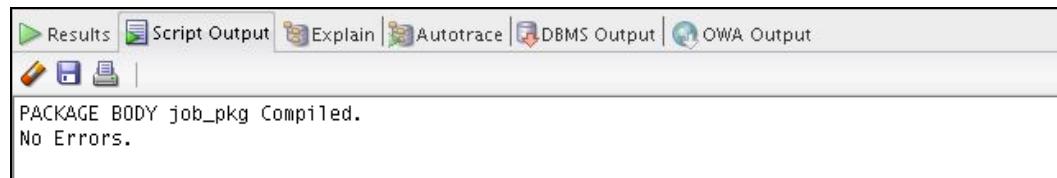
    PROCEDURE del_job (p_jobid jobs.job_id%TYPE) IS
    BEGIN
        DELETE FROM jobs
        WHERE job_id = p_jobid;
        IF SQL%NOTFOUND THEN
            RAISE_APPLICATION_ERROR(-20203, 'No jobs
deleted.');
        END IF;
    END DEL_JOB;

    FUNCTION get_job (p_jobid IN jobs.job_id%type)
    RETURN jobs.job_title%type IS
    v_title jobs.job_title%type;
    BEGIN
        SELECT job_title
        INTO v_title
        FROM jobs
        WHERE job_id = p_jobid;
        RETURN v_title;
    END get_job;

    PROCEDURE upd_job(
        p_jobid IN jobs.job_id%TYPE,
        p_jobtitle IN jobs.job_title%TYPE) IS
    BEGIN
        UPDATE jobs
        SET job_title = p_jobtitle
        WHERE job_id = p_jobid;
        IF SQL%NOTFOUND THEN
            RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
        END IF;
    END upd_job;

END job_pkg;
/
SHOW ERRORS

```



Practice Solutions 3-1: Creating and Using Packages (continued)

c) Delete the following stand-alone procedures and function you just packaged using the Procedures and Functions nodes in the Object Navigation tree:

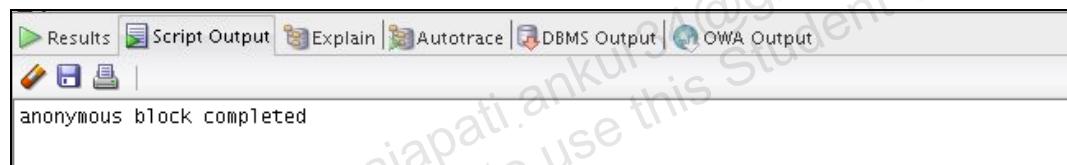
- i) The ADD_JOB, UPD_JOB, and DEL_JOB procedures
- ii) The GET_JOB function

To delete a procedure or a function, right-click the procedure's name or function's name in the Object Navigation tree, and then select Drop from the pop-up menu. The Drop window is displayed. Click Apply to drop the procedure or function. A confirmation window is displayed; click OK.

d) Invoke your ADD_JOB package procedure by passing the values IT_SYSAN and SYSTEMS ANALYST as parameters.

Open the /home/oracle/labs/plpu/solns/sol_03_01_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
EXECUTE job_pkg.add_job('IT_SYSAN', 'Systems Analyst')
```



e) Query the JOBS table to see the result.

Open the /home/oracle/labs/plpu/solns/sol_03_01_e.sql script. Click the Run Script (F5) icon or the Execute Statement (F9) on the SQL Worksheet toolbar to query the JOBS table. The code and the result (using the Run Script icon) are displayed as follows:

```
SELECT *
FROM jobs
WHERE job_id = 'IT_SYSAN';
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst		
1 rows selected			

Practice Solutions 3-1: Creating and Using Packages (continued)

2) Create and invoke a package that contains private and public constructs.

- a) Create a package specification and a package body called EMP_PKG that contains the following procedures and function that you created earlier:
 - i) ADD_EMPLOYEE procedure as a public construct
 - ii) GET_EMPLOYEE procedure as a public construct
 - iii) VALID_DEPTID function as a private construct

Open the /home/oracle/labs/plpu/solns/sol_03_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

```

Practice Solutions 3-1: Creating and Using Packages (continued)

```

p_email employees.email%TYPE,
p_job employees.job_id%TYPE DEFAULT 'SA_REP',
p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
    IF valid_deptid(p_deptid) THEN
        INSERT INTO employees(employee_id, first_name,
last_name, email,
job_id, manager_id, hire_date, salary,
commission_pct, department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid
department ID. Try again.');
    END IF;
END add_employee;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;
END emp_pkg;
/
SHOW ERRORS

```



The screenshot shows the Oracle SQL Developer interface with the 'Script Output' tab selected. The output window displays the following message:

```

PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.

```

- b) Invoke the EMP_PKG.ADD_EMPLOYEE procedure, using department ID 15 for employee Jane Harris with the email ID JAHARRIS. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.

Practice Solutions 3-1: Creating and Using Packages (continued)

Open the /home/oracle/labs/plpu/solns/sol_03_02_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

Note: You must complete step 3-2-a before performing this step. If you didn't complete step 3-2-a, run the sol_03_02_a.sql script first.

```
EXECUTE emp_pkg.add_employee ('Jane', 'Harris', 'JAHARRIS',
p_deptid => 15)
```

The screenshot shows the Oracle SQL Worksheet interface with the following error message:

```
Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('Jane', 'Harris','JAHARRIS', p_deptid => 15)
Error report:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORAG1.EMP_PKG", line 31
ORA-06512: at line 1
```

- c) Invoke the ADD_EMPLOYEE package procedure by using department ID 80 for employee David Smith with the email ID DASMITH.

Open the /home/oracle/labs/plpu/solns/sol_03_02_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
EXECUTE emp_pkg.add_employee ('David', 'Smith', 'DASMITH',
p_deptid => 80)
```

The screenshot shows the Oracle SQL Worksheet interface with the following message:

```
anonymous block completed
```

- d) Query the EMPLOYEES table to verify that the new employee was added.

Open the /home/oracle/labs/plpu/solns/sol_03_02_d.sql script. Click the Run Script (F5) icon or the Execute Statement (F9) icon (while making sure the cursor is on any of the SELECT statement code) on the SQL Worksheet toolbar to query the EMPLOYEES table. The code and the result (Execute Statement icon) are displayed as follows:

Practice Solutions 3-1: Creating and Using Packages (continued)

```
SELECT *
FROM employees
WHERE last_name = 'Smith';
```

The following output is displayed in the Results tab because we executed the code using the F9 icon.



A screenshot of the Oracle SQL Developer interface showing the results of a SQL query. The title bar includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. The main area is titled 'Results:' and contains a table with the following data:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	208	David	Smith	DASMITH (null)	19-AUG-09	SA_REP	1000	0	145	80	
2	159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	0.3	146	80
3	171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	0.15	148	80

Practices and Solutions for Lesson 4

In this practice, you modify an existing package to contain overloaded subprograms and you use forward declarations. You also create a package initialization block within a package body to populate a PL/SQL table.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 4-1: Working with Packages

In this practice, you modify the code for the EMP_PKG package that you created earlier, and then overload the ADD_EMPLOYEE procedure. Next, you create two overloaded functions called GET_EMPLOYEE in the EMP_PKG package. You also add a public procedure to EMP_PKG to populate a private PL/SQL table of valid department IDs and modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values. You also change the VALID_DEPTID validation processing function to use the private PL/SQL table of department IDs. Finally, you reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.

- 1) Modify the code for the EMP_PKG package that you created in Practice 4 step 2, and overload the ADD_EMPLOYEE procedure.
 - a) In the package specification, add a new procedure called ADD_EMPLOYEE that accepts the following three parameters:
 - i) First name
 - ii) Last name
 - iii) Department ID
 - b) Click the Run Script (F5) to create and compile the package.
 - c) Implement the new ADD_EMPLOYEE procedure in the package body as follows:
 - i) Format the email address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name.
 - ii) The procedure should call the existing ADD_EMPLOYEE procedure to perform the actual INSERT operation using its parameters and formatted email to supply the values.
 - iii) Click Run Script to create the package. Compile the package.
 - d) Invoke the new ADD_EMPLOYEE procedure using the name Samuel Joplin to be added to department 30.
 - e) Confirm that the new employee was added to the EMPLOYEES table.
- 2) In the EMP_PKG package, create two overloaded functions called GET_EMPLOYEE:
 - a) In the package specification, add the following functions:
 - i) The GET_EMPLOYEE function that accepts the parameter called p_emp_id based on the employees.employee_id%TYPE type. This function should return EMPLOYEES%ROWTYPE.
 - ii) The GET_EMPLOYEE function that accepts the parameter called p_family_name of type employees.last_name%TYPE. This function should return EMPLOYEES%ROWTYPE.
 - b) Click Run Script to re-create and compile the package.

Practice 4-1: Working with Packages (continued)

- c) In the package body:
 - i) Implement the first GET_EMPLOYEE function to query an employee using the employee's ID.
 - ii) Implement the second GET_EMPLOYEE function to use the equality operator on the value supplied in the p_family_name parameter.
 - d) Click Run Script to re-create and compile the package.
 - e) Add a utility procedure PRINT_EMPLOYEE to the EMP_PKG package as follows:
 - i) The procedure accepts an EMPLOYEES%ROWTYPE as a parameter.
 - ii) The procedure displays the following for an employee on one line, using the DBMS_OUTPUT package:
 - department_id
 - employee_id
 - first_name
 - last_name
 - job_id
 - salary
 - f) Click Run Script (F5) to create and compile the package.
 - g) Use an anonymous block to invoke the EMP_PKG.GET_EMPLOYEE function with an employee ID of 100 and family name of 'Joplin'. Use the PRINT_EMPLOYEE procedure to display the results for each row returned.
 - 3) Because the company does not frequently change its departmental data, you can improve performance of your EMP_PKG by adding a public procedure, INIT_DEPARTMENTS, to populate a private PL/SQL table of valid department IDs. Modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values.
- Note:** The sol_04_03.sql solution file script contains the code for steps a, b, and c.
- a) In the package specification, create a procedure called INIT_DEPARTMENTS with no parameters by adding the following to the package specification section before the PRINT_EMPLOYEES specification:

```
PROCEDURE init_departments;
```

 - b) In the package body, implement the INIT_DEPARTMENTS procedure to store all department IDs in a private PL/SQL index-by table named valid_departments containing BOOLEAN values.
 - i) Declare the valid_departments variable and its type definition boolean_tab_type before all procedures in the body. Enter the following at the beginning of the package body:

Practice 4-1: Working with Packages (continued)

```
TYPE boolean_tab_type IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;
```

- ii) Use the department_id column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of TRUE. Enter the INIT_DEPARTMENTS procedure declaration at the end of the package body (right after the print_employees procedure) as follows:

```
PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;
```

- c) In the body, create an initialization block that calls the INIT_DEPARTMENTS procedure to initialize the table as follows:

```
BEGIN
    init_departments;
END;
```

- d) Click Run Script (F5) to create and compile the package.

- 4) Change the VALID_DEPTID validation processing function to use the private PL/SQL table of department IDs.
- Modify the VALID_DEPTID function to perform its validation by using the PL/SQL table of department ID values. Click Run Script (F5) to create the package. Compile the package.
 - Test your code by calling ADD_EMPLOYEE using the name James Bond in department 15. What happens?
 - Insert a new department. Specify 15 for the department ID and 'Security' for the department name. Commit and verify the changes.
 - Test your code again, by calling ADD_EMPLOYEE using the name James Bond in department 15. What happens?
 - Execute the EMP_PKG.INIT_DEPARTMENTS procedure to update the internal PL/SQL table with the latest departmental data.
 - Test your code by calling ADD_EMPLOYEE using the employee name James Bond, who works in department 15. What happens?
 - Delete employee James Bond and department 15 from their respective tables, commit the changes, and refresh the department data by invoking the EMP_PKG.INIT_DEPARTMENTS procedure. Make sure you enter SET SERVEROUTPUT ON first.

Practice 4-1: Working with Packages (continued)

- 5) Reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.
- Edit the package specification and reorganize subprograms alphabetically. Click Run Script to re-create the package specification. Compile the package specification. What happens?
 - Edit the package body and reorganize all subprograms alphabetically. Click Run Script to re-create the package specification. Re-compile the package specification. What happens?
 - Correct the compilation error using a forward declaration in the body for the appropriate subprogram reference. Click Run Script to re-create the package, and then recompile the package. What happens?

Practice Solutions 4-1: Working with Packages

In this practice, you modify the code for the EMP_PKG package that you created earlier, and then overload the ADD_EMPLOYEE procedure. Next, you create two overloaded functions called GET_EMPLOYEE in the EMP_PKG package. You also add a public procedure to EMP_PKG to populate a private PL/SQL table of valid department IDs and modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values. You also change the VALID_DEPTID validation processing function to use the private PL/SQL table of department IDs. Finally, you reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.

- 1) Modify the code for the EMP_PKG package that you created in Practice 4 step 2, and overload the ADD_EMPLOYEE procedure.
 - a) In the package specification, add a new procedure called ADD_EMPLOYEE that accepts the following three parameters:
 - i) First name
 - ii) Last name
 - iii) Department ID

Open the /home/oracle/labs/plpu/solns/sol_04_01_a.sql file. The code is displayed as follows:

```

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  /* New overloaded add_employee */

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
END emp_pkg;
/

```

Practice Solutions 4-1: Working with Packages (continued)

SHOW ERRORS

- b) Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package.

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top includes icons for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there is a message box containing the text: "PACKAGE emp_pkg Compiled. No Errors."

- c) Implement the new ADD_EMPLOYEE procedure in the package body as follows:
- Format the email address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name.
 - The procedure should call the existing ADD_EMPLOYEE procedure to perform the actual INSERT operation using its parameters and formatted email to supply the values.
 - Click Run Script to create the package. Compile the package.

Open the /home/oracle/labs/plpu/solns/sol_04_01_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows (the newly added code is highlighted in bold face text in the code box below):

```

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
  BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN FALSE;
  END valid_deptid;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE);
END emp_pkg;
  
```

Practice Solutions 4-1: Working with Packages (continued)

```

p_email employees.email%TYPE,
p_job employees.job_id%TYPE DEFAULT 'SA REP',
p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30) IS

BEGIN
  IF valid_deptid(p_deptid) THEN
    INSERT INTO employees(employee_id, first_name, last_name,
                           email, job_id, manager_id, hire_date, salary,
                           commission_pct, department_id)
    VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
            p_email, p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
            p_deptid);
  ELSE
    RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try
                                again.');
  END IF;
END add_employee;

/* New overloaded add_employee procedure */

PROCEDURE add_employee(
  p_first_name employees.first_name%TYPE,
  p_last_name employees.last_name%TYPE,
  p_deptid employees.department_id%TYPE) IS
  p_email employees.email%type;
BEGIN
  p_email := UPPER(SUBSTR(p_first_name, 1,
                           1) || SUBSTR(p_last_name, 1, 7));
  add_employee(p_first_name, p_last_name, p_email, p_deptid =>
                p_deptid);
END;

/* End declaration of the overloaded add_employee procedure */

PROCEDURE get_employee(
  p.empid IN employees.employee_id%TYPE,
  p_sal OUT employees.salary%TYPE,
  p_job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO p_sal, p_job
  FROM employees
  WHERE employee_id = p.empid;
END get_employee;
END emp_pkg;
/
SHOW ERRORS

```

Practice Solutions 4-1: Working with Packages (continued)

PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.

- d) Invoke the new ADD_EMPLOYEE procedure using the name Samuel Joplin to be added to department 30.

Open the /home/oracle/labs/plpu/solns/sol_04_01_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

```
EXECUTE emp_pkg.add_employee ('Samuel', 'Joplin', 30)
```

anonymous block completed

- e) Confirm that the new employee was added to the EMPLOYEES table.

Open the /home/oracle/labs/plpu/solns/sol_04_01_e.sql script. Click anywhere on the SELECT statement, and then click the Execute Statement (F9) icon on the SQL Worksheet toolbar to execute the query. The code and the result are displayed as follows:

```
SELECT *
FROM employees
WHERE last_name = 'Joplin';
```

Results											
	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	209	Samuel	Joplin	SJOPLIN	(null)	17-JUN-09	SA_REP	1000	0	145	30

- 2) In the EMP_PKG package, create two overloaded functions called GET_EMPLOYEE:

- a) In the package specification, add the following functions:

- i) The GET_EMPLOYEE function that accepts the parameter called p_emp_id based on the employees.employee_id%TYPE type. This function should return EMPLOYEES%ROWTYPE.

Practice Solutions 4-1: Working with Packages (continued)

- ii) The GET_EMPLOYEE function that accepts the parameter called p_family_name of type employees.last_name%TYPE. This function should return EMPLOYEES%ROWTYPE.

Open the /home/oracle/labs/plpu/solns/sol_04_02_a.sql script.

```

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  /* New overloaded get_employees functions specs starts here: */

  FUNCTION get_employee(p.emp_id employees.employee_id%type)
  return employees%rowtype;

  FUNCTION get_employee(p.family_name employees.last_name%type)
  return employees%rowtype;

  /* New overloaded get_employees functions specs ends here. */

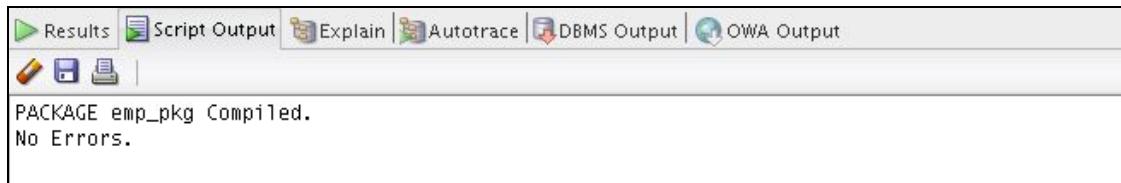
END emp_pkg;
/
SHOW ERRORS

```

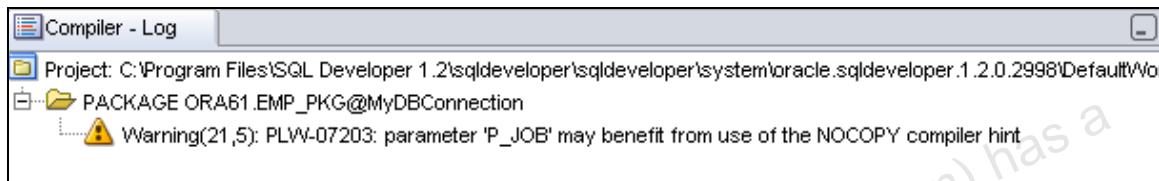
- b) Click Run Script to re-create and compile the package specification.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package's specification. The result is shown below:

Practice Solutions 4-1: Working with Packages (continued)



Note: As mentioned earlier, if your code contains an error message, you can recompile the code using the following procedure to view the details of the error or warning in the Compiler – Log tab: To compile the package specification, right-click the package's specification (or the entire package) name in the Object Navigator tree, and then select Compile from the shortcut menu. The warning is expected and is for informational purposes only.



- c) In the package body:
 - i) Implement the first GET_EMPLOYEE function to query an employee using the employee's ID.
 - ii) Implement the second GET_EMPLOYEE function to use the equality operator on the value supplied in the p_family_name parameter.

Open the /home/oracle/labs/plpu/solns/sol_04_02_c.sql script. The newly added functions are highlighted in the following code box.

```

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    
```

Practice Solutions 4-1: Working with Packages (continued)

```

p.empid IN employees.employee_id%TYPE,
p.sal OUT employees.salary%TYPE,
p.job OUT employees.job_id%TYPE);

/* New overloaded get_employees functions specs starts here: */

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype;

/* New overloaded get_employees functions specs ends here. */

END emp_pkg;
/
SHOW ERRORS

-- package body

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
        v_dummy PLS_INTEGER;
    BEGIN
        SELECT 1
        INTO v_dummy
        FROM departments
        WHERE department_id = p_deptid;
        RETURN TRUE;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE;
    END valid_deptid;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email employees.email%TYPE,
        p_job employees.job_id%TYPE DEFAULT 'SA REP',
        p_mgr employees.manager_id%TYPE DEFAULT 145,
        p_sal employees.salary%TYPE DEFAULT 1000,
        p_comm employees.commission_pct%TYPE DEFAULT 0,
        p_deptid employees.department_id%TYPE DEFAULT 30) IS
    BEGIN
        IF valid_deptid(p_deptid) THEN
            INSERT INTO employees(employee_id, first_name, last_name,
                email, job_id, manager_id, hire_date, salary,
                commission_pct, department_id)

```

Practice Solutions 4-1: Working with Packages (continued)

```

VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
        p_email, p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
        p_deptid);
ELSE
    RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
                                Try again.');
END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid =>
p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p.job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

/* New get_employee function declaration starts here */

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p.emp_id;
    RETURN rec_emp;
END;

FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p.family_name;
    RETURN rec_emp;
END;

```

Practice Solutions 4-1: Working with Packages (continued)

```

END;

/* New overloaded get_employee function declaration ends here */

END emp_pkg;
/
SHOW ERRORS

```

- d) Click Run Script to re-create the package. Compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package. The result is shown below:

```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output
| |
PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.

```

- e) Add a utility procedure PRINT_EMPLOYEE to the EMP_PKG package as follows:
- The procedure accepts an EMPLOYEES%ROWTYPE as a parameter.
 - The procedure displays the following for an employee on one line, using the DBMS_OUTPUT package:
 - department_id
 - employee_id
 - first_name
 - last_name
 - job_id
 - salary

Open the /home/oracle/labs/plpu/solns/sol_04_02_e.sql script. The newly added code is highlighted in the following code box.

```

-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
  );

```

Practice Solutions 4-1: Working with Packages (continued)

```

    p_deptid employees.department_id%TYPE DEFAULT 30);

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

/* New print_employee print_employee procedure spec */

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
        v_dummy PLS_INTEGER;
    BEGIN
        SELECT 1
        INTO v_dummy
        FROM departments
        WHERE department_id = p_deptid;
        RETURN TRUE;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
    END valid_deptid;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email employees.email%TYPE,
        p_job employees.job_id%TYPE DEFAULT 'SA REP',
        p_mgr employees.manager_id%TYPE DEFAULT 145,
        p_sal employees.salary%TYPE DEFAULT 1000,
        p_comm employees.commission_pct%TYPE DEFAULT 0,

```

Practice Solutions 4-1: Working with Packages (continued)

```

    p_deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
    IF valid_deptid(p_deptid) THEN
        INSERT INTO employees(employee_id, first_name,
last_name, email,
            job_id, manager_id, hire_date, salary, commission_pct,
department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
            p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p.sal OUT employees.salary%TYPE,
    p.job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p.emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p.emp_id;
    RETURN rec_emp;
END;

FUNCTION get_employee(p.family_name
employees.last_name%type)

```

Practice Solutions 4-1: Working with Packages (continued)

```

        return employees%rowtype IS
        rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
END;

/* New print_employees procedure declaration. */

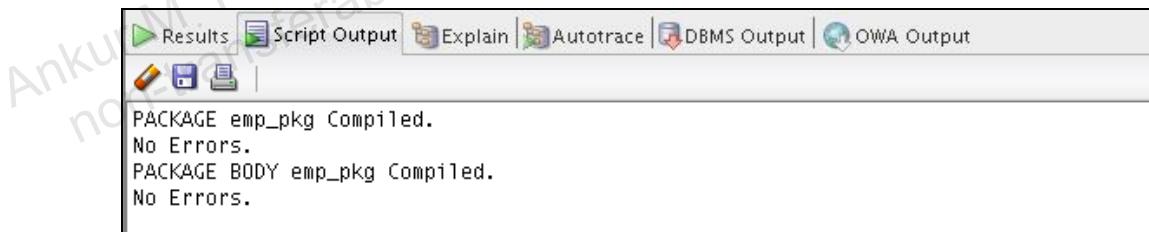
PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' ' ||
                         p_rec_emp.employee_id || ' ' ||
                         p_rec_emp.first_name || ' ' ||
                         p_rec_emp.last_name || ' ' ||
                         p_rec_emp.job_id || ' ' ||
                         p_rec_emp.salary);
END;

END emp_pkg;
/
SHOW ERRORS

```

- f) Click Run Script (F5) to create and compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package.



- g) Use an anonymous block to invoke the EMP_PKG.GET_EMPLOYEE function with an employee ID of 100 and family name of 'Joplin'. Use the PRINT_EMPLOYEE procedure to display the results for each row returned. Make sure you enter SET SERVEROUTPUT ON first.

Open the /home/oracle/labs/plpu/solns/sol_04_02_g.sql script.

```

SET SERVEROUTPUT ON
BEGIN
    emp_pkg.print_employee(emp_pkg.get_employee(100));

```

Practice Solutions 4-1: Working with Packages (continued)

```
emp_pkg.print_employee(emp_pkg.get_employee('Joplin'));
END;
/
```

```
anonymous block completed
90 100 Steven King AD_PRES 24000
30 209 Samuel Joplin SA_REP 1000
```

- 3) Because the company does not frequently change its departmental data, you can improve performance of your EMP_PKG by adding a public procedure, INIT_DEPARTMENTS, to populate a private PL/SQL table of valid department IDs. Modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values.

Note: The sol_04_03.sql solution file script contains the code for steps a, b, and c.

- a) In the package specification, create a procedure called INIT_DEPARTMENTS with no parameters by adding the following to the package specification section before the PRINT_EMPLOYEES specification:

```
PROCEDURE init_departments;
```

- b) In the package body, implement the INIT_DEPARTMENTS procedure to store all department IDs in a private PL/SQL index-by table named valid_departments containing BOOLEAN values.

- i) Declare the valid_departments variable and its type definition boolean_tab_type before all procedures in the body. Enter the following at the beginning of the package body:

```
TYPE boolean_tab_type IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;
```

- ii) Use the department_id column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of TRUE. Enter the INIT_DEPARTMENTS procedure declaration at the end of the package body (right after the print_employees procedure) as follows:

```
PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;
```

Practice Solutions 4-1: Working with Packages (continued)

- c) In the body, create an initialization block that calls the INIT_DEPARTMENTS procedure to initialize the table as follows:

```
BEGIN
    init_departments;
END;
```

**Open the /home/oracle/labs/plpu/solns/sol_04_03.sql script.
The newly added code is highlighted in the following code box.**

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email employees.email%TYPE,
        p_job employees.job_id%TYPE DEFAULT 'SA REP',
        p_mgr employees.manager_id%TYPE DEFAULT 145,
        p_sal employees.salary%TYPE DEFAULT 1000,
        p_comm employees.commission_pct%TYPE DEFAULT 0,
        p_deptid employees.department_id%TYPE DEFAULT 30);

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_deptid employees.department_id%TYPE);

    PROCEDURE get_employee(
        p.empid IN employees.employee_id%TYPE,
        p_sal OUT employees.salary%TYPE,
        p_job OUT employees.job_id%TYPE);

    FUNCTION get_employee(p_emp_id employees.employee_id%type)
        return employees%rowtype;

    FUNCTION get_employee(p_family_name
        employees.last_name%type)
        return employees%rowtype;

/* New procedure init_departments spec */

    PROCEDURE init_departments;

    PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY
```

Practice Solutions 4-1: Working with Packages (continued)

```

CREATE OR REPLACE PACKAGE BODY emp_pkg IS

/* New type */

TYPE boolean_tab_type IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;

FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
BEGIN
    SELECT 1
    INTO v_dummy
    FROM departments
    WHERE department_id = p_deptid;
    RETURN TRUE;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
    IF valid_deptid(p_deptid) THEN

        INSERT INTO employees(employee_id, first_name, last_name,
            email, job_id, manager_id, hire_date, salary,
            commission_pct, department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
            p_email, p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
            p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
                                Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS

```

Practice Solutions 4-1: Working with Packages (continued)

```

    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p.sal OUT employees.salary%TYPE,
    p.job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p.emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p.emp_id;
    RETURN rec_emp;
END;

FUNCTION get_employee(p.family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p.family_name;
    RETURN rec_emp;
END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' ' ||
    p_rec_emp.employee_id|| ' ' ||
    p_rec_emp.first_name|| ' ' ||
    p_rec_emp.last_name|| ' ' ||
    p_rec_emp.job_id|| ' ' ||
    p_rec_emp.salary);
END;

/* New init_departments procedure declaration. */

```

Practice Solutions 4-1: Working with Packages (continued)

```

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

/* call the new init_departments procedure. */

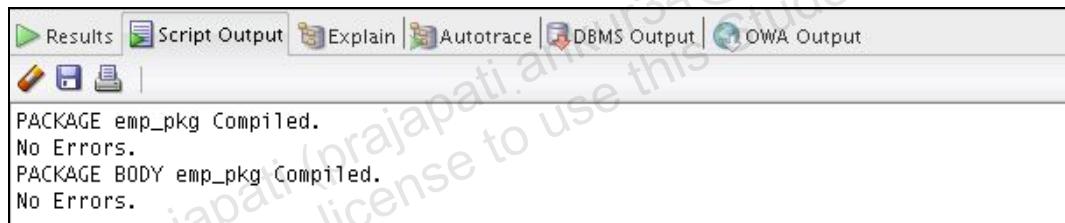
BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

- d) Click Run Script (F5) to re-create and compile the package.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package.



- 4) Change the VALID_DEPTID validation processing function to use the private PL/SQL table of department IDs.

- a) Modify the VALID_DEPTID function to perform its validation by using the PL/SQL table of department ID values. Click Run Script (F5) to create and compile the package.

Open the /home/oracle/labs/plpu/solns/sol_04_04_a.sql script. Click Run Script (F5) to create and compile the package. The newly added code is highlighted in the following code box.

```

-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',

```

Practice Solutions 4-1: Working with Packages (continued)

```

p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30);

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p_family_name
    employees.last_name%type)
    return employees%rowtype;

/* New procedure init_departments spec */

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS

TYPE boolean_tab_type IS TABLE OF BOOLEAN
    INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;

    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
        v_dummy PLS_INTEGER;
    BEGIN
        RETURN valid_departments.exists(p_deptid);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE;
    END valid_deptid;

```

Practice Solutions 4-1: Working with Packages (continued)

```

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
    IF valid_deptid(p_deptid) THEN
        INSERT INTO employees(employee_id, first_name,
            last_name, email, job_id, manager_id, hire_date,
            salary, commission_pct, department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
            p_last_name, p_email,
            p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
            Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p.emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees

```

Practice Solutions 4-1: Working with Packages (continued)

```

        WHERE employee_id = p_emp_id;
        RETURN rec_emp;
    END;

    FUNCTION get_employee(p_family_name
employees.last_name%type)
        return employees%rowtype IS
        rec_emp employees%rowtype;
    BEGIN
        SELECT * INTO rec_emp
        FROM employees
        WHERE last_name = p_family_name;
        RETURN rec_emp;
    END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' '
    p_rec_emp.employee_id|| ' '
    p_rec_emp.first_name|| ' '
    p_rec_emp.last_name|| ' '
    p_rec_emp.job_id|| ' '
    p_rec_emp.salary);
END;

/* New init_departments procedure declaration. */

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

/* call the new init_departments procedure. */

BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

The screenshot shows the Oracle SQL Developer interface with the 'Script Output' tab selected. The output window displays the following message:

```

PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.

```

Practice Solutions 4-1: Working with Packages (continued)

- b) Test your code by calling ADD_EMPLOYEE using the name James Bond in department 15. What happens?

Open the /home/oracle/labs/plpu/solns/sol_04_04_b.sql script.

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
```

Click the Run Script (F5) icon on the SQL Worksheet toolbar to test inserting a new employee. The insert operation to add the employee fails with an exception because department 15 does not exist.

```
Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
Error report:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA61.EMP_PKG", line 32
ORA-06512: at "ORA61.EMP_PKG", line 43
ORA-06512: at line 1
```

- c) Insert a new department. Specify 15 for the department ID and 'Security' for the department name. Commit and verify the changes.

Open the /home/oracle/labs/plpu/solns/sol_04_04_c.sql script. The code and result are displayed as follows:

```
INSERT INTO departments (department_id, department_name)
VALUES (15, 'Security');
COMMIT;
```

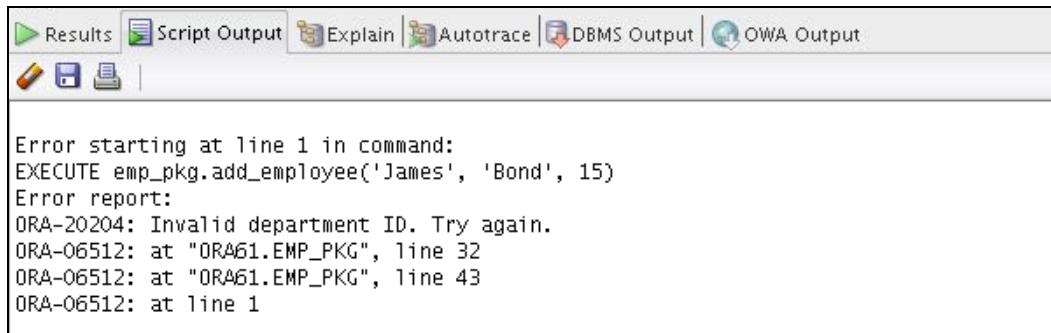
```
1 rows inserted
COMMIT succeeded.
```

- d) Test your code again, by calling ADD_EMPLOYEE using the name James Bond in department 15. What happens?

Open the /home/oracle/labs/plpu/solns/sol_04_04_d.sql script. The code and the result are displayed as follows:

```
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
```

Practice Solutions 4-1: Working with Packages (continued)



The screenshot shows the Oracle SQL Developer interface with the DBMS Output tab selected. The output window displays the following error message:

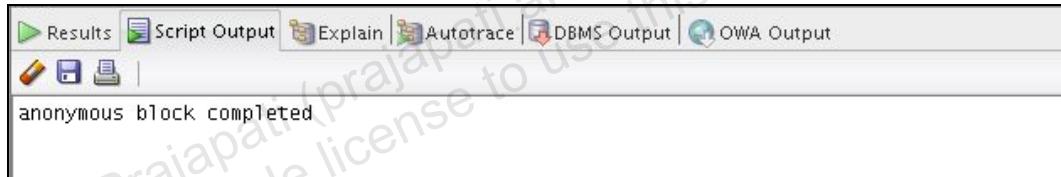
```
Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
Error report:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA61.EMP_PKG", line 32
ORA-06512: at "ORA61.EMP_PKG", line 43
ORA-06512: at line 1
```

The insert operation to add the employee fails with an exception. Department 15 does not exist as an entry in the PL/SQL associative array (index-by-table) package state variable.

- e) Execute the EMP_PKG.INIT_DEPARTMENTS procedure to update the internal PL/SQL table with the latest departmental data.

Open the /home/oracle/labs/plpu/solns/sol_04_04_e.sql script. The code and result are displayed as follows:

```
EXECUTE EMP_PKG.INIT_DEPARTMENTS
```



The screenshot shows the Oracle SQL Developer interface with the DBMS Output tab selected. The output window displays the message:

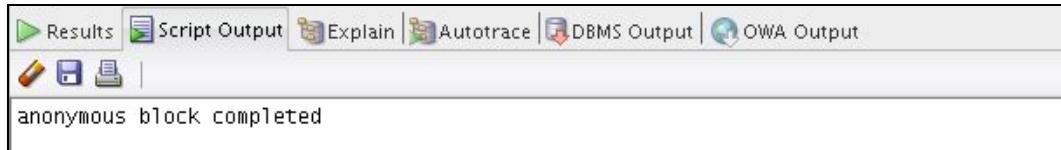
```
anonymous block completed
```

- f) Test your code by calling ADD_EMPLOYEE using the employee name James Bond, who works in department 15. What happens?

Open the /home/oracle/labs/plpu/solns/sol_04_04_f.sql script. The code and the result are displayed as follows.

```
EXECUTE emp_pkg.add_employee ('James', 'Bond', 15)
```

The row is finally inserted because the department 15 record exists in the database and the package's PL/SQL index-by table, due to invoking EMP_PKG.INIT_DEPARTMENTS, which refreshes the package state data.



The screenshot shows the Oracle SQL Developer interface with the DBMS Output tab selected. The output window displays the message:

```
anonymous block completed
```

Practice Solutions 4-1: Working with Packages (continued)

- g) Delete employee James Bond and department 15 from their respective tables, commit the changes, and refresh the department data by invoking the EMP_PKG.INIT_DEPARTMENTS procedure.

Open the /home/oracle/labs/plpu/solns/sol_04_04_g.sql script. The code and the result are displayed as follows.

```
DELETE FROM employees
WHERE first_name = 'James' AND last_name = 'Bond';
DELETE FROM departments WHERE department_id = 15;
COMMIT;
EXECUTE EMP_PKG.INIT_DEPARTMENTS
```

```
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output
|-----|
1 rows deleted
1 rows deleted
COMMIT succeeded.
anonymous block completed
```

- 5) Reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.
- a) Edit the package specification and reorganize subprograms alphabetically. Click Run Script to re-create the package specification. Compile the package specification. What happens?

Open the /home/oracle/labs/plpu/solns/sol_04_05_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package. The code and the result are displayed as follows. The package's specification subprograms are already in an alphabetical order.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  /* the package spec is already in an alphabetical order. */

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);
```

Practice Solutions 4-1: Working with Packages (continued)

```

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

FUNCTION get_employee(p.emp_id
employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p.family_name
employees.last_name%type)
    return employees%rowtype;

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

```

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for Run, Stop, and Refresh. The main area displays the message: "PACKAGE emp_pkg Compiled. No Errors."

- b) Edit the package body and reorganize all subprograms alphabetically. Click Run Script to re-create the package specification. Re-compile the package specification. What happens?

Open the /home/oracle/labs/plpu/solns/sol_04_05_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create the package. The code and the result are displayed as follows.

```

-- Package BODY
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN
        INDEX BY BINARY_INTEGER;
    valid_departments boolean_tab_type;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,

```

Practice Solutions 4-1: Working with Packages (continued)

```

p_email employees.email%TYPE,
p_job employees.job_id%TYPE DEFAULT 'SA_REP',
p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
    IF valid_deptid(p_deptid) THEN
        INSERT INTO employees(employee_id, first_name,
last_name, email,
            job_id, manager_id, hire_date, salary,
commission_pct, department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
            p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department
ID. Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email,
p_deptid => p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p_emp_id
employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees

```

Practice Solutions 4-1: Working with Packages (continued)

```

        WHERE employee_id = p_emp_id;
        RETURN rec_emp;
    END;

    FUNCTION get_employee(p_family_name
employees.last_name%type)
        return employees%rowtype IS
        rec_emp employees%rowtype;
    BEGIN
        SELECT * INTO rec_emp
        FROM employees
        WHERE last_name = p_family_name;
        RETURN rec_emp;
    END;

    PROCEDURE init_departments IS
    BEGIN
        FOR rec IN (SELECT department_id FROM departments)
        LOOP
            valid_departments(rec.department_id) := TRUE;
        END LOOP;
    END;

    PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' ' ||
                                p_rec_emp.employee_id || ' ' ||
                                p_rec_emp.first_name || ' ' ||
                                p_rec_emp.last_name || ' ' ||
                                p_rec_emp.job_id || ' ' ||
                                p_rec_emp.salary);
    END;

    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
        v_dummy PLS_INTEGER;
    BEGIN
        RETURN valid_departments.exists(p_deptid);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE;
    END valid_deptid;

BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

Practice Solutions 4-1: Working with Packages (continued)

The package does not compile successfully because the VALID_DEPTID function is referenced before it is declared.

A screenshot of the Oracle SQL Worksheet interface. The toolbar at the top includes 'Results', 'Script Output', 'Explain', 'Autotrace', 'DBMS Output', and 'OWA Output'. Below the toolbar, there are icons for 'Edit', 'Run', and 'Save'. The main area displays the following text:

```
Warning: execution completed with warning
PACKAGE BODY emp_pkg Compiled.
16/8          PLS-00313: 'VALID_DEPTID' not declared in this scope
```

- c) Correct the compilation error using a forward declaration in the body for the appropriate subprogram reference. Click Run Script to re-create the package, and then recompile the package. What happens?

Open the `/home/oracle/labs/plpu/solns/sol_04_05_c.sql` script. The function's forward declaration is highlighted in the code box below. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package. The code and the result are displayed as follows.

```
-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN
        INDEX BY BINARY_INTEGER;
    valid_departments boolean_tab_type;

/* forward declaration of valid_deptid */

    FUNCTION valid_deptid(p_deptid IN
                           departments.department_id%TYPE)
        RETURN BOOLEAN;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email employees.email%TYPE,
        p_job employees.job_id%TYPE DEFAULT 'SA_REP',
        p_mgr employees.manager_id%TYPE DEFAULT 145,
        p_sal employees.salary%TYPE DEFAULT 1000,
        p_comm employees.commission_pct%TYPE DEFAULT 0,
        p_deptid employees.department_id%TYPE DEFAULT 30) IS
    BEGIN
        IF valid_deptid(p_deptid) THEN /* valid_deptid function
referenced */
            INSERT INTO employees(employee_id, first_name,
last_name, email,
                job_id, manager_id, hire_date, salary, commission_pct,
                department_id)
```

Practice Solutions 4-1: Working with Packages (continued)

```

        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p.job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p.emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p.emp_id;
    RETURN rec_emp;
END;

FUNCTION get_employee(p.family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p.family_name;
    RETURN rec_emp;
END;

```

Practice Solutions 4-1: Working with Packages (continued)

```

END;

/* New alphabetical location of function init_departments. */

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' '
                         p_rec_emp.employee_id|| ' '
                         p_rec_emp.first_name|| ' '
                         p_rec_emp.last_name|| ' '
                         p_rec_emp.job_id|| ' '
                         p_rec_emp.salary);
END;

/* New alphabetical location of function valid_deptid. */

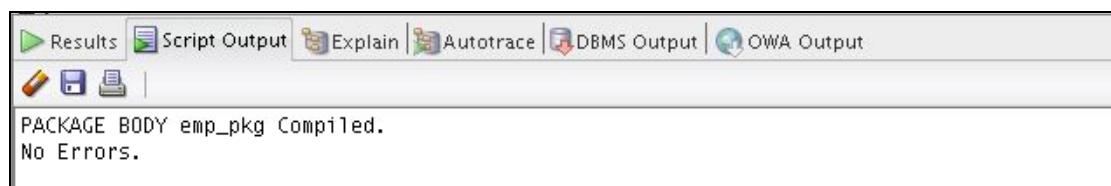
FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
BEGIN
    RETURN valid_departments.exists(p_deptid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;

BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

A forward declaration for the VALID_DEPTID function enables the package body to compile successfully as shown below:



Practices and Solutions for Lesson 5

In this practice, you use the UTL_FILE package to generate a text file report of employees in each department.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 5-1: Using the UTL_FILE Package

In this practice, you use the UTL_FILE package to generate a text file report of employees in each department. You first create and execute a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their departments. Finally, you view the generated output text file.

- 1) Create a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their departments.
 - a) Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.

Note: Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.
 - b) Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure.
- 2) Invoke the procedure using the following two arguments:
 - a) Use REPORTS_DIR as the alias for the directory object as the first parameter.
 - b) Use sal_rpt61.txt as the second parameter.
- 3) View the generated output text file as follows:
 - a) Double-click the Terminal icon on your desktop. The Terminal window is displayed.
 - b) At the \$ prompt, change to the **/home/oracle/labs/plpu/reports** folder that contains the generated output file, sal_rpt61.txt using the cd command.

Note: You can use the pwd command to list the current working directory.
 - c) List the contents of the current directory using the ls command.
 - d) Open the transferred the sal_rpt61.txt, file using gedit or an editor of your choice.

Practice Solutions 5-1: Using the UTL_FILE Package

In this practice, you use the UTL_FILE package to generate a text file report of employees in each department. You first create and execute a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their departments. Finally, you view the generated output text file.

- 1) Create a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their departments.
 - a) Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.

Note: Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.

Open the file in the /home/oracle/labs/plpu/solns/sol_05_01.sql script.

```
-- Verify with your instructor that the database initSID.ora
-- file has the directory path you are going to use with this
-- procedure.
-- For example, there should be an entry such as:
-- UTL_FILE_DIR = /home1/teachX/UTL_FILE in your initSID.ora
-- (or the SPFILE)
-- HOWEVER: The course has a directory alias provided called
-- "REPORTS_DIR" that is associated with an appropriate
-- directory. Use the directory alias name in quotes for the
-- first parameter to create a file in the appropriate
-- directory.

CREATE OR REPLACE PROCEDURE employee_report(
  p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
  f UTL_FILE.FILE_TYPE;
  CURSOR cur_avg IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                     FROM employees inner
                     GROUP BY outer.department_id)
    ORDER BY department_id;
BEGIN
  f := UTL_FILE.FOPEN(p_dir, p_filename, 'W');

```

Practice Solutions 5-1: Using the UTL_FILE Package (continued)

```

UTL_FILE.PUT_LINE(f, 'Employees who earn more than average
salary: ');
UTL_FILE.PUT_LINE(f, 'REPORT GENERATED ON ' || SYSDATE);
UTL_FILE.NEW_LINE(f);
FOR emp IN cur_avg
LOOP

    UTL_FILE.PUT_LINE(f,
RPAD(emp.last_name, 30) || ' ' ||
LPAD(NVL(TO_CHAR(emp.department_id,'9999'), '-'), 5) || ' '
||
LPAD(TO_CHAR(emp.salary, '$99,999.00'), 12));
END LOOP;
UTL_FILE.NEW_LINE(f);
UTL_FILE.PUT_LINE(f, '*** END OF REPORT ***');
UTL_FILE.FCLOSE(f);
END employee_report;
/

```

- b) Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure.
- Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure.**



- 2) Invoke the procedure using the following as arguments:
- Use REPORTS_DIR as the alias for the directory object as the first parameter.
 - Use sal_rpt61.txt as the second parameter.

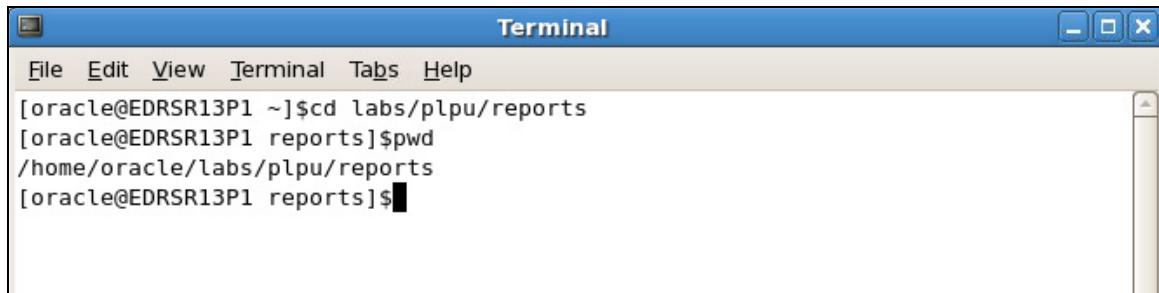
**Open the /home/oracle/labs/plpu/sols/sol_05_02.sql script.
Click the Run Script (F5) icon on the SQL Worksheet toolbar to execute the procedure. The result is shown below.**

```
-- For example, if you are student ora61, use 61 as a prefix
EXECUTE employee_report('REPORTS_DIR', 'sal_rpt61.txt')
```

- 3) View the generated output text file as follows:

Practice Solutions 5-1: Using the UTL_FILE Package (continued)

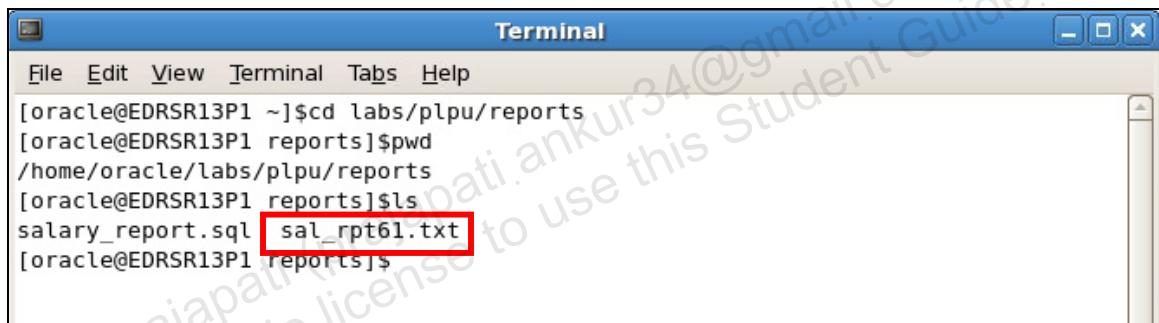
- a) Double-click the Terminal icon on your desktop. The Terminal window is displayed.
- b) At the \$ prompt, change to the `/home/oracle/labs/plpu/reports` folder that contains the generated output file, `sal_rpt61.txt` using the `cd` command as follows:



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR13P1 ~]$cd labs/plpu/reports
[oracle@EDRSR13P1 reports]$pwd
/home/oracle/labs/plpu/reports
[oracle@EDRSR13P1 reports]$
```

Note: You can use the `pwd` command to list the current working directory as shown in the screenshot above.

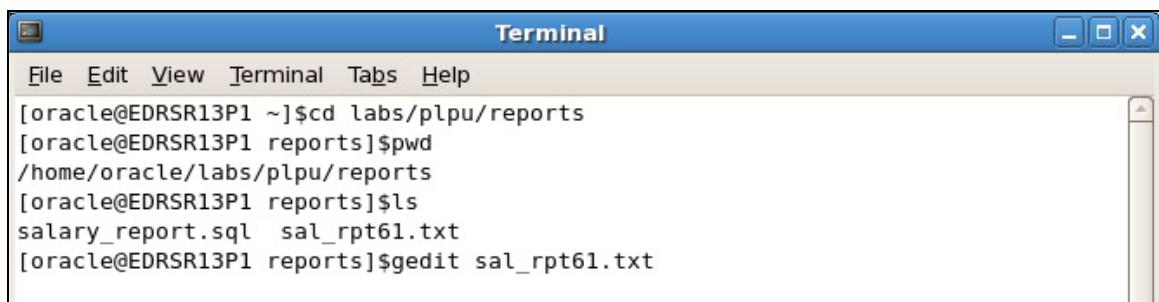
- c) List the contents of the current directory using the `ls` command as follows:



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR13P1 ~]$cd labs/plpu/reports
[oracle@EDRSR13P1 reports]$pwd
/home/oracle/labs/plpu/reports
[oracle@EDRSR13P1 reports]$ls
salary_report.sql sal_rpt61.txt
[oracle@EDRSR13P1 reports]$
```

Note the generated output file, `sal_rpt61.txt`.

- d) Open the transferred the `sal_rpt61.txt`, file using `gedit` or an editor of your choice. The report is displayed as follows:



```
Terminal
File Edit View Terminal Tabs Help
[oracle@EDRSR13P1 ~]$cd labs/plpu/reports
[oracle@EDRSR13P1 reports]$pwd
/home/oracle/labs/plpu/reports
[oracle@EDRSR13P1 reports]$ls
salary_report.sql sal_rpt61.txt
[oracle@EDRSR13P1 reports]$gedit sal_rpt61.txt
```

Practice Solutions 5-1: Using the UTL_FILE Package (continued)

Employees who earn more than average salary: REPORT GENERATED ON 18-JUN-09		
Hartstein	20	\$13,000.00
Raphaely	30	\$11,000.00
Mavris	40	\$6,500.00
Weiss	50	\$8,000.00
Fripp	50	\$8,200.00
Kaufling	50	\$7,900.00
Vollman	50	\$6,500.00
Hunold	60	\$9,000.00
Baer	70	\$10,000.00
Cambrault	80	\$11,000.00
Zlotkey	80	\$10,500.00
Tucker	80	\$10,000.00
Bernstein	80	\$9,500.00
Hall	80	\$9,000.00
Olsen	80	\$8,000.00
Cambrault	80	\$7,500.00
Tuvault	80	\$7,000.00
King	80	\$10,000.00
Sully	80	\$9,500.00
McEwen	80	\$9,000.00
Smith	80	\$8,000.00
Doran	80	\$7,500.00
Sewall	80	\$7,000.00
Vishney	80	\$10,500.00
Greene	80	\$9,500.00
Marvins	80	\$7,200.00
Lee	80	\$6,800.00
...		
Kochhar	90	\$17,000.00
Urman	100	\$7,800.00
Sciarra	100	\$7,700.00
Chen	100	\$8,200.00
Faviet	100	\$9,000.00
Greenberg	100	\$12,000.00
Popp	100	\$6,900.00
Higgins	110	\$12,000.00
Gietz	110	\$8,300.00
Grant	-	\$7,000.00
*** END OF REPORT ***		

Practices and Solutions for Lesson 6

In this practice, you create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. In addition, you create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an `INVALID` status in the `USER_OBJECTS` table.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 6-1: Using Native Dynamic SQL

In this practice, you create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. In addition, you create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an INVALID status in the USER_OBJECTS table.

- 1) Create a package called TABLE_PKG that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. The subprograms should manage optional default parameters with NULL values.

- a) Create a package specification with the following procedures:

```
PROCEDURE make(p_table_name VARCHAR2, p_col_specs VARCHAR2)
PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
                  VARCHAR2, p_cols VARCHAR2 := NULL)
PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
                  VARCHAR2, p_conditions VARCHAR2 := NULL)
PROCEDURE del_row(p_table_name VARCHAR2,
                  p_conditions VARCHAR2 := NULL);
PROCEDURE remove(p_table_name VARCHAR2)
```

- b) Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the remove procedure. This procedure should be written using the DBMS_SQL package.

- c) Execute the MAKE package procedure to create a table as follows:

```
make ('my_contacts', 'id number(4), name
      varchar2(40)');
```

- d) Describe the MY_CONTACTS table structure.

- e) Execute the ADD_ROW package procedure to add the following rows. Enable SERVEROUTPUT.

```
add_row('my_contacts', '1, ''Lauran Serhal'', 'id, name');
add_row('my_contacts', '2, ''Nancy'', 'id, name');
add_row('my_contacts', '3, ''Sunitha Patel'', 'id, name');
add_row('my_contacts', '4, ''Valli Pataballa'', 'id, name');
```

- f) Query the MY_CONTACTS table contents to verify the additions.

- g) Execute the DEL_ROW package procedure to delete a contact with an ID value of 3.

- h) Execute the UPD_ROW procedure with the following row data:

```
upd_row ('my_contacts', 'name= ''Nancy Greenberg'', 'id=2');
```

- i) Query the MY_CONTACTS table contents to verify the changes.

Practice 6-1: Using Native Dynamic SQL (continued)

- j) Drop the table by using the remove procedure and describe the MY_CONTACTS table.
- 2) Create a COMPILE_PKG package that compiles the PL/SQL code in your schema.
- In the specification, create a package procedure called MAKE that accepts the name of a PL/SQL program unit to be compiled.
 - In the package body, include the following:
 - The EXECUTE procedure used in the TABLE_PKG procedure in step 1 of this practice.
 - A private function named GET_TYPE to determine the PL/SQL object type from the data dictionary.
 - The function returns the type name (use PACKAGE for a package with a body) if the object exists; otherwise, it should return a NULL.
 - In the WHERE clause condition, add the following to the condition to ensure that only one row is returned if the name represents a PACKAGE, which may also have a PACKAGE BODY. In this case, you can only compile the complete package, but not the specification or body as separate components:

```
rownum = 1
```
 - Create the MAKE procedure by using the following information:
 - The MAKE procedure accepts one argument, name, which represents the object name.
 - The MAKE procedure should call the GET_TYPE function. If the object exists, MAKE dynamically compiles it with the ALTER statement.
 - Use the COMPILE_PKG.MAKE procedure to compile the following:
 - The EMPLOYEE_REPORT procedure
 - The EMP_PKG package
 - A nonexistent object called EMP_DATA

Practice Solutions 6-1: Using Native Dynamic SQL

In this practice, you create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. In addition, you create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an INVALID status in the USER_OBJECTS table.

- 1) Create a package called TABLE_PKG that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. The subprograms should manage optional default parameters with NULL values.

- a) Create a package specification with the following procedures:

```

PROCEDURE make(p_table_name VARCHAR2, p_col_specs VARCHAR2)
PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
    VARCHAR2, p_cols VARCHAR2 := NULL)
PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
    VARCHAR2, p_conditions VARCHAR2 := NULL)
PROCEDURE del_row(p_table_name VARCHAR2,
    p_conditions VARCHAR2 := NULL);
PROCEDURE remove(p_table_name VARCHAR2)

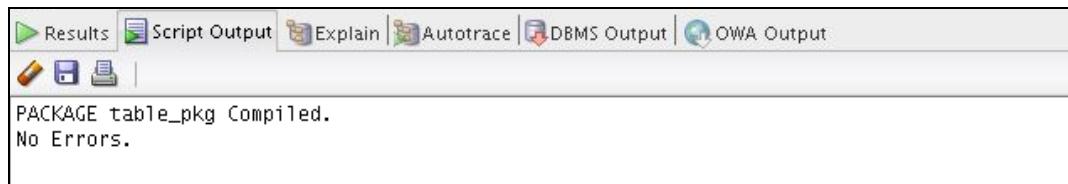
```

Open the /home/oracle/labs/plpu/solns/sol_06_01_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package specification. The code and the result are displayed as follows:

```

CREATE OR REPLACE PACKAGE table_pkg IS
  PROCEDURE make(p_table_name VARCHAR2, p_col_specs
    VARCHAR2);
  PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
    VARCHAR2, p_cols VARCHAR2 := NULL);
  PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
    VARCHAR2, p_conditions VARCHAR2 := NULL);
  PROCEDURE del_row(p_table_name VARCHAR2, p_conditions
    VARCHAR2 := NULL);
  PROCEDURE remove(p_table_name VARCHAR2);
END table_pkg;
/
SHOW ERRORS

```



Practice Solutions 6-1: Using Native Dynamic SQL (continued)

- b) Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the remove procedure. This procedure should be written using the DBMS_SQL package.

