Oracle Database 12*c*: Advanced PL/SQL

Activity Guide

D80343GC10
Edition 1.0
April 2014
D86297

Disclaimer

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

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

Restricted Rights Notice

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

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

Trademark Notice

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

Author

Sharon Sophia Stephen

Technical Contributors and Reviewers

Branislav Valny, Brent Dayley, Krishnanjani Chitta, Laszlo Czinkoczki, Nancy Greenberg, Sailaja Pasupuleti, Swarnapriya Shridhar, Wayne Abbott

This book was published using: Oracle Tutor

Table of Contents

Practices for Lesson 1: Introduction....1-1
Practices for Lesson 1: Overview....1-2
Practice 1-1: Identifying the Available SQL Developer Resources....1-3
Solution 1-1: Identifying the Available SQL Developer Resources....1-4
Practice 1-2: Creating and Using the New SQL Developer Database Connections....1-6
Solution 1-2: Creating and Using a New Oracle SQL Developer Database Connection....1-7
Practice 1-3: Browsing the HR, SH, and OE Schema Tables....1-12
Solution 1-3: Browsing the HR, SH, and OE Schema Tables....1-13
Practice 1- 4: Accessing the Oracle Database 12c Release 1 Online Documentation Library....1-17
Solution 1-4: Accessing the Oracle Database 12c, Release 1 Online Documentation Library....1-18

Practices for Lesson 2: PL/SQL Programming Concepts: Review....2-1
Practices for Lesson 2: Overview....2-2
Practice 2-1: PL/SQL Knowledge Quiz....2-3
Solution 2-1: PL/SQL Knowledge Quiz....2-5

Practices for Lesson 3: Designing PL/SQL Code....3-1
Practices for Lesson 3: Overview....3-2
Practice 3-1: Designing PL/SQL Code....3-3
Solution 3-1: Designing PL/SQL Code....3-5
Practice 3-2: Designing Using the ACCESSIBLE BY Clause....3-9
Solution 3-2: Designing Using the ACCESSIBLE BY Clause....3-10

Practices for Lesson 4: Overview of Collections....4-1
Practices for Lesson 4: Overview....4-2
Practice 4-1: Analyzing Collections....4-3
Solution 4-1: Analyzing Collections....4-5

Practices for Lesson 5: Using Collections....5-1
Practices for Lesson 5: Overview....5-2
Practice 5-1: Using Collections....5-3
Solution 5-1: Using Collections....5-6

Practices for Lesson 6: Manipulating Large Objects....6-1
Practices for Lesson 6: Overview....6-2
Practice 6-1: Working with LOBs....6-3
Solution 6-1: Working with LOBs....6-5

Practices for Lesson 7: Using Advanced Interface Methods....7-1
Practices for Lesson 7: Overview....7-2
Practice 7-1: Using Advanced Interface Methods....7-3
Solution 7-1: Using Advanced Interface Methods....7-5

Practices for Lesson 8: Performance and Tuning....8-1
Practices for Lesson 8: Overview....8-2
Practice 8-1: Performance and Tuning....8-3
Solution 8-1: Performance and Tuning....8-9

Practices for Lesson 9: Improving Performance with Caching....9-1
Practices for Lesson 9: Overview....9-2
Practice 9-1: Improving Performance with Caching....9-3
Solution 9-1: Improving Performance with Caching....9-5

Practices for Lesson 10: Analyzing PL/SQL Code 10-1
Practices for Lesson 10: Overview 10-2
Practice 10-1: Analyzing PL/SQL Code 10-3
Solution 10-1: Analyzing PL/SQL Code 10-5
Practices for Lesson 11: Profiling and Tracing PL/SQL Code 11-1
Practices for Lesson 11: Overview 11-2
Practice 11-1: Profiling and Tracing PL/SQL Code 11-3
Solution 11-1: Profiling and Tracing PL/SQL Code 11-4
Practices for Lesson 12: Implementing Fine-Grained Access Control for VPD 12-1
Practices for Lesson 12: Overview 12-2
Practice 12-1: Implementing Fine-Grained Access Control for VPD 12-3
Solution 12-1: Implementing Fine-Grained Access Control for VPD 12-6
Practices for Lesson 13: Safeguarding Your Code Against SQL Injection Attacks 13-1
Practices for Lesson 13: Overview 13-2
Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks 13-3
Solution 13-1: Safeguarding Your Code Against SQL Injection Attacks 13-5

Practices for Lesson 1: Introduction

Chapter 1

Practices for Lesson 1: Overview

Lesson Overview

This is the first of many practices in this course. The solutions (if you require them) can be found in the solutions section of each activity guide. 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, 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 the user account that you use in this course. You start SQL Developer, create a new database connection, and browse your `SH`, `HR`, and `OE` tables. You also execute SQL statements, and access and bookmark the Oracle Database 12*c* documentation and other useful websites that you can use in this course.

Practice 1-1: Identifying the Available SQL Developer Resources

Overview

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

Task

1. Familiarize yourself with Oracle SQL Developer, as needed, by referring to “Appendix B: Using SQL Developer.”
2. Access the SQL Developer Home page at: http://www.oracle.com/technology/products/database/sql_developer/index.html