Open the /home/oracle/labs/plpu/solns/sol_06_01_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package specification. The code and the result are shown below.

```

CREATE OR REPLACE PACKAGE BODY table_pkg IS
  PROCEDURE execute(p_stmt VARCHAR2) IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE(p_stmt);
      EXECUTE IMMEDIATE p_stmt;
    END;

  PROCEDURE make(p_table_name VARCHAR2, p_col_specs VARCHAR2)
  IS
    v_stmt VARCHAR2(200) := 'CREATE TABLE ||| p_table_name |||
                                (' || p_col_specs || ')';
  BEGIN
    execute(v_stmt);
  END;

  PROCEDURE add_row(p_table_name VARCHAR2, p_col_values
                    VARCHAR2, p_cols VARCHAR2 := NULL) IS
    v_stmt VARCHAR2(200) := 'INSERT INTO ||| p_table_name';
  BEGIN
    IF p_cols IS NOT NULL THEN
      v_stmt := v_stmt || ' (' || p_cols || ')';
    END IF;
    v_stmt := v_stmt || ' VALUES (' || p_col_values || ')';
    execute(v_stmt);
  END;

  PROCEDURE upd_row(p_table_name VARCHAR2, p_set_values
                    VARCHAR2, p_conditions VARCHAR2 := NULL) IS
    v_stmt VARCHAR2(200) := 'UPDATE ||| p_table_name ||| SET '
    || p_set_values;
  BEGIN
    IF p_conditions IS NOT NULL THEN
      v_stmt := v_stmt || ' WHERE ' || p_conditions;
    END IF;
    execute(v_stmt);
  END;

```

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

```

PROCEDURE del_row(p_table_name VARCHAR2, p_conditions
                  VARCHAR2 := NULL) IS
    v_stmt VARCHAR2(200) := 'DELETE FROM ' || p_table_name;
BEGIN
    IF p_conditions IS NOT NULL THEN
        v_stmt := v_stmt || ' WHERE ' || p_conditions;
    END IF;
    execute(v_stmt);
END;

PROCEDURE remove(p_table_name VARCHAR2) IS
    cur_id INTEGER;
    v_stmt VARCHAR2(100) := 'DROP TABLE ' || p_table_name;
BEGIN
    cur_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_OUTPUT.PUT_LINE(v_stmt);
    DBMS_SQLPARSE(cur_id, v_stmt, DBMS_SQL.NATIVE);
    -- Parse executes DDL statements, no EXECUTE is required.
    DBMS_SQL CLOSE_CURSOR(cur_id);
END;

END table_pkg;
/
SHOW ERRORS

```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

|

PACKAGE BODY table_pkg Compiled.
No Errors.

- c) Execute the MAKE package procedure to create a table as follows:

```
make ('my_contacts', 'id number(4), name
      varchar2(40)');
```

Open the /home/oracle/labs/plpu/solns/sol_06_01_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create the package specification. The code and the results are displayed as follows:

```
EXECUTE table_pkg.make('my_contacts', 'id number(4), name
      varchar2(40)')
```

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

The screenshot shows the Oracle SQL Developer interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for edit, save, and run. The main area displays the message "anonymous block completed".

- d) Describe the MY_CONTACTS table structure.

The code and the results are displayed as follows:

The screenshot shows the Oracle SQL Developer interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for edit, save, and run. The main area shows the SQL command "DESCRIBE my_contacts" and its results. The results table has columns Name, Null, and Type. It shows two rows: ID (NUMBER(4)) and NAME (VARCHAR2(40)). A note at the bottom says "2 rows selected".

- e) Execute the ADD_ROW package procedure to add the following rows. Enable SERVEROUTPUT.

```
add_row('my_contacts','1','Lauran Serhal','','id, name');
add_row('my_contacts','2','Nancy','','id, name');
add_row('my_contacts','3','Sunitha Patel','','id, name');
add_row('my_contacts','4','Valli Pataballa','','id, name');
```

Open the /home/oracle/labs/plpu/sols/sol_06_01_e.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to execute the script.

The screenshot shows the Oracle SQL Developer interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for edit, save, and run. The main area shows the SQL code being executed. The code includes the SET SERVEROUTPUT ON command, a BEGIN block, and four calls to the table_pkg.add_row procedure with parameters ('my_contacts', '1', 'Lauran Serhal', ''), ('my_contacts', '2', 'Nancy', ''), ('my_contacts', '3', 'Sunitha Patel', ''), and ('my_contacts', '4', 'Valli Pataballa', ''). The code ends with an END; and a slash (/).

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

```

anonymous block completed
INSERT INTO my_contacts (id, name) VALUES (1,'Lauran Serhal')
INSERT INTO my_contacts (id, name) VALUES (2,'Nancy')
INSERT INTO my_contacts (id,name) VALUES (3,'Sunitha Patel')
INSERT INTO my_contacts (id,name) VALUES (4,'Valli Pataballa')

```

- f) Query the MY_CONTACTS table contents to verify the additions.

The code and the results are displayed as follows:

```

1 | SELECT *
2 | FROM my_contacts;
3 |

```

ID	NAME
1	Lauran Serhal
2	Nancy
3	Sunitha Patel
4	Valli Pataballa

4 rows selected

- g) Execute the DEL_ROW package procedure to delete a contact with an ID value of 3.

The code and the results are displayed as follows:

```

1 | SET SERVEROUTPUT ON
2 | EXECUTE table_pkg.del_row('my_contacts', 'id=3')
3 |

```

```

anonymous block completed
DELETE FROM my_contacts WHERE id=3

```

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

- h) Execute the UPD_ROW procedure with the following row data:*

```
upd_row('my_contacts', 'name=' 'Nancy Greenberg''', 'id=2');
```

The code and the results are displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The top bar displays 'SQL Worksheet' and 'History'. Below the toolbar, the code is shown:

```
1| SET SERVEROUTPUT ON
2| EXEC table_pkg.upd_row('my_contacts', 'name=' 'Nancy Greenberg''', 'id=2')
3| 
```

The bottom pane shows the results of the execution:

```
anonymous block completed
UPDATE my_contacts SET name='Nancy Greenberg' WHERE id=2
```

- i) Query the MY_CONTACTS table contents to verify the changes.*

The code and the results are displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The top bar displays 'SQL Worksheet' and 'History'. Below the toolbar, the code is shown:

```
1| SELECT *
2| FROM my_contacts;
3| 
```

The bottom pane shows the results of the query:

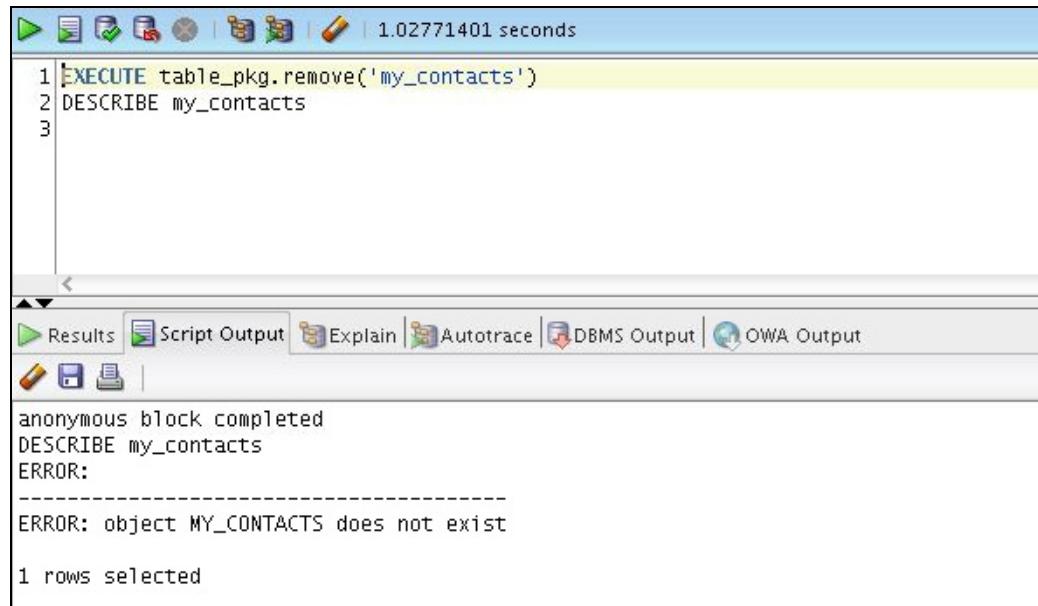
ID	NAME
1	Lauran Serhal
2	Nancy Greenberg
4	Valli Pataballa

3 rows selected

- j) Drop the table by using the remove procedure and describe the MY_CONTACTS table.*

The code and the results are displayed as follows:

Practice Solutions 6-1: Using Native Dynamic SQL (continued)



The screenshot shows an Oracle SQL Worksheet interface. The code entered is:

```

1 EXECUTE table_pkg.remove('my_contacts')
2 DESCRIBE my_contacts
3

```

The results pane shows the output of the executed statements:

```

anonymous block completed
DESCRIBE my_contacts
ERROR:
-----
ERROR: object MY_CONTACTS does not exist
1 rows selected

```

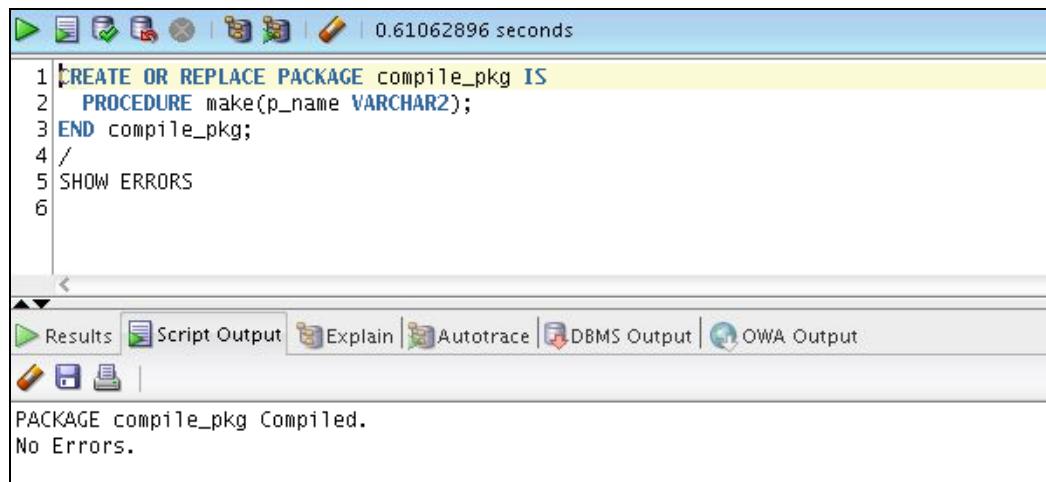
- 2) Create a COMPILE_PKG package that compiles the PL/SQL code in your schema.
 - a) In the specification, create a package procedure called MAKE that accepts the name of a PL/SQL program unit to be compiled.

Open the /home/oracle/labs/plpu/solns/sol_06_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package specification. The code and the results are shown below.

```

CREATE OR REPLACE PACKAGE compile_pkg IS
    PROCEDURE make(p_name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS

```



The screenshot shows an Oracle SQL Worksheet interface. The code entered is:

```

1 CREATE OR REPLACE PACKAGE compile_pkg IS
2     PROCEDURE make(p_name VARCHAR2);
3 END compile_pkg;
4 /
5 SHOW ERRORS
6

```

The results pane shows the output of the executed statements:

```

PACKAGE compile_pkg Compiled.
No Errors.

```

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

- b) In the package body, include the following:
- The EXECUTE procedure used in the TABLE_PKG procedure in step 1 of this practice.
 - A private function named GET_TYPE to determine the PL/SQL object type from the data dictionary.
 - The function returns the type name (use PACKAGE for a package with a body) if the object exists; otherwise, it should return a NULL.
 - In the WHERE clause condition, add the following to the condition to ensure that only one row is returned if the name represents a PACKAGE, which may also have a PACKAGE BODY. In this case, you can only compile the complete package, but not the specification or body as separate components:


```
rownum = 1
```
 - Create the MAKE procedure by using the following information:
 - The MAKE procedure accepts one argument, name, which represents the object name.
 - The MAKE procedure should call the GET_TYPE function. If the object exists, MAKE dynamically compiles it with the ALTER statement.

Open the /home/oracle/labs/plpu/solns/sol_06_02_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package body. The code and the results are displayed as follows:

```
CREATE OR REPLACE PACKAGE BODY compile_pkg IS
  PROCEDURE execute(p_stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(p_stmt);
    EXECUTE IMMEDIATE p_stmt;
  END;

  FUNCTION get_type(p_name VARCHAR2) RETURN VARCHAR2 IS
    v_proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
    */
  END;

```

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

```

SELECT object_type INTO v_proc_type
FROM user_objects
WHERE object_name = UPPER(p_name)
AND ROWNUM = 1;
RETURN v_proc_type;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN NULL;
END;

PROCEDURE make(p_name VARCHAR2) IS
    v_stmt      VARCHAR2(100);
    v_proc_type VARCHAR2(30) := get_type(p_name);
BEGIN
    IF v_proc_type IS NOT NULL THEN
        v_stmt := 'ALTER ' || v_proc_type || ' ' || p_name || '
COMPILE';
        execute(v_stmt);
    ELSE
        RAISE_APPLICATION_ERROR(-20001,
            'Subprogram '|| p_name || ' does not exist');
    END IF;
    END make;
END compile_pkg;
/
SHOW ERRORS

```



c) Use the COMPILE_PKG.MAKE procedure to compile the following:

- i) The EMPLOYEE_REPORT procedure
- ii) The EMP_PKG package
- iii) A nonexistent object called EMP_DATA

Open the file in the /home/oracle/labs/plpu/solns/sol_06_02_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to execute the package's procedure. The code and the results are shown below.

```

SET SERVEROUTPUT ON

EXECUTE compile_pkg.make('employee_report')
EXECUTE compile_pkg.make('emp_pkg')
EXECUTE compile_pkg.make('emp_data')

```

Practice Solutions 6-1: Using Native Dynamic SQL (continued)

The screenshot shows the Oracle SQL Developer interface. The top toolbar has icons for running, saving, and connecting. The status bar indicates "2.13274312 seconds". The code editor contains the following PL/SQL script:

```
1 SET SERVEROUTPUT ON
2
3 EXECUTE compile_pkg.make('employee_report')
4 EXECUTE compile_pkg.make('emp_pkg')
5 EXECUTE compile_pkg.make('emp_data')
6
```

The results tab shows the output of the execution:

```
anonymous block completed
ALTER PROCEDURE employee_report COMPILE

anonymous block completed
ALTER PACKAGE emp_pkg COMPILE

Error starting at line 5 in command:
EXECUTE compile_pkg.make('emp_data')
Error report:
ORA-20001: Subprogram 'emp_data' does not exist
ORA-06512: at "ORAG1.COMPILE_PKG", line 39
ORA-06512: at line 1
```

A watermark with the text "Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide" is diagonally across the screen.

Practices and Solutions for Lesson 7

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table. In addition, you create the add_employee procedure that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the add_employee procedure is called, whether it successfully adds a record or not.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 7-1: Using Bulk Binding and Autonomous Transactions

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table. In addition, you create the `add_employee` procedure that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the `add_employee` procedure is called, whether it successfully adds a record or not.

- 1) Update the `EMP_PKG` package with a new procedure to query employees in a specified department.
 - a) In the package specification:
 - i) Declare a `get_employees` procedure with a parameter called `dept_id`, which is based on the `employees.department_id` column type
 - ii) Define an index-by PL/SQL type as a `TABLE OF EMPLOYEES%ROWTYPE`
 - b) In the package body:
 - i) Define a private variable called `emp_table` based on the type defined in the specification to hold employee records
 - ii) Implement the `get_employees` procedure to bulk fetch the data into the table
 - c) Create a new procedure in the specification and body, called `show_employees`, which does not take arguments. The procedure displays the contents of the private PL/SQL table variable (if any data exists). Use the `print_employee` procedure that you created in an earlier practice. To view the results, click the Enable DBMS Output icon in the DBMS Output tab in SQL Developer, if you have not already done so.
 - d) Enable SERVEROUTPUT. Invoke the `emp_pkg.get_employees` procedure for department 30, and then invoke `emp_pkg.show_employees`. Repeat this for department 60.
- 2) Your manager wants to keep a log whenever the `add_employee` procedure in the package is invoked to insert a new employee into the `EMPLOYEES` table.
 - a) First, load and execute the `/home/oracle/labs/plpu/solns/sol_07_02_a.sql` script to create a log table called `LOG_NEWEMP`, and a sequence called `log_newemp_seq`.
 - b) In the `EMP_PKG` package body, modify the `add_employee` procedure, which performs the actual `INSERT` operation. Add a local procedure called `audit_newemp` as follows:
 - i) The `audit_newemp` procedure must use an autonomous transaction to insert a log record into the `LOG_NEWEMP` table.

**Practice 7-1: Using Bulk Binding and Autonomous Transactions
(continued)**

- ii) Store the USER, the current time, and the new employee name in the log table row.
- iii) Use log_newemp_seq to set the entry_id column.

Note: Remember to perform a COMMIT operation in a procedure with an autonomous transaction.

- c) Modify the add_employee procedure to invoke audit_emp before it performs the insert operation.
- d) Invoke the add_employee procedure for these new employees: Max Smart in department 20 and Clark Kent in department 10. What happens?
- e) Query the two EMPLOYEES records added, and the records in the LOG_NEWEMP table. How many log records are present?
- d) Execute a ROLLBACK statement to undo the insert operations that have not been committed. Use the same queries from step 2 e. as follows:
 - i) Use the first query to check whether the employee rows for Smart and Kent have been removed.
 - ii) Use the second query to check the log records in the LOG_NEWEMP table. How many log records are present? Why?

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table. In addition, you create the `add_employee` procedure that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the `add_employee` procedure is called, whether it successfully adds a record or not.

- 1) Update the `EMP_PKG` package with a new procedure to query employees in a specified department.
 - a) In the package specification:
 - i) Declare a `get_employees` procedure with a parameter called `dept_id`, which is based on the `employees.department_id` column type
 - ii) Define an index-by PL/SQL type as a `TABLE OF EMPLOYEES%ROWTYPE`

Open the `/home/oracle/labs/plpu/solns/sol_07_01_a.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the specification. The code and the results are displayed as follows. The newly added code is highlighted in bold letters in the code box below.

```
CREATE OR REPLACE PACKAGE emp_pkg IS

  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  FUNCTION get_employee(p.emp_id employees.employee_id%type)
  return employees%rowtype;
```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

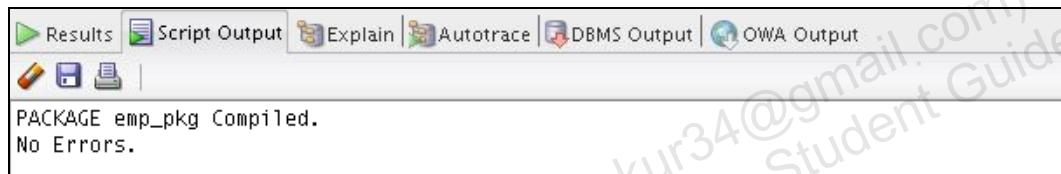
PROCEDURE get_employees(p_dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

```



b) In the package body:

- i) Define a private variable called `emp_table` based on the type defined in the specification to hold employee records
- ii) Implement the `get_employees` procedure to bulk fetch the data into the table

Open the `/home/oracle/labs/plpu/solns/sol_07_01_b.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the package body. The code and the results are shown below. The newly added code is highlighted in bold letters in the code box below.

```

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
TYPE boolean_tab_type IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;
emp_table          emp_tab_type;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
    IF valid_deptid(p_deptid) THEN

        INSERT INTO employees(employee_id, first_name,
last_name, email,
            job_id, manager_id, hire_date, salary, commission_pct,
department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
            p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p.job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p.job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p.emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p.emp_id;
    RETURN rec_emp;
END;

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS

rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
END;

/* New get_employees procedure. */

PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
END;

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' '
    p_rec_emp.employee_id|| ' '
    p_rec_emp.first_name|| ' '
    p_rec_emp.last_name|| ' '
    p_rec_emp.job_id|| ' '
    p_rec_emp.salary);
END;

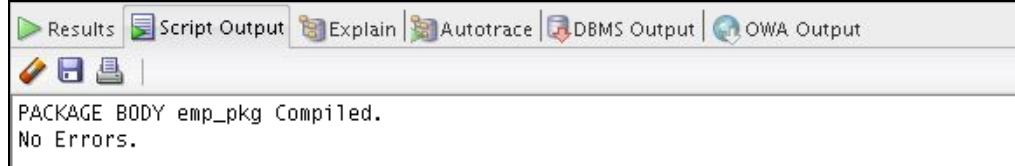
FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE) RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
BEGIN
    RETURN valid_departments.exists(p_deptid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```
BEGIN
    init_departments;

END emp_pkg;
/
SHOW ERRORS
```



- c) Create a new procedure in the specification and body, called `show_employees`, which does not take arguments. The procedure displays the contents of the private PL/SQL table variable (if any data exists). Use the `print_employee` procedure that you created in an earlier practice. To view the results, click the Enable DBMS Output icon in the DBMS Output tab in SQL Developer, if you have not already done so.

Open the `/home/oracle/labs/plpu/solns/sol_07_01_c.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to re-create and compile the package with the new procedure. The code and the results are shown below.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS
    TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email employees.email%TYPE,
        p_job employees.job_id%TYPE DEFAULT 'SA_REP',
        p_mgr employees.manager_id%TYPE DEFAULT 145,
        p_sal employees.salary%TYPE DEFAULT 1000,
        p_comm employees.commission_pct%TYPE DEFAULT 0,
        p_deptid employees.department_id%TYPE DEFAULT 30);

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_deptid employees.department_id%TYPE);

    PROCEDURE get_employee(
```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

PROCEDURE get_employees(p_dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN
        INDEX BY BINARY_INTEGER;

    valid_departments boolean_tab_type;
    emp_table          emp_tab_type;
    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
        RETURN BOOLEAN;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email      employees.email%TYPE,
        p_job        employees.job_id%TYPE DEFAULT 'SA REP',
        p_mgr        employees.manager_id%TYPE DEFAULT 145,
        p_sal        employees.salary%TYPE DEFAULT 1000,
        p_comm       employees.commission_pct%TYPE DEFAULT 0,
        p_deptid     employees.department_id%TYPE DEFAULT 30) IS
    BEGIN
        IF valid_deptid(p_deptid) THEN
            INSERT INTO employees(employee_id, first_name,
last_name, email,
                job_id, manager_id, hire_date, salary, commission_pct,
department_id)

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
          p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
      ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
      END IF;
    END add_employee;

PROCEDURE add_employee(
  p_first_name employees.first_name%TYPE,
  p_last_name employees.last_name%TYPE,
  p_deptid employees.department_id%TYPE) IS
  p_email employees.email%type;
BEGIN
  p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
  add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
  p_empid IN employees.employee_id%TYPE,
  p_sal OUT employees.salary%TYPE,
  p_job OUT employees.job_id%TYPE) IS
BEGIN
  SELECT salary, job_id
  INTO p_sal, p_job
  FROM employees
  WHERE employee_id = p_empid;
END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
  return employees%rowtype IS
  rec_emp employees%rowtype;
BEGIN
  SELECT * INTO rec_emp
  FROM employees
  WHERE employee_id = p_emp_id;
  RETURN rec_emp;
END;

FUNCTION get_employee(p_family_name
employees.last_name%type)
  return employees%rowtype IS
  rec_emp employees%rowtype;
BEGIN
  SELECT * INTO rec_emp
  FROM employees
  WHERE last_name = p_family_name;

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

        RETURN rec_emp;
    END;

    PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
    BEGIN
        SELECT * BULK COLLECT INTO emp_table
        FROM EMPLOYEES
        WHERE department_id = p_dept_id;
    END;

    PROCEDURE init_departments IS
    BEGIN
        FOR rec IN (SELECT department_id FROM departments)
        LOOP
            valid_departments(rec.department_id) := TRUE;
        END LOOP;
    END;

    PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || '||'
                                p_rec_emp.employee_id|| '||'
                                p_rec_emp.first_name|| '||'
                                p_rec_emp.last_name|| '||'
                                p_rec_emp.job_id|| '||'
                                p_rec_emp.salary);
    END;

    PROCEDURE show_employees IS
    BEGIN
        IF emp_table IS NOT NULL THEN
            DBMS_OUTPUT.PUT_LINE('Employees in Package table');
            FOR i IN 1 .. emp_table.COUNT
            LOOP
                print_employee(emp_table(i));
            END LOOP;
        END IF;
    END show_employees;

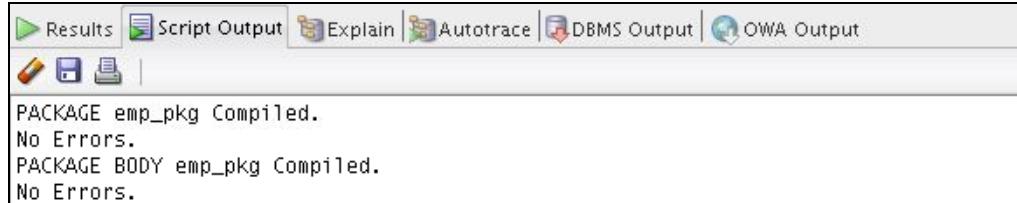
    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
        RETURN BOOLEAN IS
        v_dummy PLS_INTEGER;
    BEGIN
        RETURN valid_departments.exists(p_deptid);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE;
    END valid_deptid;

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```
BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS
```



The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top includes buttons for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are two lines of text indicating successful compilation:

```
PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.
```

- d) Enable SERVEROUTPUT. Invoke the emp_pkg.get_employees procedure for department 30, and then invoke emp_pkg.show_employees. Repeat this for department 60.

Open the /home/oracle/labs/plpu/solns/sol_07_01_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the package's procedures. The code and the results are shown below:

```
SET SERVEROUTPUT ON

EXECUTE emp_pkg.get_employees(30)
EXECUTE emp_pkg.show_employees

EXECUTE emp_pkg.get_employees(60)
EXECUTE emp_pkg.show_employees
```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

Results Script Output Explain Autotrace DBMS Output OWA Output
| |
anonymous block completed
Employees in Package table
30 114 Den Raphaely PU_MAN 11000
30 115 Alexander Khoo PU_CLERK 3100
30 116 Shelli Baida PU_CLERK 2900
30 117 Sigal Tobias PU_CLERK 2800
30 118 Guy Himuro PU_CLERK 2600
30 119 Karen Colmenares PU_CLERK 2500
30 209 Samuel Joplin SA_REP 1000

anonymous block completed
anonymous block completed
Employees in Package table
60 103 Alexander Hunold IT_PROG 9000
60 104 Bruce Ernst IT_PROG 6000
60 105 David Austin IT_PROG 4800
60 106 Valli Pataballa IT_PROG 4800
60 107 Diana Lorentz IT_PROG 4200

```

- 2) Your manager wants to keep a log whenever the add_employee procedure in the package is invoked to insert a new employee into the EMPLOYEES table.
- First, load and execute the /home/oracle/labs/plpu/solns/sol_07_02_a.sql script to create a log table called LOG_NEWEMP, and a sequence called log_newemp_seq. Open the /home/oracle/labs/plpu/solns/sol_07_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

CREATE TABLE log_newemp (
    entry_id NUMBER(6) CONSTRAINT log_newemp_pk PRIMARY KEY,
    user_id VARCHAR2(30),
    log_time DATE,
    name      VARCHAR2(60)
);

CREATE SEQUENCE log_newemp_seq;

```

```

Results Script Output Explain Autotrace DBMS Output OWA Output
| |
CREATE TABLE succeeded.
CREATE SEQUENCE succeeded.

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

- b) In the EMP_PKG package body, modify the add_employee procedure, which performs the actual INSERT operation. Add a local procedure called audit_newemp as follows:
- The audit_newemp procedure must use an autonomous transaction to insert a log record into the LOG_NEWEMP table.
 - Store the USER, the current time, and the new employee name in the log table row.
 - Use log_newemp_seq to set the entry_id column.

Note: Remember to perform a COMMIT operation in a procedure with an autonomous transaction.

Open the /home/oracle/labs/plpu/solns/sol_07_02_b.sql script. The newly added code is highlighted in bold letters in the following code box. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are displayed as follows:

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS

  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

FUNCTION get_employee(p_emp_id
employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

PROCEDURE get_employees(p_dept_id
employees.department_id%type);
PROCEDURE init_departments;
PROCEDURE print_employee(p_rec_emp employees%rowtype);
PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN
        INDEX BY BINARY_INTEGER;
    valid_departments boolean_tab_type;
    emp_table          emp_tab_type;

    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
        RETURN BOOLEAN;
    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email      employees.email%TYPE,
        p_job        employees.job_id%TYPE DEFAULT 'SA_REP',
        p_mgr        employees.manager_id%TYPE DEFAULT 145,
        p_sal        employees.salary%TYPE DEFAULT 1000,
        p_comm       employees.commission_pct%TYPE DEFAULT 0,
        p_deptid     employees.department_id%TYPE DEFAULT 30) IS
    -- New local procedure

    PROCEDURE audit_newemp IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        user_id VARCHAR2(30) := USER;
    BEGIN
        INSERT INTO log_newemp (entry_id, user_id, log_time,

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

        name)
VALUES (log_newemp_seq.NEXTVAL, user_id,
        sysdate,p_first_name||' '||p_last_name);
    COMMIT;
END audit_newemp;

BEGIN -- add_employee
IF valid_deptid(p_deptid) THEN
    INSERT INTO employees(employee_id, first_name,
last_name, email,
        job_id, manager_id, hire_date, salary,
commission_pct, department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
        p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department
ID. Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email,
p_deptid => p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p.job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p.emp_id
employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

        FROM employees
        WHERE employee_id = p_emp_id;
        RETURN rec_emp;
    END;

    FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p_family_name;
    RETURN rec_emp;
END;

/* New get_employees procedure. */

PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
END;

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' ' ||
                         p_rec_emp.employee_id|| ' ' ||
                         p_rec_emp.first_name|| ' ' ||
                         p_rec_emp.last_name|| ' ' ||
                         p_rec_emp.job_id|| ' ' ||
                         p_rec_emp.salary);
END;

PROCEDURE show_employees IS
BEGIN
    IF emp_table IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('Employees in Package table');
        FOR i IN 1 .. emp_table.COUNT
        LOOP
            print_employee(emp_table(i));
        END LOOP;
    END IF;
END;

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

        END LOOP;
    END IF;
END show_employees;

FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
BEGIN
    RETURN valid_departments.exists(p_deptid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;

BEGIN
    init_departments;
END emp_pkg;
/
SHOW ERRORS

```

PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.

- c) Modify the add_employee procedure to invoke audit_emp before it performs the insert operation.

Open the /home/oracle/labs/plpu/solns/sol_07_02_c.sql script. The newly added code is highlighted in bold letters in the following code box. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS

    TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email employees.email%TYPE,
```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

p_job employees.job_id%TYPE DEFAULT 'SA REP',
p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30);

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p.emp_id IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

FUNCTION get_employee(p.emp_id employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p.family_name
employees.last_name%type)
    return employees%rowtype;

PROCEDURE get_employees(p.dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec.emp employees%rowtype);

PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN
        INDEX BY BINARY_INTEGER;

    valid_departments boolean_tab_type;
    emp_table         emp_tab_type;

    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
        RETURN BOOLEAN;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

p_last_name employees.last_name%TYPE,
p_email employees.email%TYPE,
p_job employees.job_id%TYPE DEFAULT 'SA_REP',
p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30) IS

PROCEDURE audit_newemp IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    user_id VARCHAR2(30) := USER;
BEGIN
    INSERT INTO log_newemp (entry_id, user_id, log_time,
name)
    VALUES (log_newemp_seq.NEXTVAL, user_id,
sysdate,p_first_name||' '||p_last_name);
    COMMIT;
END audit_newemp;

BEGIN -- add_employee
    IF valid_deptid(p_deptid) THEN
        audit_newemp;
        INSERT INTO employees(employee_id, first_name,
last_name, email,
            job_id, manager_id, hire_date, salary, commission_pct,
department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
            p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
        ELSE
            RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
        END IF;
    END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%TYPE;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

    p_job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE employee_id = p.emp_id;
    RETURN rec_emp;
END;

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype IS
    rec_emp employees%rowtype;
BEGIN
    SELECT * INTO rec_emp
    FROM employees
    WHERE last_name = p.family_name;
    RETURN rec_emp;
END;

PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES
    WHERE department_id = p_dept_id;
END;

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' ' ||
    p_rec_emp.employee_id|| ' '|| )

```

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```

        p_rec_emp.first_name||' '||  

        p_rec_emp.last_name||' '||  

        p_rec_emp.job_id||' '||  

        p_rec_emp.salary);  

END;  
  

PROCEDURE show_employees IS  

BEGIN  

    IF emp_table IS NOT NULL THEN  

        DBMS_OUTPUT.PUT_LINE('Employees in Package table');  

        FOR i IN 1 .. emp_table.COUNT  

        LOOP  

            print_employee(emp_table(i));  

        END LOOP;  

    END IF;  

END show_employees;  
  

FUNCTION valid_deptid(p_deptid IN  

departments.department_id%TYPE)  

RETURN BOOLEAN IS  

    v_dummy PLS_INTEGER;  

BEGIN  

    RETURN valid_departments.exists(p_deptid);  

EXCEPTION  

    WHEN NO_DATA_FOUND THEN  

        RETURN FALSE;  

END valid_deptid;  

BEGIN  

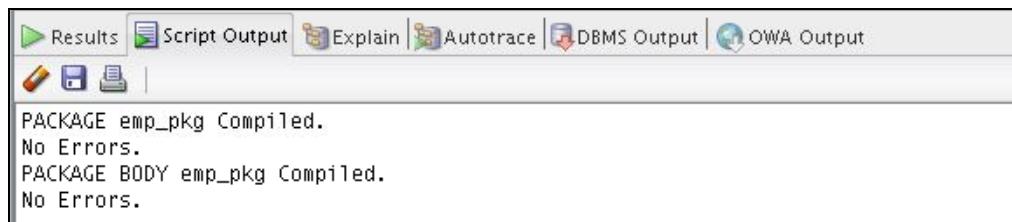
    init_departments;  

END emp_pkg;  

/  

SHOW ERRORS

```



The screenshot shows the Oracle SQL Worksheet interface with the following details:

- Toolbar icons: Results, Script Output, Explain, Autotrace, DBMS Output, OWA Output.
- Script Output pane: Displays the compilation results for the package and its body.
- Output text:

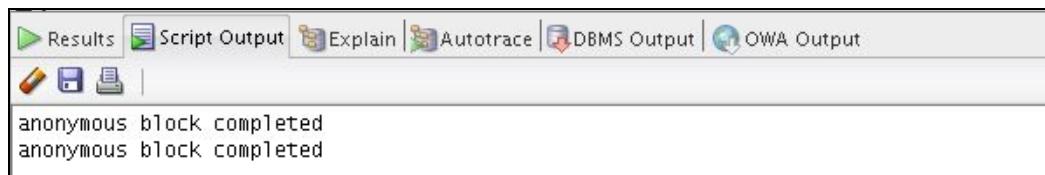
 - PACKAGE emp_pkg Compiled.
 - No Errors.
 - PACKAGE BODY emp_pkg Compiled.
 - No Errors.

- d) Invoke the add_employee procedure for these new employees: Max Smart in department 20 and Clark Kent in department 10. What happens?

Open the /home/oracle/labs/plpu/solns/sol_07_02_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are as follows.

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

```
EXECUTE emp_pkg.add_employee('Max', 'Smart', 20)
EXECUTE emp_pkg.add_employee('Clark', 'Kent', 10)
```



Both insert statements complete successfully. The log table has two log records as shown in the next step.

- e) Query the two EMPLOYEES records added, and the records in the LOG_NEWEMP table. How many log records are present?

Open the /home/oracle/labs/plpu/solns/sol_07_02_e.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are displayed as follows:

```
select department_id, employee_id, last_name, first_name
from employees
where last_name in ('Kent', 'Smart');

select * from log_newemp;
```

Employee Data			
DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	FIRST_NAME
10	212	Kent	Clark
20	211	Smart	Max
2 rows selected			
Log Table Data			
ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA61	19-AUG-09	Max Smart
2	ORA61	19-AUG-09	Clark Kent
2 rows selected			

There are two log records, one for Smart and another for Kent.

- f) Execute a ROLLBACK statement to undo the insert operations that have not been committed. Use the same queries from step 2 e. as follows:

Practice Solutions 7-1: Using Bulk Binding and Autonomous Transactions (continued)

- i) Use the first query to check whether the employee rows for Smart and Kent have been removed.
- ii) Use the second query to check the log records in the LOG_NEWEMP table. How many log records are present? Why?

ROLLBACK;

The screenshot shows the Oracle SQL Developer interface. At the top, there are tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the tabs, there are icons for New, Open, Save, and Print. The main area displays the message "ROLLBACK succeeded.".

```

1 select department_id, employee_id, last_name, first_name
2 from employees
3 where last_name in ('Kent', 'Smart');
4
5 select * from log_newemp;
6

```

The screenshot shows the Oracle SQL Developer interface with two query panes. The top pane contains the SQL code shown above. The bottom pane shows the results of the two queries. The first query returns 0 rows selected. The second query returns 2 rows selected, showing log entries for Max Smart and Clark Kent.

DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	FIRST_NAME
0 rows selected			

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA61	18-JUN-09	Max Smart
2	ORA61	18-JUN-09	Clark Kent
2 rows selected			

The two employee records are removed (rolled back). The two log records remain in the log table because they were inserted using an autonomous transaction, which is unaffected by the rollback performed in the main transaction.

Practices and Solutions for Lesson 8

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 8-1: Creating Statement and Row Triggers

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

- 1) The rows in the JOBS table store a minimum and maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.
 - a) Create a procedure called CHECK_SALARY as follows:
 - i) The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.
 - ii) The procedure uses the job ID to determine the minimum and maximum salary for the specified job.
 - iii) If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>." Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.
 - b) Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row:
 - i) The trigger must call the CHECK_SALARY procedure to carry out the business logic.
 - ii) The trigger should pass the new job ID and salary to the procedure parameters.
- 2) Test the CHECK_SAL_TRG trigger using the following cases:
 - a) Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh to department 30. What happens and why?
 - b) Update the salary of employee 115 to \$2,000. In a separate update operation, change the employee job ID to HR REP. What happens in each case?
 - c) Update the salary of employee 115 to \$2,800. What happens?
- 3) Update the CHECK_SALARY_TRG trigger to fire only when the job ID or salary values have actually changed.
 - a) Implement the business rule using a WHEN clause to check whether the JOB_ID or SALARY values have changed.

Note: Make sure that the condition handles the NULL in the OLD.column_name values if an INSERT operation is performed; otherwise, an insert operation will fail.

Practice 8-1: Creating Statement and Row Triggers (continued)

- b) Test the trigger by executing the EMP_PKG.ADD_EMPLOYEE procedure with the following parameter values:
- p_first_name: 'Eleanor'
 - p_last_name: 'Beh'
 - p_Email: 'EBEH'
 - p_Job: 'IT_PROG'
 - p_Sal: 5000
- c) Update employees with the IT_PROG job by incrementing their salary by \$2,000. What happens?
- d) Update the salary to \$9,000 for Eleanor Beh.

Hint: Use an UPDATE statement with a subquery in the WHERE clause. What happens?

- e) Change the job of Eleanor Beh to ST_MAN using another UPDATE statement with a subquery. What happens?
- 4) You are asked to prevent employees from being deleted during business hours.
- a) Write a statement trigger called DELETE_EMP_TRG on the EMPLOYEES table to prevent rows from being deleted during weekday business hours, which are from 9:00 AM to 6:00 PM.
 - b) Attempt to delete employees with JOB_ID of SA REP who are not assigned to a department.

Hint: This is employee Grant with ID 178.

Practice Solutions 8-1: Creating Statement and Row Triggers

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

- 1) The rows in the JOBS table store a minimum and maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.
 - a) Create a procedure called CHECK_SALARY as follows:
 - i) The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.
 - ii) The procedure uses the job ID to determine the minimum and maximum salary for the specified job.
 - iii) If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>". Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.

Open the /home/oracle/labs/plpu/solns/sol_08_01_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN
  SELECT min_salary, max_salary INTO v_minsal, v_maxsal
  FROM jobs
  WHERE job_id = UPPER(p_the_job);
  IF p_the_salary NOT BETWEEN v_minsal AND v_maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $' || p_the_salary || '. ' ||
      'Salaries for job '|| p_the_job ||
      ' must be between $'|| v_minsal ||' and $' || v_maxsal);
  END IF;
END;
/
SHOW ERRORS

```

The screenshot shows the Oracle SQL Worksheet interface. At the top, there are several tabs: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the tabs, there are icons for Run, Stop, and Refresh. The main area displays the following text:

```

PROCEDURE check_salary Compiled.
No Errors.

```

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)

- b) Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row:
- The trigger must call the CHECK_SALARY procedure to carry out the business logic.
 - The trigger should pass the new job ID and salary to the procedure parameters.

Open the /home/oracle/labs/plpu/solns/sol_08_01_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees
FOR EACH ROW
BEGIN
    check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```



- 2) Test the CHECK_SAL_TRIGGER using the following cases:
- Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh to department 30. What happens and why?

Open the /home/oracle/labs/plpu/solns/sol_08_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)
```

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)

```

Results Script Output Explain Autotrace DBMS Output OWA Output
| | |
Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)
Error report:
ORA-20100: Invalid salary $1000. Salaries for job SA_REP must be between $6000 and $12000
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
ORA-06512: at "ORA61.EMP_PKG", line 35
ORA-06512: at "ORA61.EMP_PKG", line 51
ORA-06512: at line 1

```

The trigger raises an exception because the `EMP_PKG.ADD_EMPLOYEE` procedure invokes an overloaded version of itself that uses the default salary of \$1,000 and a default job ID of `SA_REP`. However, the `JOBES` table stores a minimum salary of \$ 6,000 for the `SA_REP` type.

- b) Update the salary of employee 115 to \$2,000. In a separate update operation, change the employee job ID to `HR REP`. What happens in each case?

Open the `/home/oracle/labs/plpu/solns/sol_08_02_b.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

UPDATE employees
  SET salary = 2000
 WHERE employee_id = 115;

UPDATE employees
  SET job_id = 'HR REP'
 WHERE employee_id = 115;

```

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)

```

Results Script Output Explain Autotrace DBMS Output OWA Output
| | |
Error starting at line 1 in command:
UPDATE employees
  SET salary = 2000
WHERE employee_id = 115
Error report:
SQL Error: ORA-20100: Invalid salary $2000. Salaries for job PU_CLERK must be between $2500 and $5500
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'

Error starting at line 5 in command:
UPDATE employees
  SET job_id = 'HR REP'
WHERE employee_id = 115
Error report:
SQL Error: ORA-20100: Invalid salary $3100. Salaries for job HR REP must be between $4000 and $9000
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'

```

The first update statement fails to set the salary to \$2,000. The check salary trigger rule fails the update operation because the new salary for employee 115 is less than the minimum allowed for the PU_CLERK job ID.

The second update fails to change the employee's job because the current employee's salary of \$3,100 is less than the minimum for the new HR REP job ID.

- c) Update the salary of employee 115 to \$2,800. What happens?

Open the `/home/oracle/labs/plpu/solns/sol_08_02_c.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

UPDATE employees
  SET salary = 2800
WHERE employee_id = 115;

```

```

Results Script Output Explain Autotrace DBMS Output OWA Output
| | |
1 rows updated

```

The update operation is successful because the new salary falls within the acceptable range for the current job ID.

- 3) Update the CHECK_SALARY_TRG trigger to fire only when the job ID or salary values have actually changed.

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)

- a) Implement the business rule using a WHEN clause to check whether the JOB_ID or SALARY values have changed.

Note: Make sure that the condition handles the NULL in the OLD.column_name values if an INSERT operation is performed; otherwise, an insert operation will fail.

Open the /home/oracle/labs/plpu/solns/sol_08_03_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees FOR EACH ROW
WHEN (new.job_id <> NVL(old.job_id,'?') OR
      new.salary <> NVL(old.salary,0))
BEGIN
    check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```



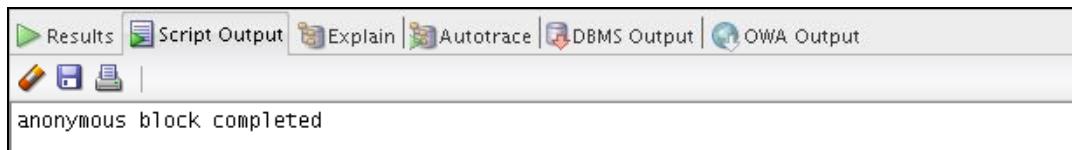
- b) Test the trigger by executing the EMP_PKG.ADD_EMPLOYEE procedure with the following parameter values:

- p_first_name: 'Eleanor'
- p_last_name: 'Beh'
- p_Email: 'EBEH'
- p_Job: 'IT_PROG'
- p_Sal: 5000

Open the /home/oracle/labs/plpu/solns/sol_08_03_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
BEGIN
    emp_pkg.add_employee('Eleanor', 'Beh', 'EBEH',
                           job => 'IT_PROG', sal => 5000);
END;
/
```

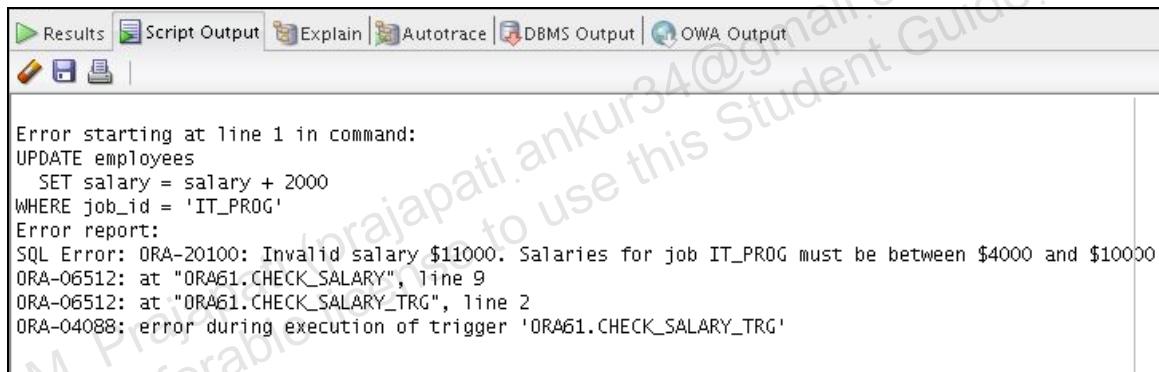
Practice Solutions 8-1: Creating Statement and Row Triggers (continued)



- c) Update employees with the IT_PROG job by incrementing their salary by \$2,000. What happens?

Open the /home/oracle/labs/plpu/solns/sol_08_03_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  SET salary = salary + 2000
 WHERE job_id = 'IT_PROG';
```



An employee's salary in the specified job type exceeds the maximum salary for that job type. No employee salaries in the IT_PROG job type are updated.

- d) Update the salary to \$9,000 for Eleanor Beh.

Open the /home/oracle/labs/plpu/solns/sol_08_03_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  SET salary = 9000
 WHERE employee_id = (SELECT employee_id
                        FROM employees
                       WHERE last_name = 'Beh');
```

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)

Results	Script Output	Explain	Autotrace	DBMS Output	OWA Output
1 rows updated					

Hint: Use an UPDATE statement with a subquery in the WHERE clause. What happens?

- e) Change the job of Eleanor Beh to ST_MAN using another UPDATE statement with a subquery. What happens?

Open the /home/oracle/labs/plpu/solns/sol_08_03_e.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  set job_id = 'ST_MAN'
WHERE employee_id = (SELECT employee_id
                      FROM employees
                     WHERE last_name = 'Beh');
```

Results	Script Output	Explain	Autotrace	DBMS Output	OWA Output
Error starting at line 1 in command: UPDATE employees set job_id = 'ST_MAN' WHERE employee_id = (SELECT employee_id FROM employees WHERE last_name = 'Beh') Error report: SQL Error: ORA-20100: Invalid salary \$9000. Salaries for job ST_MAN must be between \$5500 and \$8500 ORA-06512: at "ORA61.CHECK_SALARY", line 9 ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2 ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'					

The maximum salary of the new job type is less than the employee's current salary; therefore, the update operation fails.

- 4) You are asked to prevent employees from being deleted during business hours.
- a) Write a statement trigger called DELETE_EMP_TRG on the EMPLOYEES table to prevent rows from being deleted during weekday business hours, which are from 9:00 AM to 6:00 PM.

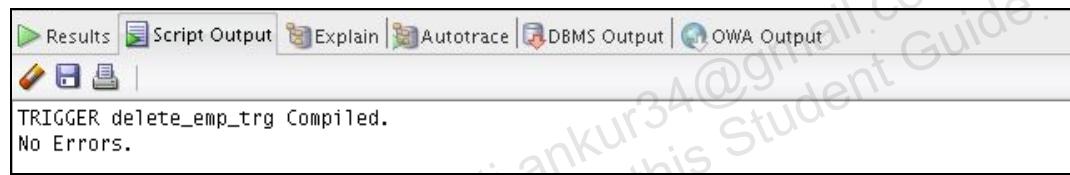
Open the /home/oracle/labs/plpu/solns/sol_08_04_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)

```

CREATE OR REPLACE TRIGGER delete_emp_trg
BEFORE DELETE ON employees
DECLARE
    the_day VARCHAR2(3) := TO_CHAR(SYSDATE, 'DY');
    the_hour PLS_INTEGER := TO_NUMBER(TO_CHAR(SYSDATE,
'HH24'));
BEGIN
    IF (the_hour BETWEEN 9 AND 18) AND (the_day NOT IN
('SAT','SUN')) THEN
        RAISE_APPLICATION_ERROR(-20150,
        'Employee records cannot be deleted during the
business
hours of 9AM and 6PM');
    END IF;
END;
/
SHOW ERRORS

```



- b) Attempt to delete employees with JOB_ID of SA REP who are not assigned to a department.

Hint: This is employee Grant with ID 178.

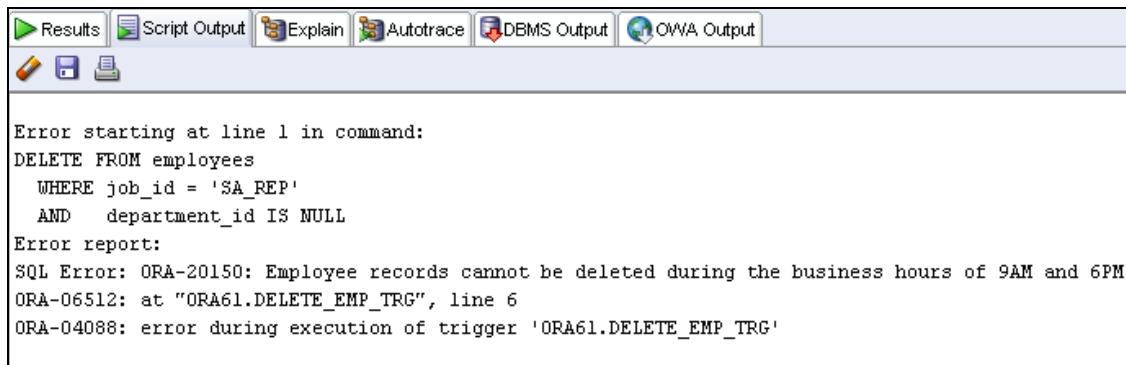
Open the /home/oracle/labs/plpu/solns/sol_08_04_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

DELETE FROM employees
WHERE job_id = 'SA REP'
AND department_id IS NULL;

```

Practice Solutions 8-1: Creating Statement and Row Triggers (continued)



The screenshot shows a SQL developer interface with the following error message:

```
Error starting at line 1 in command:  
DELETE FROM employees  
  WHERE job_id = 'SA_REP'  
    AND department_id IS NULL  
Error report:  
SQL Error: ORA-20150: Employee records cannot be deleted during the business hours of 9AM and 6PM  
ORA-06512: at "ORA61.DELETE_EMP_TRG", line 6  
ORA-04088: error during execution of trigger 'ORA61.DELETE_EMP_TRG'
```

Note: Depending on the current time on your host machine in the classroom, you may or may not be able to perform the delete operations. For example, in the screen capture above, the delete operation failed as it was performed outside the allowed business hours (based on the host machine time).

Practices and Solutions for Lesson 9

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 9-1: Managing Data Integrity Rules and Mutating Table Exceptions

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

- 1) Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salaries. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employees' salaries, the CHECK_SALARY trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.
 - a. Update your EMP_PKG package (that you last updated in Practice 8) as follows:
 - i. Add a procedure called SET_SALARY that updates the employees' salaries.
 - ii. The SET_SALARY procedure accepts the following two parameters: The job ID for those salaries that may have to be updated, and the new minimum salary for the job ID
 - b. Create a row trigger named UPD_MINSALARY_TRG on the JOBS table that invokes the EMP_PKG.SET_SALARY procedure, when the minimum salary in the JOBS table is updated for a specified job ID.
 - c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their JOB_ID is 'IT_PROG'. Then, update the minimum salary in the JOBS table to increase it by \$1,000. What happens?
- 2) To resolve the mutating table issue, create a JOBS_PKG package to maintain in memory a copy of the rows in the JOBS table. Next, modify the CHECK_SALARY procedure to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, you must create a BEFORE INSERT OR UPDATE statement trigger on the EMPLOYEES table to initialize the JOBS_PKG package state before the CHECK_SALARY row trigger is fired.
 - a. Create a new package called JOBS_PKG with the following specification:

```

PROCEDURE initialize;
FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(jobid VARCHAR2,min_salary
NUMBER);
  
```

Practice 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (Continued)

```
PROCEDURE set_maxsalary(jobid VARCHAR2, max_salary
NUMBER);
```

- b. Implement the body of JOBS_PKG as follows:
- i. Declare a private PL/SQL index-by table called jobs_tab_type that is indexed by a string type based on the JOBS.JOB_ID%TYPE.
 - ii. Declare a private variable called jobstab based on the jobs_tab_type.
 - iii. The INITIALIZE procedure reads the rows in the JOBS table by using a cursor loop, and uses the JOB_ID value for the jobstab index that is assigned its corresponding row.
 - iv. The GET_MINSalary function uses a p_jobid parameter as an index to the jobstab and returns the min_salary for that element.
 - v. The GET_MAXSalary function uses a p_jobid parameter as an index to the jobstab and returns the max_salary for that element.
 - vi. The SET_MINSalary procedure uses its p_jobid as an index to the jobstab to set the min_salary field of its element to the value in the min_salary parameter.
 - vii. The SET_MAXSalary procedure uses its p_jobid as an index to the jobstab to set the max_salary field of its element to the value in the max_salary parameter.
- c. Copy the CHECK_SALARY procedure from Practice 10, Exercise 1a, and modify the code by replacing the query on the JOBS table with statements to set the local minsal and maxsal variables with values from the JOBS_PKG data by calling the appropriate GET_*SALARY functions. This step should eliminate the mutating trigger exception.
- d. Implement a BEFORE INSERT OR UPDATE statement trigger called INIT_JOBPKG_TRG that uses the CALL syntax to invoke the JOBS_PKG.INITIALIZE procedure to ensure that the package state is current before the DML operations are performed.
- e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the IT_PROG job type by 1,000 in the JOBS table. Follow this up with a query on the employees with the IT_PROG job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?
- 3) Because the CHECK_SALARY procedure is fired by CHECK_SALARY_TRG before inserting or updating an employee, you must check whether this still works as expected.

Practice 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (Continued)

- a. Test this by adding a new employee using `EMP_PKG.ADD_EMPLOYEE` with the following parameters: ('Steve', 'Morse', 'SMORSE', and `sal => 6500`). What happens?
- b. To correct the problem encountered when adding or updating an employee:
 - i. Create a BEFORE INSERT OR UPDATE statement trigger called `EMPLOYEE_INITJOBS_TRG` on the `EMPLOYEES` table that calls the `JOBSPKG.INITIALIZE` procedure.
 - ii. Use the CALL syntax in the trigger body.
- c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the `EMPLOYEES` table by displaying the employee ID, first and last names, salary, job ID, and department ID.

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

- 1) Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salaries. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employees' salaries, the CHECK_SALARY trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.
 - a) Update your EMP_PKG package (that you last updated in Practice 8) as follows:
 - i. Add a procedure called SET_SALARY that updates the employees' salaries.
 - ii. The SET_SALARY procedure accepts the following two parameters: The job ID for those salaries that may have to be updated, and the new minimum salary for the job ID

Open the /home/oracle/labs/plpu/solns/sol_09_01_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown as follows. The newly added code is highlighted in bold letters in the following code box.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS

  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```

    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

FUNCTION get_employee(p_emp_id employees.employee_id%type)
    return employees%rowtype;

FUNCTION get_employee(p_family_name
employees.last_name%type)
    return employees%rowtype;

PROCEDURE get_employees(p_dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

PROCEDURE show_employees;

/* New set_salary procedure */

PROCEDURE set_salary(p_jobid VARCHAR2, p_min_salary NUMBER);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN
        INDEX BY BINARY_INTEGER;

    valid_departments boolean_tab_type;
    emp_table         emp_tab_type;

    FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
        RETURN BOOLEAN;

    PROCEDURE add_employee(
        p_first_name employees.first_name%TYPE,
        p_last_name employees.last_name%TYPE,
        p_email     employees.email%TYPE,
        p_job      employees.job_id%TYPE DEFAULT 'SA REP',

```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```

p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_comm employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30) IS

PROCEDURE audit_newemp IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    user_id VARCHAR2(30) := USER;
BEGIN
    INSERT INTO log_newemp (entry_id, user_id, log_time,
name)
    VALUES (log_newemp_seq.NEXTVAL, user_id,
sysdate,p_first_name||' '||p_last_name);
    COMMIT;
END audit_newemp;

BEGIN -- add_employee
    IF valid_deptid(p_deptid) THEN
        audit_newemp;
        INSERT INTO employees(employee_id, first_name,
last_name, email,
job_id, manager_id, hire_date, salary, commission_pct,
department_id)
        VALUES (employees_seq.NEXTVAL, p_first_name,
p_last_name, p_email,
p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm,
p_deptid);
    ELSE
        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
    END IF;
END add_employee;

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE) IS
    p_email employees.email%type;
BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1)||SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid
=> p_deptid);
END;

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id

```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```

INTO p_sal, p_job
FROM employees
WHERE employee_id = p.empid;
END get_employee;

FUNCTION get_employee(p_emp_id employees.employee_id%type)
  return employees%rowtype IS
  rec_emp employees%rowtype;
BEGIN
  SELECT * INTO rec_emp
  FROM employees
  WHERE employee_id = p.emp_id;
  RETURN rec_emp;
END;

FUNCTION get_employee(p_family_name
employees.last_name%type)
  return employees%rowtype IS
  rec_emp employees%rowtype;
BEGIN
  SELECT * INTO rec_emp
  FROM employees
  WHERE last_name = p.family_name;
  RETURN rec_emp;
END;

PROCEDURE get_employees(p_dept_id
employees.department_id%type) IS
BEGIN
  SELECT * BULK COLLECT INTO emp_table
  FROM EMPLOYEES
  WHERE department_id = p_dept_id;
END;

PROCEDURE init_departments IS
BEGIN
  FOR rec IN (SELECT department_id FROM departments)
  LOOP
    valid_departments(rec.department_id) := TRUE;
  END LOOP;
END;

PROCEDURE print_employee(p_rec_emp employees%rowtype) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(p_rec_emp.department_id || ' '
  p_rec_emp.employee_id|| ' '
  p_rec_emp.first_name|| ' '
  p_rec_emp.last_name|| ' '
  p_rec_emp.job_id|| ' '
  p_rec_emp.salary);
END;

```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```

PROCEDURE show_employees IS
BEGIN
    IF emp_table IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('Employees in Package table');
        FOR i IN 1 .. emp_table.COUNT
        LOOP
            print_employee(emp_table(i));
        END LOOP;
    END IF;
END show_employees;

FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
BEGIN
    RETURN valid_departments.exists(p_deptid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;

/* New set_salary procedure */

PROCEDURE set_salary(p_jobid VARCHAR2, p_min_salary NUMBER) IS
    CURSOR cur_emp IS
        SELECT employee_id
        FROM employees
        WHERE job_id = p_jobid AND salary < p_min_salary;
BEGIN
    FOR rec_emp IN cur_emp
    LOOP
        UPDATE employees
        SET salary = p_min_salary
        WHERE employee_id = rec_emp.employee_id;
    END LOOP;
END set_salary;

BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
PACKAGE emp_pkg Compiled.
No Errors.
PACKAGE BODY emp_pkg Compiled.
No Errors.
```

- b) Create a row trigger named UPD_MINSALARY_TRG on the JOBS table that invokes the EMP_PKG.SET_SALARY procedure, when the minimum salary in the JOBS table is updated for a specified job ID.

Open the /home/oracle/labs/plpu/solns/sol_09_01_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER upd_minsalary_trg
AFTER UPDATE OF min_salary ON JOBS
FOR EACH ROW
BEGIN
    emp_pkg.set_salary(:new.job_id, :new.min_salary);
END;
/
SHOW ERRORS
```

```
TRIGGER upd_minsalary_trg Compiled.
No Errors.
```

- c) Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their JOB_ID is 'IT_PROG'. Then, update the minimum salary in the JOBS table to increase it by \$1,000. What happens?

Open the /home/oracle/labs/plpu/solns/sol_09_01_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
SET min_salary = min_salary + 1000
```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
WHERE job_id = 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	6000
105	Austin	4800
106	Pataballa	4800
107	Lorentz	4200
214	Beh	9000

6 rows selected

Error starting at line 5 in command:

```
UPDATE jobs
   SET min_salary = min_salary + 1000
 WHERE job_id = 'IT_PROG'
Error report:
ORA-04091: table ORA61.JOB$ is mutating, trigger/function may not see it
ORA-06512: at "ORA61.CHECK_SALARY", line 5
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
ORA-06512: at "ORA61.EMP_PKG", line 143
ORA-06512: at "ORA61.UPD_MINSALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.UPD_MINSALARY_TRG'
04091. 00000 - "table %s.%s is mutating, trigger/function may not see it"
*Cause:    A trigger (or a user-defined plsql function that is referenced in
           this statement) attempted to look at (or modify) a table that was
           in the middle of being modified by the statement which fired it.
*Action:   Rewrite the trigger (or function) so it does not read that table.
```

The update of the `min_salary` column for job 'IT_PROG' fails because the `UPD_MINSALARY_TRG` trigger on the `JOB$` table attempts to update the employees' salaries by calling the `EMP_PKG.SET_SALARY` procedure. The `SET_SALARY` procedure causes the `CHECK_SALARY_TRG` trigger to fire (a cascading effect). The `CHECK_SALARY_TRG` calls the `CHECK_SALARY` procedure, which attempts to read the `JOB$` table data. While reading the `JOB$` table, the `CHECK_SALARY` procedure encounters the mutating table exception.

- 2) To resolve the mutating table issue, create a `JOB$_PKG` package to maintain in memory a copy of the rows in the `JOB$` table. Next, modify the `CHECK_SALARY` procedure to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, you must create a `BEFORE INSERT OR UPDATE` statement trigger on the `EMPLOYEES` table to initialize the `JOB$_PKG` package state before the `CHECK_SALARY` row trigger is fired.
 - a) Create a new package called `JOB$_PKG` with the following specification:

```
PROCEDURE initialize;
FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```

FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(jobid VARCHAR2,min_salary
NUMBER);
PROCEDURE set_maxsalary(jobid VARCHAR2,max_salary
NUMBER);

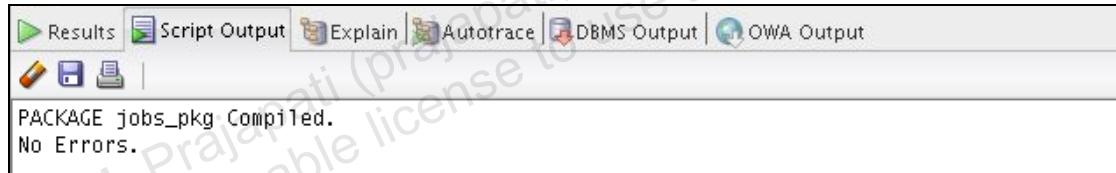
```

Open the `sol_09_02_a.sql` file in the `/home/oracle/labs/plpu/solns` folder, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

CREATE OR REPLACE PACKAGE jobs_pkg IS
  PROCEDURE initialize;
  FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER;
  FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER;
  PROCEDURE set_minsalary(p_jobid VARCHAR2, p_min_salary
NUMBER);
  PROCEDURE set_maxsalary(p_jobid VARCHAR2, p_max_salary
NUMBER);
END jobs_pkg;
/
SHOW ERRORS

```



b) Implement the body of JOBS_PKG as follows:

- i. Declare a private PL/SQL index-by table called `jobs_tab_type` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.
- ii. Declare a private variable called `jobstab` based on the `jobs_tab_type`.
- iii. The `INITIALIZE` procedure reads the rows in the `JOBS` table by using a cursor loop, and uses the `JOB_ID` value for the `jobstab` index that is assigned its corresponding row.
- iv. The `GET_MINSalary` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `min_salary` for that element.
- v. The `GET_MAXSalary` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `max_salary` for that element.

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

- vi. The SET_MINSALARY procedure uses its p_jobid as an index to the jobstab to set the min_salary field of its element to the value in the min_salary parameter.
- vii. The SET_MAXSALARY procedure uses its p_jobid as an index to the jobstab to set the max_salary field of its element to the value in the max_salary parameter.

Open the /home/oracle/labs/plpu/solns/sol_09_02_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package's body, right-click the package's name or body in the Object Navigator tree, and then select Compile.

```

CREATE OR REPLACE PACKAGE BODY jobs_pkg IS
  TYPE jobs_tab_type IS TABLE OF jobs%rowtype
    INDEX BY jobs.job_id%type;
  jobstab jobs_tab_type;

  PROCEDURE initialize IS
  BEGIN
    FOR rec_job IN (SELECT * FROM jobs)
    LOOP
      jobstab(rec_job.job_id) := rec_job;
    END LOOP;
  END initialize;

  FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(p_jobid).min_salary;
  END get_minsalary;

  FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(p_jobid).max_salary;
  END get_maxsalary;

  PROCEDURE set_minsalary(p_jobid VARCHAR2, p_min_salary
NUMBER) IS
  BEGIN
    jobstab(p_jobid).max_salary := p_min_salary;
  END set_minsalary;

  PROCEDURE set_maxsalary(p_jobid VARCHAR2, p_max_salary
NUMBER) IS
  BEGIN
    jobstab(p_jobid).max_salary := p_max_salary;
  END set_maxsalary;

END jobs_pkg;

```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

```
/  
SHOW ERRORS
```

PACKAGE BODY jobs_pkg Compiled.
No Errors.

- c) Copy the CHECK_SALARY procedure from Practice 8, Exercise 1a, and modify the code by replacing the query on the JOBS table with statements to set the local minsal and maxsal variables with values from the JOBS_PKG data by calling the appropriate GET_*SALARY functions. This step should eliminate the mutating trigger exception.

Open the /home/oracle/labs/plpu/solns/sol_09_02_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job
VARCHAR2, p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN
  /*
   ** Commented out to avoid mutating trigger exception on
   the JOBS table
  SELECT min_salary, max_salary INTO v_minsal, v_maxsal
  FROM jobs
  WHERE job_id = UPPER(p_the_job);
  */

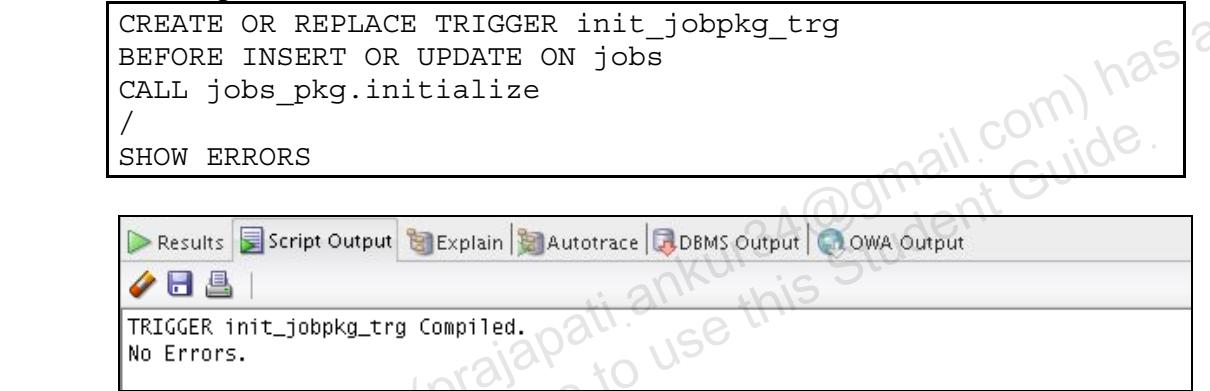
  v_minsal := jobs_pkg.get_minsalary(UPPER(p_the_job));
  v_maxsal := jobs_pkg.get_maxsalary(UPPER(p_the_job));
  IF p_the_salary NOT BETWEEN v_minsal AND v_maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $'||p_the_salary||'. ' ||
      'Salaries for job '|| p_the_job ||
      ' must be between $'|| v_minsal ||' and $' ||
      v_maxsal);
    END IF;
  END;
/
SHOW ERRORS
```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

- d) Implement a BEFORE INSERT OR UPDATE statement trigger called INIT_JOBPKG_TRG that uses the CALL syntax to invoke the JOBS_PKG.INITIALIZE procedure to ensure that the package state is current before the DML operations are performed.

Open the /home/oracle/labs/plpu/solns/sol_09_02_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER init_jobpkg_trg
BEFORE INSERT OR UPDATE ON jobs
CALL jobs_pkg.initialize
/
SHOW ERRORS
```



- e) Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the IT_PROG job type by 1,000 in the JOBS table. Follow this up with a query on the employees with the IT_PROG job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?

Open the /home/oracle/labs/plpu/solns/sol_09_02_e.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG';

SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';
```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

EMPLOYEE_ID LAST_NAME SALARY

103 Hunold 9000
104 Ernst 6000
105 Austin 4800
106 Pataballa 4800
107 Lorentz 4200
214 Beh 9000

6 rows selected

1 rows updated

EMPLOYEE_ID LAST_NAME SALARY

103 Hunold 9000
104 Ernst 6000
105 Austin 5000
106 Pataballa 5000
107 Lorentz 5000
214 Beh 9000

6 rows selected

The employees with last names Austin, Pataballa, and Lorentz have all had their salaries updated. No exception occurred during this process, and you implemented a solution for the mutating table trigger exception.

- 3) Because the CHECK_SALARY procedure is fired by CHECK_SALARY_TRG before inserting or updating an employee, you must check whether this still works as expected.
- Test this by adding a new employee using EMP_PKG.ADD_EMPLOYEE with the following parameters: ('Steve', 'Morse', 'SMORSE', and sal => 6500). What happens?

Open the /home/oracle/labs/plpu/solns/sol_09_03_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal
=> 6500)
```

Practice Solutions 9-1: Managing Data Integrity Rules and Mutating Table Exceptions (continued)

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for edit, save, and print. The main area displays the message "anonymous block completed".

- b) To correct the problem encountered when adding or updating an employee:
- Create a BEFORE INSERT OR UPDATE statement trigger called EMPLOYEE_INITJOBS_TRG on the EMPLOYEES table that calls the JOBS_PKG.INITIALIZE procedure.
 - Use the CALL syntax in the trigger body.

Open the /home/oracle/labs/plpu/solns/sol_09_03_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE TRIGGER employee_initjobs_trg
BEFORE INSERT OR UPDATE OF job_id, salary ON employees
CALL jobs_pkg.initialize
/
```

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for edit, save, and print. The main area displays the message "TRIGGER employee_initjobs_trg Compiled."

- c) Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the EMPLOYEES table by displaying the employee ID, first and last names, salary, job ID, and department ID.

Open the /home/oracle/labs/plpu/solns/sol_09_03_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top includes tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are icons for edit, save, and print. The main area displays an error report and a query result.

Error starting at line 1 in command:
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal => 6500)
Error report:
ORA-00001: unique constraint (DRA61.EMP_EMAIL_UK) violated
ORA-06512: at "DRA61.EMP_PKG", line 35
ORA-06512: at line 1
00001. 00000 - "unique constraint (%s.%s) violated"
***Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.**
For Trusted Oracle configured in DBMS MAC mode, you may see
this message if a duplicate entry exists at a different level.
***Action: Either remove the unique restriction or do not insert the key.**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	DEPARTMENT_ID
215	Steve	Morse	6500	SA_REP	30

1 rows selected

Practices and Solutions for Lesson 10

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

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 10-1: Using the PL/SQL Compiler Parameters and Warnings

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

- 1) Create and run a `lab_10_01` script to display the following information about compiler-initialization parameters by using the `USER_PLSQL_OBJECT_SETTINGS` data dictionary view. Note the settings for the `ADD_JOB_HISTORY` object.
Note: Use the Execute Statement (F9) icon to display the results in the Results tab.
 - a) Object name
 - b) Object type
 - c) The object's compilation mode
 - d) The compilation optimization level
- 2) Alter the `PLSQL_CODE_TYPE` parameter to enable native compilation for your session, and compile `ADD_JOB_HISTORY`.
 - a) Execute the `ALTER SESSION` command to enable native compilation for the session.
 - b) Compile the `ADD_JOB_HISTORY` procedure.
 - c) Rerun the `sol_10_01` script. Note the `PLSQL_CODE_TYPE` parameter.
 - d) Switch compilation to use interpreted compilation mode as follows:
- 3) Use the Tools > Preferences > PL/SQL Compiler Options region to disable all compiler warnings categories.
- 4) Edit, examine, and execute the `lab_10_04.sql` script to create the `UNREACHABLE_CODE` procedure. Click the Run Script icon (F5) to create the procedure. Use the procedure name in the Navigation tree to compile the procedure.
- 5) What are the compiler warnings that are displayed in the Compiler – Log tab, if any?
- 6) Enable all compiler-warning messages for this session using the Preferences window.
- 7) Recompile the `UNREACHABLE_CODE` procedure using the Object Navigation tree. What compiler warnings are displayed, if any?
- 8) Use the `USER_ERRORS` data dictionary view to display the compiler-warning messages details as follows.
- 9) Create a script named `warning_msgs` that uses the `EXECUTE DBMS_OUTPUT` and the `DBMS_WARNING` packages to identify the categories for the following compiler-warning message numbers: 5050, 6075, and 7100. Enable `SERVERTOUTPUT` before running the script.

Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings

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

- 1) Create and run a lab_10_01 script to display the following information about compiler-initialization parameters by using the `USER_PLSQL_OBJECT_SETTINGS` data dictionary view. Note the settings for the `ADD_JOB_HISTORY` object.

Note: Use the Execute Statement (F9) icon to display the results in the Results tab.

- a) Object name
- b) Object type
- c) The object's compilation mode
- d) The compilation optimization level

Open the `/home/oracle/labs/plpu/sols/sol_10_01.sql` script.

Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to run the query. The code and a sample of the results are shown below.

```
SELECT name, type, plsql_code_type as code_type,
       plsql_optimize_level as opt_lvl
  FROM user_plsql_object_settings;
```

NAME	TYPE	CODE_TYPE
ADD_EMPLOYEE	PROCEDURE	INTERPRETED
ADD_JOB	PROCEDURE	INTERPRETED
ADD_JOB_HISTORY	PROCEDURE	INTERPRETED
CHECK_SALARY	PROCEDURE	INTERPRETED
CHECK_SALARY_TRG	TRIGGER	INTERPRETED
COMPILE_PKG	PACKAGE	INTERPRETED
COMPILE_PKG	PACKAGE BODY	INTERPRETED
DELETE_EMP_TRG	TRIGGER	INTERPRETED
DEL_JOB	PROCEDURE	INTERPRETED
EMPLOYEE_INITJOBS_TRG	TRIGGER	INTERPRETED
EMPLOYEE_REPORT	PROCEDURE	INTERPRETED
EMP_LIST	PROCEDURE	INTERPRETED
EMP_PKG	PACKAGE	INTERPRETED
EMP_PKG	PACKAGE BODY	INTERPRETED
GET_ANNUAL_COMP	FUNCTION	INTERPRETED
GET_EMPLOYEE	PROCEDURE	INTERPRETED
GET_JOB	FUNCTION	INTERPRETED
GET_LOCATION	FUNCTION	INTERPRETED
INIT_JOBPKG_TRG	TRIGGER	INTERPRETED
JOB_PKG	PACKAGE	INTERPRETED
JOB_PKG	PACKAGE BODY	INTERPRETED
JOB_PKG	PACKAGE	INTERPRETED

...

Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)

- 2) Alter the PLSQL_CODE_TYPE parameter to enable native compilation for your session, and compile ADD_JOB_HISTORY.

- a) Execute the ALTER SESSION command to enable native compilation for the session.

Open the /home/oracle/labs/plpu/solns/sol_10_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

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

ALTER SESSION SET succeeded.

- b) Compile the ADD_JOB_HISTORY procedure.

Open the /home/oracle/labs/plpu/solns/sol_10_02_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
ALTER PROCEDURE add_job_history COMPILE;
```

ALTER PROCEDURE add_job_history succeeded.

- c) Rerun the sol_10_01.sql script. Note the PLSQL_CODE_TYPE parameter.

```
SELECT name, type, plsql_code_type as code_type,
plsql_optimize_level as opt_lvl
FROM user_plsql_object_settings;
```

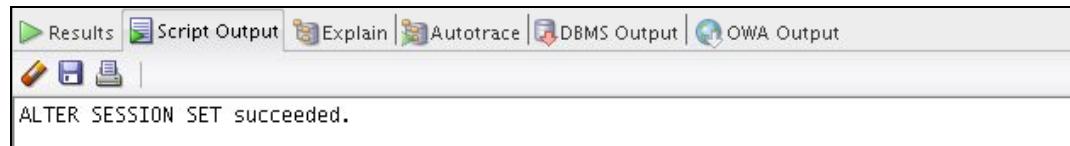
NAME	TYPE	CODE_TYPE
ADD_EMPLOYEE	PROCEDURE	INTERPRETED
ADD_JOB	PROCEDURE	INTERPRETED
ADD_JOB_HISTORY	PROCEDURE	NATIVE
CHECK_SALARY	PROCEDURE	INTERPRETED
CHECK_SALARY TRG	TRIGGER	INTERPRETED

...

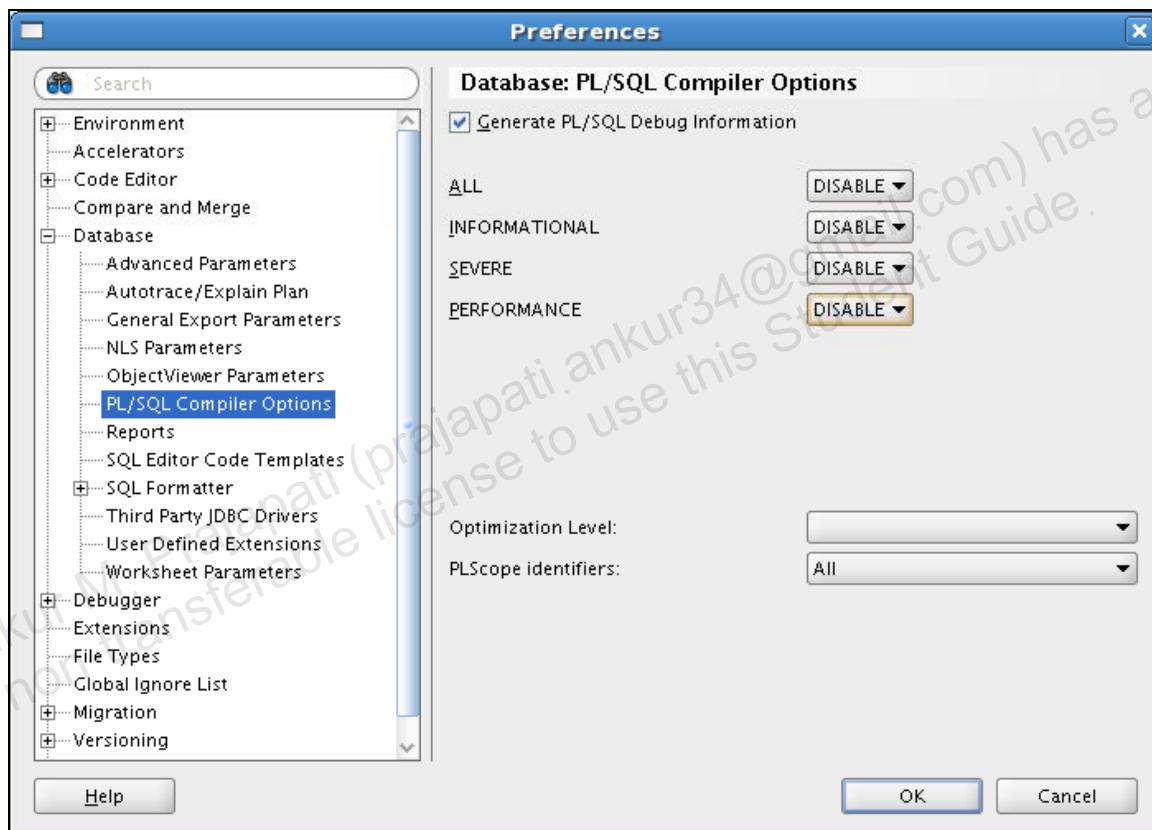
Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)

- d) Switch compilation to use interpreted compilation mode as follows:

```
ALTER SESSION SET PLSQL_CODE_TYPE = 'INTERPRETED';
```



- 3) Use the Tools > Preferences > PL/SQL Compiler Options region to disable all compiler warnings categories.



Select DISABLE for all four PL/SQL compiler warnings categories, and then click OK.

- 4) Edit, examine, and execute the `lab_10_04.sql` script to create the `UNREACHABLE_CODE` procedure. Click the Run Script icon (F5) to create and compile the procedure.

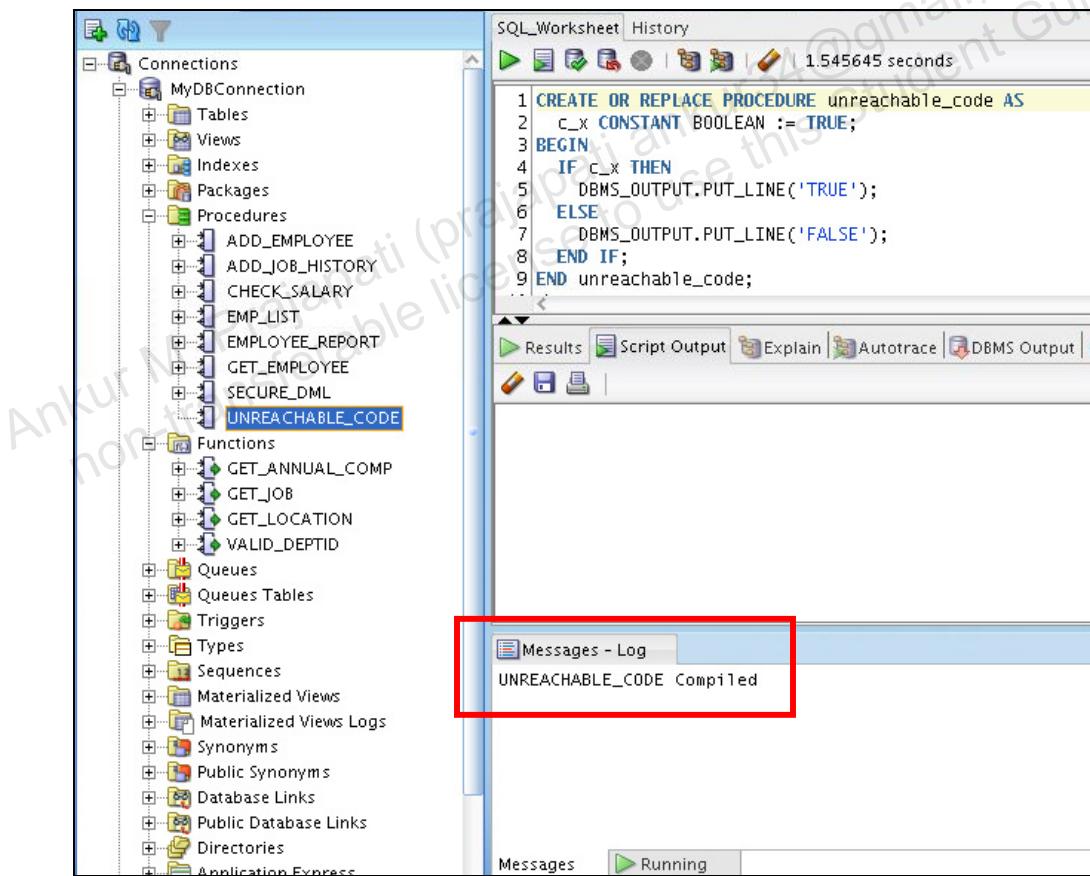
Open the `/home/oracle/labs/plpu/solns/sol_10_04.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)

```
CREATE OR REPLACE PROCEDURE unreachable_code AS
  c_x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF c_x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/
```



To view any compiler warning errors, right-click the procedure's name in the Procedures node in the Navigation tree, and then click Compile.



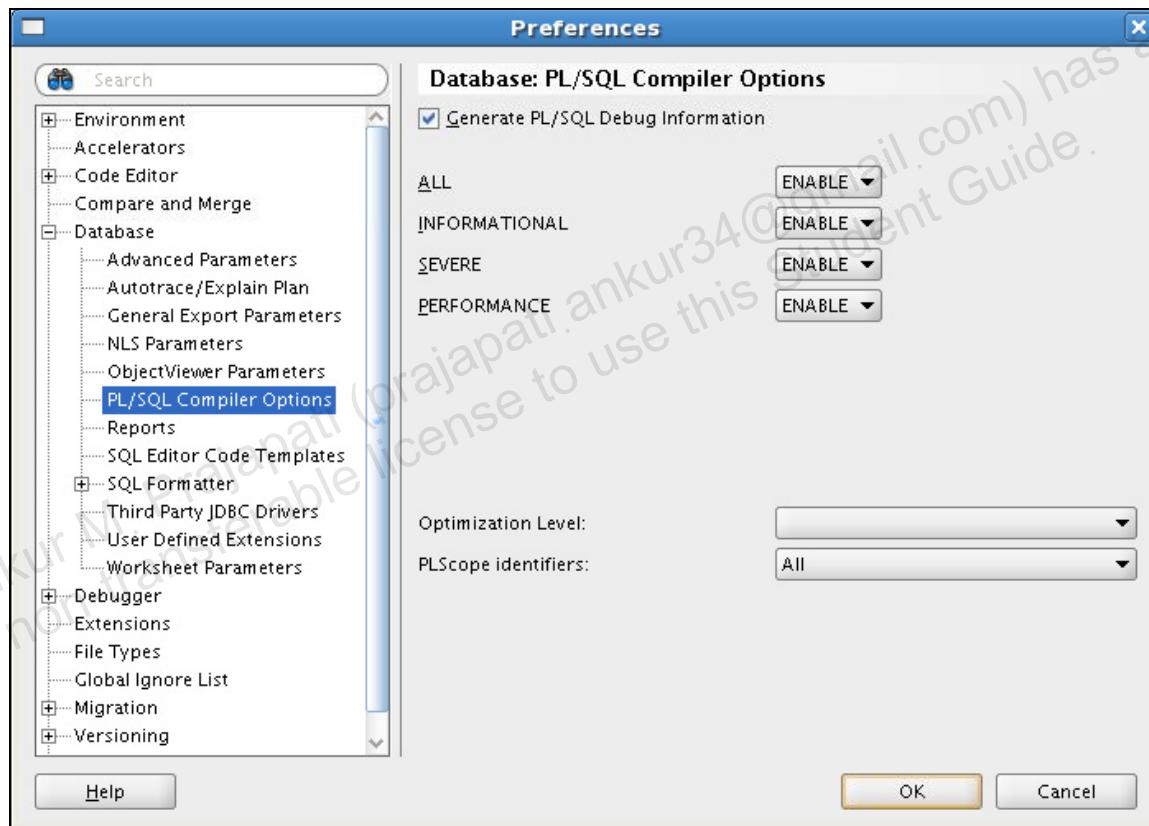
Note

- If the procedure is not displayed in the Navigation tree, click the Refresh icon in the Connections tab.

Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)

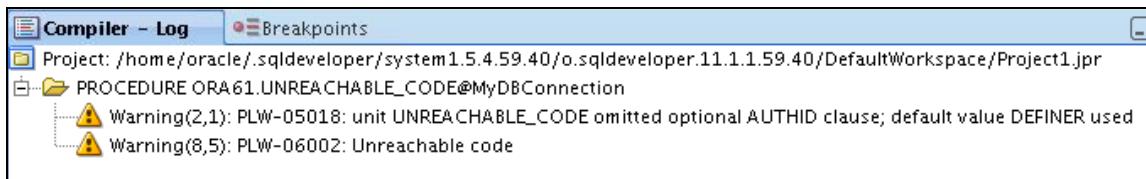
- Make sure your Messages – Log tab is displayed (select View > Log from the menu bar).
- 5) What are the compiler warnings that are displayed in the Compiler – Log tab, if any?
Note that the message in the Messages – Log tab is “UNREACHABLE_CODE Compiled” without any warning messages because you disabled the compiler warnings in step 3.
- 6) Enable all compiler-warning messages for this session using the Preferences window.

Select **ENABLE** for all four PL/SQL compiler warnings, and then click **OK**.



- 7) Recompile the UNREACHABLE_CODE procedure using the Object Navigation tree. What compiler warnings are displayed, if any?
Right-click the procedure's name in the Object Navigation tree, and then select Compile. Note the messages displayed in the Messages and Compiler subtabs in the Compiler – Log tab.

Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)



Note: If you get the following two warnings in SQL Developer, it is expected in some versions of SQL Developer. If you do get the following warnings, it is because your version of SQL Developer still uses the Oracle 11g database deprecated PLSQL_DEBUG parameter.

```
Warning (1) :PLW-06015:parameter PLSQL_DEBUG is deprecated
; use PLSQL_OPTIMIZE_LEVEL=1
```

```
Warning (1) :PLW-06013:deprecated parameter PLSQL_DEBUG
forces PLSQL_OPTIMIZE_LEVEL<=1
```

- 8) Use the USER_ERRORS data dictionary view to display the compiler-warning messages details as follows.

```
DESCRIBE user_errors
```

```
DESCRIBE user_errors
Name          Null    Type
-----        -----
NAME          NOT NULL VARCHAR2(30)
TYPE          VARCHAR2(12)
SEQUENCE      NOT NULL NUMBER
LINE          NOT NULL NUMBER
POSITION      NOT NULL NUMBER
TEXT          NOT NULL VARCHAR2(4000)
ATTRIBUTE     VARCHAR2(9)
MESSAGE_NUMBER NUMBER

8 rows selected
```

```
SELECT *
FROM user_errors;
```

Results:

	NAME	TYPE	SEQUENCE	LINE	POSITION	TEXT	ATTRIBUTE	MESSAGE_NUMBER
1	UNREACHABLE_CODE PROCEDURE		1	1	1	PLW-05018: unit UNREACHABLE_CODE omitted optional AUTHID clause; def...	WARNING	5018
2	UNREACHABLE_CODE PROCEDURE		2	0	0	PLW-06015: parameter PLSQL_DEBUG is deprecated; use PLSQL_OPTIMIZE_L...	WARNING	6015
3	UNREACHABLE_CODE PROCEDURE		3	0	0	PLW-06013: deprecated parameter PLSQL_DEBUG forces PLSQL_OPTIMIZE_L...	WARNING	6013
4	UNREACHABLE_CODE PROCEDURE		4	7	5	PLW-06002: Unreachable code	WARNING	6002

...

Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)

Note: The output was displayed in the Results tab because we used the F9 key to execute the SELECT statement. The results of the SELECT statement might be different depending on the amount of errors you had in your session.

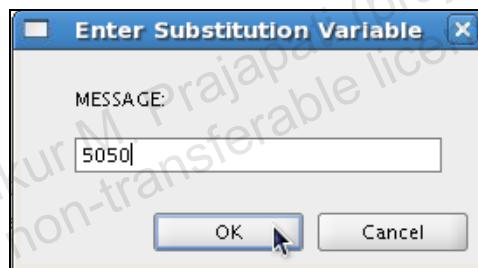
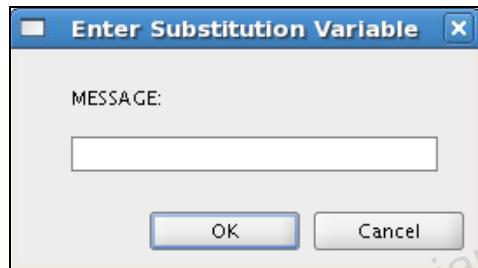
- 9) Create a script named warning_msgs that uses the EXECUTE DBMS_OUTPUT and the DBMS_WARNING packages to identify the categories for the following compiler-warning message numbers: 5050, 6075, and 7100. Enable SERVEROUTPUT before running the script.

Open the /home/oracle/labs/plpu/solns/sol_10_09.sql script.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the query.

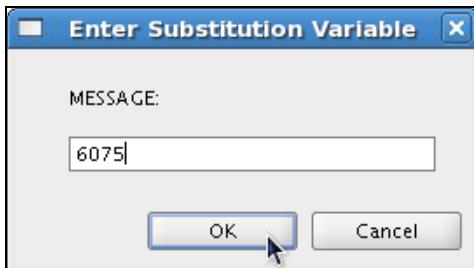
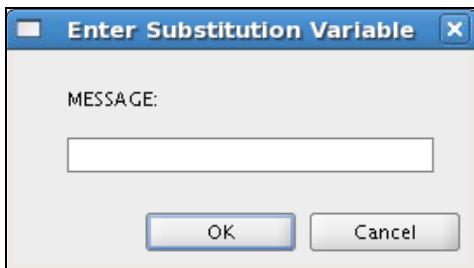
The code and the results are shown below.

```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message)) ;
```

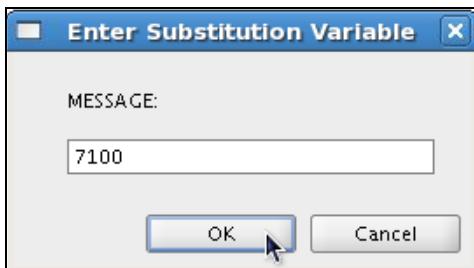
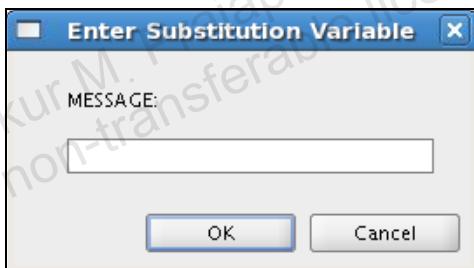


```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message)) ;
```

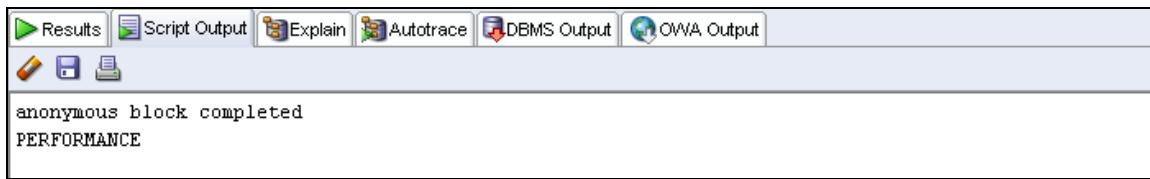
Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)



```
EXECUTE  
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```



Practice Solutions 10-1: Using the PL/SQL Compiler Parameters and Warnings (continued)



Practices and Solutions for Lesson 11

In this practice, you create a package and a procedure that use conditional compilation. In addition, you use the appropriate package to retrieve the post-processed source text of the PL/SQL unit. You also obfuscate some PL/SQL code.

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 11-1: Using Conditional Compilation

In this practice, you create a package and a procedure that use conditional compilation. In addition, you use the appropriate package to retrieve the postprocessed source text of the PL/SQL unit. You also obfuscate some PL/SQL code.

- 1) Examine and then execute the `lab_11_01.sql` script. This script sets flags for displaying debugging code and tracing information. The script also creates the `my_pkg` package and the `circle_area` procedure.
- 2) Use the `DBMS_PREPROCESSOR` subprogram to retrieve the postprocessed source text of the PL/SQL unit after processing the conditional compilation directives from `lab_11_01`. Enable `SERVERTOUTPUT`.
- 3) Create a PL/SQL script that uses the `DBMS_DB_VERSION` constant with conditional compilation. The code should test for the Oracle database version:
 - a) If the database version is less than or equal to 10.1, it should display the following error message:
`Unsupported database release.`
 - b) If the database version is 11.1 or higher, it should display the following message:
`Release 11.1 is supported.`
- 4) Consider the following code in the `lab_11_04.sql` script that uses `CREATE_WWRAPPED` to dynamically create and wrap a package specification and a package body in a database. Edit the `lab_11_04.sql` script to add the needed code to obfuscate the PL/SQL code. Save and then execute the script.

```

DECLARE
  -- the package_text variable contains the text to create
  -- the package spec and body
  package_text VARCHAR2(32767);
  FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2
AS
BEGIN
  RETURN 'CREATE PACKAGE ' || pkgname || ' AS
    PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
    PROCEDURE fire_employee (emp_id NUMBER);
  END ' || pkgname || ';';
END generate_spec;
FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2
AS
BEGIN
  RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
    PROCEDURE raise_salary (emp_id NUMBER, amount
NUMBER) IS
    BEGIN
      UPDATE employees SET salary = salary + amount
      WHERE employee_id = emp_id;
    END raise_salary;
  END';
END;
  
```

Practice 11-1: Using Conditional Compilation (continued)

```
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
    DELETE FROM employees WHERE employee_id = emp_id;
END fire_employee;
END ' || pkgname || ';' ;
END generate_body;
```

- a) Generate the package specification while passing the emp_actions parameter.
- b) Create and wrap the package specification.
- c) Generate the package body.
- d) Create and wrap the package body.
- e) Call a procedure from the wrapped package as follows:
CALL emp_actions.raise_salary(120, 100);
- f) Use the USER_SOURCE data dictionary view to verify that the code is hidden as follows:

```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

Practice Solutions 11-1: Using Conditional Compilation

In this practice, you create a package and a procedure that use conditional compilation. In addition, you use the appropriate package to retrieve the postprocessed source text of the PL/SQL unit. You also obfuscate some PL/SQL code.

- 1) Examine and then execute the `lab_11_01.sql` script. This script sets flags for displaying debugging code and tracing information. The script also creates the `my_pkg` package and the `circle_area` procedure.

Open the `/home/oracle/labs/plpu/solns/sol_11_01.sql` script.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script.

The code and the results are shown below.

```
ALTER SESSION SET PLSQL_CCFLAGS = 'my_debug:FALSE,
my_tracing:FALSE';

CREATE OR REPLACE PACKAGE my_pkg AS
  SUBTYPE my_real IS
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN NUMBER;
-- check database version
    $ELSE                                BINARY_DOUBLE;
    $END
    my_pi my_real; my_e my_real;
  END my_pkg;
/
CREATE OR REPLACE PACKAGE BODY my_pkg AS
BEGIN
  $IF DBMS_DB_VERSION.VERSION < 10 $THEN
    my_pi := 3.14016408289008292431940027343666863227;
    my_e  := 2.71828182845904523536028747135266249775;
  $ELSE
    my_pi := 3.14016408289008292431940027343666863227d;
    my_e  := 2.71828182845904523536028747135266249775d;
  $END
END my_pkg;
/

CREATE OR REPLACE PROCEDURE circle_area(radius my_pkg.my_real)
IS
  my_area my_pkg.my_real;
  my_datatype VARCHAR2(30);
BEGIN
  my_area := my_pkg.my_pi * radius * radius;
  DBMS_OUTPUT.PUT_LINE('Radius: ' || TO_CHAR(radius)

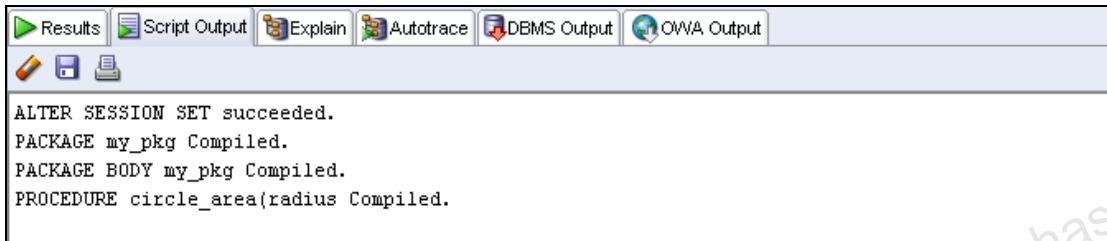
                           || ' Area: ' || TO_CHAR(my_area) );
  $IF $$my_debug $THEN
-- if my_debug is TRUE, run some debugging code
```

Practice Solutions 11-1: Using Conditional Compilation (continued)

```

SELECT DATA_TYPE INTO my_datatype FROM USER_ARGUMENTS
  WHERE OBJECT_NAME = 'CIRCLE_AREA' AND ARGUMENT_NAME =
'RADIUS';
    DBMS_OUTPUT.PUT_LINE('Datatype of the RADIUS argument is:
' || my_datatype);
$END
END;
/

```



ALTER SESSION SET succeeded.
PACKAGE my_pkg Compiled.
PACKAGE BODY my_pkg Compiled.
PROCEDURE circle_area(radius Compiled).

- 2) Use the DBMS_PREPROCESSOR subprogram to retrieve the postprocessed source text of the PL/SQL unit after processing the conditional compilation directives from lab_11_01. Enable SERVEROUTPUT.

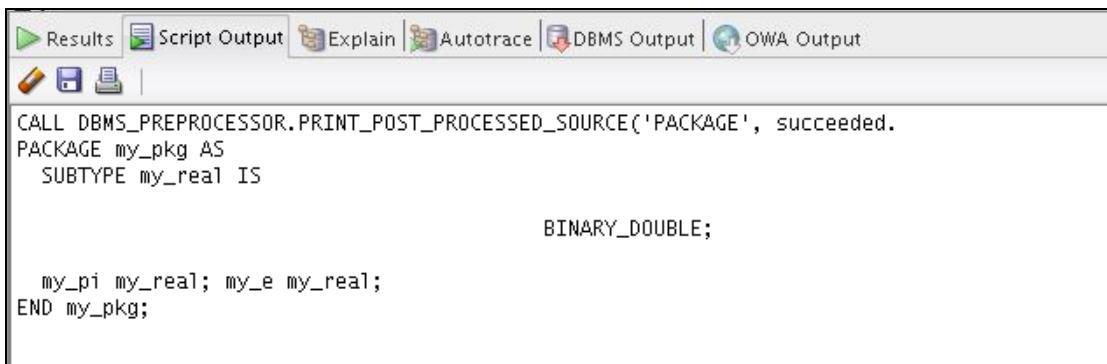
Open the /home/oracle/labs/plpu/sols/sol_11_02.sql script.
Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script.
The code and the results are shown below.

```

-- The code example assumes you are the student with the
-- account ora70. Substitute ora70 with your account
-- information.
SET SERVEROUTPUT ON

CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE ('PACKAGE',
'ORA61', 'MY_PKG');

```



CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE('PACKAGE', succeeded).
PACKAGE my_pkg AS
SUBTYPE my_real IS
 BINARY_DOUBLE;
 my_pi my_real; my_e my_real;
END my_pkg;

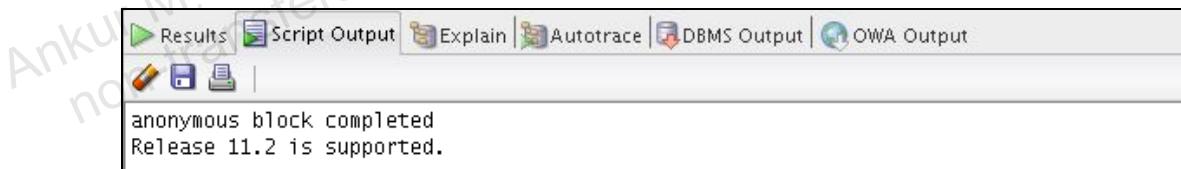
Practice Solutions 11-1: Using Conditional Compilation (continued)

- 3) Create a PL/SQL script that uses the DBMS_DB_VERSION constant with conditional compilation. The code should test for the Oracle database version:
- If the database version is less than or equal to 10.1, it should display the following error message:
Unsupported database release.
 - If the database version is 11.1 or higher, it should display the following message:
Release 11.2 is supported.

**Open the /home/oracle/labs/plpu/solns/sol_11_03.sql script.
Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.**

```
BEGIN
$IF DBMS_DB_VERSION.VER_LE_10_1 $THEN
$ERROR 'unsupported database release.' $END

$ELSE
    DBMS_OUTPUT.PUT_LINE ('Release ' ||
DBMS_DB_VERSION.VERSION || '.' ||
                                DBMS_DB_VERSION.RELEASE || ' is
supported.');
    -- Note that this COMMIT syntax is newly supported in
10.2
    COMMIT WRITE IMMEDIATE NOWAIT;
$END
END;
/
```



- 4) Consider the following code in the lab_11_04.sql script that uses CREATE_WWRAPPED to dynamically create and wrap a package specification and a package body in a database. Edit the lab_11_04.sql script to add the needed code to obfuscate the PL/SQL code. Save and then execute the script.

```
DECLARE
-- the package_text variable contains the text to create
-- the package spec and body
package_text VARCHAR2(32767);
FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2
AS
BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AS
```

Practice Solutions 11-1: Using Conditional Compilation (continued)

```

PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
PROCEDURE fire_employee (emp_id NUMBER);
END ' || pkgname || ';';
END generate_spec;
FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2
AS
BEGIN
    RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
        PROCEDURE raise_salary (emp_id NUMBER, amount
NUMBER) IS
            BEGIN
                UPDATE employees SET salary = salary + amount
WHERE employee_id = emp_id;
            END raise_salary;

        PROCEDURE fire_employee (emp_id NUMBER) IS
            BEGIN
                DELETE FROM employees WHERE employee_id = emp_id;
            END fire_employee;
        END ' || pkgname || ';';
END generate_body;

```

- Generate the package specification while passing the emp_actions parameter.
- Create and wrap the package specification.
- Generate the package body.
- Create and wrap the package body.
- Call a procedure from the wrapped package as follows:

```
CALL emp_actions.raise_salary(120, 100);
```

- Use the USER_SOURCE data dictionary view to verify that the code is hidden as follows:

```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

**Open the /home/oracle/labs/plpu/solsns/soln_11_04.sql script.
Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.**

```

DECLARE
-- the package_text variable contains the text to create
the package spec and body
    package_text VARCHAR2(32767);
    FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2
AS
BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AS

```

Practice Solutions 11-1: Using Conditional Compilation (continued)

```

        PROCEDURE raise_salary (emp_id NUMBER, amount
NUMBER);
        PROCEDURE fire_employee (emp_id NUMBER);
END ' || pkgname || ';' ;
END generate_spec;
FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2
AS
BEGIN
    RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
        PROCEDURE raise_salary (emp_id NUMBER, amount
NUMBER) IS
        BEGIN
UPDATE employees SET salary = salary + amount WHERE
employee_id = emp_id;
        END raise_salary;
        PROCEDURE fire_employee (emp_id NUMBER) IS
        BEGIN
            DELETE FROM employees WHERE employee_id = emp_id;
        END fire_employee;
        END ' || pkgname || ';' ;
END generate_body;

BEGIN
-- generate package spec
package_text := generate_spec('emp_actions');

-- create and wrap the package spec
SYS.DBMS_DDL.CREATE_WWRAPPED(package_text);

-- generate package body
package_text := generate_body('emp_actions');

-- create and wrap the package body
SYS.DBMS_DDL.CREATE_WWRAPPED(package_text);
END;
/
-- call a procedure from the wrapped package
CALL emp_actions.raise_salary(120, 100);

-- Use the USER_SOURCE data dictionary view to verify that
-- the code is hidden as follows:

SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';

```

Practice Solutions 11-1: Using Conditional Compilation (continued)

The screenshot shows the Oracle SQL Developer interface with the following details:

- Toolbar:** Results, Script Output, Explain, Autotrace, DBMS Output, OWA Output.
- Text Area:** Displays the output of an anonymous block. The output includes:
 - anonymous block completed
 - CALL emp_actions.raise_salary(120, succeeded).
 - TEXT
 -
 - PACKAGE emp_actions wrapped
 - a000000
 - 369
 - abcd
 - 9
 - 9d b6
 - J/5HL9fHra10qRCb2WqJ2palyfEwg5m49T0f9b9cWtdi46520fpHctUruHSLCampqcqqF+qc
 - UMrqAsqAcnCxznCALMY0WkSy6pqpMK4fRJlpD0mxyqREgA+laZ0vFurrql7EExfI9npppt9Cq
 - GgrV7J9GI+okHOTIiYzH47CdD0sqHab0lBYj

Practice Solutions 11-1: Using Conditional Compilation (continued)

```
PACKAGE BODY emp_actions wrapped
a000000
369
abcd
b
178 10f
Qr07JITU1etMtbw7mSy0dcAUvfUwg/BK7cusfc/GkEIY/SqSOUbuD1vf5gVxBQTo1ARbWyTT
ELXnN+WFBi6/vrXGcnAtahjKsVEC60McY8bB0p8QpoM2QH/YmdNBvQRjY3AtYEg4ofC9Camb
Ltw0+36g6kh66sqa+W7FjtMgNnus/8Mep4WsbFsLw2+cM1kxIPQkBrwblmuW8C0a2zYwxE6n
8T7byCyPrajb88NeB6PCSLwwtaLs9dw17QS2S03CKQZEoNxM4A==

2 rows selected
```

Practices and Solutions for Lesson 12

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

Note: If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or next practice.

Practice 12-1: Managing Dependencies in Your Schema

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

- 1) Create a tree structure showing all dependencies involving your add_employee procedure and your valid_deptid function.

Note: add_employee and valid_deptid were created in the lesson titled “Creating Functions.” You can run the solution scripts for Practice 3 if you need to create the procedure and function.

- a) Load and execute the utldtree.sql script, which is located in the /home/oracle/labs/plpu/labs folder.
- b) Execute the deptree_fill procedure for the add_employee procedure.
- c) Query the IDEPTREE view to see your results.
- d) Execute the deptree_fill procedure for the valid_deptid function.
- e) Query the IDEPTREE view to see your results.

If you have time, complete the following exercise:

- 2) Dynamically validate invalid objects.
 - a) Make a copy of your EMPLOYEES table, called EMPS.
 - b) Alter your EMPLOYEES table and add the column TOTSAL with data type NUMBER(9, 2).
 - c) Create and save a query to display the name, type, and status of all invalid objects.
 - d) In the compile_pkg (created in Practice 7 of the lesson titled “Using Dynamic SQL”), add a procedure called recompile that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to alter the invalid object type and compile it.
 - e) Execute the compile_pkg.recompile procedure.
 - f) Run the script file that you created in step 3 c. to check the value of the STATUS column. Do you still have objects with an INVALID status?

Practice Solutions 12-1: Managing Dependencies in Your Schema

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

- 1) Create a tree structure showing all dependencies involving your add_employee procedure and your valid_deptid function.

Note: add_employee and valid_deptid were created in the lesson titled “Creating Functions.” You can run the solution scripts for Practice 3 if you need to create the procedure and function.

- a) Load and execute the utldtree.sql script, which is located in the /home/oracle/labs/plpu/labs folder.

Open the /home/oracle/labs/plpu/solns/utldtree.sql script.

Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

Rem
Rem $Header: utldtree.sql,v 1.2 1992/10/26 16:24:44 RKOOI Stab
$
Rem
Rem Copyright (c) 1991 by Oracle Corporation
Rem NAME
Rem deptree.sql - Show objects recursively dependent on
Rem given object
Rem DESCRIPTION
Rem This procedure, view and temp table will allow you to
Rem see all objects that are (recursively) dependent on the
Rem given object.
Rem Note: you will only see objects for which you have
Rem permission.
Rem Examples:
Rem execute deptree_fill('procedure', 'scott', 'billing');
Rem select * from deptree order by seq#;
Rem
Rem execute deptree_fill('table', 'scott', 'emp');
Rem select * from deptree order by seq#;
Rem

Rem execute deptree_fill('package body', 'scott',
Rem 'accts_payable');
Rem select * from deptree order by seq#;
Rem
Rem A prettier way to display this information than
Rem select * from deptree order by seq#;
```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

```

Rem      is
Rem      select * from ideptree;
Rem      This shows the dependency relationship via indenting.
Rem      Notice that no order by clause is needed with ideptree.
Rem      RETURNS
Rem
Rem      NOTES
Rem      Run this script once for each schema that needs this
Rem      utility.
Rem      MODIFIED      (MM/DD/YY)
Rem      rkooi        10/26/92 - owner -> schema for SQL2
Rem      glumpkin     10/20/92 - Renamed from DEPTREE.SQL
Rem      rkooi        09/02/92 - change ORU errors
Rem      rkooi        06/10/92 - add rae errors
Rem      rkooi        01/13/92 - update for sys vs. regular user
Rem      rkooi        01/10/92 - fix ideptree
Rem      rkooi        01/10/92 - Better formatting, add ideptree
view
Rem      rkooi        12/02/91 - deal with cursors
Rem      rkooi        10/19/91 - Creation

DROP SEQUENCE deptree_seq
/
CREATE SEQUENCE deptree_seq cache 200
/* cache 200 to make sequence faster */

/
DROP TABLE deptree temptab
/
CREATE TABLE deptree temptab
(
    object_id          number,
    referenced_object_id number,
    nest_level         number,
    seq#               number
)
/
CREATE OR REPLACE PROCEDURE deptree_fill (type char, schema
char, name char) IS
    obj_id number;
BEGIN
    DELETE FROM deptree temptab;
    COMMITT;
    SELECT object_id INTO obj_id FROM all_objects
        WHERE owner = upper(deptree_fill.schema)

AND    object_name  = upper(deptree_fill.name)
    AND    object_type = upper(deptree_fill.type);
    INSERT INTO deptree temptab
        VALUES(obj_id, 0, 0, 0);

```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

```

INSERT INTO deptree_temptab
    SELECT object_id, referenced_object_id,
           level, deptree_seq.nextval
      FROM public_dependency
     CONNECT BY PRIOR object_id = referenced_object_id
    START WITH referenced_object_id = deptree_fill.obj_id;
EXCEPTION
    WHEN no_data_found then
        raise_application_error(-20000, 'ORU-10013: ' ||
                                type || ' ' || schema || '.' || name || ' was not
                                found.');
END;
/
DROP VIEW deptree
/
SET ECHO ON

REM This view will succeed if current user is sys. This view
REM shows which shared cursors depend on the given object. If
REM the current user is not sys, then this view get an error
REM either about lack of privileges or about the non-existence
REM of REM table x$kglxss.

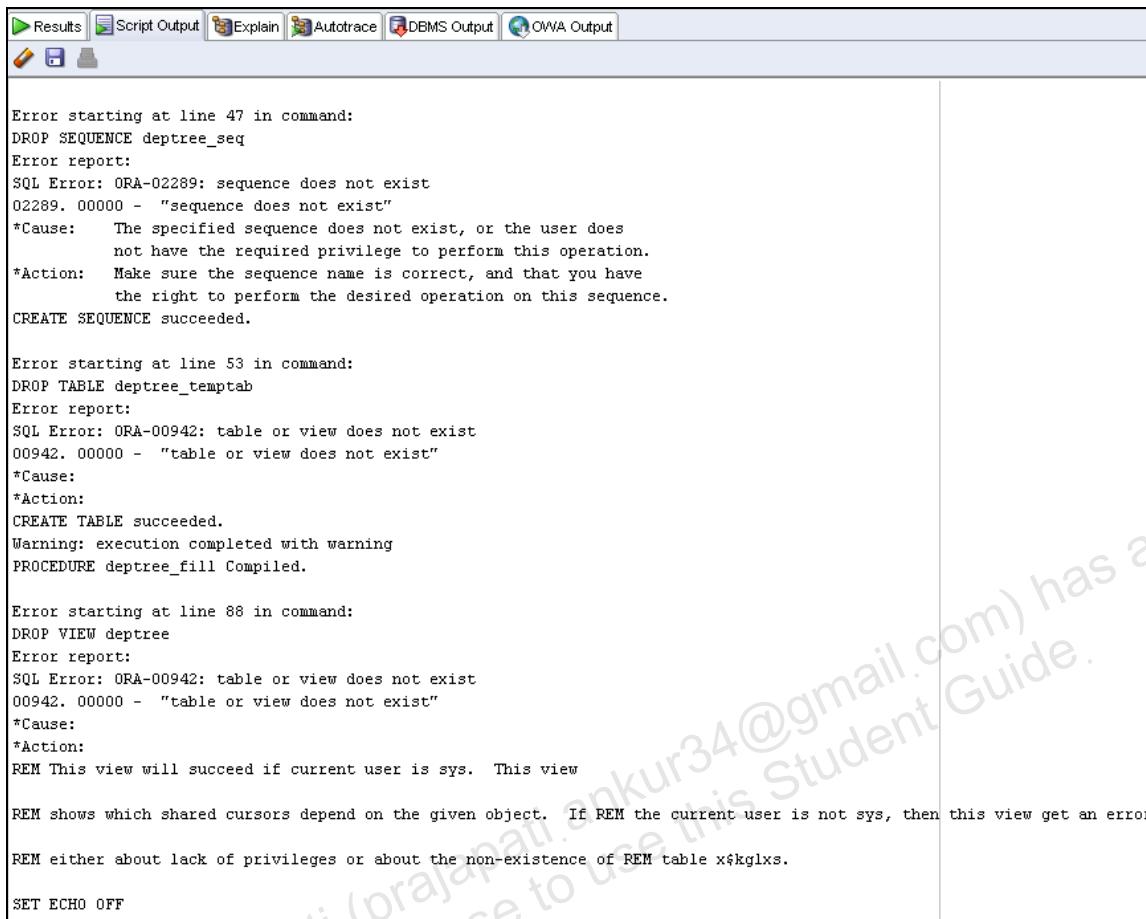
SET ECHO OFF
CREATE VIEW sys.deptree
  (nested_level, type, schema, name, seq#)
AS
  SELECT d.nest_level, o.object_type, o.owner, o.object_name,
d.seq#
  FROM deptree_temptab d, dba_objects o
 WHERE d.object_id = o.object_id (+)
UNION ALL
  SELECT d.nest_level+1, 'CURSOR', '<shared>',
'||c.kglnaobj||', d.seq#+.5
  FROM deptree_temptab d, x$kgldp k, x$kglob g, obj$ o, user$u,
x$kglob c,
x$kglxss a
 WHERE d.object_id = o.obj#
  AND o.name = g.kglnaobj
  AND o.owner# = u.user#
  AND u.name = g.kglnaown
  AND g.kglhdadr = k.kglrfhdl
  AND k.kglhdadr = a.kglhdadr /* make sure it is not a
transitive */
  AND k.kgldepno = a.kglxsdep /* reference, but a direct
one */
  AND k.kglhdadr = c.kglhdadr
  AND c.kglhdnsp = 0 /* a cursor */

```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

```
/  
  
SET ECHO ON  
  
REM This view will succeed if current user is not sys. This view  
REM does *not* show which shared cursors depend on the given  
REM object.  
REM If the current user is sys then this view will get an  
error  
REM indicating that the view already exists (since prior view  
REM create will have succeeded).  
  
SET ECHO OFF  
CREATE VIEW deptree  
  (nested_level, type, schema, name, seq#)  
AS  
  select d.nest_level, o.object_type, o.owner, o.object_name,  
d.seq#  
  FROM deptree temptab d, all_objects o  
  WHERE d.object_id = o.object_id (+)  
/  
DROP VIEW ideptree  
/  
CREATE VIEW ideptree (dependencies)  
AS  
  SELECT lpad(' ', 3*(max(nested_level))) || max(nvl(type, '<no  
permission>'))  
  || ' ' || schema || decode(type, NULL, '', '.') || name  
  FROM deptree  
  GROUP BY seq# /* So user can omit sort-by when selecting  
from ideptree */  
/
```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)



The screenshot shows the Oracle SQL Developer interface with several tabs at the top: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. The main pane displays the following error messages:

```
Error starting at line 47 in command:  
DROP SEQUENCE deptree_seq  
Error report:  
SQL Error: ORA-02289: sequence does not exist  
02289. 00000 - "sequence does not exist"  
*Cause: The specified sequence does not exist, or the user does  
not have the required privilege to perform this operation.  
*Action: Make sure the sequence name is correct, and that you have  
the right to perform the desired operation on this sequence.  
CREATE SEQUENCE succeeded.  
  
Error starting at line 53 in command:  
DROP TABLE deptree temptab  
Error report:  
SQL Error: ORA-00942: table or view does not exist  
00942. 00000 - "table or view does not exist"  
*Cause:  
*Action:  
CREATE TABLE succeeded.  
Warning: execution completed with warning  
PROCEDURE deptree_fill Compiled.  
  
Error starting at line 88 in command:  
DROP VIEW deptree  
Error report:  
SQL Error: ORA-00942: table or view does not exist  
00942. 00000 - "table or view does not exist"  
*Cause:  
*Action:  
REM This view will succeed if current user is sys. This view  
  
REM shows which shared cursors depend on the given object. If REM the current user is not sys, then this view get an error  
REM either about lack of privileges or about the non-existence of REM table x$kglixs.  
  
SET ECHO OFF
```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

```
Error starting at line 98 in command:
CREATE VIEW sys.deptree
  (nested_level, type, schema, name, seq#)
AS
  SELECT d.nest_level, o.object_type, o.owner, o.object_name, d.seq#
  FROM deptree_temptab d, dba_objects o
  WHERE d.object_id = o.object_id (+)
UNION ALL
  SELECT d.nest_level+1, 'CURSOR', '<shared>', ''||c.kglnaobj||'', d.seq#+.5
  FROM deptree_temptab d, x$kgldp k, x$kglob g, obj$ o, user$ u, x$kglob c,
       x$kglx$ a
  WHERE d.object_id = o.obj#
    AND o.name = g.kglnaobj
    AND o.owner# = u.user#
    AND u.name = g.kglnaown
    AND g.kglhdadr = k.kglrfhdl
    AND k.kglhdadr = a.kglhdadr /* make sure it is not a transitive */
    AND k.kgldepno = a.kglxsdep /* reference, but a direct one */
    AND k.kglhdadr = c.kglhdadr
    AND c.kglhdnsp = 0 /* a cursor */
Error at Command Line:102 Column:7
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
REM This view will succeed if current user is not sys. This view
REM does *not* show which shared cursors depend on the given
REM object.
REM If the current user is sys then this view will get an error
REM indicating that the view already exists (since prior view
```

```
REM create will have succeeded).

SET ECHO OFF

CREATE VIEW succeeded.

Error starting at line 136 in command:
DROP VIEW ideptree
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
CREATE VIEW succeeded.
```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

- b) Execute the deptree_fill procedure for the add_employee procedure.

Open the /home/oracle/labs/plpu/solns/sol_12_01_b.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE deptree_fill('PROCEDURE', USER, 'add_employee')
```



- c) Query the IDEPTREE view to see your results.

Open the /home/oracle/labs/plpu/solns/sol_12_01_c.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT * FROM IDEPTREE;
```



- d) Execute the deptree_fill procedure for the valid_deptid function.

Open the /home/oracle/labs/plpu/solns/sol_12_01_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE deptree_fill('FUNCTION', USER, 'valid_deptid')
```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are three icons: a pencil, a square, and a document. The main area displays the message "anonymous block completed".

- e) Query the IDEPTREE view to see your results.

Open the /home/oracle/labs/plpu/solns/sol_12_01_e.sql script. Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT * FROM IDEPTREE;
```

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are three icons: a pencil, a square, and a document. The main area displays the message "DEPENDENCIES" followed by a dashed line. Underneath the dashed line, it lists "PROCEDURE ORA61.ADD_EMPLOYEE" and "FUNCTION ORA61.VALID_DEPTID". At the bottom, it says "2 rows selected".

If you have time, complete the following exercise:

- 2) Dynamically validate invalid objects.
a) Make a copy of your EMPLOYEES table, called EMPS.

Open the /home/oracle/labs/plpu/solns/sol_12_02_a.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE TABLE emps AS
SELECT * FROM employees;
```

The screenshot shows the Oracle SQL Worksheet interface. The toolbar at the top has tabs for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are three icons: a pencil, a square, and a document. The main area displays the message "CREATE TABLE succeeded."

- b) Alter your EMPLOYEES table and add the column TOTSAL with data type NUMBER (9, 2).

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

Open the `/home/oracle/labs/plpu/solns/sol_12_02_b.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
ALTER TABLE employees
ADD (totsal NUMBER(9, 2));
```

ALTER TABLE employees succeeded.

- c) Create and save a query to display the name, type, and status of all invalid objects.

Open the `/home/oracle/labs/plpu/solns/sol_12_02_c.sql` script. Click the Execute Statement (F9) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
1.EMP_ACTIONS	PACKAGE BODY	INVALID
2.UPD_MINSALARY_TRG	TRIGGER	INVALID
3.DELETE_EMP_TRG	TRIGGER	INVALID
4.EMP_PKG	PACKAGE BODY	INVALID
5.EMP_PKG	PACKAGE	INVALID
6.GET_EMPLOYEE	PROCEDURE	INVALID
7.UPDATE_JOB_HISTORY	TRIGGER	INVALID
8.SECURE_EMPLOYEES	TRIGGER	INVALID

- d) In the `compile_pkg` (created in Practice 6 of the lesson titled “Using Dynamic SQL”), add a procedure called `recompile` that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to alter the invalid object type and compile it.

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

Open the /home/oracle/labs/plpu/solns/sol_12_02_d.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. The newly added code is highlighted in bold letters in the following code box.

```

CREATE OR REPLACE PACKAGE compile_pkg IS
  PROCEDURE make(name VARCHAR2);
  PROCEDURE recompile;
END compile_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY compile_pkg IS

  PROCEDURE execute(stmt VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
  END;

  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
    /*
     * The ROWNUM = 1 is added to the condition
     * to ensure only one row is returned if the
     * name represents a PACKAGE, which may also
     * have a PACKAGE BODY. In this case, we can
     * only compile the complete package, but not
     * the specification or body as separate
     * components.
    */
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(name)
    AND ROWNUM = 1;
    RETURN proc_type;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN NULL;
  END;

  PROCEDURE make(name VARCHAR2) IS
    stmt          VARCHAR2(100);
    proc_type    VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      stmt := 'ALTER ' || proc_type || ' ' || name || '
COMPILE';
    END IF;
  END;

```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

```

execute(stmt);
ELSE
    RAISE_APPLICATION_ERROR(-20001,
        'Subprogram '''|| name ||''' does not exist');
END IF;
END make;

PROCEDURE recompile IS
    stmt VARCHAR2(200);
    obj_name user_objects.object_name%type;
    obj_type user_objects.object_type%type;
BEGIN
    FOR objrec IN (SELECT object_name, object_type
                    FROM user_objects
                   WHERE status = 'INVALID'
                     AND object_type <> 'PACKAGE BODY')
    LOOP
        stmt := 'ALTER '|| objrec.object_type ||' '|
                objrec.object_name ||' COMPILE';
        execute(stmt);
    END LOOP;
END recompile;

END compile_pkg;
/
SHOW ERRORS

```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

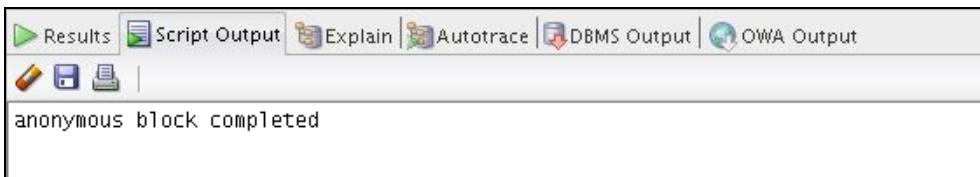
PACKAGE compile_pkg Compiled.
No Errors.
PACKAGE BODY compile_pkg Compiled.
No Errors.

- e) Execute the `compile_pkg.recompile` procedure.

Open the `/home/oracle/labs/plpu/solns/sol_12_02_e.sql` script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE compile_pkg.recompile
```

Practice Solutions 12-1: Managing Dependencies in Your Schema (continued)

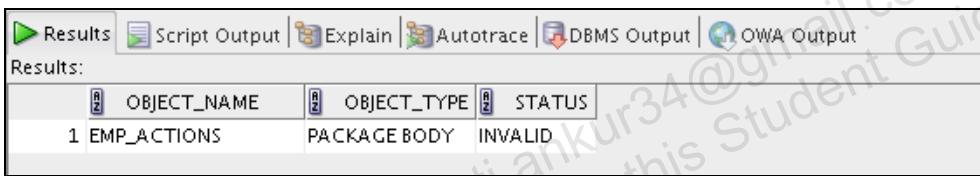


anonymous block completed

- f) Run the script file that you created in step 3 c. to check the value of the STATUS column. Do you still have objects with an INVALID status?

Open the /home/oracle/labs/plpu/sols/sol_12_02_f.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT object_name, object_type, status  
FROM USER_OBJECTS  
WHERE status = 'INVALID';
```



OBJECT_NAME	OBJECT_TYPE	STATUS
EMP_ACTIONS	PACKAGE BODY	INVALID

Appendix AP

Additional Practices and Solutions

Table of Contents

Practice 1.....	3
Practice 1-1: Creating a New SQL Developer Database Connection.....	4
Practice 1-2: Adding a New Job to the JOBS Table.....	6
Practice 1-3: Adding a New Row to the JOB_HISTORY Table.....	7
Practice 1-4: Updating the Minimum and Maximum Salaries for a Job	8
Practice 1-5: Monitoring Employees Salaries.....	9
Practice 1-6: Retrieving the Total Number of Years of Service for an Employee	10
Practice 1-7: Retrieving the Total Number of Different Jobs for an Employee	11
Practice 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions.....	12
Practice 1-9: Creating a Trigger to Ensure that the Employees' Salaries Are Within the Acceptable Range	13
Practice Solutions: 1-1: Creating a New SQL Developer Database Connection.....	14
Practice Solutions 1-2: Adding a New Job to the JOBS Table.....	16
Practice Solutions 1-3: Adding a New Row to the JOB_HISTORY Table.....	19
Practice Solution 1-4: Updating the Minimum and Maximum Salaries for a Job	23
Practice Solutions 1-5: Monitoring Employees Salaries	28
Practice Solutions 1-6: Retrieving the Total Number of Years of Service for an Employee	33
Practice Solutions 1-7: Retrieving the Total Number of Different Jobs for an Employee	37
Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions	40
Practice Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range	47
Practice 2.....	51
Practice 2-1: Creating the VIDEO_PKG Package.....	52
Practice Solutions 2-1: Creating the VIDEO_PKG Package.....	54

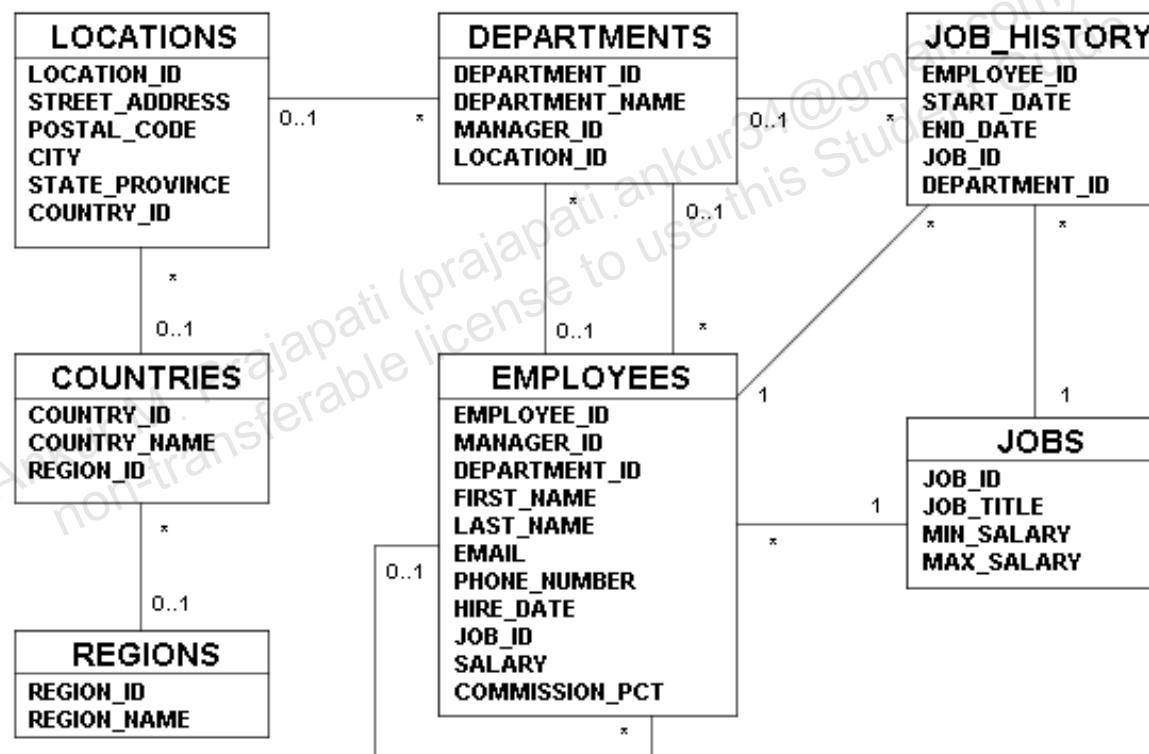
Practice 1

The additional practices are provided as a supplement to the course *Oracle Database 11g: Develop PL/SQL Program Units*. In these practices, you apply the concepts that you learned in the course. The additional practices comprise two lessons.

Lesson 1 provides supplemental exercises to create stored procedures, functions, packages, and triggers, and to use the Oracle-supplied packages with SQL Developer or SQL*Plus as the development environment. The tables used in this portion of the additional practice include EMPLOYEES, JOBS, JOB_HISTORY, and DEPARTMENTS.

An entity relationship diagram is provided at the start of each practice. Each entity relationship diagram displays the table entities and their relationships. More detailed definitions of the tables and the data contained in them is provided in the appendix titled “Additional Practices: Table Descriptions and Data.”

The Human Resources (HR) Schema Entity Relationship Diagram



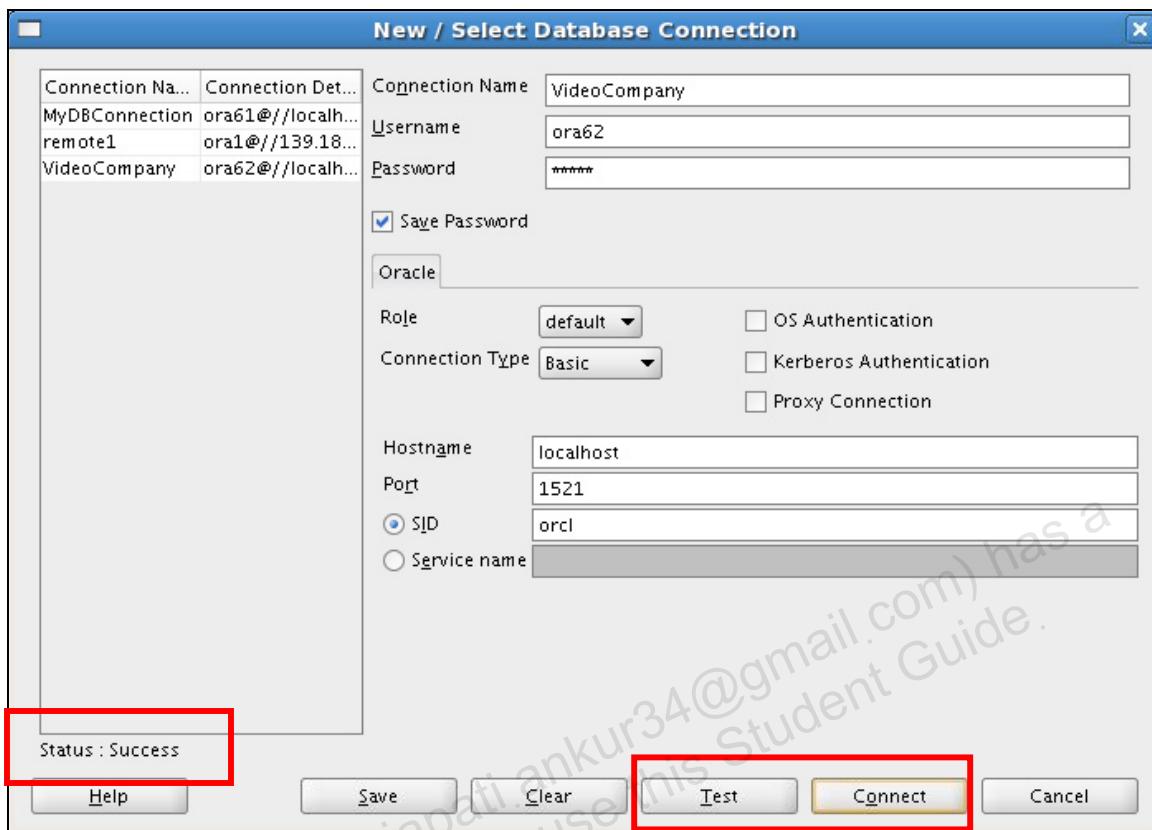
Practice 1-1: Creating a New SQL Developer Database Connection

In this practice, you start SQL Developer using your connection information and create a new database connection.

Start up SQL Developer using the user ID and password that are provided to you by the instructor such as ora62.

- 1) Create a database connection using the following information:
 - a) Connection Name: VideoCompany
 - b) Username: ora62
 - c) Password: ora62
 - d) Hostname: Enter the host name for your PC
 - e) Port: 1521
 - f) SID: ORCL
- 2) Test the new connection. If the Status is Success, connect to the database using this new connection:
 - a) Double-click the VideoCompany 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 1-1: Creating a New SQL Developer Database Connection (continued)



Practice 1-2: Adding a New Job to the JOBS Table

In this practice, you create a subprogram to add a new job into the JOBS table.

- 1) Create a stored procedure called NEW_JOB to enter a new order into the JOBS table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.
- 2) Enable SERVEROUTPUT, and then invoke the procedure to add a new job with job ID 'SY_ANAL', job title 'System Analyst', and minimum salary of 6000.
- 3) Check whether a row was added and note the new job ID for use in the next exercise. Commit the changes.

Practice 1-3: Adding a New Row to the JOB_HISTORY Table

In this practice, you add a new row to the JOB_HISTORY table for an existing employee.

- 1) Create a stored procedure called ADD_JOB_HIST to add a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID ('SY_ANAL') that you created in exercise 1 b.
 - a) The procedure should provide two parameters, one for the employee ID who is changing the job, and the second for the new job ID.
 - b) Read the employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table.
 - c) Make the hire date of this employee as start date and today's date as end date for this row in the JOB_HISTORY table.
 - d) Change the hire date of this employee in the EMPLOYEES table to today's date.
 - e) Update the job ID of this employee to the job ID passed as parameter (use the 'SY_ANAL' job ID) and salary equal to the minimum salary for that job ID + 500.

Note: Include exception handling to handle an attempt to insert a nonexistent employee.

- 2) Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the ADD_JOB_HIST procedure.
- 3) Enable SERVEROUTPUT, and then execute the procedure with employee ID 106 and job ID 'SY_ANAL' as parameters.
- 4) Query the JOB_HISTORY and EMPLOYEES tables to view your changes for employee 106, and then commit the changes.
- 5) Reenable the triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables.

Practice 1-4: Updating the Minimum and Maximum Salaries for a Job

In this practice, you create a program to update the minimum and maximum salaries for a job in the JOBS table.

- 1) Create a stored procedure called UPD_JOBSAL to update the minimum and maximum salaries for a specific job ID in the JOBS table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the JOBS table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the JOBS table is locked.
Hint: The resource locked/busy error number is -54.
- 2) Enable SERVEROUTPUT, and then execute the UPD_JOBSAL procedure by using a job ID of 'SY_ANAL', a minimum salary of 7000 and a maximum salary of 140.
Note: This should generate an exception message.
- 3) Disable triggers on the EMPLOYEES and JOBS tables.
- 4) Execute the UPD_JOBSAL procedure using a job ID of 'SY_ANAL', a minimum salary of 7000, and a maximum salary of 14000.
- 5) Query the JOBS table to view your changes, and then commit the changes.
- 6) Enable the triggers on the EMPLOYEES and JOBS tables.

Practice 1-5: Monitoring Employees Salaries

In this practice, you create a procedure to monitor whether employees have exceeded their average salaries for their job type.

- 1) Disable the SECURE_EMPLOYEES trigger.
- 2) In the EMPLOYEES table, add an EXCEED_AVGSAL column to store up to three characters and a default value of NO. Use a check constraint to allow the values YES or NO.
- 3) Create a stored procedure called CHECK_AVGSAL that checks whether each employee's salary exceeds the average salary for the JOB_ID.
 - a) The average salary for a job is calculated from the information in the JOBS table.
 - b) If the employee's salary exceeds the average for his or her job, then update the EXCEED_AVGSAL column in the EMPLOYEES table to a value of YES; otherwise, set the value to NO.
 - c) Use a cursor to select the employee's rows using the FOR UPDATE option in the query.
 - d) Add exception handling to account for a record being locked.
Hint: The resource locked/busy error number is -54.
 - e) Write and use a local function called GET_JOB_AVGSAL to determine the average salary for a job ID specified as a parameter.
- 4) Execute the CHECK_AVGSAL procedure. To view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary and the exceed_avgsal indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

Note: These exercises can be used for extra practice when discussing how to create functions.

Practice 1-6: Retrieving the Total Number of Years of Service for an Employee

In this practice, you create a subprogram to retrieve the number of years of service for a specific employee.

- 1) Create a stored function called GET_YEARS_SERVICE to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.
- 2) Invoke the GET_YEARS_SERVICE function in a call to DBMS_OUTPUT.PUT_LINE for an employee with ID 999.
- 3) Display the number of years of service for employee 106 with DBMS_OUTPUT.PUT_LINE invoking the GET_YEARS_SERVICE function. Make sure to enable SERVEROUTPUT.
- 4) Query the JOB_HISTORY and EMPLOYEES tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those that you get when you run these queries.

Practice 1-7: Retrieving the Total Number of Different Jobs for an Employee

In this practice, you create a program to retrieve the number of different jobs that an employee worked on during his or her service.

- 1) Create a stored function called GET_JOB_COUNT to retrieve the total number of different jobs on which an employee worked.
 - a) The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job.
 - b) Add exception handling to account for an invalid employee ID.
 - c) Write a UNION of two queries and count the rows retrieved into a PL/SQL table.
 - d) Use a FETCH with BULK COLLECT INTO to obtain the unique jobs for the employee.
- 2) Invoke the function for the employee with the ID of 176. Make sure to enable SERVEROUTPUT.

Note: These exercises can be used for extra practice when discussing how to create packages.

Practice 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions

In this practice, you create a package called EMPJOB_PKG that contains your NEW_JOB, ADD_JOB_HIST, UPD_JOBSAL procedures, as well as your GET_YEARS_SERVICE and GET_JOB_COUNT functions.

- 1) Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.
- 2) Create the package body with the subprogram implementation; remember to remove, from the subprogram implementations, any types that you moved into the package specification.
- 3) Invoke your EMPJOB_PKG.NEW_JOB procedure to create a new job with the ID PR_MAN, the job title Public Relations Manager, and the salary 6250. Make sure to enable SERVEROUTPUT.
- 4) Invoke your EMPJOB_PKG.ADD_JOB_HIST procedure to modify the job of employee ID 110 to job ID PR_MAN.
Note: You need to disable the UPDATE_JOB_HISTORY trigger before you execute the ADD_JOB_HIST procedure, and reenable the trigger after you have executed the procedure.
- 5) Query the JOBS, JOB_HISTORY, and EMPLOYEES tables to verify the results.

Note: These exercises can be used for extra practice when discussing how to create database triggers.

Practice 1-9: Creating a Trigger to Ensure that the Employees' Salaries Are Within the Acceptable Range

In this practice, you create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.

- 1) Create a trigger called CHECK_SAL_RANGE that is fired before every row that is updated in the MIN_SALARY and MAX_SALARY columns in the JOBS table.
 - a) For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the EMPLOYEES table falls within the new range of salaries specified for this job ID.
 - b) Include exception handling to cover a salary range change that affects the record of any existing employee.
- 2) Test the trigger using the SY_ANAL job, setting the new minimum salary to 5000, and the new maximum salary to 7000. Before you make the change, write a query to display the current salary range for the SY_ANAL job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the JOBS table for the specified job ID.
- 3) Using the SY_ANAL job, set the new minimum salary to 7,000, and the new maximum salary to 18000. Explain the results.

Practice Solutions: 1-1: Creating a New SQL Developer Database Connection

In this practice, you start SQL Developer using your connection information and create a new database connection.

- 1) Start up SQL Developer using the user ID and password that are provided to you by the instructor such as ora62.

Click the SQL Developer icon on your desktop.



- 2) Create a database connection using the following information:

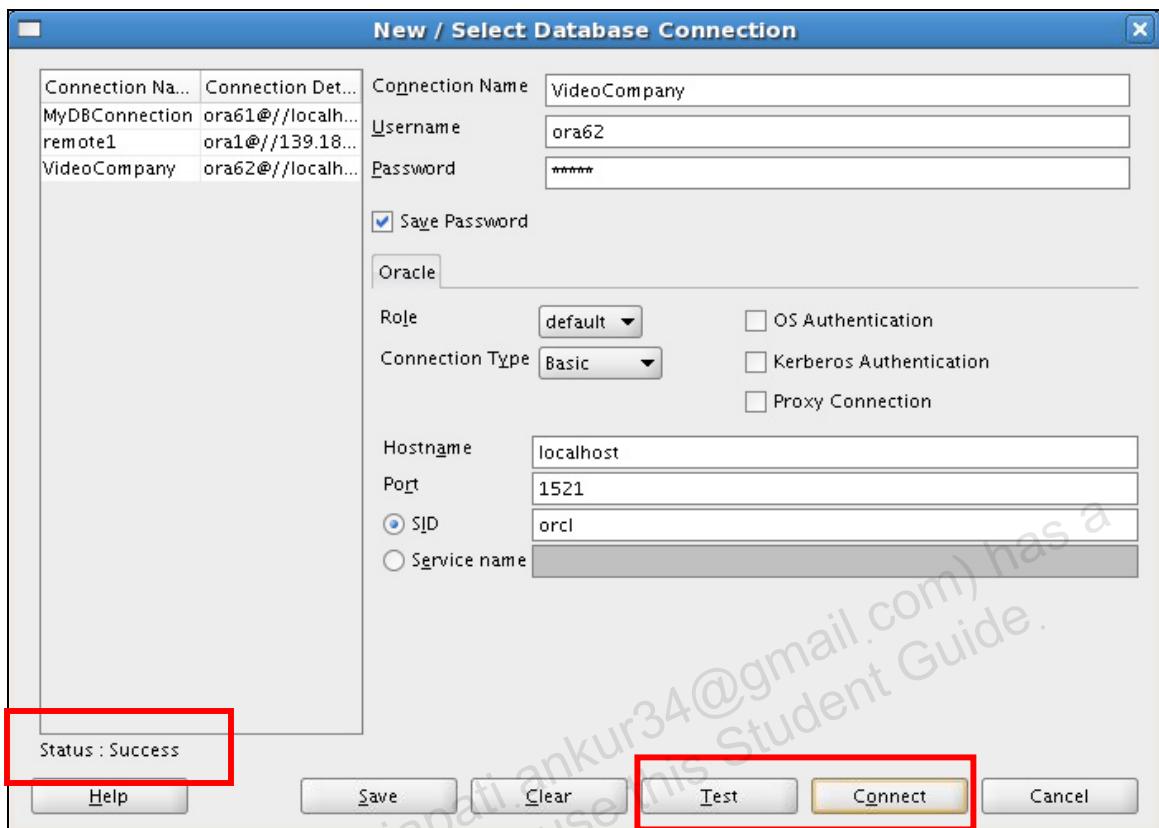
- a) Connection Name: VideoCompany
- b) Username: ora62
- c) Password: ora62
- d) Hostname: Enter the host name for your PC
- e) Port: 1521
- f) SID: ORCL

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 is displayed. Use the preceding information provided 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. Substitute the username, password, host name, and service name with the appropriate information as provided by your instructor. The following is a sample of the newly created database connection for student ora62:

- 3) Test the new connection. If the status is Success, connect to the database using this new connection:
 - a) Double-click the VideoCompany 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: Creating a New SQL Developer Database Connection (continued)



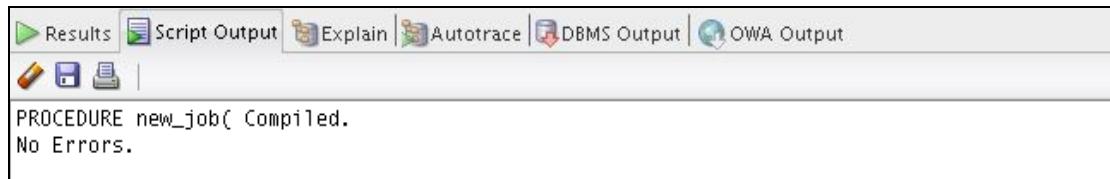
Practice Solutions 1-2: Adding a New Job to the JOBS Table

In this practice, you create a subprogram to add a new job into the JOBS table.

- 1) Create a stored procedure called NEW_JOB to enter a new order into the JOBS table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.

Open the /home/oracle/labs/plpu/sols/sol_ap_01_02_01.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to create and compile the procedure. When prompted to select a connection, select the new VideoCompany connection. The code, connection prompt, and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE new_job(
    p_jobid  IN jobs.job_id%TYPE,
    p_title   IN jobs.job_title%TYPE,
    v_minsal  IN jobs.min_salary%TYPE) IS
    v_maxsal  jobs.max_salary%TYPE := 2 * v_minsal;
BEGIN
    INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
    VALUES (p_jobid, p_title, v_minsal, v_maxsal);
    DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
    DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_title || ' ' ||
                           v_minsal || ' ' || v_maxsal);
END new_job;
/
SHOW ERRORS
```

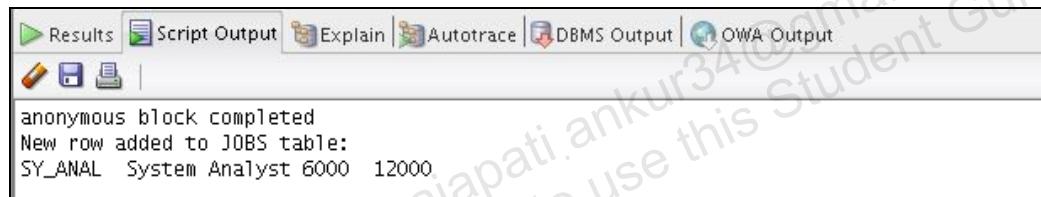


- 2) Enable SERVEROUTPUT, and then invoke the procedure to add a new job with job ID 'SY_ANAL', job title 'System Analyst', and minimum salary of 6000.

Practice Solutions 1-2: Adding a New Job to the JOBS Table (continued)

Run the `/home/oracle/labs/plpu/sols/sol_ap_01_02_02.sql` script. When prompted to select a connection, select the new VideoCompany connection. The code, connection prompt, and the results are displayed as follows:

```
SET SERVEROUTPUT ON
EXECUTE new_job ('SY_ANAL', 'System Analyst', 6000)
```



- 3) Check whether a row was added and note the new job ID for use in the next exercise.
Commit the changes.

Run the `/home/oracle/labs/plpu/sols/sol_ap_01_02_03.sql` script. The code, connection prompt, and the results are displayed as follows:

```
SELECT *
FROM   jobs
WHERE  job_id = 'SY_ANAL';
COMMIT;
```



**Practice Solutions 1-2: Adding a New Job to the JOBS Table
(continued)**

The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The query executed was:

```
INSERT INTO JOBS (JOB_ID, JOB_TITLE, MIN_SALARY, MAX_SALARY)
VALUES ('SY_ANAL', 'System Analyst', 6000, 12000);
```

The output shows the inserted row:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	6000	12000

Below the table, the message '1 rows selected' is displayed, followed by 'COMMIT succeeded.'

Practice Solutions 1-3: Adding a New Row to the JOB_HISTORY Table

In this practice, you add a new row to the JOB_HISTORY table for an existing employee.

- 1) Create a stored procedure called ADD_JOB_HIST to add a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID ('SY_ANAL') that you created in exercise 1 b.
 - a) The procedure should provide two parameters, one for the employee ID who is changing the job, and the second for the new job ID.
 - b) Read the employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table.
 - c) Make the hire date of this employee as start date and today's date as end date for this row in the JOB_HISTORY table.
 - d) Change the hire date of this employee in the EMPLOYEES table to today's date.
 - e) Update the job ID of this employee to the job ID passed as parameter (use the 'SY_ANAL' job ID) and salary equal to the minimum salary for that job ID + 500.

Note: Include exception handling to handle an attempt to insert a nonexistent employee.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_03_01.sql script. The code, connection prompt, and the results are displayed as follows:

```

CREATE OR REPLACE PROCEDURE add_job_hist(
  p_emp_id    IN employees.employee_id%TYPE,
  p_new_jobid IN jobs.job_id%TYPE) IS
BEGIN
  INSERT INTO job_history
    SELECT employee_id, hire_date, SYSDATE, job_id,
           department_id
      FROM   employees
     WHERE  employee_id = p_emp_id;
  UPDATE employees
    SET   hire_date = SYSDATE,
          job_id = p_new_jobid,
          salary = (SELECT min_salary + 500
                      FROM   jobs
                     WHERE  job_id = p_new_jobid)
   WHERE employee_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
                        ' details to the JOB_HISTORY
table');
  DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
                        ' || '

```

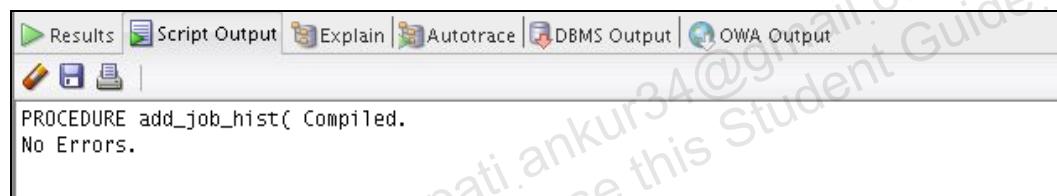
Practice Solutions 1-3: Adding a New Row to the JOB_HISTORY Table (continued)

```

    p_emp_id || ' to ' || p_new_jobid);

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20001, 'Employee does not
exist!');
END add_job_hist;
/
SHOW ERRORS

```



- 2) Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the ADD_JOB_HIST procedure.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_03_02.sql script. The code, the connection prompt, and the results are displayed as follows:

```

ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;
ALTER TABLE job_history DISABLE ALL TRIGGERS;

```



Practice Solutions 1-3: Adding a New Row to the JOB_HISTORY Table (continued)

```

Results Script Output Explain Autotrace DBMS Output OWA Output
ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.
ALTER TABLE job_history succeeded.

```

- 3) Enable SERVEROUTPUT, and then execute the procedure with employee ID 106 and job ID 'SY_ANAL' as parameters.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_03_03.sql script. The code, the connection prompt, and the results are displayed as follows:

```

SET SERVEROUTPUT ON
EXECUTE add_job_hist(106, 'SY_ANAL')

```



```

Results Script Output Explain Autotrace DBMS Output OWA Output
anonymous block completed
Added employee 106 details to the JOB_HISTORY table
Updated current job of employee 106 to SY_ANAL

```

- 4) Query the JOB_HISTORY and EMPLOYEES tables to view your changes for employee 106, and then commit the changes.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_03_04.sql script. The code, the connection prompt, and the results are displayed as follows:

```

SELECT * FROM job_history
WHERE employee_id = 106;

SELECT job_id, salary FROM employees
WHERE employee_id = 106;

COMMIT;

```

Practice Solutions 1-3: Adding a New Row to the JOB_HISTORY Table (continued)

The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The output displays the following data:

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
106	05-FEB-98	26-JUN-09	IT_PROG	60

1 rows selected.

JOB_ID	SALARY
SY_ANAL	6500

1 rows selected.

COMMIT succeeded.

- 5) Reenable the triggers on the EMPLOYEES, JOBS and JOB_HISTORY tables.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_03_05.sql script. The code, the connection prompt, and the results are displayed as follows:

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE jobs ENABLE ALL TRIGGERS;
ALTER TABLE job_history ENABLE ALL TRIGGERS;
```



The screenshot shows the Oracle SQL Developer interface with the 'Results' tab selected. The output displays the following messages:

```
ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.
ALTER TABLE job_history succeeded.
```

Practice Solution 1-4: Updating the Minimum and Maximum Salaries for a Job

In this practice, you create a program to update the minimum and maximum salaries for a job in the JOBS table.

- 1) Create a stored procedure called UPD_JOBSAL to update the minimum and maximum salaries for a specific job ID in the JOBS table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the JOBS table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the JOBS table is locked.
- Hint:** The resource locked/busy error number is -54.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_04_01.sql script. The code, the connection prompt, and the results are displayed as follows:

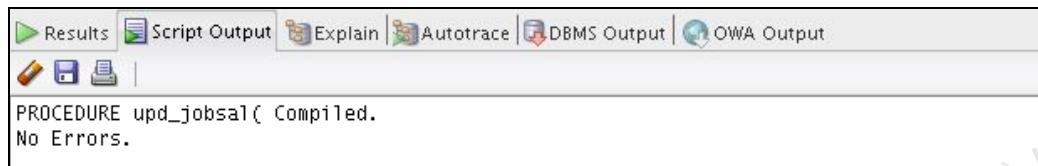
```

CREATE OR REPLACE PROCEDURE upd_jobsal(
    p_jobid      IN jobs.job_id%type,
    p_new_minsal IN jobs.min_salary%type,
    p_new_maxsal IN jobs.max_salary%type) IS
    v_dummy        PLS_INTEGER;
    e_resource_busy EXCEPTION;
    e_sal_error    EXCEPTION;
    PRAGMA          EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
    IF (p_new_maxsal < p_new_minsal) THEN
        RAISE e_sal_error;
    END IF;
    SELECT 1 INTO v_dummy
    FROM jobs
    WHERE job_id = p_jobid
    FOR UPDATE OF min_salary NOWAIT;
    UPDATE jobs
        SET min_salary = p_new_minsal,
            max_salary = p_new_maxsal
        WHERE job_id = p_jobid;
EXCEPTION
    WHEN e_resource_busy THEN
        RAISE_APPLICATION_ERROR (-20001,
            'Job information is currently locked, try later.');
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'This job ID does not
exist');
    WHEN e_sal_error THEN
        RAISE_APPLICATION_ERROR(-20001,
            'Data error: Max salary should be more than min salary');
END upd_jobsal;

```

Practice Solution 1-4: Updating the Minimum and Maximum Salaries for a Job (continued)

```
/  
SHOW ERRORS
```



- 2) Enable SERVEROUTPUT, and then execute the UPD_JOBSAL procedure by using a job ID of 'SY_ANAL', a minimum salary of 7000 and a maximum salary of 140.

Note: This should generate an exception message.

Run the `/home/oracle/labs/plpu/sols/sol_ap_01_04_02.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
SET SERVEROUTPUT ON  
  
EXECUTE upd_jobsal('SY_ANAL', 7000, 140)
```



Practice Solution 1-4: Updating the Minimum and Maximum Salaries for a Job (continued)

The screenshot shows the SQL*Plus interface with the following output:

```

Line 1: SQLPLUS Command Skipped: SET SERVEROUTPUT ON

Error starting at line 3 in command:
EXECUTE upd_jobsal('SY_ANAL', 7000, 140)
Error report:
ORA-20001: Data error: Max salary should be more than min salary
ORA-06512: at "ORA62.UPD_JOBSAL", line 28
ORA-06512: at line 1

```

- 3) Disable triggers on the EMPLOYEES and JOBS tables.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_04_03.sql script. The code, the connection prompt, and the results are displayed as follows:

```

ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;

```



The screenshot shows the SQL*Plus interface with the following output:

```

ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.

```

- 4) Execute the UPD_JOBSAL procedure using a job ID of 'SY_ANAL', a minimum salary of 7000, and a maximum salary of 14000.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_04_04.sql script. The code, the connection prompt, and the results are displayed as follows:

```

EXECUTE upd_jobsal('SY_ANAL', 7000, 14000)

```

Practice Solution 1-4: Updating the Minimum and Maximum Salaries for a Job (continued)

The screenshot shows the Oracle SQL Developer interface. At the top is the 'Select Connection' dialog box, which displays the message 'Select the connection you wish to use from the list, or create a new connection.' A dropdown menu labeled 'Connection:' shows 'VideoCompany'. Below the dropdown are 'Help', 'OK', and 'Cancel' buttons. To the right of the dialog is the main SQL developer window. The title bar of the window says 'anonymous block completed'. The toolbar at the top of the window includes 'Results', 'Script Output', 'Explain', 'Autotrace', 'DBMS Output', and 'OWA Output'. The main area of the window is currently empty, indicating that the anonymous block has been completed.

- 5) Query the JOBS table to view your changes, and then commit the changes.

Run the `/home/oracle/labs/plpu/solns/sol_ap_01_04_05.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
SELECT *
FROM   jobs
WHERE  job_id = 'SY_ANAL';
```

The screenshot shows the Oracle SQL Developer interface. At the top is the 'Select Connection' dialog box, identical to the one in the previous screenshot. The main SQL developer window below shows the results of a query on the JOBS table. The results are presented in a tabular format:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

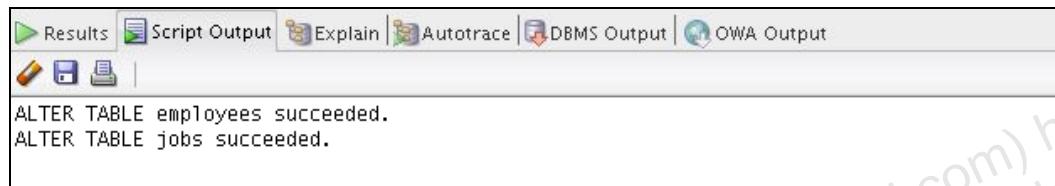
Below the table, it says '1 rows selected'.

- 6) Enable the triggers on the EMPLOYEES and JOBS tables.

Run the `/home/oracle/labs/plpu/solns/sol_ap_01_04_06.sql` script. The code, the connection prompt, and the results are displayed as follows:

Practice Solution 1-4: Updating the Minimum and Maximum Salaries for a Job (continued)

```
ALTER TABLE employees ENABLE ALL TRIGGERS;  
ALTER TABLE jobs ENABLE ALL TRIGGERS;
```



Practice Solutions 1-5: Monitoring Employees Salaries

In this practice, you create a procedure to monitor whether employees have exceeded their average salaries for their job type.

- 1) Disable the SECURE_EMPLOYEES trigger.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_05_01.sql script. The code, the connection prompt, and the results are displayed as follows:

```
ALTER TRIGGER secure_employees DISABLE;
```

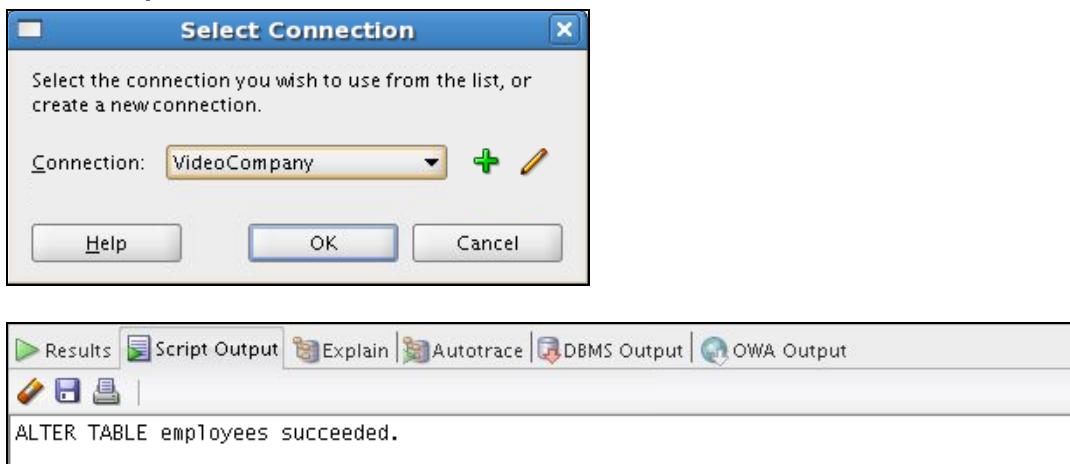


- 2) In the EMPLOYEES table, add an EXCEED_AVGSAL column to store up to three characters and a default value of NO. Use a check constraint to allow the values YES or NO.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_05_02.sql script. The code, the connection prompt, and the results are displayed as follows:

```
ALTER TABLE employees (
    ADD (exceed_avgsal VARCHAR2(3) DEFAULT 'NO'
        CONSTRAINT employees_exceed_avgsal_ck
        CHECK (exceed_avgsal IN ('YES', 'NO')));
```

Practice Solutions 1-5: Monitoring Employees Salaries (continued)



- 3) Create a stored procedure called CHECK_AVGSAL that checks whether each employee's salary exceeds the average salary for the JOB_ID.
 - a) The average salary for a job is calculated from the information in the JOBS table.
 - b) If the employee's salary exceeds the average for his or her job, then update the EXCEED_AVGSAL column in the EMPLOYEES table to a value of YES; otherwise, set the value to NO.
 - c) Use a cursor to select the employee's rows using the FOR UPDATE option in the query.
 - d) Add exception handling to account for a record being locked.
Hint: The resource locked/busy error number is -54.
 - e) Write and use a local function called GET_JOB_AVGSAL to determine the average salary for a job ID specified as a parameter.

Run the `/home/oracle/labs/plpu/solns/sol_ap_01_05_03.sql` script. The code, the connection prompt, and the results are displayed as follows:

```

CREATE OR REPLACE PROCEDURE check_avgsal IS
  emp_exceed_avgsal_type employees.exceed_avgsal%type;
  CURSOR c_emp_csr IS
    SELECT employee_id, job_id, salary
    FROM employees
    FOR UPDATE;
  e_resource_busy EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_resource_busy, -54);
  FUNCTION get_job_avgsal (jobid VARCHAR2) RETURN NUMBER IS
    avg_sal employees.salary%type;
  BEGIN
    SELECT (max_salary + min_salary)/2 INTO avg_sal
    FROM jobs
    WHERE job_id = jobid;
  
```

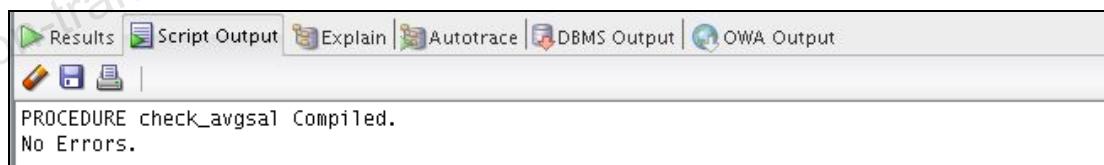
Practice Solutions 1-5: Monitoring Employees Salaries (continued)

```

        RETURN avg_sal;
      END;

      BEGIN
        FOR emprec IN c_emp_csr
        LOOP
          emp_exceed_avgsal_type := 'NO';
          IF emprec.salary >= get_job_avgsal(emprec.job_id) THEN
            emp_exceed_avgsal_type := 'YES';
          END IF;
          UPDATE employees
            SET exceed_avgsal = emp_exceed_avgsal_type
            WHERE CURRENT OF c_emp_csr;
        END LOOP;
      EXCEPTION
        WHEN e_resource_busy THEN
          ROLLBACK;
          RAISE_APPLICATION_ERROR (-20001, 'Record is busy, try
later.');
      END check_avgsal;
    /
SHOW ERRORS

```



- 4) Execute the CHECK_AVGSAL procedure. To view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary and the exceed_avgsal indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

Note: These exercises can be used for extra practice when discussing how to create functions.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_05_04.sql script. The code, the connection prompt, and the results are displayed as follows:

Practice Solutions 1-5: Monitoring Employees Salaries (continued)

```
EXECUTE check_avgsal

SELECT e.employee_id, e.job_id, (j.max_salary-j.min_salary/2)
job_avgsal,
      e.salary, e.exceed_avgsal avg_exceeded
FROM   employees e, jobs j
WHERE  e.job_id = j.job_id
and e.exceed_avgsal = 'YES';

COMMIT;
```



Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

anonymous block completed

EMPLOYEE_ID	JOB_ID	JOB_AVGSAL	SALARY	AVG_EXCEEDED
103	IT_PROG	8000	9000	YES
109	FI_ACCOUNT	6900	9000	YES
110	FI_ACCOUNT	6900	8200	YES
111	FI_ACCOUNT	6900	7700	YES
112	FI_ACCOUNT	6900	7800	YES
113	FI_ACCOUNT	6900	6900	YES
120	ST_MAN	5750	8000	YES
121	ST_MAN	5750	8200	YES
122	ST_MAN	5750	7900	YES
137	ST_CLERK	4000	3600	YES
141	ST_CLERK	4000	3500	YES
150	SA REP	9000	10000	YES
151	SA REP	9000	9500	YES
152	SA REP	9000	9000	YES
156	SA REP	9000	10000	YES
157	SA REP	9000	9500	YES
158	SA REP	9000	9000	YES
162	SA REP	9000	10500	YES
163	SA REP	9000	9500	YES
168	SA REP	9000	11500	YES
169	SA REP	9000	10000	YES
170	SA REP	9000	9600	YES
174	SA REP	9000	11000	YES
184	SH_CLERK	4250	4200	YES
185	SH_CLERK	4250	4100	YES

**Practice Solutions 1-5: Monitoring Employees Salaries
(continued)**

192	SH_CLERK	4250	4000	YES
201	MK_MAN	10500	13000	YES
203	HR REP	7000	6500	YES
204	PR REP	8250	10000	YES
206	AC ACCOUNT	6900	8300	YES

30 rows selected

COMMIT succeeded.

Practice Solutions 1-6: Retrieving the Total Number of Years of Service for an Employee

In this practice, you create a subprogram to retrieve the number of years of service for a specific employee.

- 1) Create a stored function called GET_YEARS_SERVICE to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_06_01.sql script. The code, the connection prompt, and the results are displayed as follows:

```

CREATE OR REPLACE FUNCTION get_years_service(
    p_emp.empid_type IN employees.employee_id%TYPE) RETURN
NUMBER IS
CURSOR c_jobh_csr IS
    SELECT MONTHS_BETWEEN(end_date, start_date)/12
v_years_in_job
    FROM job_history
    WHERE employee_id = p_emp.empid_type;
v_years_service NUMBER(2) := 0;
v_years_in_job NUMBER(2) := 0;
BEGIN
    FOR jobh_rec IN c_jobh_csr
    LOOP
        EXIT WHEN c_jobh_csr%NOTFOUND;
        v_years_service := v_years_service +
jobh_rec.v_years_in_job;
    END LOOP;
    SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO
v_years_in_job
    FROM employees
    WHERE employee_id = p_emp.empid_type;
    v_years_service := v_years_service + v_years_in_job;
    RETURN ROUND(v_years_service);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20348,
            'Employee with ID '|| p_emp.empid_type || ' does not
exist.');
        RETURN NULL;
    END get_years_service;
/
SHOW ERRORS

```

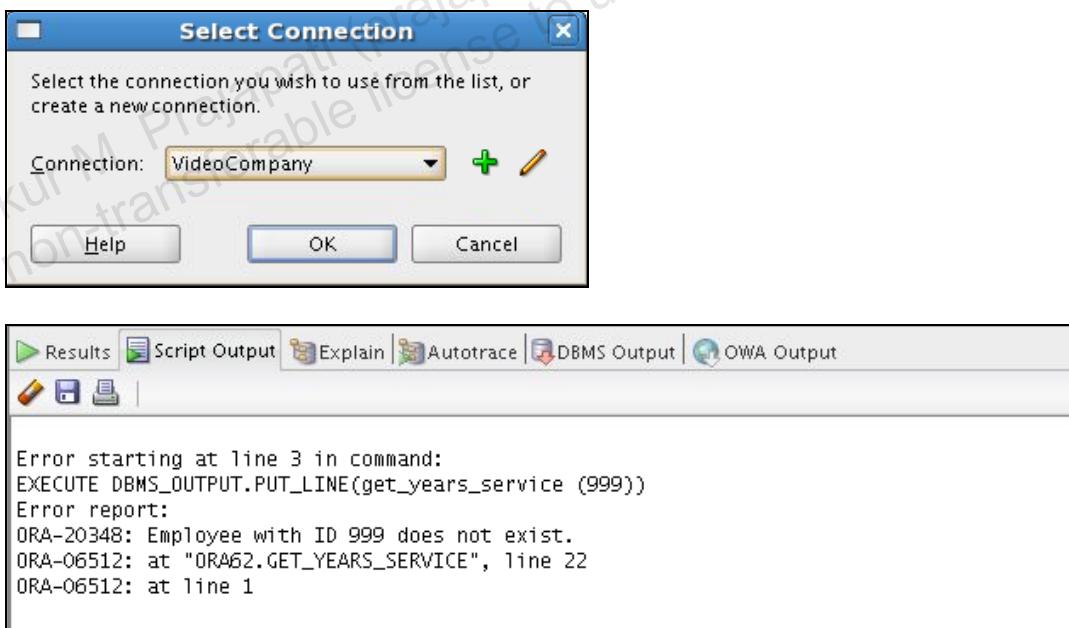
Practice Solutions 1-6: Retrieving the Total Number of Years of Service for an Employee (continued)



- 2) Invoke the GET_YEARS_SERVICE function in a call to DBMS_OUTPUT.PUT_LINE for an employee with ID 999.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_06_02.sql script. The code, the connection prompt, and the results are displayed as follows:

```
EXECUTE DBMS_OUTPUT.PUT_LINE(get_years_service(999))
```



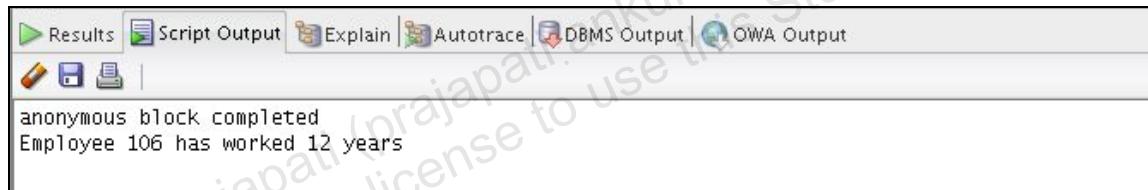
- 3) Display the number of years of service for employee 106 with DBMS_OUTPUT.PUT_LINE invoking the GET_YEARS_SERVICE function. Make sure to enable SERVEROUTPUT.

Practice Solutions 1-6: Retrieving the Total Number of Years of Service for an Employee (continued)

Run the `/home/oracle/labs/plpu/solns/sol_ap_01_06_03.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
SET SERVEROUTPUT ON

BEGIN
    DBMS_OUTPUT.PUT_LINE (
        'Employee 106 has worked ' || get_years_service(106) || ' '
    years');
END;
/
```



- 4) Query the JOB_HISTORY and EMPLOYEES tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those you get when you run these queries.

Run the `/home/oracle/labs/plpu/solns/sol_ap_01_06_04.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
SELECT employee_id, job_id,
       MONTHS_BETWEEN(end_date, start_date)/12 duration
  FROM job_history;

SELECT job_id, MONTHS_BETWEEN(SYSDATE, hire_date)/12 duration
  FROM employees
 WHERE employee_id = 106;
```

Practice Solutions 1-6: Retrieving the Total Number of Years of Service for an Employee (continued)

Results Script Output Explain Autotrace DBMS Output OWA Output

EMPLOYEE_ID	JOB_ID	DURATION
102	IT_PROG	5.52956989247311827956989247311827956989
101	AC_ACCOUNT	4.09946236559139784946236559139784946237
101	AC_MGR	3.38172043010752688172043010752688172043
201	MK_REP	3.83870967741935483870967741935483870968
114	ST_CLERK	1.7688172043010752688172043010752688172
122	ST_CLERK	0.9973118279569892473118279569892473118283
200	AD_ASST	5.75
176	SA REP	0.7688172043010752688172043010752688172042
176	SA MAN	0.9973118279569892473118279569892473118283
200	AC_ACCOUNT	4.49731182795698924731182795698924731183
106	IT_PROG	11.53968678439864595778574273197929111908

11 rows selected

JOB_ID	DURATION
SY_ANAL	0

1 rows selected

Practice Solutions 1-7: Retrieving the Total Number of Different Jobs for an Employee

In this practice, you create a program to retrieve the number of different jobs that an employee worked on during his or her service.

- 1) Create a stored function called GET_JOB_COUNT to retrieve the total number of different jobs on which an employee worked.

- a) The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job.

- b) Add exception handling to account for an invalid employee ID.
Hint: Use the distinct job IDs from the JOB_HISTORY table, and exclude the current job ID if it is one of the job IDs on which the employee has already worked.
- c) Write a UNION of two queries and count the rows retrieved into a PL/SQL table.
- d) Use a FETCH with BULK COLLECT INTO to obtain the unique jobs for the employee.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_07_01.sql script. The code, the connection prompt, and the results are displayed as follows:

```

CREATE OR REPLACE FUNCTION get_job_count(
    p_emp.empid_type IN employees.employee_id%TYPE) RETURN
NUMBER IS
    TYPE jobs_table_type IS TABLE OF jobs.job_id%type;
    v_jobtab jobs_table_type;
    CURSOR c_empjob_csr IS
        SELECT job_id
        FROM job_history
        WHERE employee_id = p_emp.empid_type
        UNION
        SELECT job_id
        FROM employees
        WHERE employee_id = p_emp.empid_type;
BEGIN
    OPEN c_empjob_csr;
    FETCH c_empjob_csr BULK COLLECT INTO v_jobtab;
    CLOSE c_empjob_csr;
    RETURN v_jobtab.count;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20348,
            'Employee with ID '|| p_emp.empid_type || ' does not
exist!');
    RETURN NULL;

```

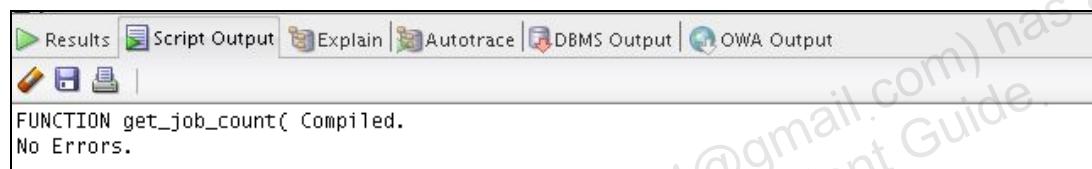
Practice Solutions 1-7: Retrieving the Total Number of Different Jobs for an Employee (continued)

```

END get_job_count;
/
SHOW ERRORS

FUNCTION get_job_count( Compiled.
No Errors.

```



- 2) Invoke the function for the employee with the ID of 176. Make sure to enable SERVEROUTPUT.

Note: These exercises can be used for extra practice when discussing how to create packages.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_07_02.sql script. The code, the connection prompt, and the results are displayed as follows:

```

SET SERVEROUTPUT ON

BEGIN
  DBMS_OUTPUT.PUT_LINE('Employee 176 worked on ' ||
    get_job_count(176) || ' different jobs.');
END;
/

```



Practice Solutions 1-7: Retrieving the Total Number of Different Jobs for an Employee (continued)

The screenshot shows the Oracle SQL Developer interface. At the top, there is a toolbar with several icons: Results (highlighted in green), Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there is a menu bar with Edit, Object Navigator, Database, Tools, Help, and a separator line. The main area contains the following text:

```
anonymous block completed
Employee 176 worked on 2 different jobs.
```

Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions

In this practice, you create a package called EMPJOB_PKG that contains your NEW_JOB, ADD_JOB_HIST, UPD_JOBSAL procedures, as well as your GET_YEARS_SERVICE and GET_JOB_COUNT functions.

- 1) Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_08_01.sql script. The code, the connection prompt, and the results are displayed as follows:

```
CREATE OR REPLACE PACKAGE empjob_pkg IS
  TYPE jobs_table_type IS TABLE OF jobs.job_id%type;

  PROCEDURE add_job_hist(
    p_emp_id IN employees.employee_id%TYPE,
    p_new_jobid IN jobs.job_id%TYPE);

  FUNCTION get_job_count(
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

  FUNCTION get_years_service(
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

  PROCEDURE new_job(
    p_jobid IN jobs.job_id%TYPE,
    p_title IN jobs.job_title%TYPE,
    p_minsal IN jobs.min_salary%TYPE);

  PROCEDURE upd_jobsal(
    p_jobid IN jobs.job_id%type,
    p_new_minsal IN jobs.min_salary%type,
    p_new_maxsal IN jobs.max_salary%type);
END empjob_pkg;
/
SHOW ERRORS
```



Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions (continued)

The screenshot shows the Oracle SQL Developer interface with the 'Script Output' tab selected. The output window displays the message: 'PACKAGE empjob_pkg Compiled. No Errors.' This indicates that the package has been successfully created and compiled without any errors.

- 2) Create the package body with the subprogram implementation; remember to remove from the subprogram implementations any types that you moved into the package specification.

Run the /home/oracle/labs/plpu/solns/sol_ap_01_08_02.sql script. The code, the connection prompt, and the results are displayed as follows:

```

CREATE OR REPLACE PACKAGE BODY empjob_pkg IS
  PROCEDURE add_job_hist(
    p_emp_id IN employees.employee_id%TYPE,
    p_new_jobid IN jobs.job_id%TYPE) IS
  BEGIN
    INSERT INTO job_history
      SELECT employee_id, hire_date, SYSDATE, job_id,
department_id
        FROM employees
       WHERE employee_id = p_emp_id;
    UPDATE employees
      SET hire_date = SYSDATE,
          job_id = p_new_jobid,
          salary = (SELECT min_salary + 500
                     FROM jobs
                    WHERE job_id = p_new_jobid)
       WHERE employee_id = p_emp_id;
    DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
' details to the JOB_HISTORY table');
    DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
|| p_emp_id|| ' to '|| p_new_jobid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR (-20001, 'Employee does not
exist!');
  END add_job_hist;

  FUNCTION get_job_count(
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
    v_jobtab jobs_table_type;
    CURSOR c_empjob_csr IS
      SELECT job_id
        FROM job_history
       WHERE employee_id = p_emp_id
      UNION

```

Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions (continued)

```

SELECT job_id
  FROM employees
 WHERE employee_id = p_emp_id;
BEGIN
  OPEN c.empjob_csr;
  FETCH c.empjob_csr BULK COLLECT INTO v_jobtab;
  CLOSE c.empjob_csr;
  RETURN v_jobtab.count;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_id ||' does not exist!');
    RETURN 0;
END get_job_count;

FUNCTION get_years_service(
  p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
CURSOR c.jobh_csr IS
  SELECT MONTHS_BETWEEN(end_date, start_date)/12
v_years_in_job
  FROM job_history
 WHERE employee_id = p_emp_id;
  v_years_service NUMBER(2) := 0;
  v_years_in_job NUMBER(2) := 0;
BEGIN
  FOR jobh_rec IN c.jobh_csr
  LOOP
    EXIT WHEN c.jobh_csr%NOTFOUND;
    v_years_service := v_years_service +
jobh_rec.v_years_in_job;
  END LOOP;
  SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO
v_years_in_job
  FROM employees
 WHERE employee_id = p_emp_id;
  v_years_service := v_years_service + v_years_in_job;
  RETURN ROUND(v_years_service);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_id ||' does not exist.');
    RETURN 0;
END get_years_service;

PROCEDURE new_job(
  p_jobid IN jobs.job_id%TYPE,
  p_title IN jobs.job_title%TYPE,
  p_minsal IN jobs.min_salary%TYPE) IS
  v_maxsal jobs.max_salary%TYPE := 2 * p_minsal;
BEGIN

```

Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions (continued)

```

        INSERT INTO jobs(job_id, job_title, min_salary,
max_salary)
VALUES (p_jobid, p_title, p_minsal, v_maxsal);
DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_title || ' ' ||
p_minsal || ' ' || v_maxsal);

END new_job;

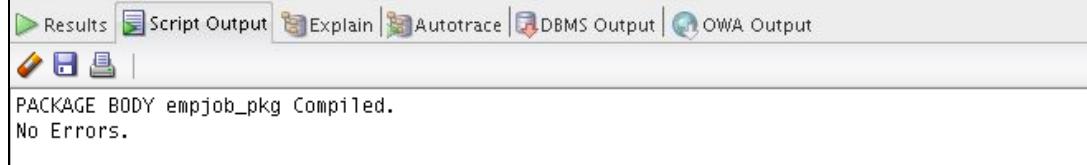
PROCEDURE upd_jobsal(
    p_jobid IN jobs.job_id%type,
    p_new_minsal IN jobs.min_salary%type,
    p_new_maxsal IN jobs.max_salary%type) IS
    v_dummy PLS_INTEGER;
    e_resource_busy EXCEPTION;
    e_sal_error EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
    IF (p_new_maxsal < p_new_minsal) THEN
        RAISE e_sal_error;
    END IF;
    SELECT 1 INTO v_dummy
    FROM jobs
    WHERE job_id = p_jobid
    FOR UPDATE OF min_salary NOWAIT;
    UPDATE jobs
        SET min_salary = p_new_minsal,
            max_salary = p_new_maxsal
        WHERE job_id = p_jobid;
EXCEPTION
    WHEN e_resource_busy THEN
        RAISE_APPLICATION_ERROR (-20001,
            'Job information is currently locked, try later.');
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'This job ID does not
exist');
    WHEN e_sal_error THEN
        RAISE_APPLICATION_ERROR(-20001,
            'Data error: Max salary should be more than min
salary');
    END upd_jobsal;
END empjob_pkg;
/
SHOW ERRORS

```

Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions (continued)



The screenshot shows the 'Select Connection' dialog box. The connection dropdown is set to 'VideoCompany'. There are 'OK' and 'Cancel' buttons at the bottom.



The results window displays the message: 'PACKAGE BODY empjob_pkg Compiled. No Errors.'

- 3) Invoke your EMPJOB_PKG.NEW_JOB procedure to create a new job with the ID PR_MAN, the job title Public Relations Manager, and the salary 6250. Make sure to enable SERVEROUTPUT.
- Run the /home/oracle/labs/plpu/sols/sol_ap_01_08_03.sql script. The code, the connection prompt, and the results are displayed as follows:**

```
SET SERVEROUTPUT ON
EXECUTE empjob_pkg.new_job('PR_MAN', 'Public Relations
Manager', 6250)
```



The screenshot shows the 'Select Connection' dialog box. The connection dropdown is set to 'VideoCompany'. There are 'OK' and 'Cancel' buttons at the bottom.



The results window displays the output of the anonymous block: 'anonymous block completed' and 'New row added to JOBS table: PR_MAN Public Relations Manager 6250 12500'.

Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions (continued)

- 4) Invoke your EMPJOB_PKG.ADD_JOB_HIST procedure to modify the job of employee ID 110 to job ID PR_MAN.

Note: You need to disable the UPDATE_JOB_HISTORY trigger before you execute the ADD_JOB_HIST procedure, and reenable the trigger after you have executed the procedure.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_08_04.sql script. The code, the connection prompt, and the results are displayed as follows:

```
ALTER TRIGGER update_job_history DISABLE;
EXECUTE empjob_pkg.add_job_hist(110, 'PR_MAN')
ALTER TRIGGER update_job_history ENABLE;
```



```
ALTER TRIGGER update_job_history succeeded.
anonymous block completed
Added employee 110 details to the JOB_HISTORY table
Updated current job of employee 110 to PR_MAN

ALTER TRIGGER update_job_history succeeded.
```

- 5) Query the JOBS, JOB_HISTORY, and EMPLOYEES tables to verify the results.

Note: These exercises can be used for extra practice when discussing how to create database triggers.

Run the /home/oracle/labs/plpu/sols/sol_ap_01_08_05.sql script. The code, the connection prompt, and the results are displayed as follows:

```
SELECT * FROM jobs WHERE job_id = 'PR_MAN';
SELECT * FROM job_history WHERE employee_id = 110;
SELECT job_id, salary FROM employees WHERE employee_id = 110;
```

Practice Solutions 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions (continued)A screenshot of the Oracle SQL Developer interface. The top navigation bar includes "Results", "Script Output", "Explain", "Autotrace", "DBMS Output", and "OWA Output". Below the toolbar, there are icons for edit, insert, delete, and refresh. The main area displays three separate SQL queries. The first query shows a single row for the job PR_MAN with details: JOB_ID (PR_MAN), JOB_TITLE (Public Relations Manager), MIN_SALARY (6250), and MAX_SALARY (12500). The second query shows a single row for employee 110 with details: EMPLOYEE_ID (110), START_DATE (28-SEP-97), END_DATE (19-AUG-09), JOB_ID (FI_ACCOUNT), and DEPARTMENT_ID (100). The third query shows a single row for the job PR_MAN with details: JOB_ID (PR_MAN) and SALARY (6750).

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
PR_MAN	Public Relations Manager	6250	12500

1 rows selected

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
110	28-SEP-97	19-AUG-09	FI_ACCOUNT	100

1 rows selected

JOB_ID	SALARY
PR_MAN	6750

1 rows selected

Practice Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range

In this practice, you create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.

- 1) Create a trigger called CHECK_SAL_RANGE that is fired before every row that is updated in the MIN_SALARY and MAX_SALARY columns in the JOBS table.
 - a) For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the EMPLOYEES table falls within the new range of salaries specified for this job ID.
 - b) Include exception handling to cover a salary range change that affects the record of any existing employee.

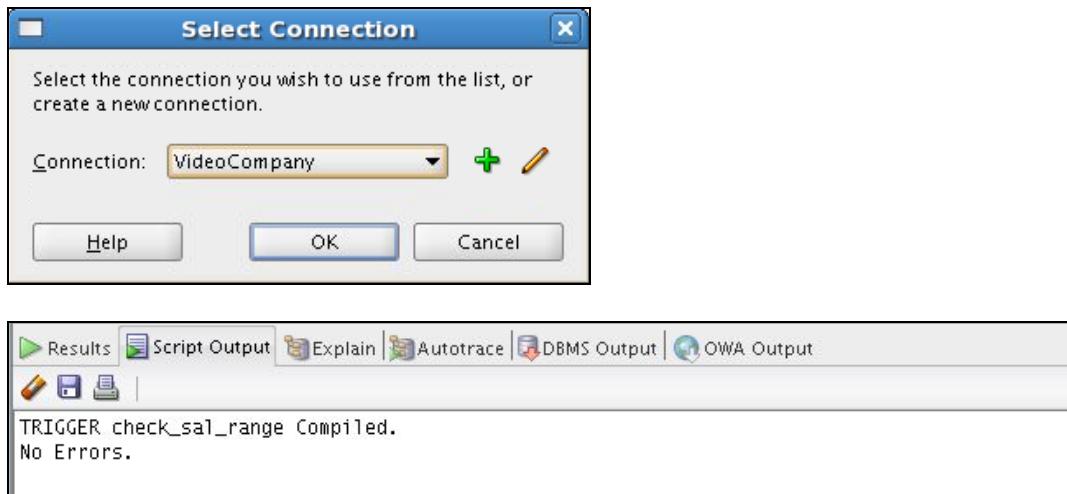
Run the `/home/oracle/labs/plpu/solns/sol_ap_01_09_01.sql` script. The code, the connection prompt, and the results are displayed as follows:

```

CREATE OR REPLACE TRIGGER check_sal_range
BEFORE UPDATE OF min_salary, max_salary ON jobs
FOR EACH ROW
DECLARE
  v_minsal employees.salary%TYPE;
  v_maxsal employees.salary%TYPE;
  e_invalid_salrange EXCEPTION;
BEGIN
  SELECT MIN(salary), MAX(salary) INTO v_minsal, v_maxsal
  FROM employees
  WHERE job_id = :NEW.job_id;
  IF (v_minsal < :NEW.min_salary) OR (v_maxsal >
  :NEW.max_salary) THEN
    RAISE e_invalid_salrange;
  END IF;
EXCEPTION
  WHEN e_invalid_salrange THEN
    RAISE_APPLICATION_ERROR(-20550,
      'Employees exist whose salary is out of the specified
      range. ' ||
      'Therefore the specified salary range cannot be
      updated.');
  END check_sal_range;
/
SHOW ERRORS

```

Practice Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range (continued)



- 2) Test the trigger using the SY_ANAL job, setting the new minimum salary to 5000, and the new maximum salary to 7000. Before you make the change, write a query to display the current salary range for the SY_ANAL job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the JOBS table for the specified job ID.

Run the `/home/oracle/labs/plpu/sols/sol_ap_01_09_02.sql` script. The code, the connection prompt, and the results are displayed as follows:

```

SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';

SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'SY_ANAL';

UPDATE jobs
SET min_salary = 5000, max_salary = 7000
WHERE job_id = 'SY_ANAL';

SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';

```



Practice Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range (continued)

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

JOB_ID JOB_TITLE MIN_SALARY MAX_SALARY

SY_ANAL System Analyst 7000 14000

1 rows selected

EMPLOYEE_ID LAST_NAME SALARY

106 Pataballa 6500

1 rows selected

1 rows updated

JOB_ID JOB_TITLE MIN_SALARY MAX_SALARY

SY_ANAL System Analyst 5000 7000

1 rows selected

- 3) Using the SY_ANAL job, set the new minimum salary to 7,000, and the new maximum salary to 18000. Explain the results.

Run the `/home/oracle/labs/plpu/sols/sol_ap_01_09_03.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
UPDATE jobs
SET min_salary = 7000, max_salary = 18000
WHERE job_id = 'SY_ANAL';
```



Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Error starting at line 1 in command:
`UPDATE jobs
SET min_salary = 7000, max_salary = 18000
WHERE job_id = 'SY_ANAL'`

Error report:
SQL Error: ORA-20550: Employees exist whose salary is out of the specified range. Therefore the specified salary range cannot be updated.
ORA-06512: at "ORA62.CHECK_SAL_RANGE", line 14
ORA-04088: error during execution of trigger 'ORA62.CHECK_SAL_RANGE'

Practice Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range (continued)

The update fails to change the salary range due to the functionality provided by the `CHECK_SAL_RANGE` trigger because employee 106 who has the `SY_ANAL` job ID has a salary of 6500, which is less than the minimum salary for the new salary range specified in the `UPDATE` statement.

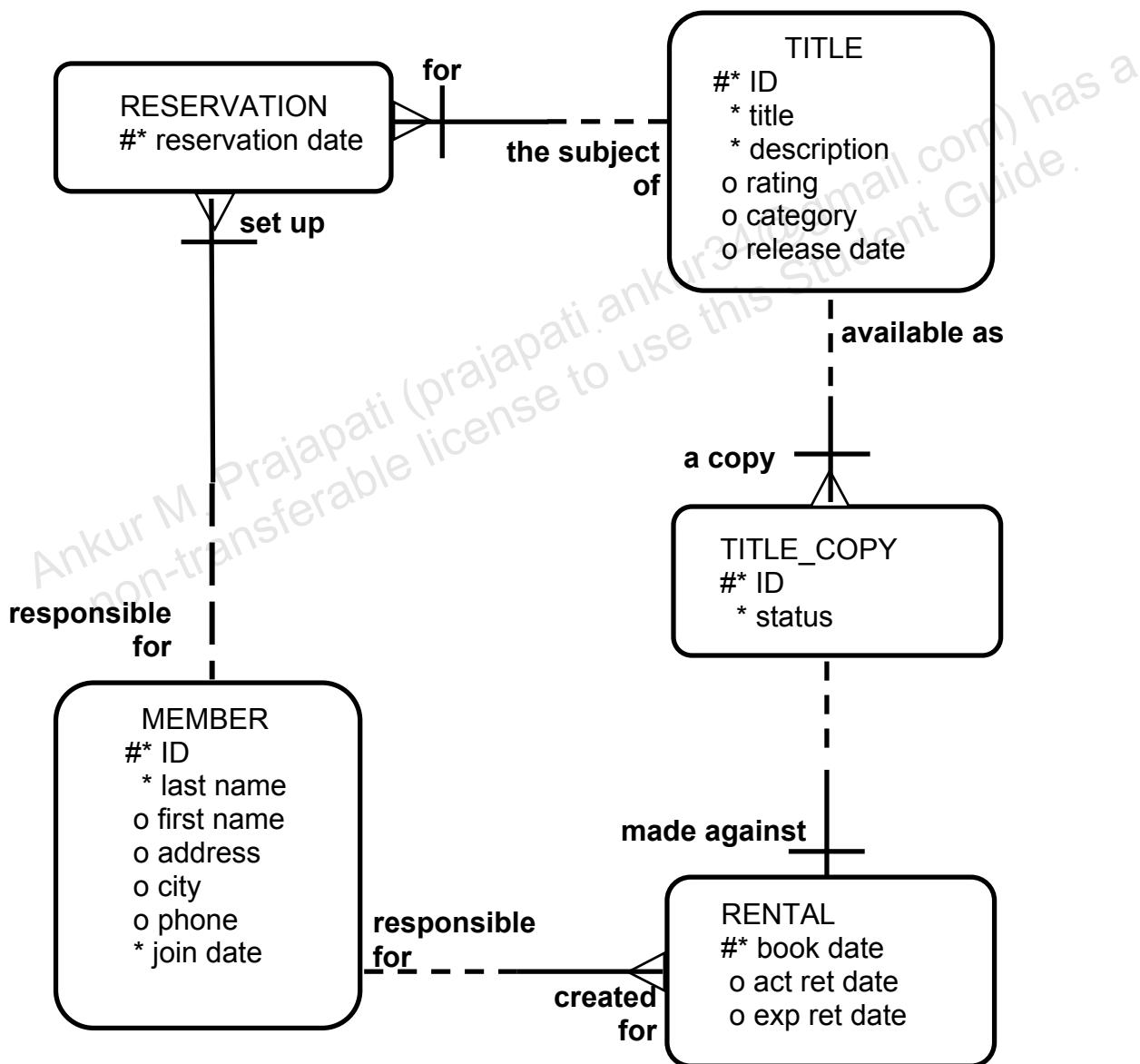
Practice 2

In this case study, you create a package named VIDEO_PKG that contains procedures and functions for a video store application. This application enables customers to become a member of the video store. Any member can rent movies, return rented movies, and reserve movies. Additionally, you create a trigger to ensure that any data in the video tables is modified only during business hours.

Create the package by using SQL*Plus and use the DBMS_OUTPUT Oracle-supplied package to display messages.

The video store database contains the following tables: TITLE, TITLE_COPY, RENTAL, RESERVATION, and MEMBER.

The video store database entity relationship diagram



Practice 2-1: Creating the **VIDEO_PKG** Package

In this practice, you create a package named **VIDEO_PKG** that contains procedures and functions for a video store application.

- 1) Load and execute the `/home/oracle/labs/plpu/labs/buildvid1.sql` script to create all the required tables and sequences that are needed for this exercise.
- 2) Load and execute the `/home/oracle/labs/plpu/labs/buildvid2.sql` script to populate all the tables created through the `buildvid1.sql` script.
- 3) Create a package named **VIDEO_PKG** with the following procedures and functions:
 - a) **NEW_MEMBER:** A public procedure that adds a new member to the **MEMBER** table. For the member ID number, use the sequence **MEMBER_ID_SEQ**; for the join date, use **SYSDATE**. Pass all other values to be inserted into a new row as parameters.
 - b) **NEW_RENTAL:** An overloaded public function to record a new rental. Pass the title ID number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as **AVAILABLE** in the **TITLE_COPY** table for one copy of this title, then update this **TITLE_COPY** table and set the status to **RENTED**. If there is no copy available, the function must return **NULL**. Then, insert a new record into the **RENTAL** table identifying the booked date as today's date, the copy ID number, the member ID number, the title ID number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return **NULL**, and display a list of the customers' names that match and their ID numbers.
 - c) **RETURN_MOVIE:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title ID, the copy ID, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the **RENTAL** table and set the actual return date to today's date. Update the status in the **TITLE_COPY** table based on the status parameter passed into the procedure.
 - d) **RESERVE_MOVIE:** A private procedure that executes only if all the video copies requested in the **NEW_RENTAL** procedure have a status of **RENTED**. Pass the member ID number and the title ID number to this procedure. Insert a new record into the **RESERVATION** table and record the reservation date, member ID number, and title ID number. Print a message indicating that a movie is reserved and its expected date of return.

Practice 2-1: Creating the *VIDEO_PKG* Package (continued)

- e) **EXCEPTION_HANDLER:** A private procedure that is called from the exception handler of the public programs. Pass the SQLCODE number to this procedure, and the name of the program (as a text string) where the error occurred. Use RAISE_APPLICATION_ERROR to raise a customized error. Start with a unique key violation (-1) and foreign key violation (-2292). Allow the exception handler to raise a generic error for any other errors.
- 4) Use the following scripts located in the /home/oracle/labs/plpu/soln directory to test your routines:
- Add two members using sol_ap_02_01_04_a.sql.
 - Add new video rentals using sol_ap_02_01_04_b.sql.
 - Return movies using the sol_ap_02_01_04_c.sql.
- 5) The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 12:00 PM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.
- Create a stored procedure called TIME_CHECK that checks the current time against business hours. If the current time is not within business hours, use the RAISE_APPLICATION_ERROR procedure to give an appropriate message.
 - Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your TIME_CHECK procedure from each of these triggers.
 - Test your triggers.

Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

Practice Solutions 2-1: Creating the VIDEO_PKG Package

In this practice, you create a package named VIDEO_PKG that contains procedures and functions for a video store application.

- 1) Load and execute the /home/oracle/labs/plpu/labs/buildvid1.sql script to create all the required tables and sequences that are needed for this exercise.

Run the /home/oracle/labs/plpu/labs/buildvid1.sql script. The code, the connection prompt, and the results are displayed as follows:

```

SET ECHO OFF
/* Script to build the Video Application (Part 1 -
buildvid1.sql)
   for the Oracle Introduction to Oracle with Procedure
Builder course.
   Created by: Debby Kramer Creation date: 12/10/95
   Last updated: 2/13/96
   Modified by Nagavalli Pataballa on 26-APR-2001
      For the course Introduction to Oracle9i: PL/SQL
      This part of the script creates tables and sequences that
are used
      by Part B of the Additional Practices of the course.
      Ignore the errors which appear due to dropping of table.
*/
DROP TABLE rental CASCADE CONSTRAINTS;
DROP TABLE reservation CASCADE CONSTRAINTS;
DROP TABLE title_copy CASCADE CONSTRAINTS;
DROP TABLE title CASCADE CONSTRAINTS;
DROP TABLE member CASCADE CONSTRAINTS;

PROMPT Please wait while tables are created.....

CREATE TABLE MEMBER
  (member_id  NUMBER (10)          CONSTRAINT member_id_pk
PRIMARY KEY
, last_name  VARCHAR2(25)
  CONSTRAINT member_last_nn NOT NULL
, first_name VARCHAR2(25)
, address    VARCHAR2(100)
, city       VARCHAR2(30)
, phone      VARCHAR2(25)
, join_date  DATE DEFAULT SYSDATE
  CONSTRAINT join_date_nn NOT NULL)
/
CREATE TABLE TITLE
  (title_id   NUMBER(10)
  CONSTRAINT title_id_pk PRIMARY KEY
, title      VARCHAR2(60)

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        CONSTRAINT title_nn NOT NULL
, description VARCHAR2(400)
    CONSTRAINT title_desc_nn NOT NULL
, rating      VARCHAR2(4)
    CONSTRAINT title_rating_ck CHECK (rating IN
('G','PG','R','NC17','NR'))
, category     VARCHAR2(20) DEFAULT 'DRAMA'
    CONSTRAINT title_categ_ck CHECK (category IN
('DRAMA','COMEDY','ACTION','CHILD','SCIFI','DOCUMENTARY'))
, release_date DATE)
/
CREATE TABLE TITLE_COPY
(copy_id      NUMBER(10)
, title_id    NUMBER(10)
    CONSTRAINT copy_title_id_fk
        REFERENCES title(title_id)
, status       VARCHAR2(15)
    CONSTRAINT copy_status_nn NOT NULL
    CONSTRAINT copy_status_ck CHECK (status IN ('AVAILABLE',
'DESTROYED',
'RENTED', 'RESERVED'))
, CONSTRAINT copy_title_id_pk PRIMARY KEY(copy_id,
title_id))
/
CREATE TABLE RENTAL
(book_date DATE DEFAULT SYSDATE
, copy_id   NUMBER(10)
, member_id NUMBER(10)
    CONSTRAINT rental_mbr_id_fk REFERENCES member(member_id)
, title_id  NUMBER(10)
, act_ret_date DATE
, exp_ret_date DATE DEFAULT SYSDATE+2
, CONSTRAINT rental_copy_title_id_fk FOREIGN KEY (copy_id,
title_id)
    REFERENCES title_copy(copy_id,title_id)
, CONSTRAINT rental_id_pk PRIMARY KEY(book_date, copy_id,
title_id, member_id))
/
CREATE TABLE RESERVATION
(res_date    DATE
, member_id NUMBER(10)
, title_id  NUMBER(10)
, CONSTRAINT res_id_pk PRIMARY KEY(res_date, member_id,
title_id))
/
PROMPT Tables created.
DROP SEQUENCE title_id_seq;
DROP SEQUENCE member_id_seq;

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```
PROMPT Creating Sequences...
CREATE SEQUENCE member_id_seq
  START WITH 101
  NOCACHE

CREATE SEQUENCE title_id_seq
  START WITH 92
  NOCACHE
/

PROMPT Sequences created.

PROMPT Run buildvid2.sql now to populate the above tables.
```



AnkurVikramTransfer: (brajapankur34@gmail.com) has a license to use this Student Guide.

Results Script Output Explain Autotrace DBMS Output OWA Output

Error starting at line 12 in command:
DROP TABLE rental CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 13 in command:
DROP TABLE reservation CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 14 in command:
DROP TABLE title_copy CASCADE CONSTRAINTS
Error report:

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 15 in command:
DROP TABLE title CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 16 in command:
DROP TABLE member CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
Please wait while tables are created....
CREATE TABLE succeeded.
Tables created.

Error starting at line 80 in command:
DROP SEQUENCE title_id_seq
Error report:
SQL Error: ORA-02289: sequence does not exist
02289. 00000 - "sequence does not exist"
*Cause: The specified sequence does not exist, or the user does
       not have the required privilege to perform this operation.
*Action: Make sure the sequence name is correct, and that you have
       the right to perform the desired operation on this sequence.

Error starting at line 81 in command:
DROP SEQUENCE member_id_seq
Error report:
Error report:
SQL Error: ORA-02289: sequence does not exist
02289. 00000 - "sequence does not exist"
*Cause: The specified sequence does not exist, or the user does
       not have the required privilege to perform this operation.
*Action: Make sure the sequence name is correct, and that you have
       the right to perform the desired operation on this sequence.

Creating Sequences...
CREATE SEQUENCE succeeded.
CREATE SEQUENCE succeeded.
Sequences created.
Run buildvid2.sql now to populate the above tables.

```

- 2) Load and execute the /home/oracle/labs/plpu/labs/buildvid2.sql script to populate all the tables created through the buildvid1.sql script.

Run the /home/oracle/labs/plpu/labs/buildvid2.sql script. The code, the connection prompt, and the results are displayed as follows:

```
/* Script to build the Video Application (Part 2 -
buildvid2.sql)
```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

This part of the script populates the tables that are
created using
buildvid1.sql
These are used by Part B of the Additional Practices of the
course.
You should run the script buildvid1.sql before running this
script to
    create the above tables.
*/

INSERT INTO member
VALUES  (member_id_seq.NEXTVAL, 'Velasquez', 'Carmen',
'283 King Street', 'Seattle', '587-99-6666', '03-MAR-90');
INSERT INTO member
VALUES  (member_id_seq.NEXTVAL, 'Ngao', 'LaDoris',
'5 Modrany', 'Bratislava', '586-355-8882', '08-MAR-90');
INSERT INTO member
VALUES  (member_id_seq.NEXTVAL,'Nagayama', 'Midori',
'68 Via Centrale', 'Sao Paolo', '254-852-5764', '17-JUN-
91');
INSERT INTO member
VALUES  (member_id_seq.NEXTVAL,'Quick-To-See','Mark',
'6921 King Way', 'Lagos', '63-559-777', '07-APR-90');
INSERT INTO member
VALUES  (member_id_seq.NEXTVAL, 'Ropeburn', 'Audry',
'86 Chu Street', 'Hong Kong', '41-559-87', '04-MAR-90');
INSERT INTO member
VALUES  (member_id_seq.NEXTVAL, 'Urguhart', 'Molly',
'3035 Laurier Blvd.', 'Quebec', '418-542-9988','18-JAN-
91');
INSERT INTO member
VALUES (member_id_seq.NEXTVAL, 'Menchu', 'Roberta',
'Boulevard de Waterloo 41', 'Brussels', '322-504-2228',
'14-MAY-90');
INSERT INTO member
VALUES (member_id_seq.NEXTVAL, 'Biri', 'Ben',
'398 High St.', 'Columbus', '614-455-9863', '07-APR-90');
INSERT INTO member
VALUES (member_id_seq.NEXTVAL, 'Catchpole', 'Antoinette',
'88 Alfred St.', 'Brisbane', '616-399-1411', '09-FEB-92');

COMMIT;

INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Willie and Christmas Too',
'All of Willie''s friends made a Christmas list for Santa,
but Willie has yet to create his own wish list.', 'G',
'CHILD', '05-OCT-95');

INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

VALUES (TITLE_ID_SEQ.NEXTVAL, 'Alien Again', 'Another
installment of science fiction history. Can the heroine save
the planet from the alien life form?', 'R', 'SCIFI',
'19-MAY-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'The Glob', 'A meteor crashes
near a small American town and unleashes carivorous goo in
this classic.', 'NR', 'SCIFI', '12-AUG-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'My Day Off', 'With a little
luck and a lot of ingenuity, a teenager skips school for a day
in New York.', 'PG', 'COMEDY', '12-JUL-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Miracles on Ice', 'A six-
year-old has doubts about Santa Claus. But she discovers that
miracles really do exist.', 'PG', 'DRAMA', '12-SEP-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Soda Gang', 'After
discovering a cache of drugs, a young couple find themselves
pitted against a vicious gang.', 'NR', 'ACTION', '01-JUN-95');
INSERT INTO title (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Interstellar Wars',
'Futuristic interstellar action movie. Can the rebels save the
humans from the evil Empire?', 'PG', 'SCIFI','07-JUL-77');

COMMIT;

INSERT INTO title_copy VALUES (1,92, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,93, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,93, 'RENTED');
INSERT INTO title_copy VALUES (1,94, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (3,95, 'RENTED');
INSERT INTO title_copy VALUES (1,96, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,97, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,98, 'RENTED');
INSERT INTO title_copy VALUES (2,98, 'AVAILABLE');
COMMIT;
INSERT INTO reservation VALUES (sysdate-1, 101, 93);
INSERT INTO reservation VALUES (sysdate-2, 106, 102);

COMMIT;

```


Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

1 rows inserted
1 rows inserted
COMMIT succeeded.
1 rows inserted
COMMIT succeeded.
** Tables built and data loaded **

```

- 3) Create a package named VIDEO_PKG with the following procedures and functions:
- NEW_MEMBER:** A public procedure that adds a new member to the MEMBER table. For the member ID number, use the sequence MEMBER_ID_SEQ; for the join date, use SYSDATE. Pass all other values to be inserted into a new row as parameters.
 - NEW_RENTAL:** An overloaded public function to record a new rental. Pass the title ID number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as AVAILABLE in the TITLE_COPY table for one copy of this title, then update this TITLE_COPY table and set the status to RENTED. If there is no copy available, the function must return NULL. Then, insert a new record into the RENTAL table identifying the booked date as today's date, the copy ID number, the member ID number, the title ID number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return NULL, and display a list of the customers' names that match and their ID numbers.
 - RETURN_MOVIE:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title ID, the copy ID, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the RENTAL table and set the actual return date to today's date. Update the status in the TITLE_COPY table based on the status parameter passed into the procedure.
 - RESERVE_MOVIE:** A private procedure that executes only if all the video copies requested in the NEW_RENTAL procedure have a status of RENTED. Pass the member ID number and the title ID number to this procedure. Insert a new record into the RESERVATION table and record the reservation date, member ID number, and title ID number. Print a message indicating that a movie is reserved and its expected date of return.

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

- e) **EXCEPTION_HANDLER:** A private procedure that is called from the exception handler of the public programs. Pass the SQLCODE number to this procedure, and the name of the program (as a text string) where the error occurred. Use RAISE_APPLICATION_ERROR to raise a customized error. Start with a unique key violation (-1) and foreign key violation (-2292). Allow the exception handler to raise a generic error for any other errors.

Run the /home/oracle/labs/plpu/solns/sol_ap_02_01_03.sql script. The code, the connection prompt, and the results are displayed as follows:

VIDEO_PKG Package Specification

```
CREATE OR REPLACE PACKAGE video_pkg IS
  PROCEDURE new_member
    (p_lname      IN member.last_name%TYPE,
     p_fname      IN member.first_name%TYPE      DEFAULT NULL,
     p_address    IN member.address%TYPE         DEFAULT NULL,
     p_city       IN member.city%TYPE            DEFAULT NULL,
     p_phone      IN member.phone%TYPE           DEFAULT NULL);

  FUNCTION new_rental
    (p_memberid   IN rental.member_id%TYPE,
     p_titleid    IN rental.title_id%TYPE)
  RETURN DATE;

  FUNCTION new_rental
    (p_membername IN member.last_name%TYPE,
     p_titleid    IN rental.title_id%TYPE)
  RETURN DATE;

  PROCEDURE return_movie
    (p_titleid    IN rental.title_id%TYPE,
     p_copyid     IN rental.copy_id%TYPE,
     p_sts        IN title_copy.status%TYPE);
END video_pkg;
/
SHOW ERRORS
```

```
CREATE OR REPLACE PACKAGE BODY video_pkg IS
  PROCEDURE exception_handler(errcode IN NUMBER, context IN VARCHAR2) IS
  BEGIN
    IF errcode = -1 THEN
      RAISE_APPLICATION_ERROR(-20001,
        'The number is assigned to this member is already in
use, |||
      'try again.');
    ELSIF errcode = -2291 THEN
```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        RAISE_APPLICATION_ERROR(-20002, context ||
        ' has attempted to use a foreign key value that is
invalid');
    ELSE
        RAISE_APPLICATION_ERROR(-20999, 'Unhandled error in ' ||
        context || '. Please contact your application ' ||
        'administrator with the following information: ' ||
        CHR(13) || SQLERRM);
    END IF;
END exception_handler;
PROCEDURE reserve_movie
(memberid IN reservation.member_id%TYPE,
 titleid IN reservation.title_id%TYPE) IS
CURSOR rented_csr IS
    SELECT exp_ret_date
    FROM rental
    WHERE title_id = titleid
    AND act_ret_date IS NULL;
BEGIN
    INSERT INTO reservation (res_date, member_id, title_id)
VALUES (SYSDATE, memberid, titleid);
    COMMIT;
    FOR rented_rec IN rented_csr LOOP
        DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on:
        ' || rented_rec.exp_ret_date);
        EXIT WHEN rented_csr%found;
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'RESERVE_MOVIE');
END reserve_movie;

PROCEDURE return_movie(
titleid IN rental.title_id%TYPE,
copyid IN rental.copy_id%TYPE,
sts IN title_copy.status%TYPE) IS
v_dummy VARCHAR2(1);
CURSOR res_csr IS
    SELECT *
    FROM reservation
    WHERE title_id = titleid;
BEGIN
    SELECT '' INTO v_dummy
    FROM title
    WHERE title_id = titleid;
    UPDATE rental
    SET act_ret_date = SYSDATE
    WHERE title_id = titleid
    AND copy_id = copyid AND act_ret_date IS NULL;

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        UPDATE title_copy
          SET status = UPPER(sts)
        WHERE title_id = titleid AND copy_id = copyid;
      FOR res_rec IN res_csr LOOP
        IF res_csr%FOUND THEN
          DBMS_OUTPUT.PUT_LINE('Put this movie on hold -- ' ||
                               'reserved by member #' || res_rec.member_id);
        END IF;
      END LOOP;
    EXCEPTION
      WHEN OTHERS THEN
        exception_handler(SQLCODE, 'RETURN_MOVIE');
    END return_movie;

  FUNCTION new_rental(
    memberid  IN rental.member_id%TYPE,
    titleid   IN rental.title_id%TYPE) RETURN DATE IS
    CURSOR copy_csr IS
      SELECT * FROM title_copy
      WHERE title_id = titleid
      FOR UPDATE;
    flag      BOOLEAN := FALSE;
  BEGIN

    FOR copy_rec IN copy_csr LOOP
      IF copy_rec.status = 'AVAILABLE' THEN
        UPDATE title_copy
          SET status = 'RENTED'
        WHERE CURRENT OF copy_csr;
        INSERT INTO rental(book_date, copy_id, member_id,
                           title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, memberid,
                           titleid, SYSDATE + 3);
        flag := TRUE;
        EXIT;
      END IF;
    END LOOP;
    COMMIT;
    IF flag THEN
      RETURN (SYSDATE + 3);
    ELSE
      reserve_movie(memberid, titleid);
      RETURN NULL;
    END IF;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_RENTAL');
  END new_rental;

  FUNCTION new_rental(
    membername IN member.last_name%TYPE,

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        titleid    IN rental.title_id%TYPE) RETURN DATE IS
CURSOR copy_csr IS
    SELECT * FROM title_copy
    WHERE title_id = titleid
    FOR UPDATE;
flag  BOOLEAN  := FALSE;
memberid  member.member_id%TYPE;
CURSOR member_csr IS
    SELECT member_id, last_name, first_name
    FROM member
    WHERE LOWER(last_name) = LOWER(membername)
    ORDER BY last_name, first_name;
BEGIN
    SELECT member_id INTO memberid
    FROM member
    WHERE lower(last_name) = lower(membername);
FOR copy_rec IN copy_csr LOOP
    IF copy_rec.status = 'AVAILABLE' THEN
        UPDATE title_copy
        SET status = 'RENTED'
        WHERE CURRENT OF copy_csr;
        INSERT INTO rental (book_date, copy_id, member_id,
                           title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, memberid,
                titleid, SYSDATE + 3);
        flag := TRUE;
        EXIT;
    END IF;
END LOOP;
COMMIT;
IF flag THEN
    RETURN(SYSDATE + 3);
ELSE
    reserve_movie(memberid, titleid);
    RETURN NULL;
END IF;
EXCEPTION
WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE(
        'Warning! More than one member by this name.');
    FOR member_rec IN member_csr LOOP
        DBMS_OUTPUT.PUT_LINE(member_rec.member_id || CHR(9) ||
        member_rec.last_name || ', ' ||
        member_rec.first_name);
    END LOOP;
    RETURN NULL;
WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
END new_rental;

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

PROCEDURE new_member(
    lname      IN member.last_name%TYPE,
    fname      IN member.first_name%TYPE      DEFAULT NULL,
    address    IN member.address%TYPE        DEFAULT NULL,
    city       IN member.city%TYPE           DEFAULT NULL,
    phone      IN member.phone%TYPE          DEFAULT NULL) IS
BEGIN
    INSERT INTO member(member_id, last_name, first_name,
                       address, city, phone, join_date)
        VALUES (member_id_seq.NEXTVAL, lname, fname,
                address, city, phone, SYSDATE);
    COMMIT;
CREATE OR REPLACE PACKAGE BODY video_pkg IS
    PROCEDURE exception_handler(errcode IN NUMBER, p_context IN VARCHAR2) IS
    BEGIN
        IF errcode = -1 THEN
            RAISE_APPLICATION_ERROR(-20001,
                'The number is assigned to this member is already in
use, ' ||
                'try again.');
        ELSIF errcode = -2291 THEN
            RAISE_APPLICATION_ERROR(-20002, p_context ||
                ' has attempted to use a foreign key value that is
invalid');
        ELSE
            RAISE_APPLICATION_ERROR(-20999, 'Unhandled error in ' ||
                p_context || '. Please contact your application ' ||
                'administrator with the following information: ' ||
                CHR(13) || SQLERRM);
        END IF;
    END exception_handler;

    PROCEDURE reserve_movie
        (p_memberid  IN reservation.member_id%TYPE,
         p_titleid   IN reservation.title_id%TYPE) IS
    CURSOR c_rented_csr IS
        SELECT exp_ret_date
        FROM rental
        WHERE title_id = p_titleid
        AND act_ret_date IS NULL;
    BEGIN
        INSERT INTO reservation (res_date, member_id, title_id)
        VALUES (SYSDATE, p_memberid, p_titleid);
        COMMIT;
        FOR rented_rec IN c_rented_csr LOOP
            DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on:
'
                || rented_rec.exp_ret_date);
        EXIT WHEN c_rented_csr%found;
    END;

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'RESERVE_MOVIE');
END reserve_movie;

PROCEDURE return_movie(
    p_titleid IN rental.title_id%TYPE,
    p_copyid IN rental.copy_id%TYPE,
    p_sts IN title_copy.status%TYPE) IS
    v_dummy VARCHAR2(1);
    CURSOR c_res_csr IS
        SELECT *
        FROM reservation
        WHERE title_id = p_titleid;
BEGIN
    SELECT '' INTO v_dummy
    FROM title
    WHERE title_id = p_titleid;
UPDATE rental
    SET act_ret_date = SYSDATE
    WHERE title_id = p_titleid
    AND copy_id = p_copyid AND act_ret_date IS NULL;
UPDATE title_copy
    SET status = UPPER(p_sts)
    WHERE title_id = p_titleid AND copy_id = p_copyid;
FOR res_rec IN c_res_csr LOOP
    IF c_res_csr%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Put this movie on hold -- ' ||
            'reserved by member #' || res_rec.member_id);
    END IF;
END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'RETURN_MOVIE');
END return_movie;

FUNCTION new_rental(
    p_memberid IN rental.member_id%TYPE,
    p_titleid IN rental.title_id%TYPE) RETURN DATE IS
    CURSOR c_copy_csr IS
        SELECT * FROM title_copy
        WHERE title_id = p_titleid
        FOR UPDATE;
    v_flag BOOLEAN := FALSE;
BEGIN
    FOR copy_rec IN c_copy_csr LOOP
        IF copy_rec.status = 'AVAILABLE' THEN
            UPDATE title_copy
            SET status = 'RENTED'
    END IF;
END LOOP;
    RETURN SYSDATE;
END new_rental;

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        WHERE CURRENT OF c_copy_csr;
        INSERT INTO rental(book_date, copy_id, member_id,
                           title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, p_memberid,
                p_titleid, SYSDATE + 3);
        v_flag := TRUE;
        EXIT;
    END IF;
END LOOP;
COMMIT;
IF v_flag THEN
    RETURN (SYSDATE + 3);
ELSE
    reserve_movie(p_memberid, p_titleid);
    RETURN NULL;
END IF;
EXCEPTION
WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
    RETURN NULL;
END new_rental;

FUNCTION new_rental(
    p_membername IN member.last_name%TYPE,
    p_titleid     IN rental.title_id%TYPE) RETURN DATE IS
CURSOR c_copy_csr IS
    SELECT * FROM title_copy
        WHERE title_id = p_titleid
        FOR UPDATE;
v_flag BOOLEAN := FALSE;
v_memberid member.member_id%TYPE;
CURSOR c_member_csr IS
    SELECT member_id, last_name, first_name
        FROM member
        WHERE LOWER(last_name) = LOWER(p_membername)
        ORDER BY last_name, first_name;
BEGIN
    SELECT member_id INTO v_memberid
        FROM member
        WHERE lower(last_name) = lower(p_membername);
    FOR copy_rec IN c_copy_csr LOOP
        IF copy_rec.status = 'AVAILABLE' THEN
            UPDATE title_copy
                SET status = 'RENTED'
                WHERE CURRENT OF c_copy_csr;
            INSERT INTO rental (book_date, copy_id, member_id,
                               title_id, exp_ret_date)
            VALUES (SYSDATE, copy_rec.copy_id, v_memberid,
                    p_titleid, SYSDATE + 3);
            v_flag := TRUE;
        END IF;
    END LOOP;
    COMMIT;
    RETURN (SYSDATE + 3);
EXCEPTION
WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
    RETURN NULL;
END new_rental;

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        EXIT;
    END IF;
END LOOP;
COMMIT;
IF v_flag THEN
    RETURN(SYSDATE + 3);
ELSE
    reserve_movie(v_memberid, p_titleid);
    RETURN NULL;
END IF;
EXCEPTION
WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE(
        'Warning! More than one member by this name.');
    FOR member_rec IN c_member_csr LOOP
        DBMS_OUTPUT.PUT_LINE(member_rec.member_id || CHR(9) ||
            member_rec.last_name || ', ' ||
            member_rec.first_name);
    END LOOP;
    RETURN NULL;
WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
    RETURN NULL;
END new_rental;

PROCEDURE new_member(
    p_lname      IN member.last_name%TYPE,
    p_fname       IN member.first_name%TYPE      DEFAULT NULL,
    p_address     IN member.address%TYPE         DEFAULT NULL,
    p_city        IN member.city%TYPE            DEFAULT NULL,
    p_phone       IN member.phone%TYPE          DEFAULT NULL) IS
BEGIN
    INSERT INTO member(member_id, last_name, first_name,
                      address, city, phone, join_date)
        VALUES(member_id_seq.NEXTVAL, p_lname, p_fname,
               p_address, p_city, p_phone, SYSDATE);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'NEW_MEMBER');
END new_member;
END video_pkg;
/
SHOW ERRORS

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

The screenshot shows the Oracle SQL Developer interface. At the top, there is a 'Select Connection' dialog box with the title 'Select Connection'. It contains a dropdown menu labeled 'Connection: VideoCompany' and a green '+' button. Below the dialog is a toolbar with icons for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. The main area displays the following text:

```

PACKAGE video_pkg Compiled.
No Errors.
PACKAGE BODY video_pkg Compiled.
No Errors.

```

- 4) Use the following scripts located in the /home/oracle/labs/plpu/soln directory to test your routines. Make sure you enable SERVEROUTPUT:
- Add two members using sol_ap_02_01_04_a.sql.
- Run the**
/home/oracle/labs/plpu/solns/sol_ap_02_01_04_a.sql script.
The code, the connection prompt, and the results are displayed as follows:

```

SET SERVEROUTPUT ON
EXECUTE video_pkg.new_member('Haas', 'James', 'Chestnut
Street', 'Boston', '617-123-4567')
EXECUTE video_pkg.new_member('Biri', 'Allan', 'Hiawatha
Drive', 'New York', '516-123-4567')

```

The screenshot shows the Oracle SQL Developer interface. At the top, there is a 'Select Connection' dialog box with the title 'Select Connection'. It contains a dropdown menu labeled 'Connection: VideoCompany' and a green '+' button. Below the dialog is a toolbar with icons for Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. The main area displays the following text:

```

anonymous block completed
anonymous block completed

```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

- b) Add new video rentals using sol_ap_02_01_04_b.sql.

Run the

/home/oracle/labs/plpu/solns/sol_ap_02_01_04_b.sql script.
The code, the connection prompt, and the results are displayed as follows:

```
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(110, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(109, 93))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(107, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental('Biri', 97))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(97, 97))
```



```
anonymous block completed
02-JUL-09

anonymous block completed
02-JUL-09

anonymous block completed
Movie reserved. Expected back on: 28-JUN-09

anonymous block completed
Warning! More than one member by this name.
111 Biri, Allan
108 Biri, Ben

Error starting at line 7 in command:
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(97, 97))
Error report:
ORA-20002: NEW_RENTAL has attempted to use a foreign key value that is invalid
ORA-06512: at "ORA62.VIDEO_PKG", line 9
ORA-06512: at "ORA62.VIDEO_PKG", line 103
ORA-06512: at line 1
```

- c) Return movies using the sol_ap_02_01_04_c.sql script.

Run the

/home/oracle/labs/plpu/solns/sol_ap_02_01_04_c.sql script.
The code, the connection prompt, and the results are displayed as follows:

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```
SET SERVEROUTPUT ON

EXECUTE video_pkg.return_movie(98, 1, 'AVAILABLE')
EXECUTE video_pkg.return_movie(95, 3, 'AVAILABLE')
EXECUTE video_pkg.return_movie(111, 1, 'RENTED')
```



```
anonymous block completed
Put this movie on hold -- reserved by member #107

anonymous block completed

Error starting at line 5 in command:
EXECUTE video_pkg.return_movie(111, 1, 'RENTED')
Error report:
ORA-20999: Unhandled error in RETURN_MOVIE. Please contact your application administrator with the following information:
ORA-01403: no data found
ORA-06512: at "ORA62.VIDEO_PKG", line 12
ORA-06512: at "ORA62.VIDEO_PKG", line 69
ORA-06512: at line 1
```

- 5) The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 12:00 PM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.
- Create a stored procedure called TIME_CHECK that checks the current time against business hours. If the current time is not within business hours, use the RAISE_APPLICATION_ERROR procedure to give an appropriate message.

Run the

/home/oracle/labs/plpu/solns/sol_ap_02_01_05_a.sql script.
The code, the connection prompt, and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE time_check IS
BEGIN
  IF ((TO_CHAR(SYSDATE, 'D') BETWEEN 1 AND 6) AND
      (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT
       BETWEEN
```

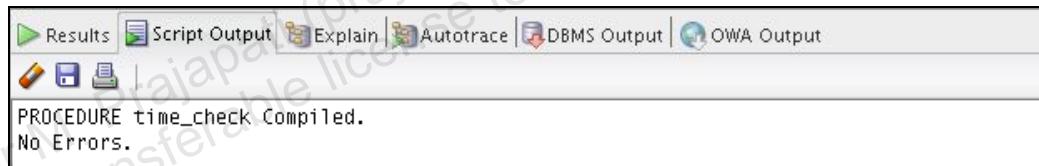
Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

        TO_DATE('08:00', 'hh24:mi') AND TO_DATE('22:00',
'hh24:mi'))
    OR ((TO_CHAR(SYSDATE, 'D') = 7)
        AND (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'),
'hh24:mi') NOT BETWEEN
            TO_DATE('08:00', 'hh24:mi') AND TO_DATE('24:00',
'hh24:mi'))) THEN
        RAISE_APPLICATION_ERROR(-20999,
        'Data changes restricted to office hours.');
    END IF;
END time_check;
/
SHOW ERRORS

PROCEDURE time_check Compiled.
No Errors.

```



- b) Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your TIME_CHECK procedure from each of these triggers.

Run the

/home/oracle/labs/plpu/solns/sol_ap_02_01_05_b.sql script.
The code and the result are displayed as follows:

```

CREATE OR REPLACE TRIGGER member_trig
    BEFORE INSERT OR UPDATE OR DELETE ON member
    CALL time_check
/

CREATE OR REPLACE TRIGGER rental_trig
    BEFORE INSERT OR UPDATE OR DELETE ON rental
    CALL time_check

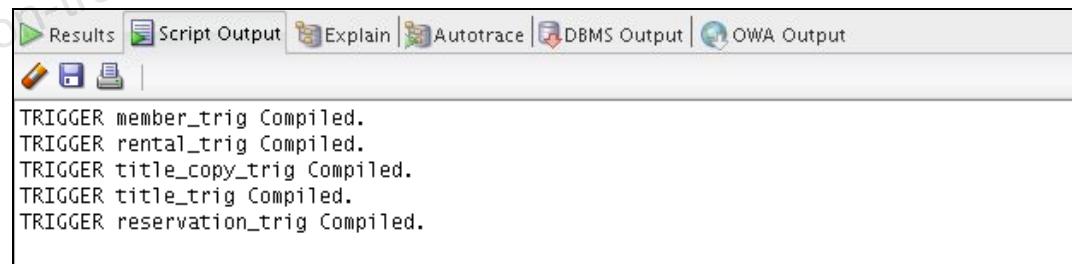
```

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

```

/
CREATE OR REPLACE TRIGGER title_copy_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title_copy
CALL time_check
/
CREATE OR REPLACE TRIGGER title_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title
CALL time_check
/
CREATE OR REPLACE TRIGGER reservation_trig
  BEFORE INSERT OR UPDATE OR DELETE ON reservation
CALL time_check
/
TRIGGER member_trig Compiled.
TRIGGER rental_trig Compiled.
TRIGGER title_copy_trig Compiled.
TRIGGER title_trig Compiled.
TRIGGER reservation_trig Compiled.

```



- c) Test your triggers.

Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

Practice Solutions 2-1: Creating the VIDEO_PKG Package (continued)

Run the

/home/oracle/labs/plpu/solns/sol_ap_02_01_05_C.sql script.

The code and the result are displayed as follows:

```
-- First determine current timezone and time
SELECT SESSIONTIMEZONE,
       TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY HH24:MI')
CURR_DATE
FROM DUAL;

-- Change your time zone usinge [+|-]HH:MI format such that
-- the current time returns a time between 6pm and 8am

ALTER SESSION SET TIME_ZONE='-07:00';

-- Add a new member (for a sample test)

EXECUTE video_pkg.new_member('Elias', 'Elliane', 'Vine
Street', 'California', '789-123-4567')

BEGIN video_pkg.new_member('Elias', 'Elliane', 'Vine
Street', 'California', '789-123-4567'); END;

-- Restore the original time zone for your session.

ALTER SESSION SET TIME_ZONE='00:00';
```

SESSIONTIMEZONE	CURR_DATE
Etc/GMT-7	29-JUN-2009 21:51

1 rows selected

ALTER SESSION SET succeeded.
anonymous block completed
ALTER SESSION SET succeeded.

B

Table Descriptions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Schema Description

Overall Description

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

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

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

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

Human Resources (HR)

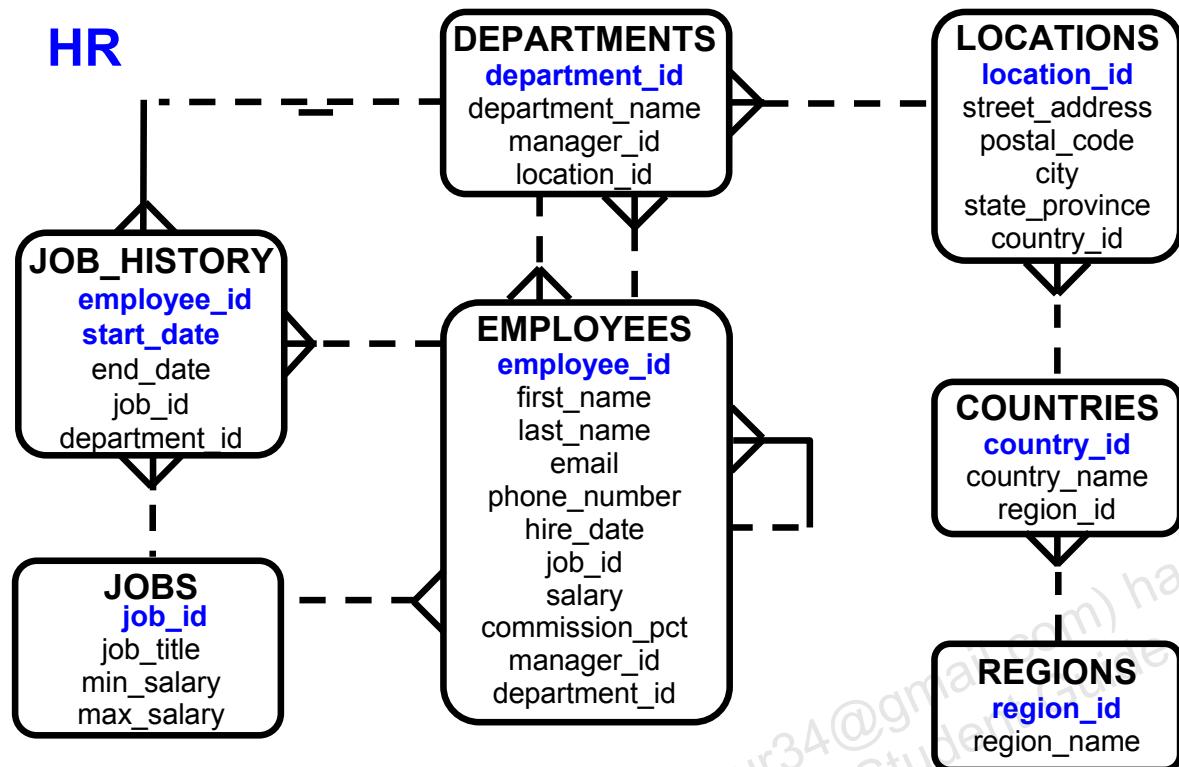
This is the schema that is used in this course. In the Human Resource (HR) records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

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

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

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

The HR Entity Relationship Diagram



The Human Resources (HR) Table Descriptions

DESCRIBE countries

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

SELECT * FROM countries

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

The Human Resources (HR) Table Descriptions (continued)

DESCRIBE departments

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

SELECT * FROM departments

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10 Administration	200	1700
2	20 Marketing	201	1800
3	30 Purchasing	114	1700
4	40 Human Resources	203	2400
5	50 Shipping	121	1500
6	60 IT	103	1400
7	70 Public Relations	204	2700
8	80 Sales	145	2500
9	90 Executive	100	1700
10	100 Finance	108	1700
11	110 Accounting	205	1700
12	120 Treasury	(null)	1700
13	130 Corporate Tax	(null)	1700
14	140 Control And Credit	(null)	1700
15	150 Shareholder Services	(null)	1700
16	160 Benefits	(null)	1700
17	170 Manufacturing	(null)	1700
18	180 Construction	(null)	1700
19	190 Contracting	(null)	1700
20	200 Operations	(null)	1700
21	210 IT Support	(null)	1700
22	220 NOC	(null)	1700
23	230 IT Helpdesk	(null)	1700
24	240 Government Sales	(null)	1700
25	250 Retail Sales	(null)	1700
26	260 Recruiting	(null)	1700
27	270 Payroll	(null)	1700
28	980 Education	(null)	2500
29	280 Training	(null)	2400

The Human Resources (HR) Table Descriptions (continued)

DESCRIBE employees

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

SELECT * FROM employees

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100 Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	(null)	(null)	90
1	100 Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	(null)	(null)	90
3	102 Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	(null)	100	90
4	103 Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	(null)	102	60
5	104 Bruce	Ernst	BERNSTEIN	590.423.4568	21-MAY-91	IT_PROG	6000	(null)	103	60
6	105 David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800	(null)	103	60
7	106 Valli	Pataballa	VPAATABALLA	590.423.4560	05-FEB-98	IT_PROG	4800	(null)	103	60
8	107 Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200	(null)	103	60
9	108 Nancy	Greenberg	NGRGRNBERG	515.124.4569	17-AUG-94	FI_MGR	12000	(null)	101	100
10	109 Daniel	Faviet	DFAFVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000	(null)	108	100
11	110 John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200	(null)	108	100
12	111 Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700	(null)	108	100
13	112 Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800	(null)	108	100
14	113 Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900	(null)	108	100
15	114 Den	Raphaely	DRAPEL	515.127.4561	07-DEC-94	PU_MAN	11000	(null)	100	30
16	115 Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100	(null)	114	30
17	116 Shelli	Baida	SBAIL	515.127.4563	24-DEC-97	PU_CLERK	2900	(null)	114	30
18	117 Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800	(null)	114	30
19	118 Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600	(null)	114	30
20	119 Karen	Colmenares	KCOCOLMENARES	515.127.4566	10-AUG-99	PU_CLERK	2500	(null)	114	30
21	120 Matthew	Weiss	MWEWEISS	650.123.1234	18-JUL-96	ST_MAN	8000	(null)	100	50
22	121 Adam	Fripp	AFRIPPI	650.123.2234	10-APR-97	ST_MAN	8200	(null)	100	50
23	122 Payam	Kaufling	PKAUFLING	650.123.3234	01-MAY-95	ST_MAN	7900	(null)	100	50
24	123 Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500	(null)	100	50
25	124 Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800	(null)	100	50
26	125 Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200	(null)	120	50
27	126 Irene	Mikkilineni	IMIKKILINENI	650.124.1224	28-SEP-98	ST_CLERK	2700	(null)	120	50
28	127 James	Landry	JLAJLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400	(null)	120	50
29	128 Steven	Markle	SMAJMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200	(null)	120	50
30	129 Laura	Bissot	LBISBISOT	650.124.5234	20-AUG-97	ST_CLERK	3300	(null)	121	50
31	130 Mozhe	Atkinson	MATMOSHE	650.124.6234	30-OCT-97	ST_CLERK	2800	(null)	121	50
32	131 James	Marlow	JAMMARLOW	650.124.7234	16-FEB-97	ST_CLERK	2500	(null)	121	50
33	132 TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100	(null)	121	50
34	133 Jason	Mallin	JMAMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300	(null)	122	50
35	134 Michael	Rogers	MROJROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900	(null)	122	50
36	135 Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400	(null)	122	50
37	136 Hazel	Philtanker	PHILANKER	650.127.1634	06-FEB-00	ST_CLERK	2200	(null)	122	50
38	137 Renske	Ladwig	RLALADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600	(null)	123	50

The Human Resources (HR) Table Descriptions (continued)

Employees (continued)

39	138 Stephen	Stiles	SSTI...	650.121.2034	26-OCT-97	ST_CLERK	3200	(null)	123	50
40	139 John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700	(null)	123	50
41	140 Joshua	Patel	JPAT...	650.121.1834	06-APR-98	ST_CLERK	2500	(null)	123	50
42	141 Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500	(null)	124	50
43	142 Curtis	Davies	CDA...	650.121.2994	29-JAN-97	ST_CLERK	3100	(null)	124	50
44	143 Randall	Matos	RMA...	650.121.2874	15-MAR-98	ST_CLERK	2600	(null)	124	50
45	144 Peter	Vargas	PVA...	650.121.2004	09-JUL-98	ST_CLERK	2500	(null)	124	50
46	145 John	Russell	JRU...	011.44.1344.42...	01-OCT-96	SA_MAN	14000	0.4	100	80
47	146 Karen	Partners	KPA...	011.44.1344.46...	05-JAN-97	SA_MAN	13500	0.3	100	80
48	147 Alberto	Errazuriz	AER...	011.44.1344.42...	10-MAR-97	SA_MAN	12000	0.3	100	80
49	148 Gerald	Cambrault	GCA...	011.44.1344.61...	15-OCT-99	SA_MAN	11000	0.3	100	80
50	149 Eleni	Zlotkey	EZL...	011.44.1344.42...	29-JAN-00	SA_MAN	10500	0.2	100	80
51	150 Peter	Tucker	PTU...	011.44.1344.12...	30-JAN-97	SA REP	10000	0.3	145	80
52	151 David	Bernstein	DBE...	011.44.1344.34...	24-MAR-97	SA REP	9500	0.25	145	80
53	152 Peter	Hall	PHALL	011.44.1344.47...	20-AUG-97	SA REP	9000	0.25	145	80
54	153 Christopher	Olsen	COL...	011.44.1344.49...	30-MAR-98	SA REP	8000	0.2	145	80
55	154 Nanette	Cambrault	NCA...	011.44.1344.98...	09-DEC-98	SA REP	7500	0.2	145	80
56	155 Oliver	Tuvault	OTU...	011.44.1344.48...	23-NOV-99	SA REP	7000	0.15	145	80
57	156 Janette	King	JKING	011.44.1345.42...	30-JAN-96	SA REP	10000	0.35	146	80
58	157 Patrick	Sully	PSU...	011.44.1345.92...	04-MAR-96	SA REP	9500	0.35	146	80
59	158 Allan	McEwen	AMC...	011.44.1345.82...	01-AUG-96	SA REP	9000	0.35	146	80
60	159 Lindsey	Smith	LSMI...	011.44.1345.72...	10-MAR-97	SA REP	8000	0.3	146	80
61	160 Louise	Doran	LDO...	011.44.1345.62...	15-DEC-97	SA REP	7500	0.3	146	80
62	161 Sarath	Sewall	SSE...	011.44.1345.52...	03-NOV-98	SA REP	7000	0.25	146	80
63	162 Clara	Vishney	CVIS...	011.44.1346.12...	11-NOV-97	SA REP	10500	0.25	147	80
64	163 Danielle	Greene	DGR...	011.44.1346.22...	19-MAR-99	SA REP	9500	0.15	147	80
65	164 Mattea	Marvins	MMA...	011.44.1346.32...	24-JAN-00	SA REP	7200	0.1	147	80
66	165 David	Lee	DLEE	011.44.1346.52...	23-FEB-00	SA REP	6800	0.1	147	80
67	166 Sundar	Ande	SAN...	011.44.1346.62...	24-MAR-00	SA REP	6400	0.1	147	80
68	167 Amit	Banda	ABA...	011.44.1346.72...	21-APR-00	SA REP	6200	0.1	147	80
69	168 Lisa	Ozer	LOZER	011.44.1343.92...	11-MAR-97	SA REP	11500	0.25	148	80
70	169 Harrison	Bloom	HBL...	011.44.1343.82...	23-MAR-98	SA REP	10000	0.2	148	80
71	170 Tayler	Fox	TFOX	011.44.1343.72...	24-JAN-98	SA REP	9600	0.2	148	80
72	171 William	Smith	WSM...	011.44.1343.62...	23-FEB-99	SA REP	7400	0.15	148	80
73	172 Elizabeth	Bates	EBA...	011.44.1343.52...	24-MAR-99	SA REP	7300	0.15	148	80
74	173 Sundita	Kumar	SKU...	011.44.1343.32...	21-APR-00	SA REP	6100	0.1	148	80

The Human Resources (HR) Table Descriptions (continued)

Employees (continued)

75	174 Ellen	Abel	EABEL	011.44.1644.42...	11-MAY-96	SA_REP	11000	0.3	149	80
76	175 Alyssa	Hutton	AHULL	011.44.1644.42...	19-MAR-97	SA_REP	8800	0.25	149	80
77	176 Jonathon	Taylor	JTA...	011.44.1644.42...	24-MAR-98	SA_REP	8600	0.2	149	80
78	177 Jack	Livingston	JLIVI...	011.44.1644.42...	23-APR-98	SA_REP	8400	0.2	149	80
79	178 Kimberly	Grant	KGR...	011.44.1644.42...	24-MAY-99	SA_REP	7000	0.15	149	(null)
80	179 Charles	Johnson	CJO...	011.44.1644.42...	04-JAN-00	SA_REP	6200	0.1	149	80
81	180 Winston	Taylor	WT...	650.507.9876	24-JAN-98	SH_CLERK	3200	(null)	120	50
82	181 Jean	Fleur	JFLE...	650.507.9877	23-FEB-98	SH_CLERK	3100	(null)	120	50
83	182 Martha	Sullivan	MSU...	650.507.9878	21-JUN-99	SH_CLERK	2500	(null)	120	50
84	183 Girard	Geoni	GGE...	650.507.9879	03-FEB-00	SH_CLERK	2800	(null)	120	50
85	184 Nandita	Sarchand	NSA...	650.509.1876	27-JAN-96	SH_CLERK	4200	(null)	121	50
86	185 Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100	(null)	121	50
87	186 Julia	Dellinger	JDEL...	650.509.3876	24-JUN-98	SH_CLERK	3400	(null)	121	50
88	187 Anthony	Cabrio	ACA...	650.509.4876	07-FEB-99	SH_CLERK	3000	(null)	121	50
89	188 Kelly	Chung	KCH...	650.505.1876	14-JUN-97	SH_CLERK	3800	(null)	122	50
90	189 Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600	(null)	122	50
91	190 Timothy	Gates	TGA...	650.505.3876	11-JUL-98	SH_CLERK	2900	(null)	122	50
92	191 Randall	Perkins	RPE...	650.505.4876	19-DEC-99	SH_CLERK	2500	(null)	122	50
93	192 Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000	(null)	123	50
94	193 Britney	Everett	BEV...	650.501.2876	03-MAR-97	SH_CLERK	3900	(null)	123	50
95	194 Samuel	McCain	SMC...	650.501.3876	01-JUL-98	SH_CLERK	3200	(null)	123	50
96	195 Vance	Jones	VJO...	650.501.4876	17-MAR-99	SH_CLERK	2800	(null)	123	50
97	196 Alana	Walsh	AW...	650.507.9811	24-APR-98	SH_CLERK	3100	(null)	124	50
98	197 Kevin	Feehey	KFEE...	650.507.9822	23-MAY-98	SH_CLERK	3000	(null)	124	50
99	198 Donald	O'Connell	DOC...	650.507.9833	21-JUN-99	SH_CLERK	2600	(null)	124	50
100	199 Douglas	Grant	DGR...	650.507.9844	13-JAN-00	SH_CLERK	2600	(null)	124	50
101	200 Jennifer	Whalen	JWH...	515.123.4444	17-SEP-87	AD_ASST	4400	(null)	101	10
102	201 Michael	Hartstein	MHA...	515.123.5555	17-FEB-96	MK_MAN	13000	(null)	100	20
103	202 Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK REP	6000	(null)	201	20
104	203 Susan	Mavris	SMA...	515.123.7777	07-JUN-94	HR REP	6500	(null)	101	40
105	204 Hermann	Baer	HBA...	515.123.8888	07-JUN-94	PR REP	10000	(null)	101	70
106	205 Shelley	Higgins	SHIG...	515.123.8080	07-JUN-94	AC_MGR	12000	(null)	101	110
107	206 William	Gietz	WGI...	515.123.8181	07-JUN-94	AC_ACCOUNT	8300	(null)	205	110

The Human Resources (HR) Table Descriptions (continued)

DESCRIBE job_history

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

SELECT * FROM job_history

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
1	102 13-JAN-93	24-JUL-98	IT_PROG	60
2	101 21-SEP-89	27-OCT-93	AC_ACCOUNT	110
3	101 28-OCT-93	15-MAR-97	AC_MGR	110
4	201 17-FEB-96	19-DEC-99	MK_REP	20
5	114 24-MAR-98	31-DEC-99	ST_CLERK	50
6	122 01-JAN-99	31-DEC-99	ST_CLERK	50
7	200 17-SEP-87	17-JUN-93	AD_ASST	90
8	176 24-MAR-98	31-DEC-98	SA_REP	80
9	176 01-JAN-99	31-DEC-99	SA_MAN	80
10	200 01-JUL-94	31-DEC-98	AC_ACCOUNT	90

The Human Resources (HR) Table Descriptions (continued)

DESCRIBE jobs

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

SELECT * FROM jobs

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1 AD_PRES	President	20000	40000
2 AD_VP	Administration Vice President	15000	30000
3 AD_ASST	Administration Assistant	3000	6000
4 FI_MGR	Finance Manager	8200	16000
5 FI_ACCOUNT	Accountant	4200	9000
6 AC_MGR	Accounting Manager	8200	16000
7 AC_ACCOUNT	Public Accountant	4200	9000
8 SA_MAN	Sales Manager	10000	20000
9 SA_REP	Sales Representative	6000	12000
10 PU_MAN	Purchasing Manager	8000	15000
11 PU_CLERK	Purchasing Clerk	2500	5500
12 ST_MAN	Stock Manager	5500	8500
13 ST_CLERK	Stock Clerk	2000	5000
14 SH_CLERK	Shipping Clerk	2500	5500
15 IT_PROG	Programmer	4000	10000
16 MK_MAN	Marketing Manager	9000	15000
17 MK_REP	Marketing Representative	4000	9000
18 HR_REP	Human Resources Representative	4000	9000
19 PR_REP	Public Relations Representative	4500	10500

The Human Resources (HR) Table Descriptions (continued)

DESCRIBE locations

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

SELECT * FROM locations

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

The Human Resources (HR) Table Descriptions (continued)

DESCRIBE regions

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

SELECT * FROM locations

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

C

Using SQL Developer

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (ankur34@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

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

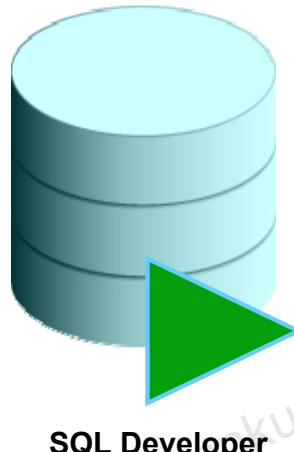
- List the key features of Oracle SQL Developer
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports



Copyright © 2009, Oracle. All rights reserved.

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.



ORACLE

Copyright © 2009, Oracle. 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* that 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.

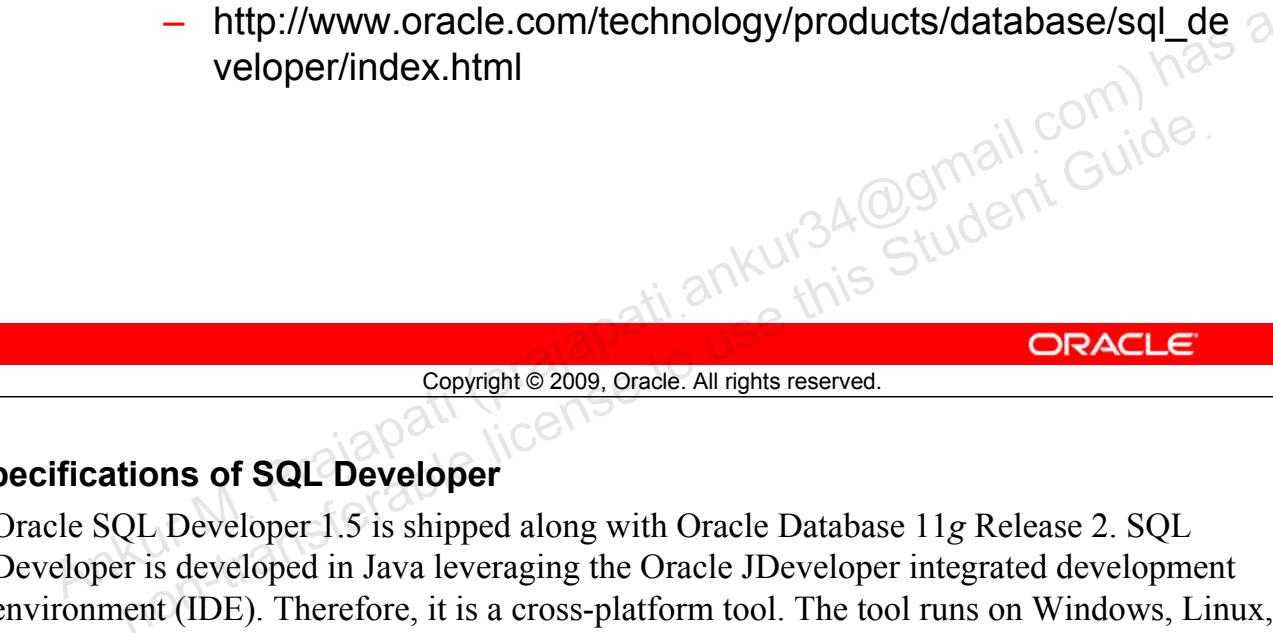
Additionally, SQL Developer includes support for Oracle Application Express 3.0.1 (Oracle APEX).

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



Copyright © 2009, Oracle. 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

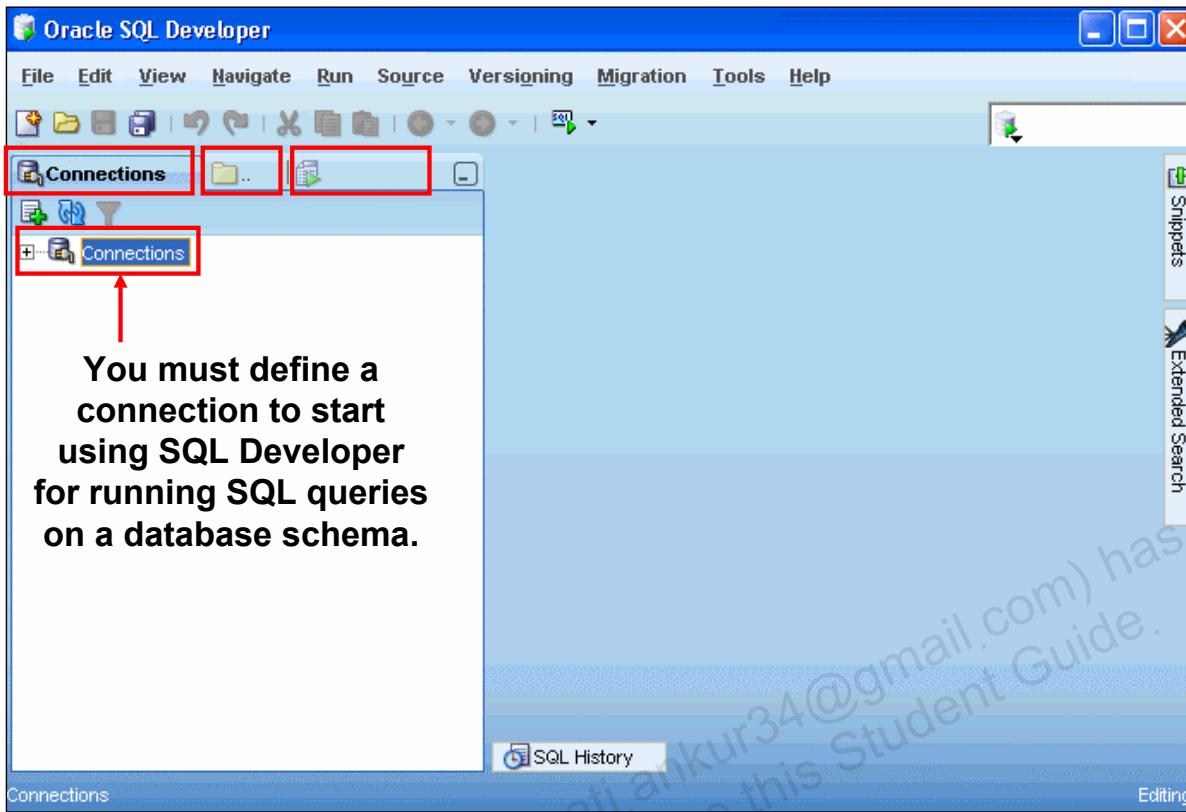
For Oracle Database versions earlier than Oracle Database 11g Release 2, you will have to download and install SQL Developer. SQL Developer 1.5 is freely downloadable from the following link:

http://www.oracle.com/technology/products/database/sql_developer/index.html.

For instructions on how to install SQL Developer, you can visit the following link:

http://download.oracle.com/docs/cd/E12151_01/index.htm

SQL Developer 1.5 Interface



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/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 various panes and for executing subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **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

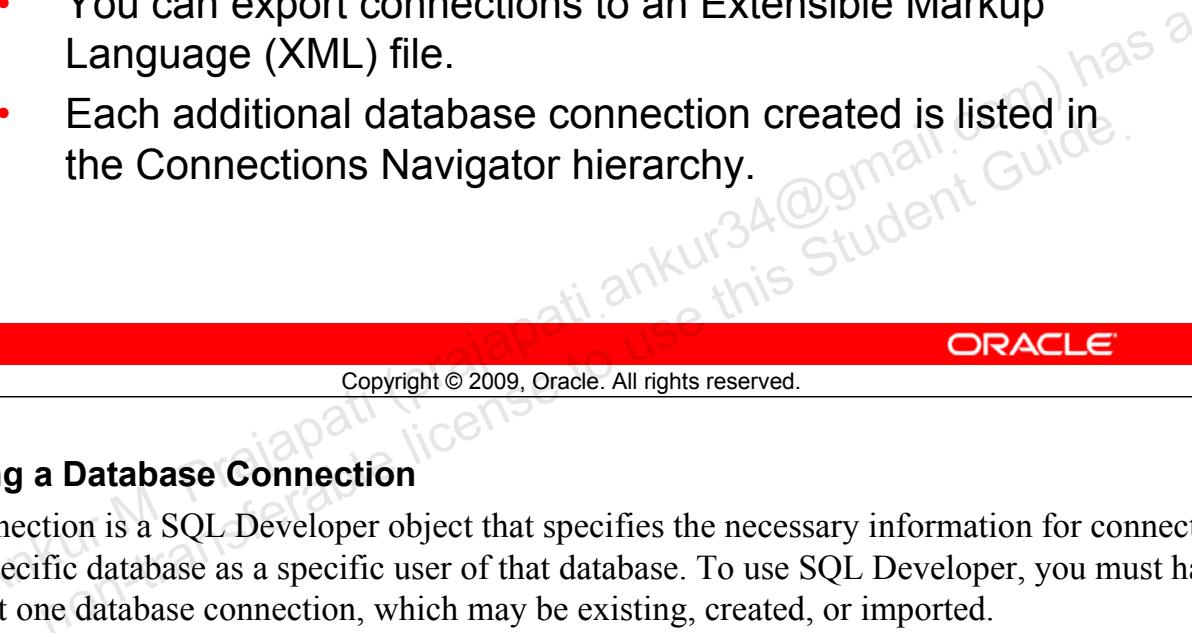
Note: The Run menu also contains options that are relevant when a function or procedure is selected for debugging. These are the same options that are found in the Debug menu in version 1.2.

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 © 2009, Oracle. 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.

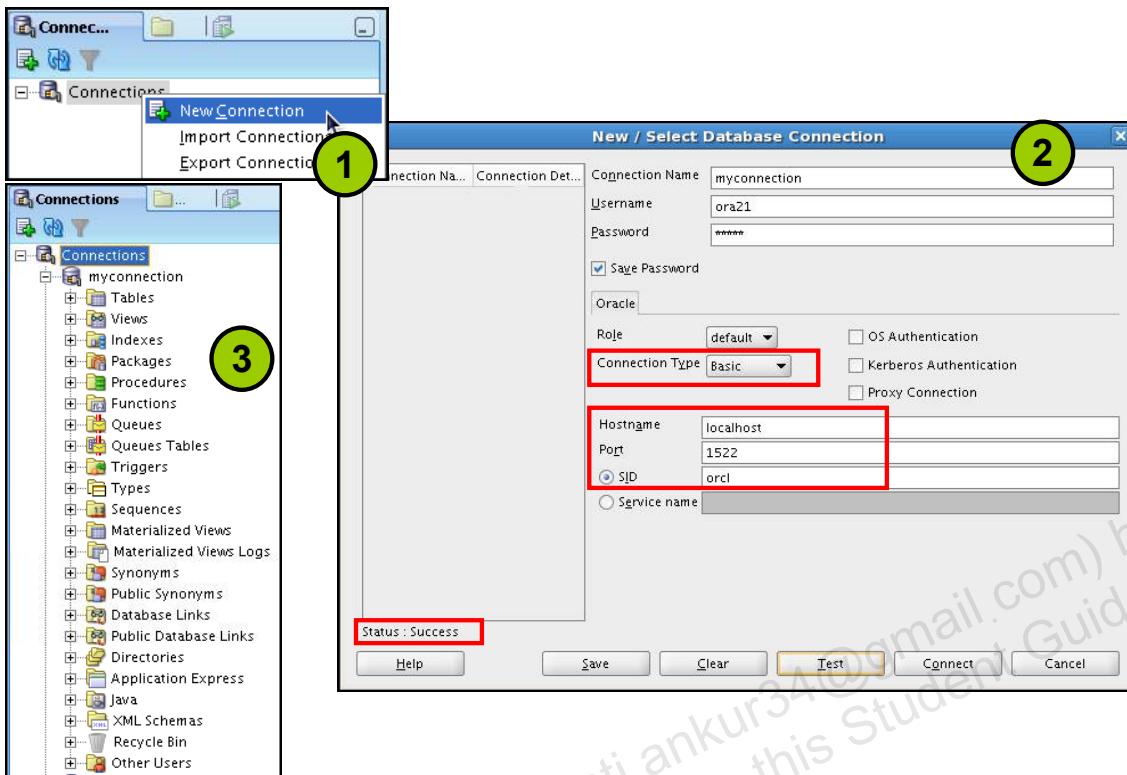
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically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



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

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

Browsing Database Objects

Use the Connections Navigator to:

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

The screenshot shows the Oracle SQL Developer interface. On the left, the Connections Navigator displays a tree structure of database objects under 'myconnection'. The 'EMPLOYEES' table is selected and highlighted with a red box. On the right, the main workspace shows the 'EMPLOYEES' table definition across several tabs: Columns, Data, Constraints, Grants, Statistics, Triggers, Flashback, and Dependencies. The 'Columns' tab is selected and highlighted with a red box. The table structure is as follows:

Column Name	Data Type	Nullable	Data Default	Collation
EMPLOYEE_ID	NUMBER(6,0)	No	(null)	
FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	
LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	
EMAIL	VARCHAR2(25 BYTE)	No	(null)	
PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	
HIRE_DATE	DATE	No	(null)	
JOB_ID	VARCHAR2(10 BYTE)	No	(null)	
SALARY	NUMBER(8,2)	Yes	(null)	
COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	
MANAGER_ID	NUMBER(6,0)	Yes	(null)	
DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	

ORACLE

Copyright © 2009, Oracle. 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.

You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, 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 Oracle SQL Developer interface. The top status bar indicates "1.69686902 seconds" and the connection name "myconnection". The main window displays the results of the DESCRIBE EMPLOYEES command. The output shows the column names, whether they are nullable (NULL or NOT NULL), and their data types. There are 11 rows selected.

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

11 rows selected

ORACLE

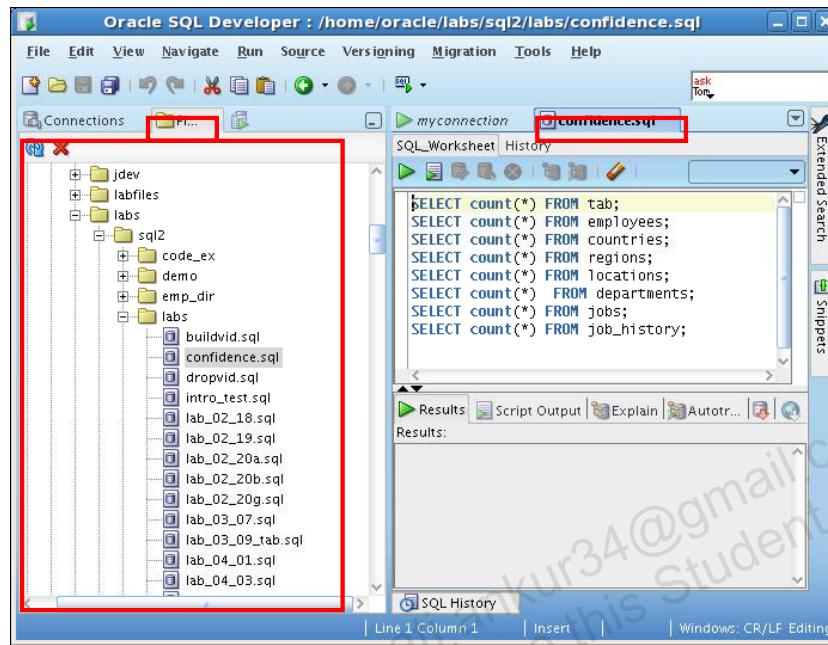
Copyright © 2009, Oracle. All rights reserved.

Displaying the Table Structure

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

Browsing Files

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



ORACLE

Copyright © 2009, Oracle. All rights reserved.

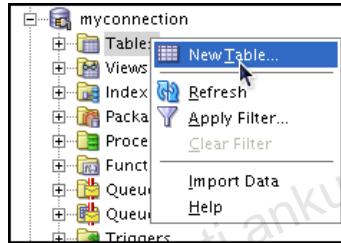
Browsing Database Objects

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

- To view the files 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 © 2009, Oracle. All rights reserved.

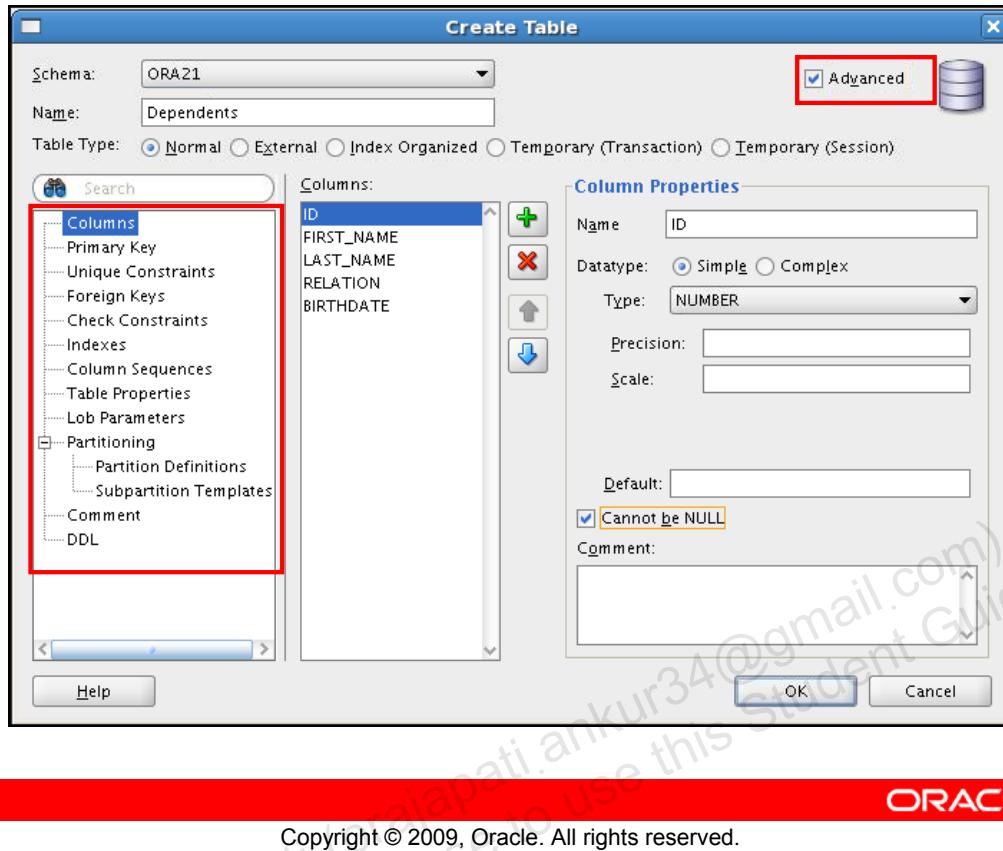
Creating a Schema Object

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

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

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

Creating a New Table: Example



ORACLE

Copyright © 2009, Oracle. 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.

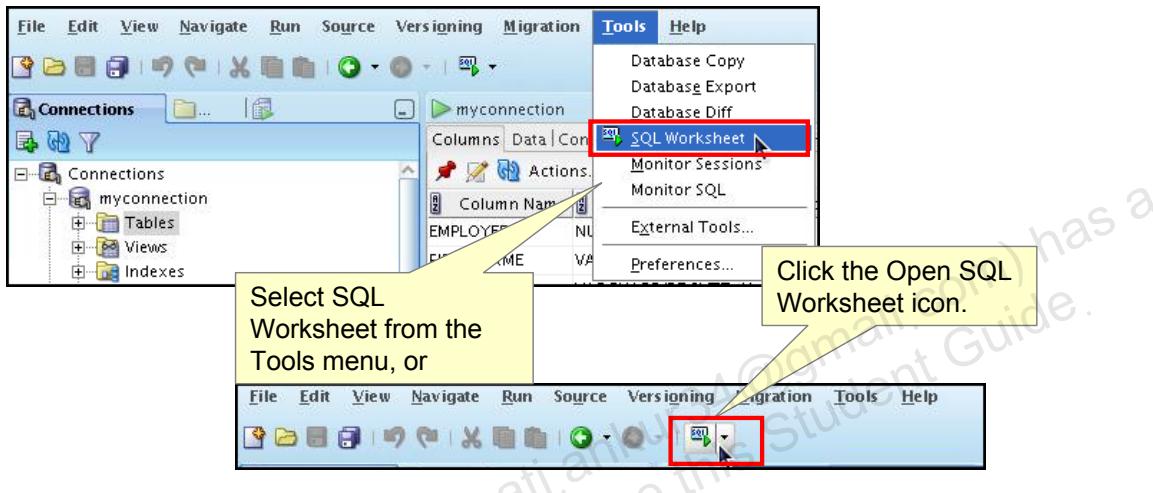
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables.
2. Select Create 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 by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created; to do so, right-click the table in the Connections Navigator and select Edit.

Using the SQL Worksheet

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



ORACLE

Copyright © 2009, Oracle. 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.

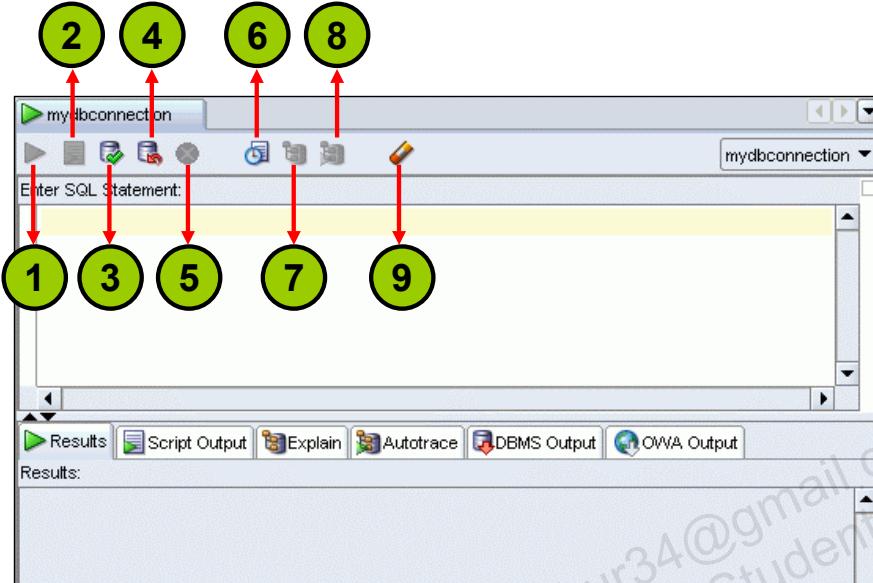
You can specify actions that can be processed by the database connection associated with the worksheet, such as:

- 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 © 2009, Oracle. All rights reserved.

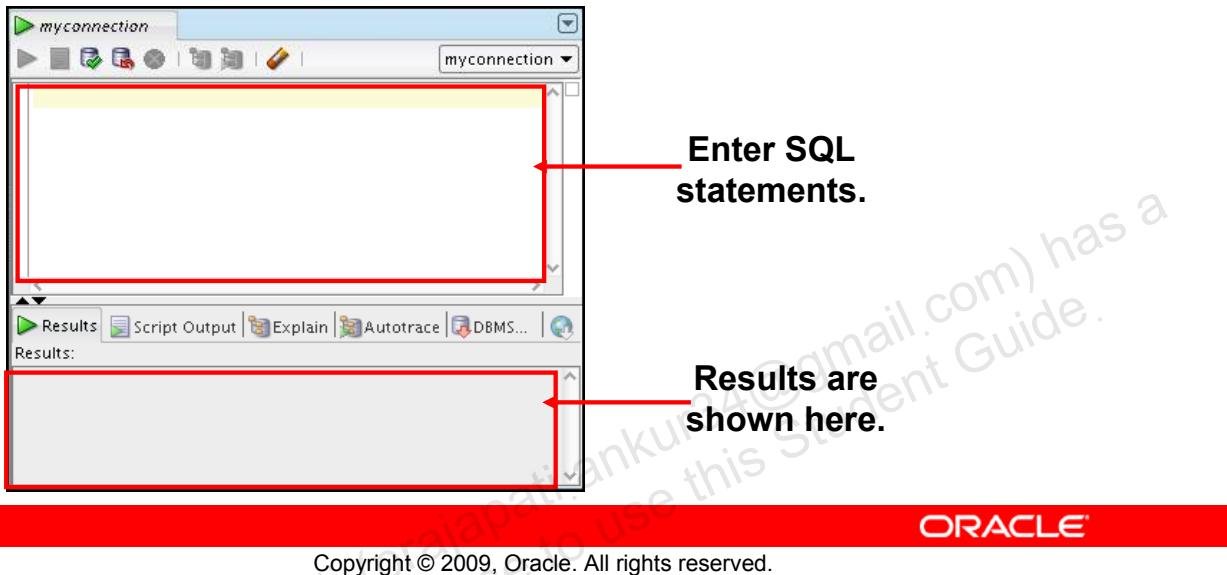
Using the SQL Worksheet (continued)

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. You can use the SQL Worksheet toolbar that contains icons 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.



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.

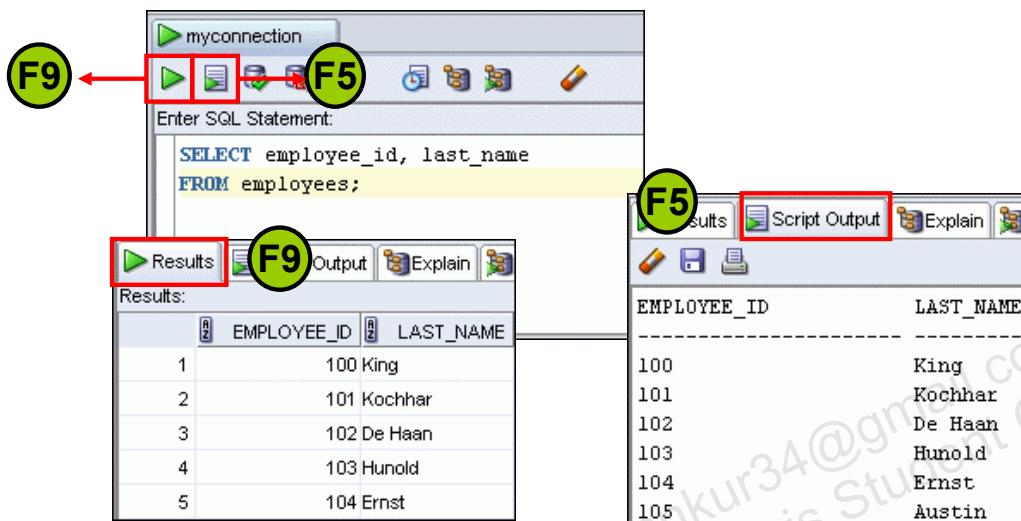
The SQL Worksheet currently supports a number of SQL*Plus commands. Commands not supported by the SQL Worksheet are ignored and are not sent to the Oracle database. Through the SQL Worksheet, you can execute SQL statements and some of the SQL*Plus commands.

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.

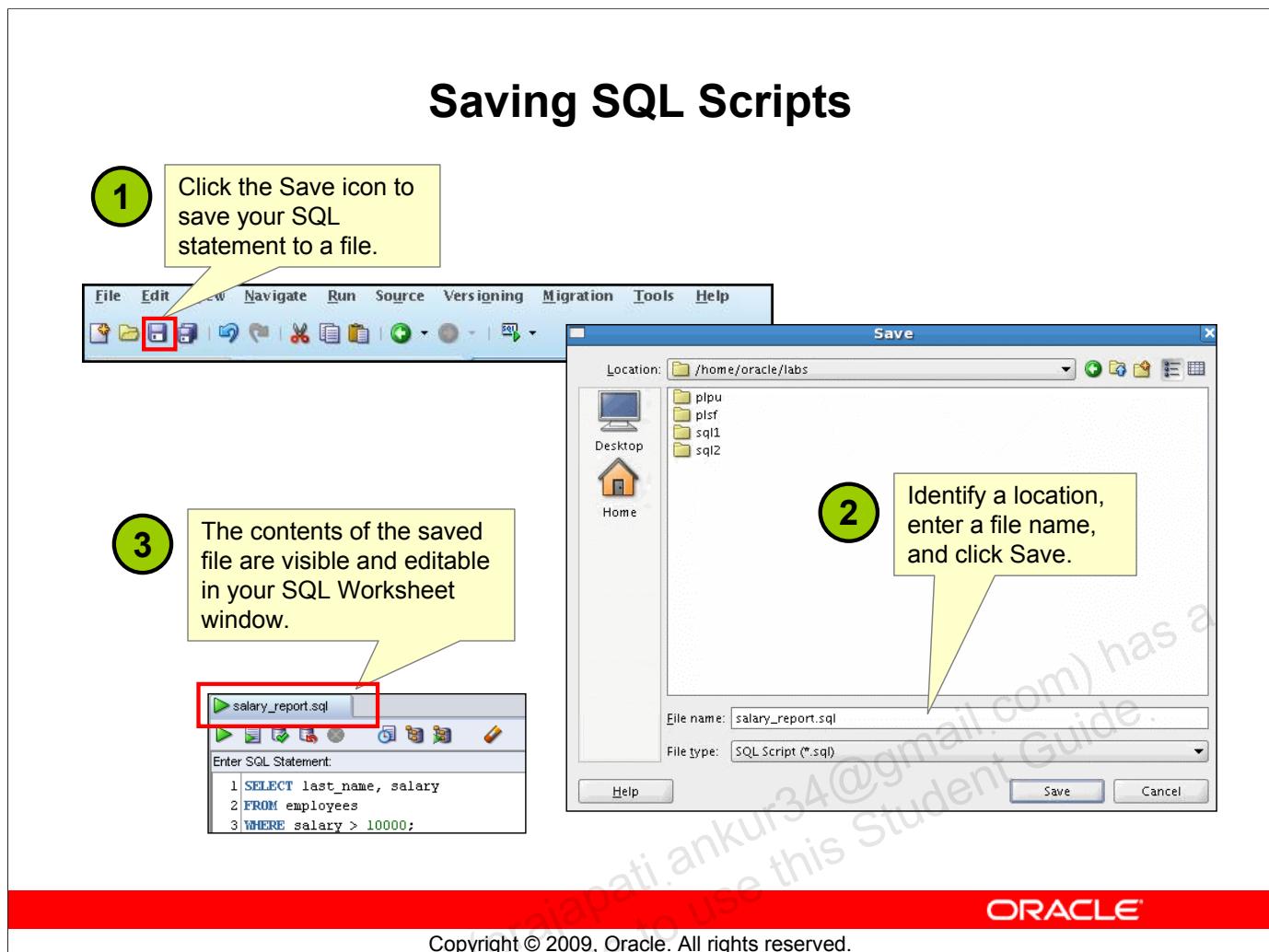


ORACLE

Copyright © 2009, Oracle. 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.



ORACLE

Copyright © 2009, Oracle. 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:

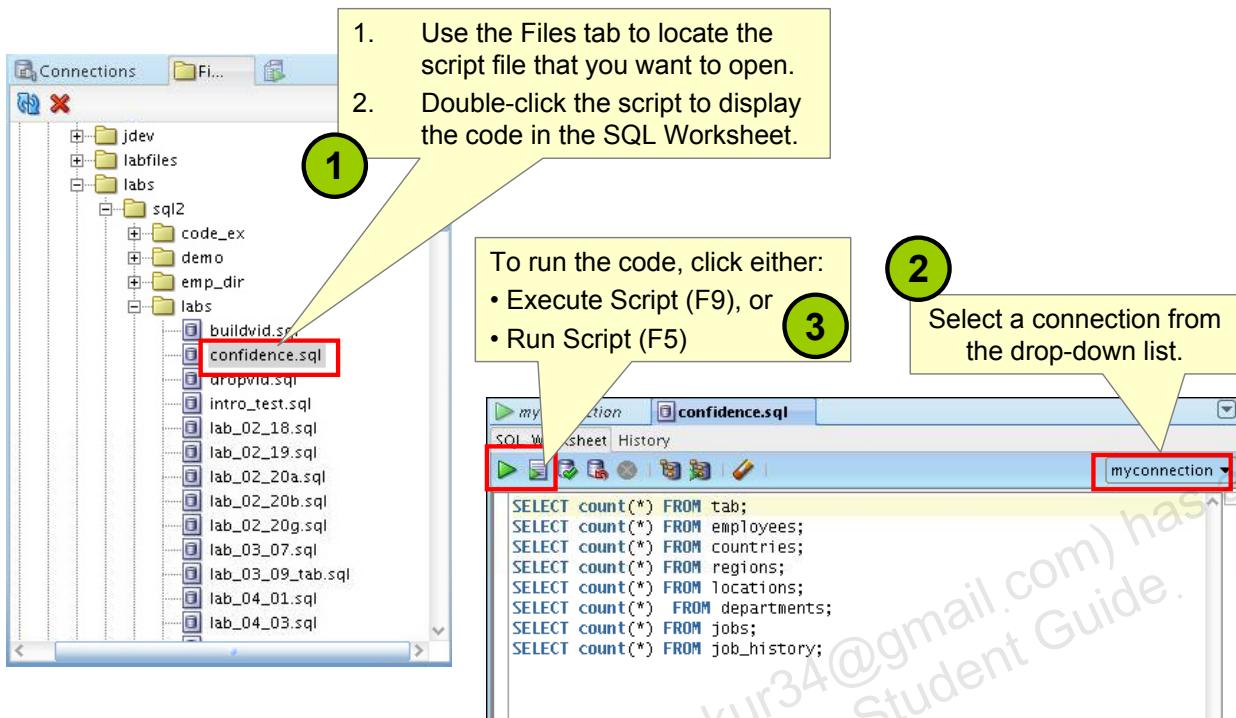
1. Click the Save icon or use the File > Save menu item.
2. In the Windows Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

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 © 2009, Oracle. All rights reserved.

Executing Saved Script Files: Method 1

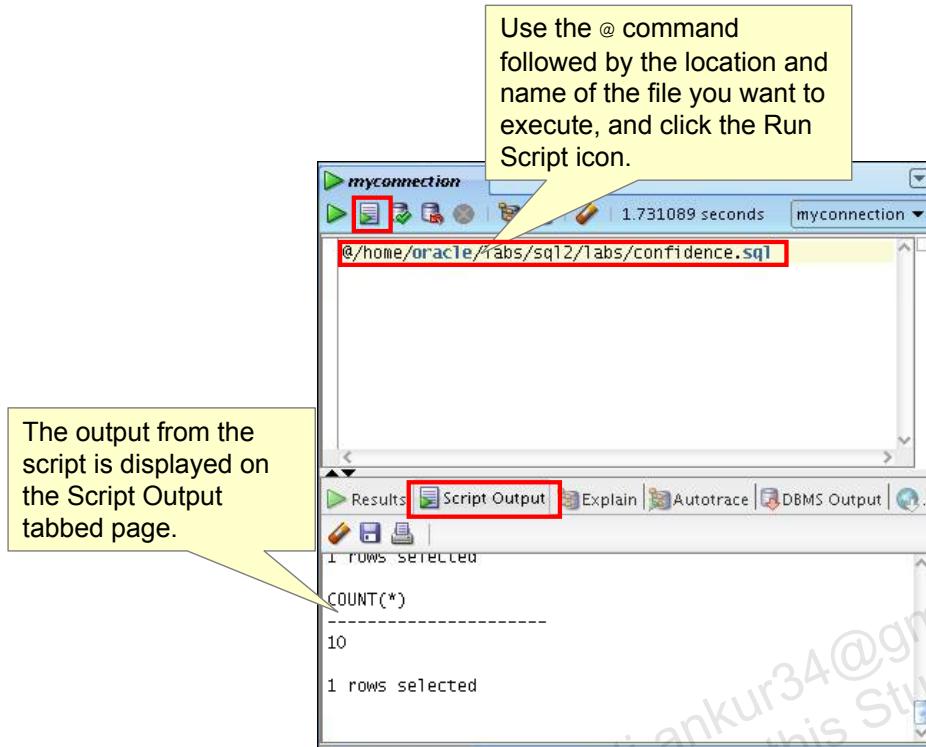
To open a script file and display the code in the SQL Worksheet area, perform the following:

1. In the files navigator select (or navigate to) the script file that you want to open.
2. Double-click to open. The 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 also:

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 you want to use for the script execution.

Executing Saved Script Files: Method 2



ORACLE

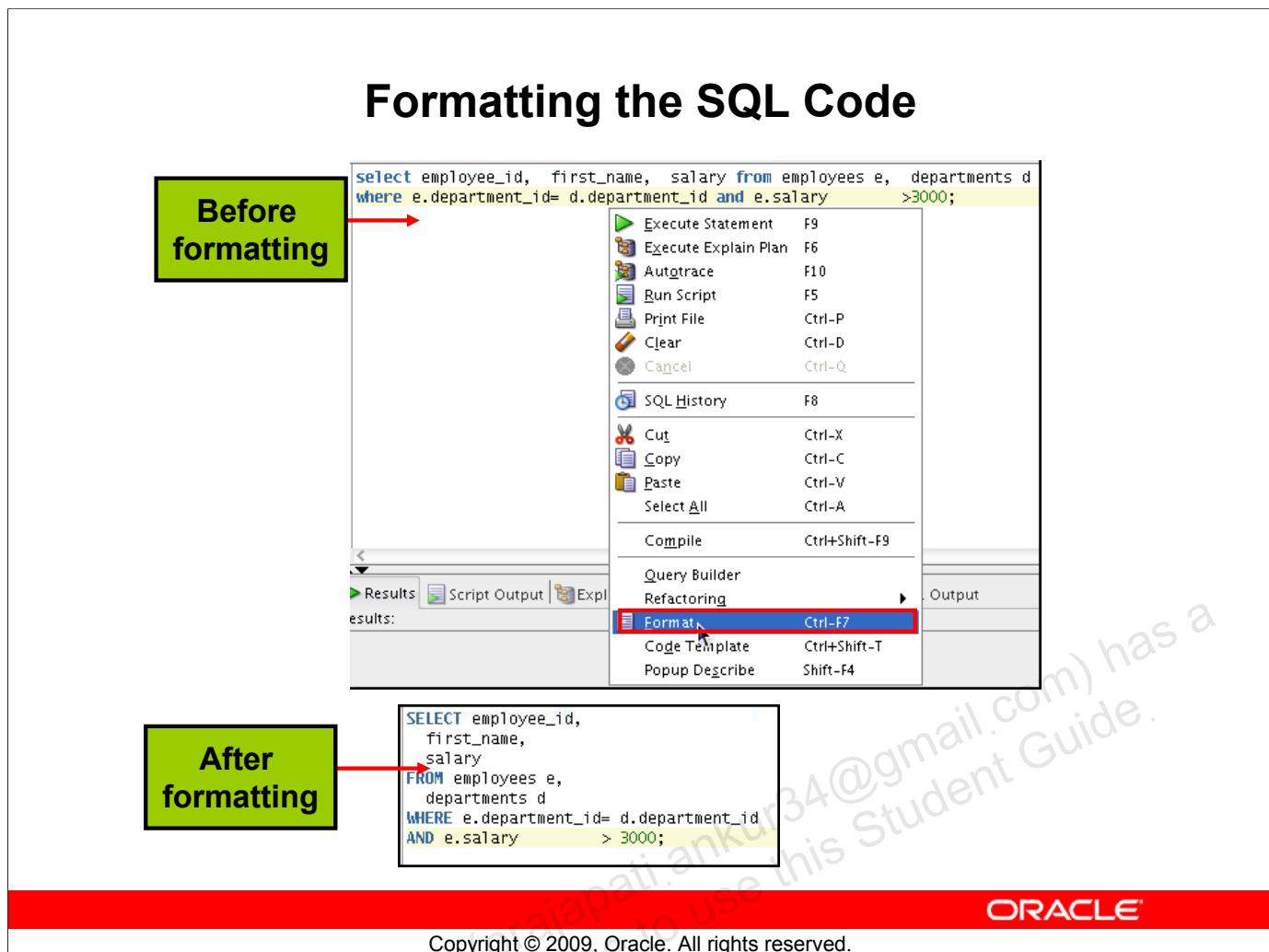
Copyright © 2009, Oracle. All rights reserved.

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 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.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Formatting the SQL Code

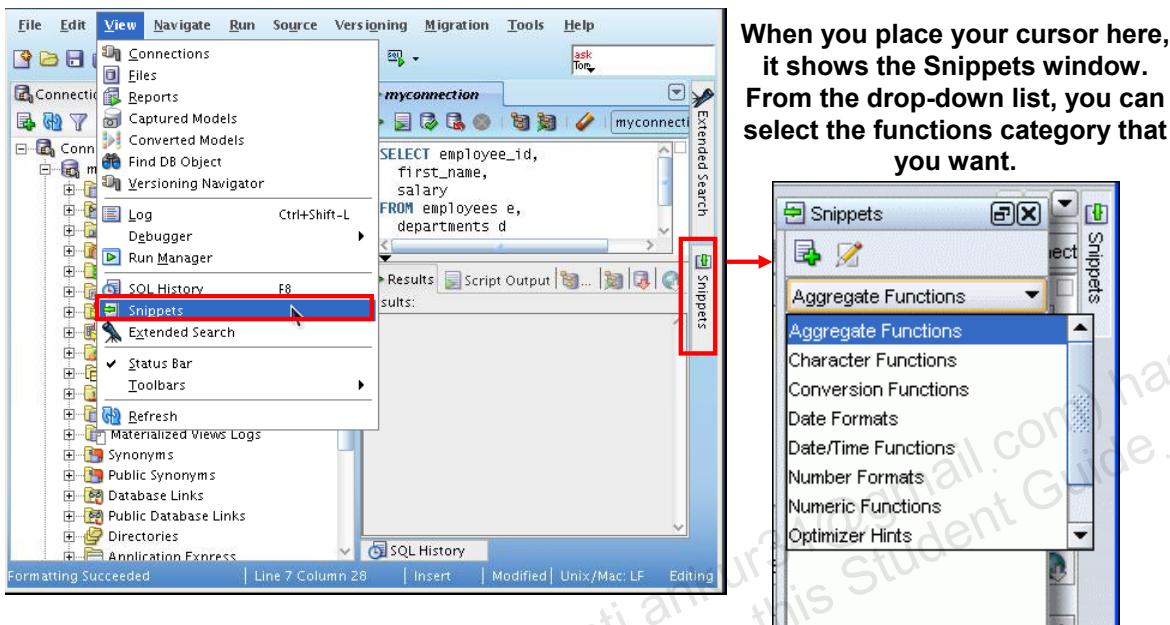
You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

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.



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

ORACLE

Copyright © 2009, Oracle. All rights reserved.

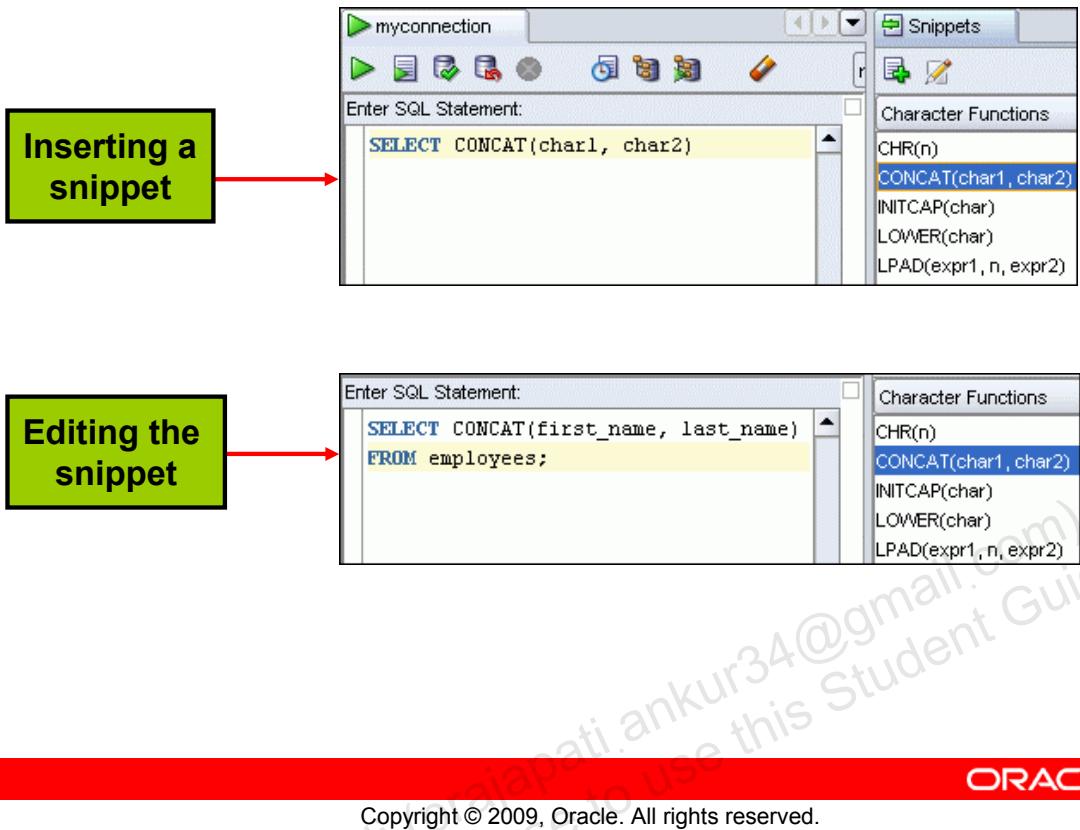
Using Snippets

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



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Using Snippets: Example

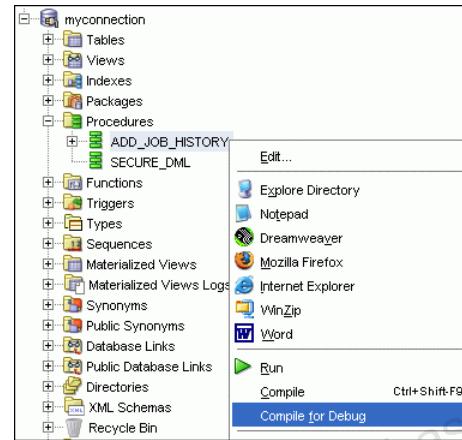
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function 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 © 2009, Oracle. All rights reserved.

Debugging Procedures and Functions

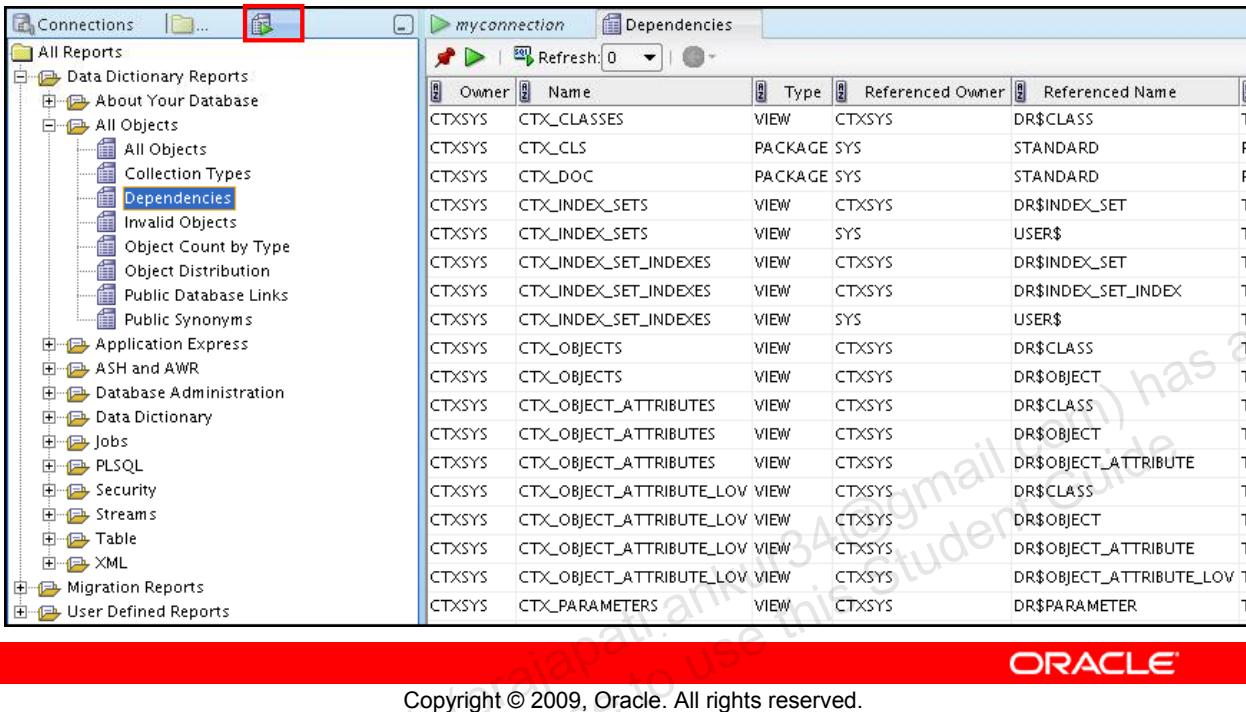
In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **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.



The screenshot shows the Oracle SQL Developer interface. The left sidebar has a tree view of report categories. A red box highlights the 'Reports' tab icon at the top of the sidebar. The main pane shows a table of database objects under the connection 'myconnection'. The table has columns: Owner, Name, Type, Referenced Owner, and Referenced Name. Most rows belong to the owner 'CTXSYS'. The table includes rows for various system objects like CTX_CLASSES, CTX_CLS, CTX_DOC, etc.

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 © 2009, Oracle. All rights reserved.

Database Reporting

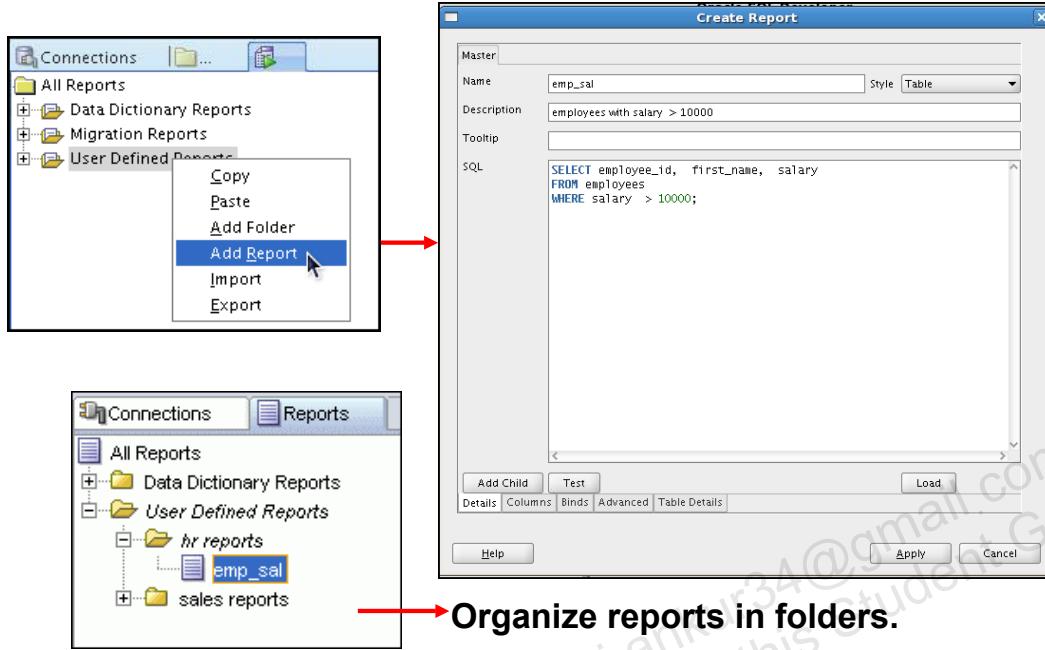
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- 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.



ORACLE

Copyright © 2009, Oracle. 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:

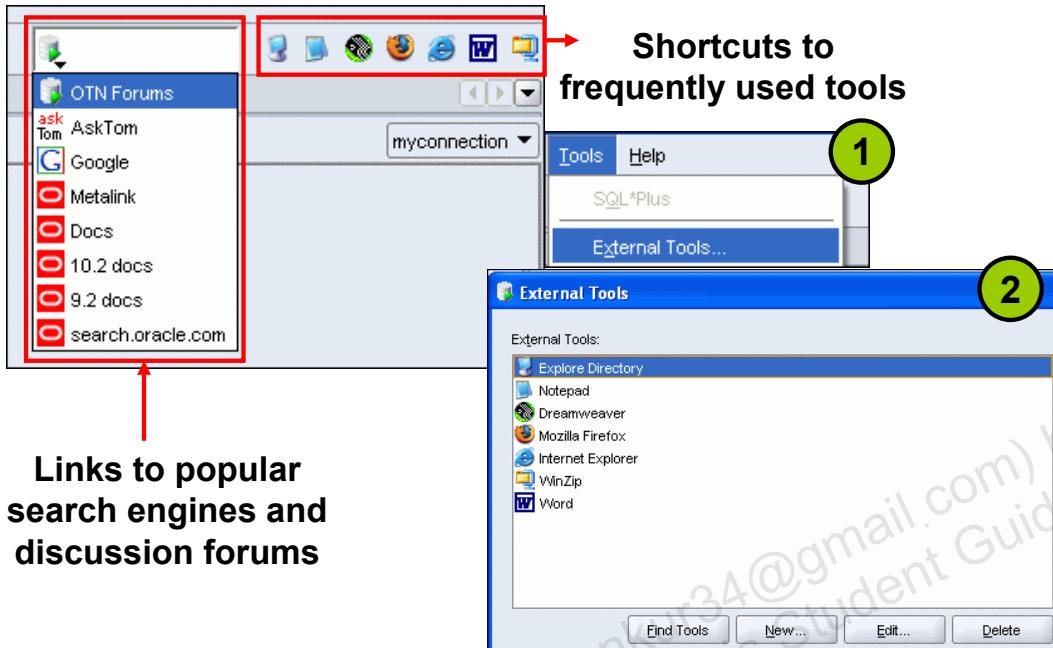
1. Right-click the User Defined Reports node under Reports, and select Add Report.
2. In the Create Report Dialog box, specify the report name and the SQL query to retrieve information for the report. Then, click Apply.

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 © 2009, Oracle. All rights reserved.

Search Engines and External Tools

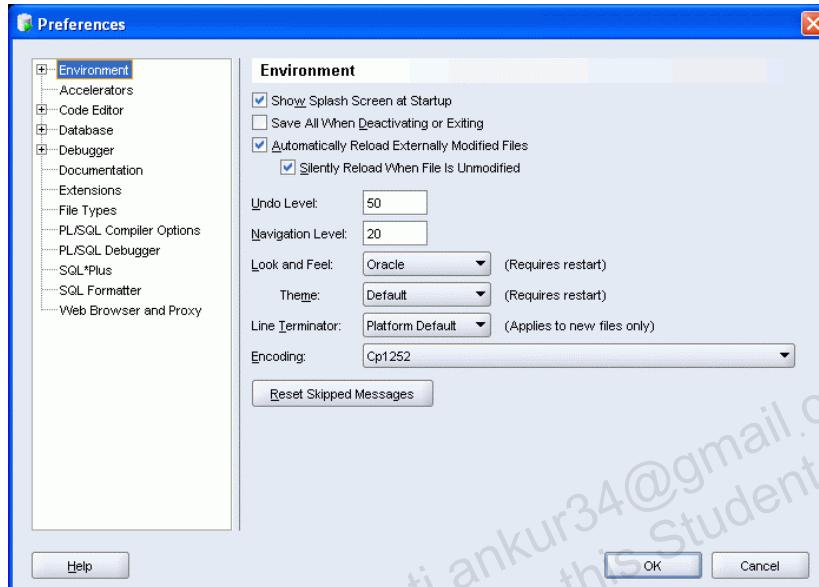
To enhance productivity of the SQL developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to 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 © 2009, Oracle. 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, 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

The screenshot shows a terminal window titled "Terminal". The user has run the command `locate windowinglayout.xml`, which has found several files named `windowinglayout.xml` in different directories under `/home/oracle/.sqldeveloper/system1.5.4.59.40/o.ide.11.1.1.0.22.49.48`. The user then changes directory to `/home/oracle/.sqldeveloper/system1.5.4.59.41/o.ide.11.1.1.0.22.49.48` and runs the command `rm windowinglayout.xml` to delete the file. The terminal window has a red bar at the bottom with the text "Copyright © 2009, Oracle. All rights reserved." and the ORACLE logo.

```
[oracle@EDRSR5P1 ~]$ locate windowinglayout.xml
/home/oracle/.sqldeveloper/system1.5.4.59.40/o.ide.11.1.1.0.22.49.48/windowinglayout.xml
/home/oracle/.sqldeveloper/system1.5.4.59.41/o.ide.11.1.1.0.22.49.48/windowinglayout.xml
[oracle@EDRSR5P1 ~]$ cd /home/oracle/.sqldeveloper/system1.5.4.59.41/o.ide.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 © 2009, Oracle. 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 from SQL Developer.
2. Open a terminal window and use the `locate` command to find the location of `windowinglayout.xml`.
3. Go to the directory which 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

The red bar spans most of the page width, with a white space on the left side.

ORACLE

Copyright © 2009, Oracle. 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.

Using SQL*Plus

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

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

The red bar spans most of the page width, centered horizontally.

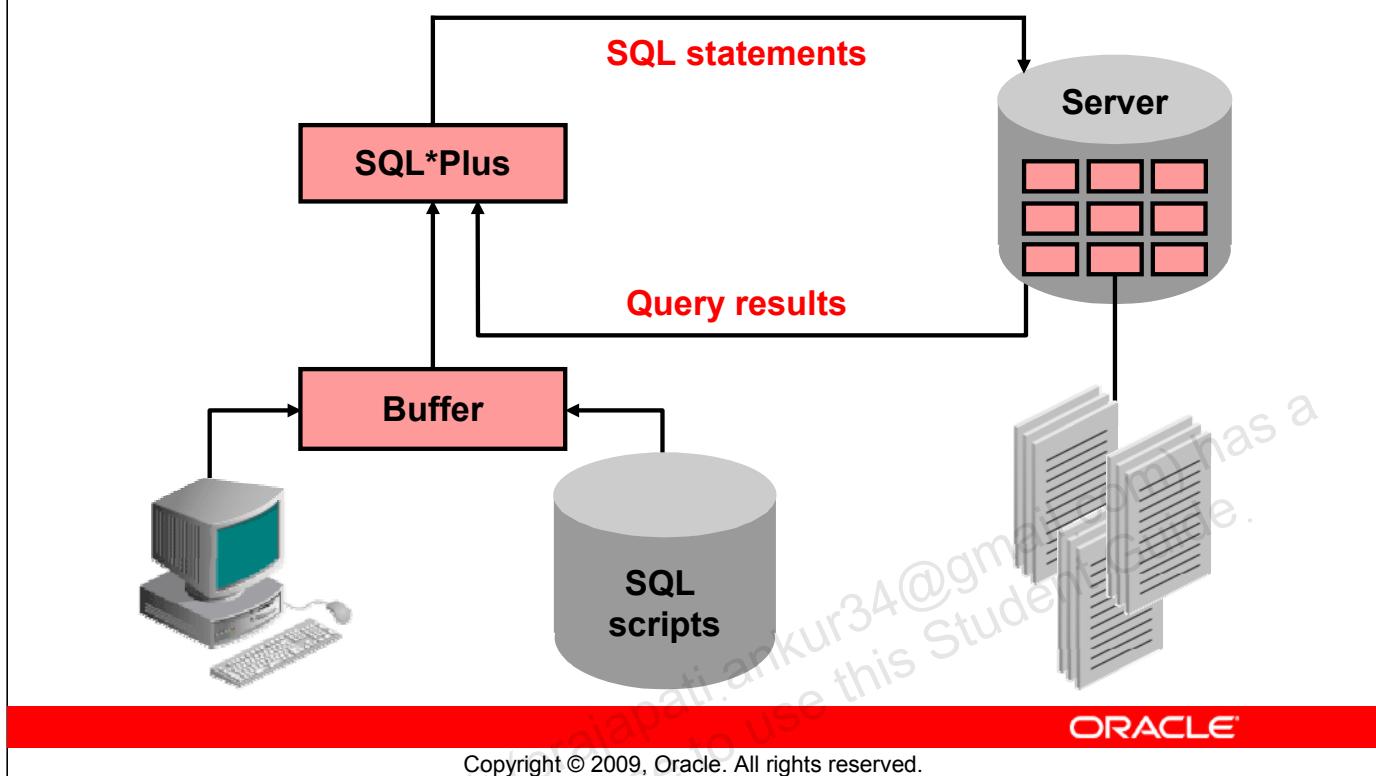
ORACLE

Copyright © 2009, Oracle. 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 using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

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

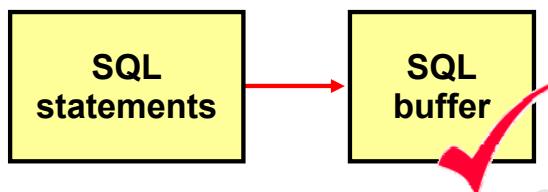
Features of SQL*Plus

- 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

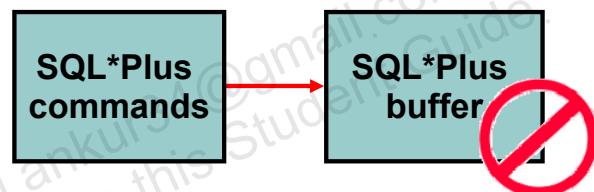
SQL

- 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 © 2009, Oracle. 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 from the file to buffer to edit.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

SQL*Plus

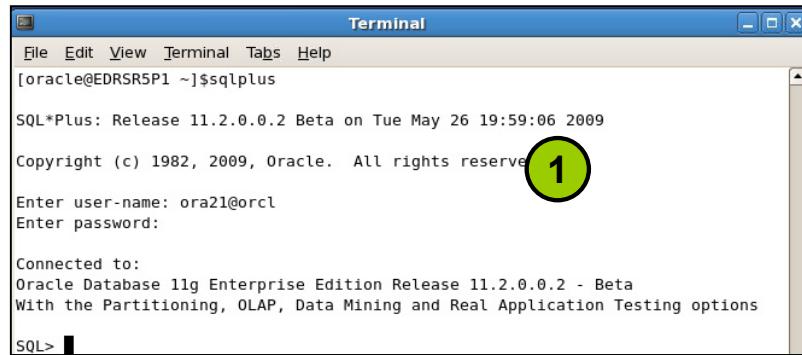
SQL*Plus is an environment in which you can:

- 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:

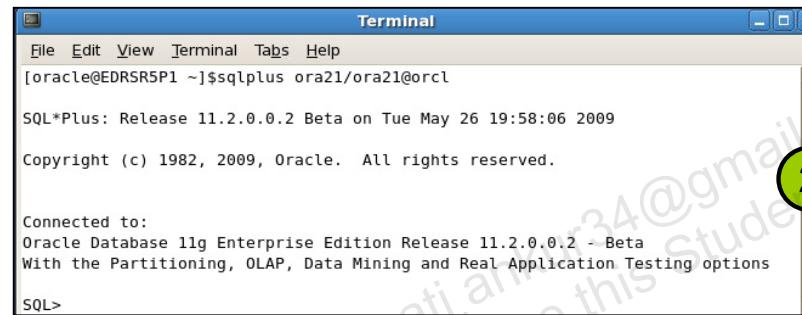
Category	Purpose
Environment	Affect the general behavior of SQL statements for the session.
Format	Format query results.
File manipulation	Save, load, and run script files.
Execution	Send SQL statements from the SQL buffer to the Oracle server.
Edit	Modify SQL statements in the buffer.
Interaction	Create 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/ora21@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 © 2009, Oracle. 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.

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
```



Copyright © 2009, Oracle. All rights reserved.

Displaying the Table Structure

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

In the syntax:

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

To describe the DEPARTMENTS table, use 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)



Copyright © 2009, Oracle. All rights reserved.

Displaying the Table Structure (continued)

The example 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*



Copyright © 2009, Oracle. 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
A [PPEND] <i>text</i>	Adds <i>text</i> to the end of the current line
C [HANGE] / <i>old</i> / <i>new</i>	Changes <i>old</i> text to <i>new</i> in the current line
C [HANGE] / <i>text</i> /	Deletes <i>text</i> from the current line
CL [EAR] BUFF [ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL <i>n</i>	Deletes line <i>n</i>
DEL <i>m n</i>	Deletes lines <i>m</i> to <i>n</i> inclusive

Guidelines

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



Copyright © 2009, Oracle. 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>
O <i>text</i>	Inserts a line before line 1

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

Using LIST, n, and APPEND

```
LIST
```

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

```
1
```

```
1* SELECT last_name
```

```
A , job_id
```

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

```
LIST
```

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



Copyright © 2009, Oracle. 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
```



Copyright © 2009, Oracle. All rights reserved.

Using the CHANGE Command

- Use L [LIST] to display the contents of the buffer.
- Use the C [CHANGE] 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 [LIST] command to verify the new contents of the buffer.

SQL*Plus File Commands

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



Copyright © 2009, Oracle. 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> [REP[LACE] APP[END]]	Saves current contents of SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.
<code>GET filename [.ext]</code>	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
<code>STA[RT] filename [.ext]</code>	Runs a previously saved command file
<code>@ filename</code>	Runs a previously saved command file (same as START)
<code>ED[IT]</code>	Invokes the editor and saves the buffer contents to a file named afiedt.buf
<code>ED[IT] [filename [.ext]]</code>	Invokes the editor to edit the contents of a saved file
<code>SPO[OL] [filename [.ext]] OFF OUT</code>	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
<code>EXIT</code>	Quits SQL*Plus

Using the SAVE 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
```

```
-----
```

```
Kochhar
```

```
100
```

```
90
```

```
...
```

```
107 rows selected.
```



Copyright © 2009, Oracle. 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
```

SERVERTOUTPUT Command

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

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

Copyright © 2009, Oracle. All rights reserved.

SERVERTOUTPUT 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 you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

Copyright © 2009, Oracle. 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]]  
[STATISTICS]
```

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



Copyright © 2009, Oracle. 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 the 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

The red bar spans most of the page width, with the Oracle logo positioned at its right end.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

JDeveloper

Using JDeveloper

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (ankur34@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

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

- List the key features of Oracle JDeveloper
- Create a database connection in JDeveloper
- Manage database objects in JDeveloper
- Use JDeveloper to execute SQL Commands
- Create and run PL/SQL Program Units

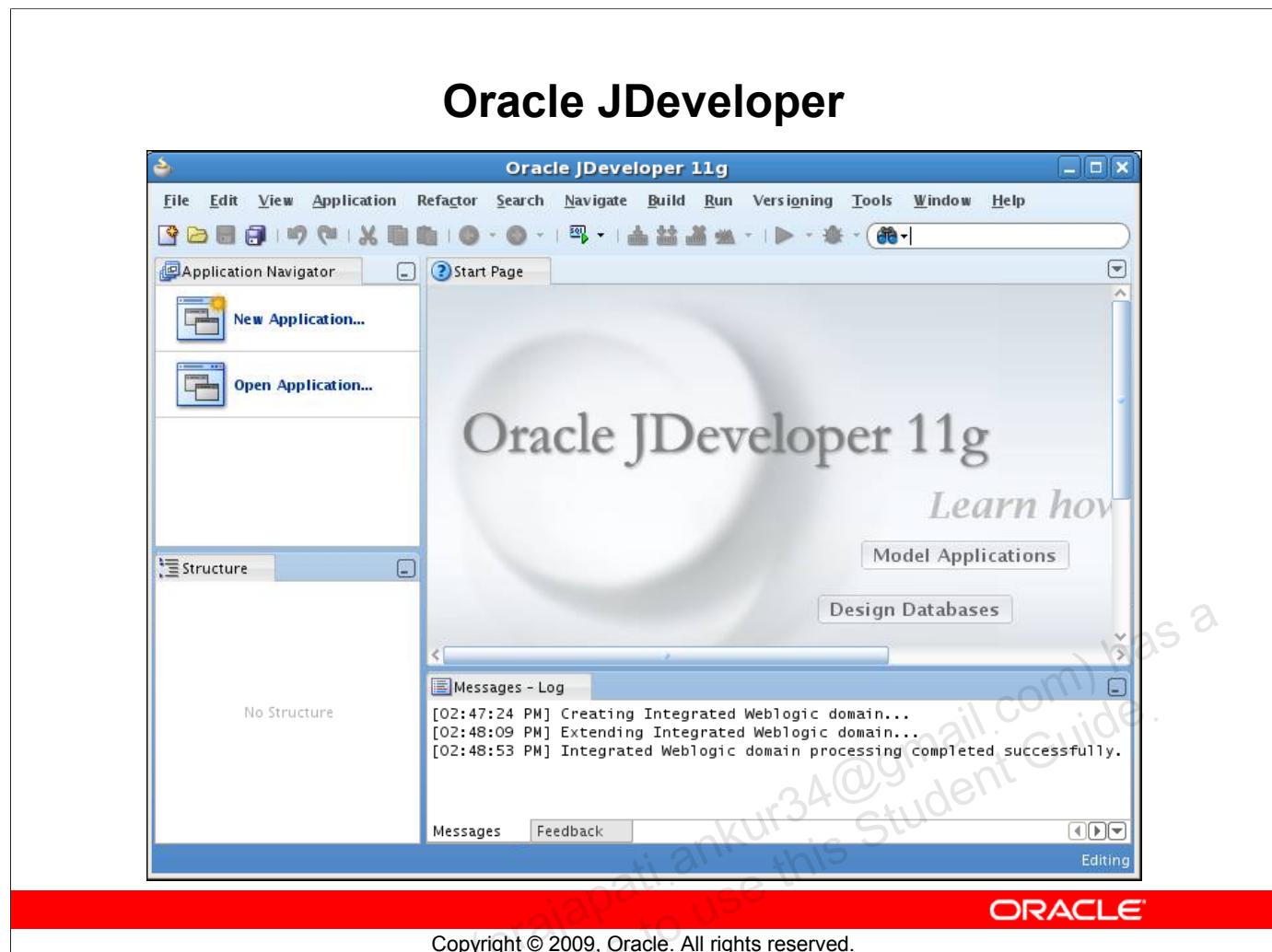
The red bar spans most of the width of the page, with a white space at the top and bottom.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Objectives

In this appendix, you are introduced to the tool JDeveloper. You learn how to use JDeveloper for your database development tasks.

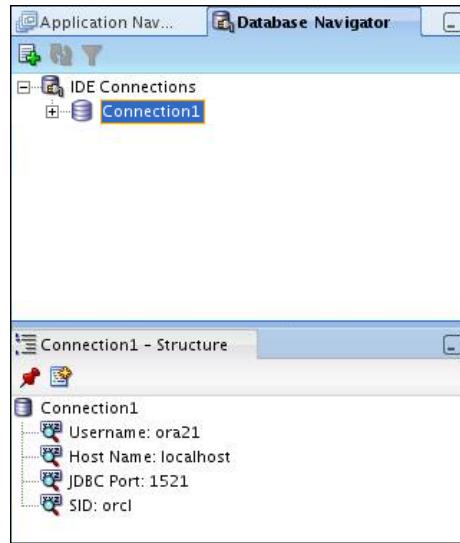


Oracle JDeveloper

Oracle JDeveloper is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC) from modeling to deploying. It has the features to use the latest industry standards for Java, XML, and SQL while developing an application.

Oracle JDeveloper 11g initiates a new approach to J2EE development with features that enable visual and declarative development. This innovative approach makes J2EE development simple and efficient.

Database Navigator

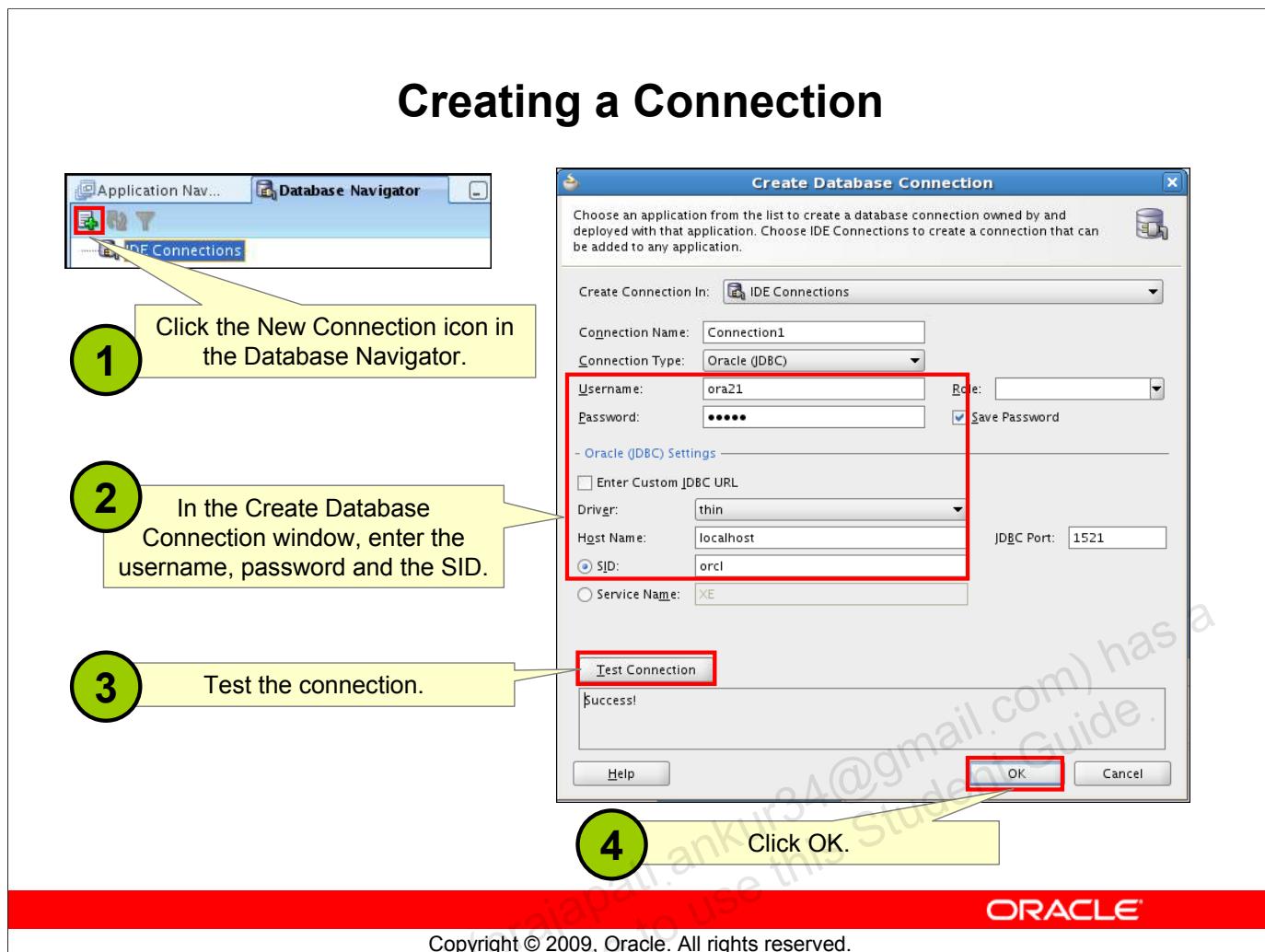


ORACLE

Copyright © 2009, Oracle. All rights reserved.

Database Navigator

Using Oracle JDeveloper, you can store the information necessary to connect to a database in an object called “connection.” A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.



Creating a Connection

A connection is an object that specifies the necessary information for connecting to a specific database as a specific user of that database. You can create and test connections for multiple databases and for multiple schemas.

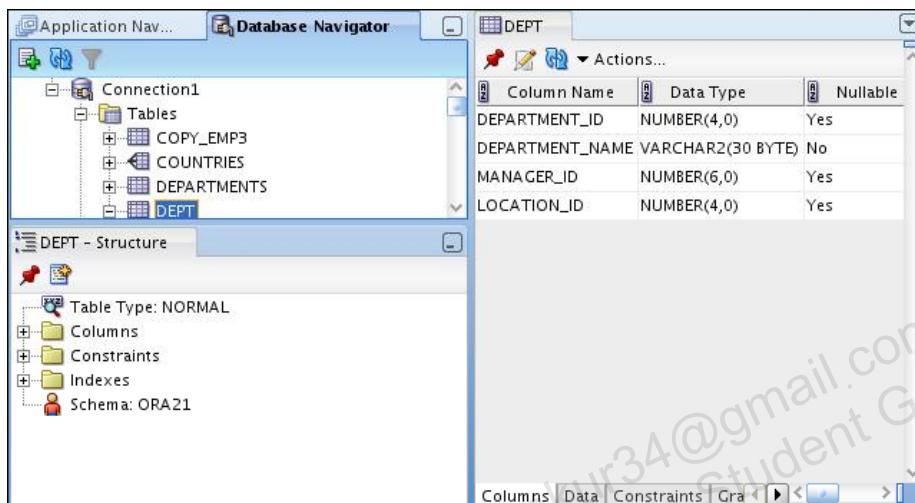
To create a database connection, perform the following steps:

1. Click the New Connection icon in the Database Navigator.
2. In the Create Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to. Enter the SID of the database that you want to connect to.
3. Click Test to ensure that the connection has been set correctly.
4. Click OK.

Browsing Database Objects

Use the Database Navigator to:

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



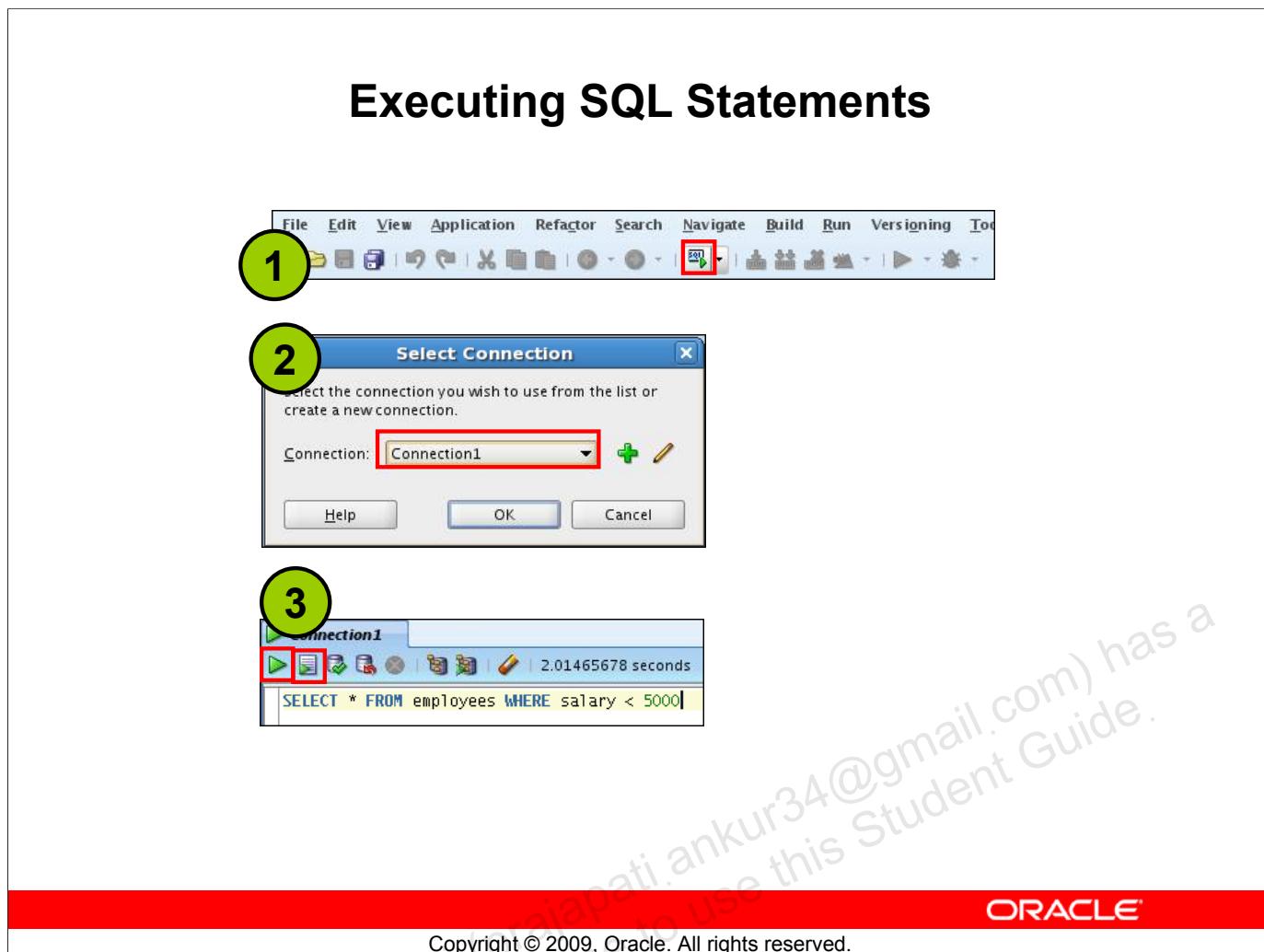
ORACLE

Copyright © 2009, Oracle. All rights reserved.

Browsing Database Objects

After you create a database connection, you can use the Database Navigator to browse through many objects in a database schema including tables, views, indexes, packages, procedures, triggers, and types.

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



Executing SQL Statements

To execute a SQL statement, perform the following steps:

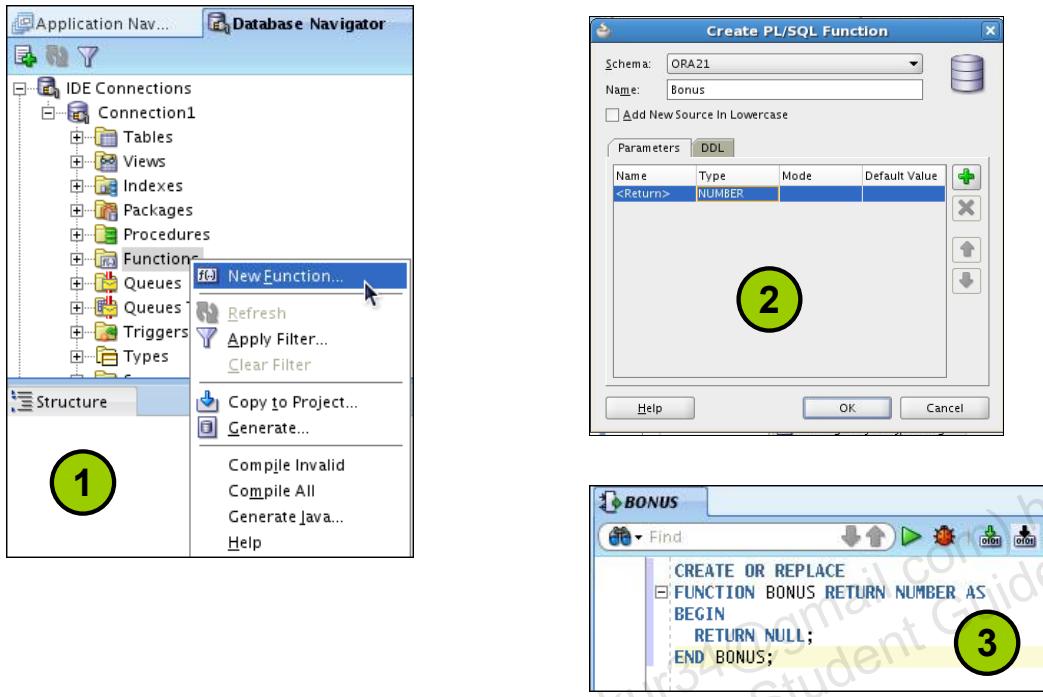
1. Click the Open SQL Worksheet icon.
2. Select the connection.
3. Execute the SQL command by clicking:
 - The **Execute statement** button or by pressing F9. The output is as follows:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME
1	100	Steven	King
2	101	Neena	Kochhar

- The **Run Script** button or by pressing F5. The output is as follows:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME
100	Steven	King

Creating Program Units



Skeleton of the function

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating Program Units

To create a PL/SQL program unit:

1. Select View > Database Navigator. Select and expand a database connection. Right-click a folder corresponding to the object type (Procedures, Packages, Functions). Select “New [Procedures|Packages|Functions]”.
2. Enter a valid name for the function, package, or procedure, and click OK.
3. A skeleton definition is created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling

The screenshot shows the 'Messages - Log' window from Oracle JDeveloper. It displays the message 'Compiling...' followed by the context 'MakeSelectedCommand selection=Element containing fi'. Below this, it shows the command '/home/oracle/Middleware/jdk160_11/jre/bin/java -jar / [5:14:32 PM] Successful compilation: 0 errors, 0 warnings.' The window has tabs for 'Messages' and 'Feedback' at the bottom.

Compilation with errors

The screenshot shows the 'Compiler - Log' window from Oracle JDeveloper. It lists a project named 'Project1' under 'Project: /home/oracle/jdeveloper/mywork/Application1/Project1/'. Underneath, it shows a file named 'Hello0' with three errors: 'Error(7,13): duplicate definition of variable a in constructor Hello0', 'Error(7,15): ; expected', and 'Error(8,28): variable a might not have been initialized'. The window has tabs for 'Messages', 'Feedback', and 'Compiler' at the bottom.

Compilation without errors

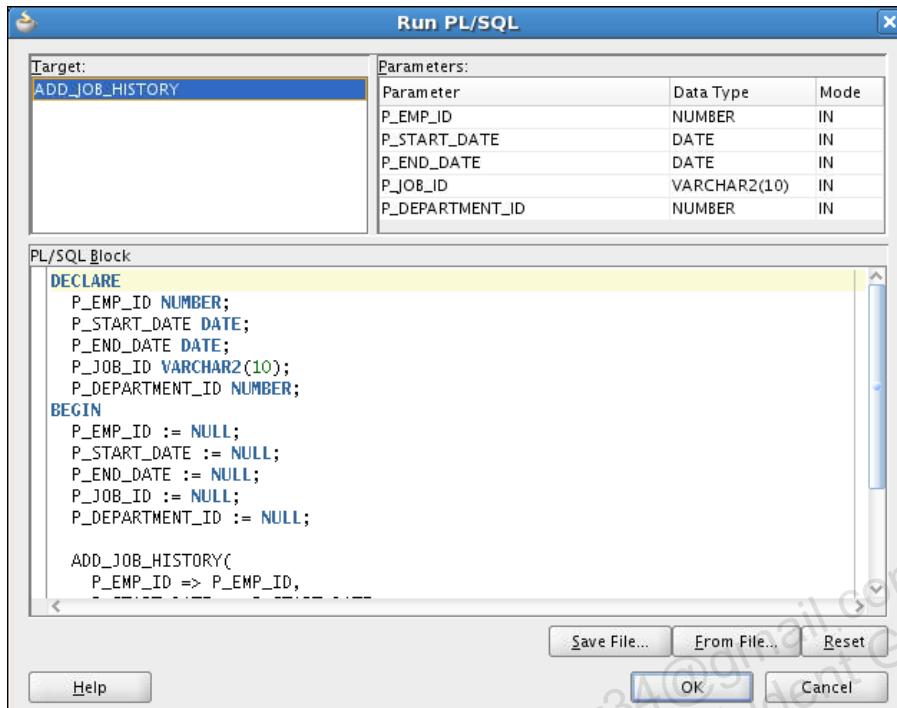
ORACLE

Copyright © 2009, Oracle. All rights reserved.

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator, and then select Compile. Alternatively, you can press CTRL + SHIFT + F9 to compile.

Running a Program Unit



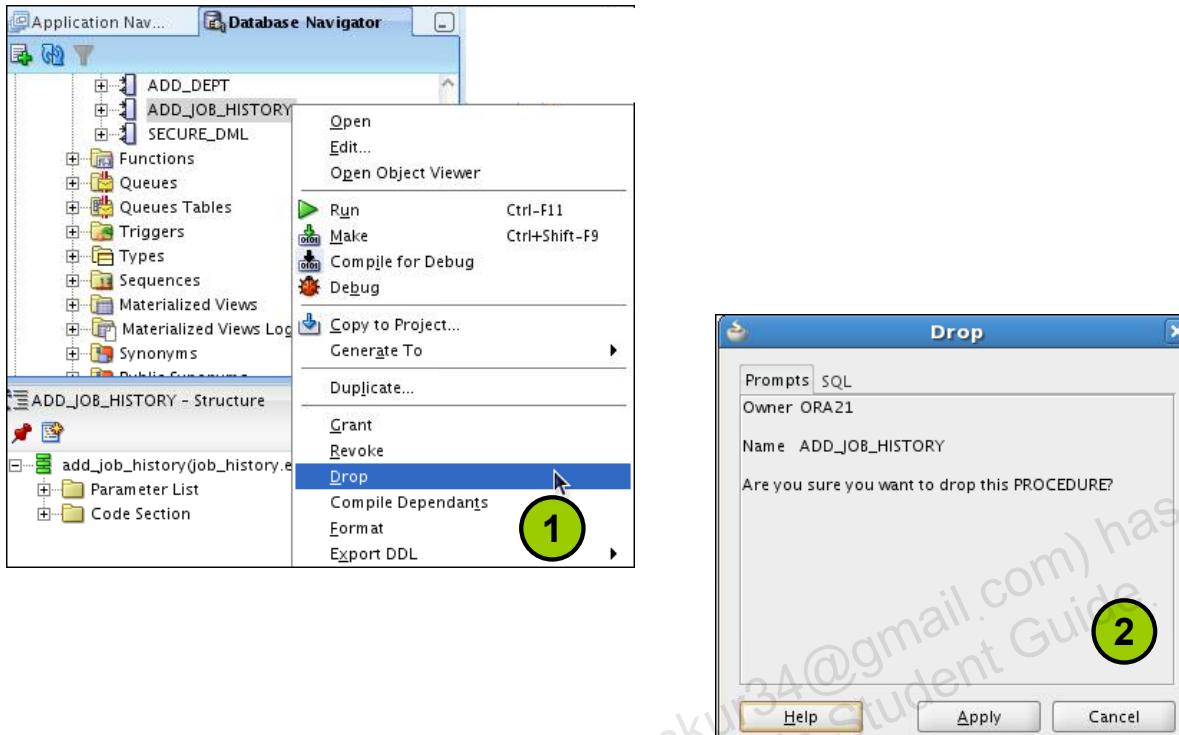
ORACLE

Copyright © 2009, Oracle. All rights reserved.

Running a Program Unit

To execute the program unit, right-click the object and select Run. The Run PL/SQL dialog box appears. You may need to change the NULL values with reasonable values that are passed into the program unit. After you change the values, click OK. The output is displayed in the Message-Log window.

Dropping a Program Unit



ORACLE

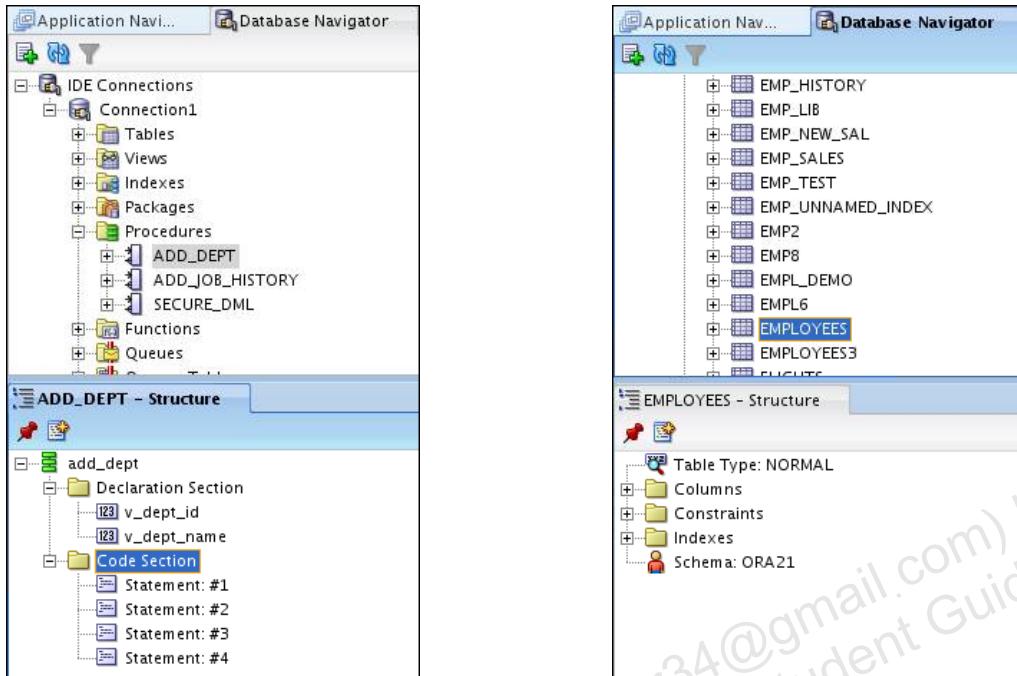
Copyright © 2009, Oracle. All rights reserved.

Dropping a Program Unit

To drop a program unit:

1. Right-click the object and select Drop.
The Drop Confirmation dialog box appears.
2. Click Apply.
The object is dropped from the database.

Structure Window



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Structure Window

The Structure window offers a structural view of the data in the document that is currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in a variety of ways. The structures that are available for display are based on document type. For a Java file, you can view code structure, UI structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, tracking always the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, select a different structure tab.

Editor Window

The screenshot shows the Oracle SQL Developer interface. The title bar has tabs for "Hello.java", "Connection1", "ADD_JOB_HISTORY" (which is the active tab), and "SECURE_DML". Below the title bar is a toolbar with icons for Find, Save, Undo, Redo, and others. The main editor area contains the following PL/SQL code:

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

ORACLE

Copyright © 2009, Oracle. All rights reserved.

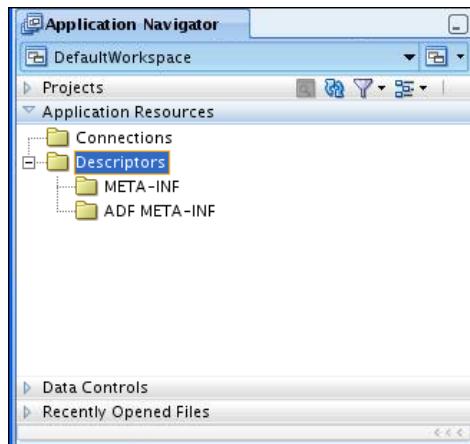
Editor Window

You can view your project files all in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

The tabs at the top of the editor window are the document tabs. Selecting a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

Application Navigator



ORACLE

Copyright © 2009, Oracle. All rights reserved.

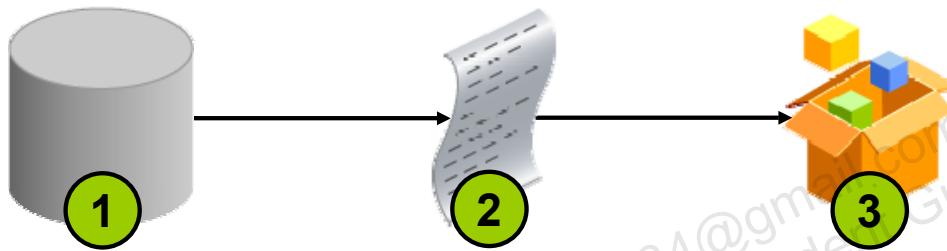
Application Navigator

Application Navigator gives you a logical view of your application and the data that it contains. Application Navigator provides an infrastructure that the different extensions can plug in to and use to organize their data and menus in a consistent, abstract manner. While Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, Unified Modeling Language (UML) diagrams, Enterprise JavaBeans (EJB), or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

1. Create a database connection.
2. Create a deployment profile.
3. Deploy the objects.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Deploying Java Stored Procedures

Create a deployment profile for Java stored procedures, and then deploy the classes and, optionally, any public static methods in JDeveloper using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call-specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

The screenshot shows two windows of Oracle SQL Developer. The top window displays the Java code for `TrimLob.java`, which contains a main method that establishes a database connection. The bottom window shows the creation of a PL/SQL procedure named `TRIMLOBPROC`. A green circle labeled '1' points to the Java code, and a green circle labeled '2' points to the PL/SQL code.

```
public class TrimLob {
    public static void main (String args[]) throws SQLException {
        Connection conn=null;
        if (System.getProperty("oracle.jserver.version") != null)
        {
            conn = DriverManager.getConnection("jdbc:default:connection:");
        }
        else
        {
            DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
            conn = DriverManager.getConnection("jdbc:oracle:thin:scott/tiger");
        }
    }
}
```

```
CREATE OR REPLACE PROCEDURE TRIMLOBPROC
as Language java
name 'TrimLob.main(java.lang.String[])';
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Publishing Java to PL/SQL

The slide shows the Java code and illustrates how to publish the Java code in a PL/SQL procedure.

How Can I Learn More About JDeveloper 11g ?

Topic	Web site
Oracle JDeveloper Product Page	http://www.oracle.com/technology/products/jdev/index.html
Oracle JDeveloper 11g Tutorials	http://www.oracle.com/technology/obe/obe11jdev/11/index.html
Oracle JDeveloper 11g Product Documentation	http://www.oracle.com/technology/documentation/jdev.html
Oracle JDeveloper 11g Discussion Forum	http://forums.oracle.com/forums/forum.jspa?forumID=83

Summary

In this appendix, you should have learned how to use JDeveloper to do the following:

- List the key features of Oracle JDeveloper
- Create a database connection in JDeveloper
- Manage database objects in JDeveloper
- Use JDeveloper to execute SQL Commands
- Create and run PL/SQL Program Units

The red bar spans most of the page width, with a white space at the top and bottom.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

R Review of PL/SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (ankur34@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

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

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



Copyright © 2009, Oracle. All rights reserved.

Block Structure for Anonymous PL/SQL Blocks

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

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Anonymous Blocks

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

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

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

Declaring PL/SQL Variables

- Syntax:

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

- Examples:

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

Copyright © 2009, Oracle. All rights reserved.

Declaring PL/SQL Variables

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

In the syntax:

Identifier is the name of the variable

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

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

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

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

Declaring Variables with the %TYPE Attribute: Examples

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



Copyright © 2009, Oracle. All rights reserved.

Declaring Variables with the %TYPE Attribute

Declare variables to store the name of an employee.

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

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

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

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

Creating a PL/SQL Record

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

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



Copyright © 2009, Oracle. All rights reserved.

Creating a PL/SQL Record

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

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

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

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

%ROWTYPE Attribute: Examples

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

```
dept_record    departments%ROWTYPE;
```

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

```
emp_record    employees%ROWTYPE;
```



Copyright © 2009, Oracle. All rights reserved.

Examples

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

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

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

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

Creating a PL/SQL Table

```

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

```



Copyright © 2009, Oracle. All rights reserved.

Creating a PL/SQL Table

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

Referencing a PL/SQL Table

Syntax

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

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

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

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

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

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

SELECT Statements in PL/SQL: Example

The INTO clause is mandatory.

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



Copyright © 2009, Oracle. All rights reserved.

INTO Clause

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

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

Queries Must Return One and Only One Row

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

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

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

Inserting Data: Example

Add new employee information to the EMPLOYEES table.

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



Copyright © 2009, Oracle. All rights reserved.

Inserting Data

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

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

Updating Data: Example

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

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



Copyright © 2009, Oracle. All rights reserved.

Updating Data

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

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

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

Deleting Data: Example

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

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



Copyright © 2009, Oracle. All rights reserved.

Deleting Data: Example

Delete a specific job:

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

Controlling Transactions with the COMMIT and ROLLBACK Statements

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

The red bar spans most of the width of the slide, centered horizontally.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Controlling Transactions with the COMMIT and ROLLBACK Statements

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

Syntax

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

In this syntax, WORK is for compliance with ANSI standards.

Note: The transaction control commands are all valid within PL/SQL, although the host environment may place some restriction on their use.

You can also include explicit locking commands (such as LOCK TABLE and SELECT . . . FOR UPDATE) in a block. They stay in effect until the end of the transaction. Also, one PL/SQL block does not necessarily imply one transaction.

IF, THEN, and ELSIF Statements: Example

For a given value entered, return a calculated value.

```
.
.
.
IF v_start > 100 THEN
    v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
    v_start := 0.5 * v_start;
ELSE
    v_start := 0.1 * v_start;
END IF;
.
.
```



Copyright © 2009, Oracle. All rights reserved.

IF, THEN, and ELSIF Statements

When possible, use the ELSIF clause instead of nesting IF statements. The code is easier to read and understand, and the logic is clearly identified. If the action in the ELSE clause consists purely of another IF statement, it is more convenient to use the ELSIF clause. This makes the code clearer by removing the need for nested END IFs at the end of each further set of conditions and actions.

Example

```
IF condition1 THEN
    statement1;
ELSIF condition2 THEN
    statement2;
ELSIF condition3 THEN
    statement3;
END IF;
```

The statement in the slide is further defined as follows:

For a given value entered, return a calculated value. If the entered value is over 100, then the calculated value is two times the entered value. If the entered value is between 50 and 100, then the calculated value is 50% of the starting value. If the entered value is less than 50, then the calculated value is 10% of the starting value.

Note: Any arithmetic expression containing null values evaluates to null.

Basic Loop: Example

```
DECLARE
  v_ordid      order_items.order_id%TYPE := 101;
  v_counter    NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO order_items(order_id,line_item_id)
    VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```



Copyright © 2009, Oracle. All rights reserved.

Basic Loop: Example

The basic loop example shown in the slide is defined as follows:

Insert the first 10 new line items for order number 101.

Note: A basic loop enables execution of its statements at least once, even if the condition has been met upon entering the loop.

FOR Loop: Example

Insert the first 10 new line items for order number 101.

```
DECLARE
    v_ordid      order_items.order_id%TYPE := 101;
BEGIN
    FOR i IN 1..10 LOOP
        INSERT INTO order_items(order_id,line_item_id)
        VALUES(v_ordid, i);
    END LOOP;
END;
```



Copyright © 2009, Oracle. All rights reserved.

FOR Loop: Example

The slide shows a FOR loop that inserts 10 rows into the `order_items` table.

WHILE Loop: Example

```
ACCEPT p_price PROMPT 'Enter the price of the item: '
ACCEPT p_itemtot -
  PROMPT 'Enter the maximum total for purchase of item: '
DECLARE
  ...
  v_qty          NUMBER(8) := 1;
  v_running_total NUMBER(7,2) := 0;

BEGIN
  ...
  WHILE v_running_total < &p_itemtot LOOP
    ...
    v_qty := v_qty + 1;
    v_running_total := v_qty * &p_price;
  END LOOP;
  ...

```



Copyright © 2009, Oracle. All rights reserved.

WHILE Loop: Example

In the example in the slide, the quantity increases with each iteration of the loop until the quantity is no longer less than the maximum price allowed for spending on the item.

SQL Implicit Cursor Attributes

You can use SQL cursor attributes to test the outcome of your SQL statements.

SQL Cursor Attributes	Description
SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value)
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows
SQL%ISOPEN	Boolean attribute that always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed



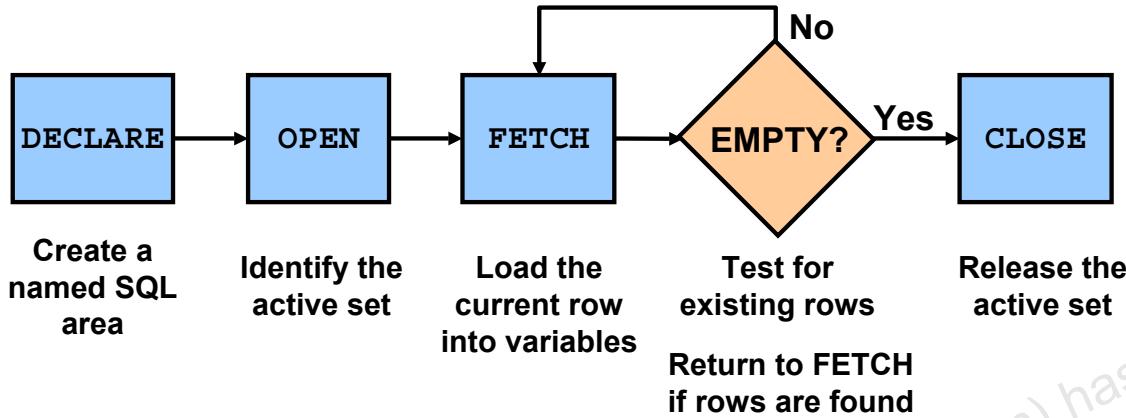
Copyright © 2009, Oracle. All rights reserved.

SQL Implicit Cursor Attributes

SQL cursor attributes enable you to evaluate what happened when the implicit cursor was last used. You use these attributes in PL/SQL statements such as functions. You cannot use them in SQL statements.

You can use the SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, and SQL%ISOPEN attributes in the exception section of a block to gather information about the execution of a DML statement. In PL/SQL, a DML statement that does not change any rows is not seen as an error condition, whereas the SELECT statement will return an exception if it cannot locate any rows.

Controlling Explicit Cursors



Copyright © 2009, Oracle. All rights reserved.

ORACLE

Explicit Cursors

Controlling Explicit Cursors Using Four Commands

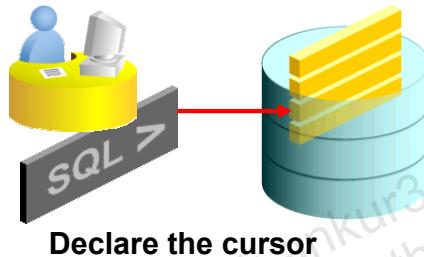
1. Declare the cursor by naming it and defining the structure of the query to be performed within it.
2. Open the cursor. The **OPEN** statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor. The **FETCH** statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore, each fetch accesses a different row returned by the query. In the flow diagram in the slide, each fetch tests the cursor for any existing rows. If rows are found, it loads the current row into variables; otherwise, it closes the cursor.
4. Close the cursor. The **CLOSE** statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Controlling Explicit Cursors: Declaring the Cursor

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;

  CURSOR c2 IS
    SELECT *
    FROM   departments
    WHERE  department_id = 10;
BEGIN
  ...

```



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Explicit Cursor Declaration

Retrieve the employees one by one.

```
DECLARE
  v_empid  employees.employee_id%TYPE;
  v_ename  employees.last_name%TYPE;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  ...

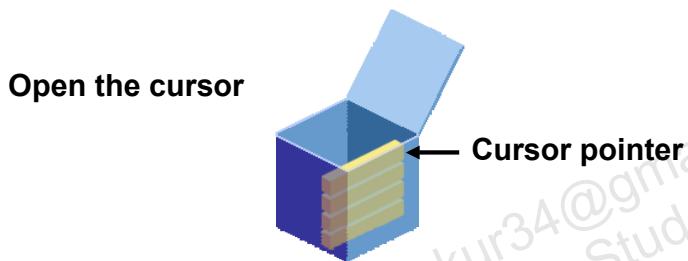
```

Note: You can reference variables in the query, but you must declare them before the CURSOR statement.

Controlling Explicit Cursors: Opening the Cursor

```
OPEN  cursor_name;
```

- Open the cursor to execute the query and identify the active set.
- If the query returns no rows, no exception is raised.
- Use cursor attributes to test the outcome after a fetch.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

OPEN Statement

Open the cursor to execute the query and identify the result set, which consists of all rows that meet the query search criteria. The cursor now points to the first row in the result set.

In the syntax, `cursor_name` is the name of the previously declared cursor.

OPEN is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area that eventually contains crucial processing information
2. Parses the SELECT statement
3. Binds the input variables—that is, sets the value for the input variables by obtaining their memory addresses
4. Identifies the result set—that is, the set of rows that satisfy the search criteria. Rows in the result set are not retrieved into variables when the OPEN statement is executed. Rather, the FETCH statement retrieves the rows.
5. Positions the pointer just before the first row in the active set

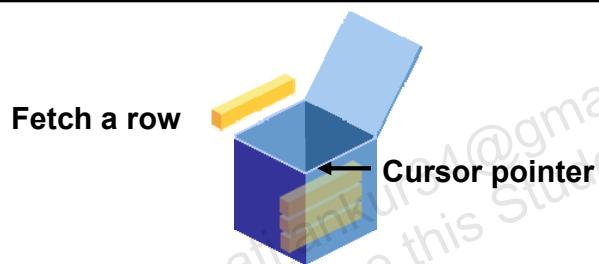
Note: If the query returns no rows when the cursor is opened, then PL/SQL does not raise an exception. However, you can test the cursor's status after a fetch.

For cursors declared by using the FOR UPDATE clause, the OPEN statement also locks those rows.

Controlling Explicit Cursors: Fetching Data from the Cursor

```
FETCH c1 INTO v_empid, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
  -- Process the retrieved data
  ...
END;
```



ORACLE

Copyright © 2009, Oracle. All rights reserved.

FETCH Statement

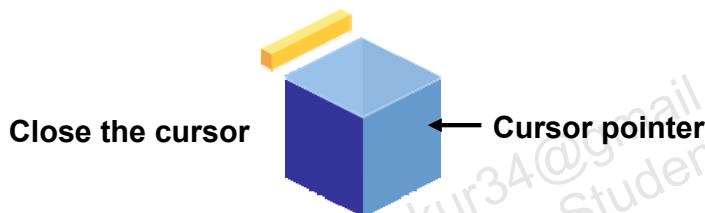
You use the `FETCH` statement to retrieve the current row values into output variables. After the `fetch`, you can manipulate the variables by further statements. For each column value returned by the query associated with the cursor, there must be a corresponding variable in the `INTO` list. Also, their data types must be compatible. Retrieve the first 10 employees one by one:

```
DECLARE
  v_empid  employees.employee_id%TYPE;
  v_ename   employees.last_name%TYPE;
  i         NUMBER := 1;
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  OPEN c1;
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename;
    ...
  END LOOP;
END;
```

Controlling Explicit Cursors: Closing the Cursor

```
CLOSE    cursor_name;
```

- Close the cursor after completing the processing of the rows.
- Reopen the cursor, if required.
- Do not attempt to fetch data from a cursor after it has been closed.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

CLOSE Statement

The CLOSE statement disables the cursor, and the result set becomes undefined. Close the cursor after completing the processing of the SELECT statement. This step allows the cursor to be reopened, if required. Therefore, you can establish an active set several times.

In the syntax, `cursor_name` is the name of the previously declared cursor.

Do not attempt to fetch data from a cursor after it has been closed, or the `INVALID_CURSOR` exception will be raised.

Note: The CLOSE statement releases the context area. Although it is possible to terminate the PL/SQL block without closing cursors, you should always close any cursor that you declare explicitly in order to free up resources. There is a maximum limit to the number of open cursors per user, which is determined by the `OPEN_CURSORS` parameter in the database parameter field. By default, the maximum number of `OPEN_CURSORS` is 50.

```
...
FOR i IN 1..10 LOOP
    FETCH c1 INTO v_empid, v_ename; ...
END LOOP;
CLOSE c1;
END;
```

Explicit Cursor Attributes

Obtain status information about a cursor.

Attribute	Type	Description
ISOPEN	BOOLEAN	Evaluates to TRUE if the cursor is open
%NOTFOUND	BOOLEAN	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	BOOLEAN	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	NUMBER	Evaluates to the total number of rows returned so far



Copyright © 2009, Oracle. All rights reserved.

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining status information about a cursor. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a DML statement.

Note: Do not reference cursor attributes directly in a SQL statement.

Cursor FOR Loops: Example

Retrieve employees one by one until there are no more left.

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM   employees;
BEGIN
  FOR emp_record IN c1 LOOP
    -- implicit open and implicit fetch occur
    IF emp_record.employee_id = 134 THEN
      ...
    END LOOP; -- implicit close occurs
END;
```



Copyright © 2009, Oracle. All rights reserved.

Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. The cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched. In the example in the slide, `emp_record` in the cursor for loop is an implicitly declared record that is used in the FOR LOOP construct.

FOR UPDATE Clause: Example

Retrieve the orders for amounts over \$1,000 that were processed today.

```
DECLARE
  CURSOR c1 IS
    SELECT customer_id, order_id
    FROM   orders
    WHERE  order_date = SYSDATE
           AND order_total > 1000.00
    ORDER BY customer_id
    FOR UPDATE NOWAIT;
```



Copyright © 2009, Oracle. All rights reserved.

FOR UPDATE Clause

If the database server cannot acquire the locks on the rows it needs in a SELECT FOR UPDATE, then it waits indefinitely. You can use the NOWAIT clause in the SELECT FOR UPDATE statement and test for the error code that returns due to failure to acquire the locks in a loop. Therefore, you can retry opening the cursor *n* times before terminating the PL/SQL block.

If you intend to update or delete rows by using the WHERE CURRENT OF clause, you must specify a column name in the FOR UPDATE OF clause.

If you have a large table, you can achieve better performance by using the LOCK TABLE statement to lock all rows in the table. However, when using LOCK TABLE, you cannot use the WHERE CURRENT OF clause and must use the notation WHERE *column* = *identifier*.

WHERE CURRENT OF Clause: Example

```
DECLARE
  CURSOR c1 IS
    SELECT salary FROM employees
    FOR UPDATE OF salary NOWAIT;
BEGIN
  ...
  FOR emp_record IN c1 LOOP
    UPDATE ...
      WHERE CURRENT OF c1;
  ...
END LOOP;
COMMIT;
END;
```



Copyright © 2009, Oracle. All rights reserved.

WHERE CURRENT OF Clause

You can update rows based on criteria from a cursor.

Additionally, you can write your DELETE or UPDATE statement to contain the WHERE CURRENT OF cursor_name clause to refer to the latest row processed by the FETCH statement. When you use this clause, the cursor you reference must exist and must contain the FOR UPDATE clause in the cursor query; otherwise, you get an error. This clause enables you to apply updates and deletes to the currently addressed row without the need to explicitly reference the ROWID pseudocolumn.

Trapping Predefined Oracle Server Errors

- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX

The red bar spans most of the page width, with the Oracle logo positioned at its right end.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Trapping Predefined Oracle Server Errors

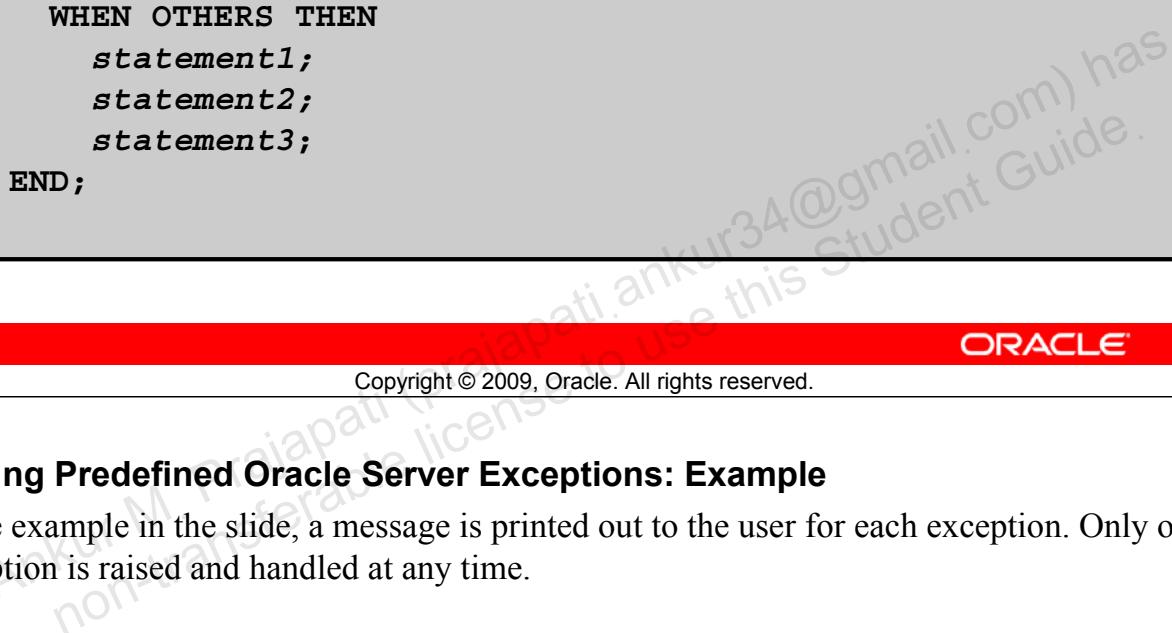
Trap a predefined Oracle server error by referencing its standard name within the corresponding exception-handling routine.

Note: PL/SQL declares predefined exceptions in the STANDARD package.

It is a good idea to always consider the NO_DATA_FOUND and TOO_MANY_ROWS exceptions, which are the most common.

Trapping Predefined Oracle Server Errors: Example

```
BEGIN   SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

A large, faint watermark is visible across the slide, reading "HackerMantra.com - All rights reserved. Non-Commercial Use Only. This watermark is used to prevent unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates."

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Trapping Predefined Oracle Server Exceptions: Example

In the example in the slide, a message is printed out to the user for each exception. Only one exception is raised and handled at any time.

Non-Predefined Error

Trap for Oracle server error number –2292, which is an integrity constraint violation.

```

DECLARE
    1 e_products_invalid EXCEPTION;
    PRAGMA EXCEPTION_INIT (
        2 e_products_invalid, -2292);
    v_message VARCHAR2(50);
BEGIN
    . . .
    3 EXCEPTION
        WHEN e_products_invalid THEN
            :g_message := 'Product ID
                           specified is not valid.';
    . . .
END;

```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Trapping a Non-Predefined Oracle Server Exception

1. Declare the name for the exception within the declarative section.

Syntax

```
exception EXCEPTION;
```

In this syntax, *exception* is the name of the exception.

2. Associate the declared exception with the standard Oracle server error number, using the PRAGMA EXCEPTION_INIT statement.

Syntax

```
PRAGMA EXCEPTION_INIT(exception, error_number);
```

In this syntax:

<i>exception</i>	Is the previously declared exception
<i>error_number</i>	Is a standard Oracle server error number

3. Reference the declared exception within the corresponding exception-handling routine.

In the example in the slide: If there is product in stock, halt processing and print a message to the user.

User-Defined Exceptions: Example

```
[DECLARE]
  e_amount_remaining EXCEPTION;
  .
  .
BEGIN
  .
  .
  RAISE e_amount_remaining; 2
  .
  .
EXCEPTION
  WHEN e_amount_remaining THEN
    :g_message := 'There is still an amount
      in stock.';
  .
  .
END;
```

1

2

3

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Trapping User-Defined Exceptions

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name for the user-defined exception within the declarative section.
Syntax: `exception EXCEPTION;`
where: `exception` Is the name of the exception
2. Use the RAISE statement to raise the exception explicitly within the executable section.
Syntax: `RAISE exception;`
where: `exception` Is the previously declared exception
3. Reference the declared exception within the corresponding exception-handling routine.

In the example in the slide: This customer has a business rule that states that a product cannot be removed from its database if there is any inventory left in stock for this product. Because there are no constraints in place to enforce this rule, the developer handles it explicitly in the application. Before performing a DELETE on the PRODUCT_INFORMATION table, the block queries the INVENTORIES table to see whether there is any stock for the product in question. If there is stock, raise an exception.

Note: Use the RAISE statement by itself within an exception handler to raise the same exception back to the calling environment.

RAISE_APPLICATION_ERROR Procedure

```
raise_application_error (error_number,  
message[, {TRUE | FALSE}]);
```

- Enables you to issue user-defined error messages from stored subprograms
- Is called from an executing stored subprogram only

The red bar contains the word "ORACLE" in white capital letters.

Copyright © 2009, Oracle. All rights reserved.

RAISE_APPLICATION_ERROR Procedure

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax, *error_number* is a user-specified number for the exception between –20,000 and –20,999. The *message* is the user-specified message for the exception. It is a character string that is up to 2,048 bytes long.

TRUE | FALSE is an optional Boolean parameter. If TRUE, the error is placed on the stack of previous errors. If FALSE (the default), the error replaces all previous errors.

Example:

```
...  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RAISE_APPLICATION_ERROR (-20201,  
                                'Manager is not a valid employee.');
```

```
END;
```

RAISE_APPLICATION_ERROR Procedure

- Is used in two different places:
 - Executable section
 - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

The red bar contains the ORACLE logo.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

RAISE_APPLICATION_ERROR Procedure: Example

```
...
DELETE FROM employees
WHERE manager_id = v_mgr;
IF SQL%NOTFOUND THEN
  RAISE_APPLICATION_ERROR (-20202,
    'This is not a valid manager');
END IF;
...
```

Summary

In this appendix, you should have learned how to:

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



Copyright © 2009, Oracle. All rights reserved.

Summary

The Oracle server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor FOR loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor FOR loops do this implicitly. If you are updating or deleting rows, lock the rows by using a FOR UPDATE clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a WHERE CURRENT OF clause in conjunction with the FOR UPDATE clause to reference the current row fetched by the cursor.

G

Studies for Implementing Triggers

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (ankur34@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

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

- Enhance database security with triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities using triggers



Copyright © 2009, Oracle. All rights reserved.

Lesson Aim

In this lesson, you learn to develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server. In some cases, it may be sufficient to refrain from using triggers and accept the functionality provided by the Oracle server.

This lesson covers the following business application scenarios:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Controlling Security Within the Server

Using database security with the GRANT statement.

```
GRANT SELECT, INSERT, UPDATE, DELETE  
  ON employees  
  TO clerk;          -- database role  
GRANT clerk TO scott;
```



Copyright © 2009, Oracle. All rights reserved.

Controlling Security Within the Server

Develop schemas and roles within the Oracle server to control the security of data operations on tables according to the identity of the user.

- Base privileges upon the username supplied when the user connects to the database.
- Determine access to tables, views, synonyms, and sequences.
- Determine query, data-manipulation, and data-definition privileges.

Controlling Security with a Database Trigger

```
CREATE OR REPLACE TRIGGER secure_emp
  BEFORE INSERT OR UPDATE OR DELETE ON employees
DECLARE
  dummy PLS_INTEGER;
BEGIN
  IF (TO_CHAR (SYSDATE, 'DY') IN ('SAT','SUN')) THEN
    RAISE_APPLICATION_ERROR(-20506,'You may only
      change data during normal business hours.');
  END IF;
  SELECT COUNT(*) INTO dummy FROM holiday
  WHERE holiday_date = TRUNC (SYSDATE);
  IF dummy > 0 THEN
    RAISE_APPLICATION_ERROR(-20507,
      'You may not change data on a holiday.');
  END IF;
END;
/
```



Copyright © 2009, Oracle. All rights reserved.

Enforcing Data Integrity Within the Server

```
ALTER TABLE employees ADD  
CONSTRAINT ck_salary CHECK (salary >= 500);
```



Copyright © 2009, Oracle. All rights reserved.

Enforcing Data Integrity Within the Server

You can enforce data integrity within the Oracle server and develop triggers to handle more complex data integrity rules.

The standard data integrity rules are not null, unique, primary key, and foreign key.

Use these rules to:

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

Example

The code sample in the slide ensures that the salary is at least \$500.

Protecting Data Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER check_salary
  BEFORE UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.salary < OLD.salary)
BEGIN
  RAISE_APPLICATION_ERROR (-20508,
    'Do not decrease salary.');
END;
/
```



Copyright © 2009, Oracle. All rights reserved.

Protecting Data Integrity with a Trigger

Protect data integrity with a trigger and enforce nonstandard data integrity checks.

- Provide variable default values.
- Enforce dynamic constraints.
- Enable and disable dynamically.
- Incorporate declarative constraints within the definition of a table to protect data integrity.

Example

The code sample in the slide ensures that the salary is never decreased.

Enforcing Referential Integrity Within the Server

```
ALTER TABLE employees
  ADD CONSTRAINT emp_deptno_fk
    FOREIGN KEY (department_id)
      REFERENCES departments(department_id)
    ON DELETE CASCADE;
```



Copyright © 2009, Oracle. All rights reserved.

Enforcing Referential Integrity Within the Server

Incorporate referential integrity constraints within the definition of a table to prevent data inconsistency and enforce referential integrity within the server.

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

Example

When a department is removed from the DEPARTMENTS parent table, cascade the deletion to the corresponding rows in the EMPLOYEES child table.

Protecting Referential Integrity with a Trigger

```
CREATE OR REPLACE TRIGGER cascade_updates
AFTER UPDATE OF department_id ON departments
FOR EACH ROW
BEGIN
  UPDATE employees
    SET employees.department_id=:NEW.department_id
    WHERE employees.department_id=:OLD.department_id;
  UPDATE job_history
    SET department_id=:NEW.department_id
    WHERE department_id=:OLD.department_id;
END;
/
```



Copyright © 2009, Oracle. All rights reserved.

Protecting Referential Integrity with a Trigger

The following referential integrity rules are not supported by declarative constraints:

- Cascade updates.
- Set to NULL for updates and deletions.
- Set to a default value on updates and deletions.
- Enforce referential integrity in a distributed system.
- Enable and disable dynamically.

You can develop triggers to implement these integrity rules.

Example

Enforce referential integrity with a trigger. When the value of DEPARTMENT_ID changes in the DEPARTMENTS parent table, cascade the update to the corresponding rows in the EMPLOYEES child table.

For a complete referential integrity solution using triggers, a single trigger is not enough.

Replicating a Table Within the Server

```
CREATE MATERIALIZED VIEW emp_copy
NEXT sysdate + 7
AS SELECT * FROM employees@ny;
```

The red bar spans the width of the slide content area.
ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating a Materialized View

Materialized views enable you to maintain copies of remote data on your local node for replication purposes. You can select data from a materialized view as you would from a normal database table or view. A materialized view is a database object that contains the results of a query, or a copy of some database on a query. The `FROM` clause of the query of a materialized view can name tables, views, and other materialized views.

When a materialized view is used, replication is performed implicitly by the Oracle server. This performs better than using user-defined PL/SQL triggers for replication. Materialized views:

- Copy data from local and remote tables asynchronously, at user-defined intervals
- Can be based on multiple master tables
- Are read-only by default, unless using the Oracle Advanced Replication feature
- Improve the performance of data manipulation on the master table

Alternatively, you can replicate tables using triggers.

The example in the slide creates a copy of the remote EMPLOYEES table from New York. The `NEXT` clause specifies a date-time expression for the interval between automatic refreshes.

Replicating a Table with a Trigger

```
CREATE OR REPLACE TRIGGER emp_replica
  BEFORE INSERT OR UPDATE ON employees FOR EACH ROW
BEGIN /* Proceed if user initiates data operation,
NOT through the cascading trigger.*/
  IF INSERTING THEN
    IF :NEW.flag IS NULL THEN
      INSERT INTO employees@sf
      VALUES (:new.employee_id, ..., 'B');
      :NEW.flag := 'A';
    END IF;
  ELSE /* Updating. */
    IF :NEW.flag = :OLD.flag THEN
      UPDATE employees@sf
      SET ename=:NEW.last_name, ..., flag=:NEW.flag
      WHERE employee_id = :NEW.employee_id;
    END IF;
    IF :OLD.flag = 'A' THEN :NEW.flag := 'B';
      ELSE :NEW.flag := 'A';
    END IF;
  END IF;
END;
```



Copyright © 2009, Oracle. All rights reserved.

Replicating a Table with a Trigger

You can replicate a table with a trigger. By replicating a table, you can:

- Copy tables synchronously, in real time
- Base replicas on a single master table
- Read from replicas as well as write to them

Note: Excessive use of triggers can impair the performance of data manipulation on the master table, particularly if the network fails.

Example

In New York, replicate the local EMPLOYEES table to San Francisco.

Computing Derived Data Within the Server

```
UPDATE departments
  SET total_sal=(SELECT SUM(salary)
                  FROM employees
                 WHERE employees.department_id =
                      departments.department_id);
```



Copyright © 2009, Oracle. All rights reserved.

Computing Derived Data Within the Server

By using the server, you can schedule batch jobs or use the database Scheduler for the following scenarios:

- Compute derived column values asynchronously, at user-defined intervals.
- Store derived values only within database tables.
- Modify data in one pass to the database and calculate derived data in a second pass.

Alternatively, you can use triggers to keep running computations of derived data.

Example

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

Computing Derived Values with a Trigger

```
CREATE PROCEDURE increment_salary
  (id NUMBER, new_sal NUMBER) IS
BEGIN
  UPDATE departments
  SET    total_sal = NVL (total_sal, 0)+ new_sal
  WHERE  department_id = id;
END increment_salary;
```

```
CREATE OR REPLACE TRIGGER compute_salary
AFTER INSERT OR UPDATE OF salary OR DELETE
ON employees FOR EACH ROW
BEGIN
  IF DELETING THEN      increment_salary(
    :OLD.department_id, (-1*:OLD.salary));
  ELSIF UPDATING THEN  increment_salary(
    :NEW.department_id, (:NEW.salary-:OLD.salary));
  ELSE                 increment_salary(
    :NEW.department_id,:NEW.salary); --INSERT
  END IF;
END;
```

Copyright © 2009, Oracle. All rights reserved.

Computing Derived Data Values with a Trigger

By using a trigger, you can perform the following tasks:

- Compute derived columns synchronously, in real time.
- Store derived values within database tables or within package global variables.
- Modify data and calculate derived data in a single pass to the database.

Example

Keep a running total of the salary for each department in the special TOTAL_SALARY column of the DEPARTMENTS table.

Logging Events with a Trigger

```

CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF quantity_on_hand, reorder_point
ON inventories FOR EACH ROW
DECLARE
  dsc product_descriptions.product_description%TYPE;
  msg_text VARCHAR2(2000);
BEGIN
  IF :NEW.quantity_on_hand <=
    :NEW.reorder_point THEN
    SELECT product_description INTO dsc
    FROM product_descriptions
    WHERE product_id = :NEW.product_id;
    msg_text := 'ALERT: INVENTORY LOW ORDER:' ||
      'Yours,' || CHR(10) || user || '.' || CHR(10);
  ELSIF :OLD.quantity_on_hand >=
    :NEW.quantity_on_hand THEN
    msg_text := 'Product #' || ... CHR(10);
  END IF;
  UTL_MAIL.SEND('inv@oracle.com', 'ord@oracle.com',
    message=>msg_text, subject='Inventory Notice');
END;

```

Copyright © 2009, Oracle. All rights reserved.

Logging Events with a Trigger

In the server, you can log events by querying data and performing operations manually. This sends an email message when the inventory for a particular product has fallen below the acceptable limit. This trigger uses the Oracle-supplied package UTL_MAIL to send the email message.

Logging Events Within the Server

1. Query data explicitly to determine whether an operation is necessary.
2. Perform the operation, such as sending a message.

Using Triggers to Log Events

1. Perform operations implicitly, such as firing off an automatic electronic memo.
2. Modify data and perform its dependent operation in a single step.
3. Log events automatically as data is changing.

Logging Events with a Trigger (continued)

Logging Events Transparently

In the trigger code:

- CHR(10) is a carriage return
- Reorder_point is not NULL
- Another transaction can receive and read the message in the pipe

Example

```

CREATE OR REPLACE TRIGGER notify_reorder_rep
BEFORE UPDATE OF amount_in_stock, reorder_point
ON inventory FOR EACH ROW
DECLARE
    dsc product.descrip%TYPE;
    msg_text VARCHAR2(2000);
BEGIN
    IF :NEW.amount_in_stock <= :NEW.reorder_point THEN
        SELECT descrip INTO dsc
        FROM PRODUCT WHERE prodid = :NEW.product_id;
        msg_text := 'ALERT: INVENTORY LOW ORDER:' || CHR(10) ||
        'It has come to my personal attention that, due to recent' ||
        CHR(10) || 'transactions, our inventory for product #' ||
        TO_CHAR(:NEW.product_id) || '-- ' || dsc ||
        ' -- has fallen below acceptable levels.' || CHR(10) ||
        'Yours,' || CHR(10) || user || '.' || CHR(10) || CHR(10);
    ELSIF :OLD.amount_in_stock >= :NEW.amount_in_stock THEN
        msg_text := 'Product #' || TO_CHAR(:NEW.product_id)
        || ' ordered.' || CHR(10) || CHR(10);
    END IF;
    UTL_MAIL.SEND('inv@oracle.com', 'ord@oracle.com',
        message => msg_text, subject => 'Inventory Notice');
END;

```

Summary

In this appendix, you should have learned how to:

- Enhance database security with triggers
- Enforce data integrity with DML triggers
- Maintain referential integrity using triggers
- Use triggers to replicate data between tables
- Use triggers to automate computation of derived data
- Provide event-logging capabilities using triggers



Copyright © 2009, Oracle. All rights reserved.

Using the DBMS_SCHEDULER and HTP Packages

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

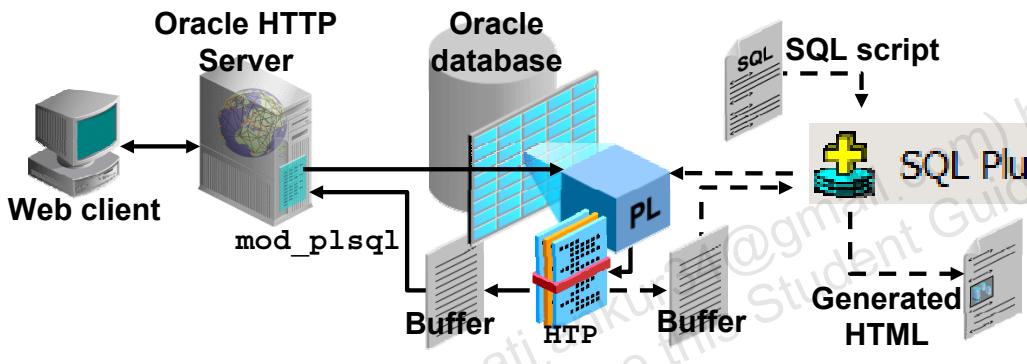
- Use the `HTP` package to generate a simple Web page
- Call the `DBMS_SCHEDULER` package to schedule PL/SQL code for execution



Copyright © 2009, Oracle. All rights reserved.

Generating Web Pages with the HTP Package

- The HTP package procedures generate HTML tags.
- The HTP package is used to generate HTML documents dynamically and can be invoked from:
 - A browser using Oracle HTTP Server and PL/SQL Gateway (`mod_plsql`) services
 - A SQL*Plus script to display HTML output



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Generating Web Pages with the HTP Package

The HTP package contains procedures that are used to generate HTML tags. The HTML tags that are generated typically enclose the data provided as parameters to the various procedures. The slide illustrates two ways in which the HTP package can be used:

- Most likely your procedures are invoked by the PL/SQL Gateway services, via the `mod_plsql` component supplied with Oracle HTTP Server, which is part of the Oracle Application Server product (represented by solid lines in the graphic).
- Alternatively (as represented by dotted lines in the graphic), your procedure can be called from *SQL*Plus* that can display the generated HTML output, which can be copied and pasted to a file. This technique is used in this course because Oracle Application Server software is not installed as a part of the course environment.

Note: The HTP procedures output information to a session buffer held in the database server. In the Oracle HTTP Server context, when the procedure completes, the `mod_plsql` component automatically receives the buffer contents, which are then returned to the browser as the HTTP response. In *SQL*Plus*, you must manually execute:

- A `SET SERVEROUTPUT ON` command
- The procedure to generate the HTML into the buffer
- The `OWA_UTIL.SHOWPAGE` procedure to display the buffer contents

Using the HTP Package Procedures

- Generate one or more HTML tags. For example:

```
htp.bold('Hello');           -- <B>Hello</B>
htp.print('Hi <B>World</B>'); -- Hi <B>World</B>
```

- Are used to create a well-formed HTML document:

<pre><code>BEGIN htp.htmlOpen; ----- htp.headOpen; ----- htp.title('Welcome'); -- htp.headClose; ----- htp.bodyOpen; ----- htp.print('My home page'); htp.bodyClose; ----- htp.htmlClose; ----- END;</code></pre>	-- Generates: <pre><code><HTML> <HEAD> <TITLE>Welcome</TITLE> </HEAD> <BODY> My home page </BODY> </HTML></code></pre>
---	---



Copyright © 2009, Oracle. All rights reserved.

Using the HTP Package Procedures

The HTP package is structured to provide a one-to-one mapping of a procedure to standard HTML tags. For example, to display bold text on a Web page, the text must be enclosed in the HTML tag pair `` and ``. The first code box in the slide shows how to generate the word Hello in HTML bold text by using the equivalent HTP package procedure—that is, `HTP.BOLD`. The `HTP.BOLD` procedure accepts a text parameter and ensures that it is enclosed in the appropriate HTML tags in the HTML output that is generated.

The `HTP.PRINT` procedure copies its text parameter to the buffer. The example in the slide shows how the parameter supplied to the `HTP.PRINT` procedure can contain HTML tags. This technique is recommended only if you need to use HTML tags that cannot be generated by using the set of procedures provided in the HTP package.

The second example in the slide provides a PL/SQL block that generates the basic form of an HTML document. The example serves to illustrate how each of the procedures generates the corresponding HTML line in the enclosed text box on the right.

The benefit of using the HTP package is that you create well-formed HTML documents, eliminating the need to manually enter the HTML tags around each piece of data.

Note: For information about all the HTP package procedures, refer to *PL/SQL Packages and Types Reference*.

Creating an HTML File with SQL*Plus

To create an HTML file with SQL*Plus, perform the following steps:

1. Create a SQL script with the following commands:

```
SET SERVEROUTPUT ON
ACCEPT procname PROMPT "Procedure: "
EXECUTE &procname
EXECUTE owa_util.showpage
UNDEFINE proc
```

2. Load and execute the script in SQL*Plus, supplying values for substitution variables.
3. Select, copy, and paste the HTML text that is generated in the browser to an HTML file.
4. Open the HTML file in a browser.



Copyright © 2009, Oracle. All rights reserved.

Creating an HTML File with SQL*Plus

The example in the slide shows the steps for generating HTML by using any procedure and saving the output into an HTML file. You should perform the following steps:

1. Turn on server output with the SET SERVEROUTPUT ON command. Without this, you receive exception messages when running procedures that have calls to the HTP package.
2. Execute the procedure that contains calls to the HTP package.
Note: This does *not* produce output, unless the procedure has calls to the DBMS_OUTPUT package.
3. Execute the OWA_UTIL.SHOWPAGE procedure to display the text. This call actually displays the HTML content that is generated from the buffer.

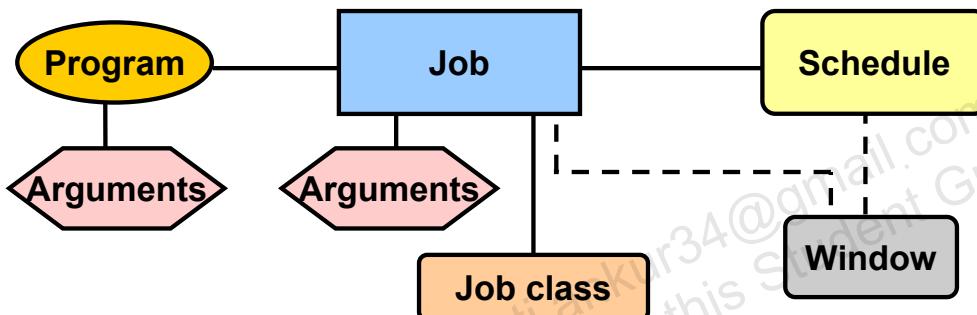
The ACCEPT command prompts for the name of the procedure to execute. The call to OWA_UTIL.SHOWPAGE displays the HTML tags in the browser window. You can then copy and paste the generated HTML tags from the browser window into an HTML file, typically with an .htm or .html extension.

Note: If you are using SQL*Plus, then you can use the SPOOL command to direct the HTML output directly to an HTML file.

The DBMS_SCHEDULER Package

The database Scheduler comprises several components to enable jobs to be run. Use the DBMS_SCHEDULER package to create each job with:

- A unique job name
- A program (“what” should be executed)
- A schedule (“when” it should run)



ORACLE

Copyright © 2009, Oracle. All rights reserved.

The DBMS_SCHEDULER Package

Oracle Database provides a collection of subprograms in the DBMS_SCHEDULER package to simplify management and to provide a rich set of functionality for complex scheduling tasks. Collectively, these subprograms are called the Scheduler and can be called from any PL/SQL program. The Scheduler enables database administrators and application developers to control when and where various tasks take place. By ensuring that many routine database tasks occur without manual intervention, you can lower operating costs, implement more reliable routines, and minimize human error.

The diagram shows the following architectural components of the Scheduler:

- A **job** is the combination of a program and a schedule. Arguments required by the program can be provided with the program or the job. All job names have the format [schema .] name. When you create a job, you specify the job name, a program, a schedule, and (optionally) job characteristics that can be provided through a **job class**.
- A **program** determines what should be run. Every automated job involves a particular executable, whether it is a PL/SQL block, a stored procedure, a native binary executable, or a shell script. A program provides metadata about a particular executable and may require a list of arguments.
- A **schedule** specifies when and how many times a job is executed.

The DBMS_SCHEDULER Package (continued)

- A **job class** defines a category of jobs that share common resource usage requirements and other characteristics. At any given time, each job can belong to only a single job class. A job class has the following attributes:
 - A database **service name**. The jobs in the job class will have an affinity to the particular service specified—that is, the jobs will run on the instances that cater to the specified service.
 - A **resource consumer group**, which classifies a set of user sessions that have common resource-processing requirements. At any given time, a user session or job class can belong to a single resource consumer group. The resource consumer group that the job class associates with determines the resources that are allocated to the job class.
- A **window** is represented by an interval of time with a well-defined beginning and end, and is used to activate different resource plans at different times.

The slide focuses on the job component as the primary entity. However, a program, a schedule, a window, and a job class are components that can be created as individual entities that can be associated with a job to be executed by the Scheduler. When a job is created, it may contain all the information needed inline—that is, in the call that creates the job. Alternatively, creating a job may reference a program or schedule component that was previously defined. Examples of this are discussed on the next few pages.

For more information about the Scheduler, see the online course titled *Oracle Database 11g: Configure and Manage Jobs with the Scheduler*.

Creating a Job

- A job can be created in several ways by using a combination of inline parameters, named Programs, and named Schedules.
- You can create a job with the CREATE_JOB procedure by:
 - Using inline information with the “what” and the schedule specified as parameters
 - Using a named (saved) program and specifying the schedule inline
 - Specifying what should be done inline and using a named Schedule
 - Using named Program and Schedule components



Copyright © 2009, Oracle. All rights reserved.

Creating a Job

The component that causes something to be executed at a specified time is called a **job**. Use the DBMS_SCHEDULER.CREATE_JOB procedure of the DBMS_SCHEDULER package to create a job, which is in a disabled state by default. A job becomes active and scheduled when it is explicitly enabled. To create a job, you:

- Provide a name in the format [schema.] name
- Need the CREATE JOB privilege

Note: A user with the CREATE ANY JOB privilege can create a job in any schema except the SYS schema. Associating a job with a particular class requires the EXECUTE privilege for that class.

In simple terms, a job can be created by specifying all the job details—the program to be executed (what) and its schedule (when)—in the arguments of the CREATE_JOB procedure. Alternatively, you can use predefined Program and Schedule components. If you have a named Program and Schedule, then these can be specified or combined with inline arguments for maximum flexibility in the way a job is created.

A simple logical check is performed on the schedule information (that is, checking the date parameters when a job is created). The database checks whether the end date is after the start date. If the start date refers to a time in the past, then the start date is changed to the current date.

Creating a Job with Inline Parameters

Specify the type of code, code, start time, and frequency of the job to be run in the arguments of the CREATE_JOB procedure.

```
-- Schedule a PL/SQL block every hour:  
  
BEGIN  
    DBMS_SCHEDULER.CREATE_JOB (  
        job_name => 'JOB_NAME',  
        job_type => 'PLSQL_BLOCK',  
        job_action => 'BEGIN ...; END;',  
        start_date => SYSTIMESTAMP,  
        repeat_interval=>'FREQUENCY=HOURLY; INTERVAL=1',  
        enabled => TRUE);  
END;  
/
```



Copyright © 2009, Oracle. All rights reserved.

Creating a Job with Inline Parameters

You can create a job to run a PL/SQL block, stored procedure, or external program by using the DBMS_SCHEDULER.CREATE_JOB procedure. The CREATE_JOB procedure can be used directly without requiring you to create Program or Schedule components.

The example in the slide shows how you can specify all the job details inline. The parameters of the CREATE_JOB procedure define “what” is to be executed, the schedule, and other job attributes. The following parameters define what is to be executed:

- The job_type parameter can be one of the following three values:
 - PLSQL_BLOCK for any PL/SQL block or SQL statement. This type of job cannot accept arguments.
 - STORED_PROCEDURE for any stored stand-alone or packaged procedure. The procedures can accept arguments that are supplied with the job.
 - EXECUTABLE for an executable command-line operating system application
- The schedule is specified by using the following parameters:
 - The start_date accepts a time stamp, and the repeat_interval is string-specified as a calendar or PL/SQL expression. An end_date can be specified.

Note: String expressions that are specified for repeat_interval are discussed later. The example specifies that the job should run every hour.

Creating a Job Using a Program

- Use CREATE_PROGRAM to create a program:

```
BEGIN
  DBMS_SCHEDULER.CREATE_PROGRAM(
    program_name => 'PROG_NAME',
    program_type => 'PLSQL_BLOCK',
    program_action => 'BEGIN ...; END;');
END;
```

- Use overloaded CREATE_JOB procedure with its program_name parameter:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
    program_name => 'PROG_NAME',
    start_date => SYSTIMESTAMP,
    repeat_interval => 'FREQ=DAILY',
    enabled => TRUE);
END;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating a Job Using a Program

The DBMS_SCHEDULER.CREATE_PROGRAM procedure defines a program that must be assigned a unique name. Creating the program separately for a job enables you to:

- Define the action once and then reuse this action within multiple jobs
- Change the schedule for a job without having to re-create the PL/SQL block
- Change the program executed without changing all the jobs

The program action string specifies a procedure, executable name, or PL/SQL block depending on the value of the program_type parameter, which can be:

- PLSQL_BLOCK to execute an anonymous block or SQL statement
- STORED_PROCEDURE to execute a stored procedure, such as PL/SQL, Java, or C
- EXECUTABLE to execute operating system command-line programs

The example shown in the slide demonstrates calling an anonymous PL/SQL block. You can also call an external procedure within a program, as in the following example:

```
DBMS_SCHEDULER.CREATE_PROGRAM(program_name =>
  'GET_DATE',
  program_action => '/usr/local/bin/date',
  program_type => 'EXECUTABLE');
```

To create a job with a program, specify the program name in the program_name argument in the call to the DBMS_SCHEDULER.CREATE_JOB procedure, as shown in the slide.

Creating a Job for a Program with Arguments

- Create a program:

```
DBMS_SCHEDULER.CREATE_PROGRAM(
    program_name => 'PROG_NAME',
    program_type => 'STORED PROCEDURE',
    program_action => 'EMP_REPORT');
```

- Define an argument:

```
DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT (
    program_name => 'PROG_NAME',
    argument_name => 'DEPT_ID',
    argument_position=> 1, argument_type=> 'NUMBER',
    default_value => '50');
```

- Create a job specifying the number of arguments:

```
DBMS_SCHEDULER.CREATE_JOB ('JOB_NAME', program_name
    => 'PROG_NAME', start_date => SYSTIMESTAMP,
    repeat_interval => 'FREQ=DAILY',
    number_of_arguments => 1, enabled => TRUE);
```

Copyright © 2009, Oracle. All rights reserved.

Creating a Job for a Program with Arguments

Programs, such as PL/SQL or external procedures, may require input arguments. Using the DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT procedure, you can define an argument for an existing program. The DEFINE_PROGRAM_ARGUMENT procedure parameters include the following:

- program_name specifies an existing program that is to be altered.
- argument_name specifies a unique argument name for the program.
- argument_position specifies the position in which the argument is passed when the program is called.
- argument_type specifies the data type of the argument value that is passed to the called program.
- default_value specifies a default value that is supplied to the program if the job that schedules the program does not provide a value.

The slide shows how to create a job executing a program with one argument. The program argument default value is 50. To change the program argument value for a job, use:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE (
    job_name => 'JOB_NAME',
    argument_name => 'DEPT_ID', argument_value => '80');
```

Creating a Job Using a Schedule

- Use CREATE_SCHEDULE to create a schedule:

```
BEGIN
    DBMS_SCHEDULER.CREATE_SCHEDULE ('SCHED_NAME',
        start_date => SYSTIMESTAMP,
        repeat_interval => 'FREQ=DAILY',
        end_date => SYSTIMESTAMP +15);
END;
```

- Use CREATE_JOB by referencing the schedule in the schedule_name parameter:

```
BEGIN
    DBMS_SCHEDULER.CREATE_JOB('JOB_NAME',
        schedule_name => 'SCHED_NAME',
        job_type => 'PLSQL_BLOCK',
        job_action => 'BEGIN ...; END;',
        enabled => TRUE);
END;
```



Copyright © 2009, Oracle. All rights reserved.

Creating a Job Using a Schedule

You can create a common schedule that can be applied to different jobs without having to specify the schedule details each time. The following are the benefits of creating a schedule:

- It is reusable and can be assigned to different jobs.
- Changing the schedule affects all jobs using the schedule. The job schedules are changed once, not multiple times.

A schedule is precise to only the nearest second. Although the TIMESTAMP data type is more accurate, the Scheduler rounds off anything with a higher precision to the nearest second.

The start and end times for a schedule are specified by using the TIMESTAMP data type. The end_date for a saved schedule is the date after which the schedule is no longer valid. The schedule in the example is valid for 15 days after using it with a specified job.

The repeat_interval for a saved schedule must be created by using a calendaring expression. A NULL value for repeat_interval specifies that the job runs only once.

Note: You cannot use PL/SQL expressions to express the repeat interval for a saved schedule.

Setting the Repeat Interval for a Job

- Using a calendaring expression:

```
repeat_interval=> 'FREQ=HOURLY; INTERVAL=4'
repeat_interval=> 'FREQ=DAILY'
repeat_interval=> 'FREQ=MINUTELY; INTERVAL=15'
repeat_interval=> 'FREQ=YEARLY;
    BYMONTH=MAR, JUN, SEP, DEC;
    BYMONTHDAY=15'
```

- Using a PL/SQL expression:

```
repeat_interval=> 'SYSDATE + 36/24'
repeat_interval=> 'SYSDATE + 1'
repeat_interval=> 'SYSDATE + 15/(24*60)'
```

Copyright © 2009, Oracle. All rights reserved.

Creating a Job Using a Schedule

You can create a common schedule that can be applied to different jobs without having to specify the schedule details each time. The following are the benefits of creating a schedule:

- It is reusable and can be assigned to different jobs.
- Changing the schedule affects all jobs using the schedule. The job schedules are changed once, not multiple times.

A schedule is precise to only the nearest second. Although the TIMESTAMP data type is more accurate, the Scheduler rounds off anything with a higher precision to the nearest second.

The start and end times for a schedule are specified by using the TIMESTAMP data type. The `end_date` for a saved schedule is the date after which the schedule is no longer valid. The schedule in the example is valid for 15 days after being used with a specified job.

The `repeat_interval` for a saved schedule must be created by using a calendaring expression. A NULL value for `repeat_interval` specifies that the job runs only once.

Note: You cannot use PL/SQL expressions to express the repeat interval for a saved schedule.

Creating a Job Using a Named Program and Schedule

- Create a named program called PROG_NAME by using the CREATE_PROGRAM procedure.
- Create a named schedule called SCHED_NAME by using the CREATE_SCHEDULE procedure.
- Create a job referencing the named program and schedule:

```
BEGIN
    DBMS_SCHEDULER.CREATE_JOB (
        'JOB_NAME',
        program_name => 'PROG_NAME',
        schedule_name => 'SCHED_NAME',
        enabled => TRUE);
END;
/
```



Copyright © 2009, Oracle. All rights reserved.

Creating a Job Using a Named Program and Schedule

The example in the slide shows the final form for using the DBMS_SCHEDULER.CREATE_JOB procedure. In this example, the named program (PROG_NAME) and schedule (SCHED_NAME) are specified in their respective parameters in the call to the DBMS_SCHEDULER.CREATE_JOB procedure.

With this example, you can see how easy it is to create jobs by using a predefined program and schedule.

Some jobs and schedules can be too complex to cover in this course. For example, you can create windows for recurring time plans and associate a resource plan with a window. A resource plan defines attributes about the resources required during the period defined by execution window.

For more information, refer to the online course titled *Oracle Database 11g: Configure and Manage Jobs with the Scheduler*.

Managing Jobs

- Run a job:

```
DBMS_SCHEDULER.RUN_JOB('SCHEMA.JOB_NAME');
```

- Stop a job:

```
DBMS_SCHEDULER.STOP_JOB('SCHEMA.JOB_NAME');
```

- Drop a job even if it is currently running:

```
DBMS_SCHEDULER.DROP_JOB('JOB_NAME', TRUE);
```

Copyright © 2009, Oracle. All rights reserved.

Managing Jobs

After a job has been created, you can:

- Run the job by calling the RUN_JOB procedure specifying the name of the job. The job is immediately executed in your current session.
- Stop the job by using the STOP_JOB procedure. If the job is running currently, it is stopped immediately. The STOP_JOB procedure has two arguments:
 - job_name:** Is the name of the job to be stopped
 - force:** Attempts to gracefully terminate a job. If this fails and force is set to TRUE, then the job slave is terminated. (Default value is FALSE.) To use force, you must have the MANAGE SCHEDULER system privilege.
- Drop the job with the DROP_JOB procedure. This procedure has two arguments:
 - job_name:** Is the name of the job to be dropped
 - force:** Indicates whether the job should be stopped and dropped if it is currently running (Default value is FALSE.)

If the DROP_JOB procedure is called and the job specified is currently running, then the command fails unless the force option is set to TRUE. If the force option is set to TRUE, then any instance of the job that is running is stopped and the job is dropped.

Note: To run, stop, or drop a job that belongs to another user, you need ALTER privileges on that job or the CREATE ANY JOB system privilege.

Data Dictionary Views

- [DBA | ALL | USER]_SCHEDULER_JOBS
- [DBA | ALL | USER]_SCHEDULER_RUNNING_JOBS
- [DBA | ALL]_SCHEDULER_JOB_CLASSES
- [DBA | ALL | USER]_SCHEDULER_JOB_LOG
- [DBA | ALL | USER]_SCHEDULER_JOB_RUN_DETAILS
- [DBA | ALL | USER]_SCHEDULER_PROGRAMS

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Data Dictionary Views

The DBA_SCHEDULER_JOB_LOG view shows all completed job instances, both successful and failed.

To view the state of your jobs, use the following query:

```
SELECT job_name, program_name, job_type, state  
FROM USER_SCHEDULER_JOBS;
```

To determine which instance a job is running on, use the following query:

```
SELECT owner, job_name, running_instance,  
resource_consumer_group  
FROM DBA_SCHEDULER_RUNNING_JOBS;
```

To determine information about how a job ran, use the following query:

```
SELECT job_name, instance_id, req_start_date,  
actual_start_date, status  
FROM ALL_SCHEDULER_JOB_RUN_DETAILS;
```

To determine the status of your jobs, use the following query:

```
SELECT job_name, status, error#, run_duration, cpu_used  
FROM USER_SCHEDULER_JOB_RUN_DETAILS;
```

Summary

In this appendix, you should have learned how to:

- Use the HTP package to generate a simple Web page
- Call the DBMS_SCHEDULER package to schedule PL/SQL code for execution



Copyright © 2009, Oracle. All rights reserved.

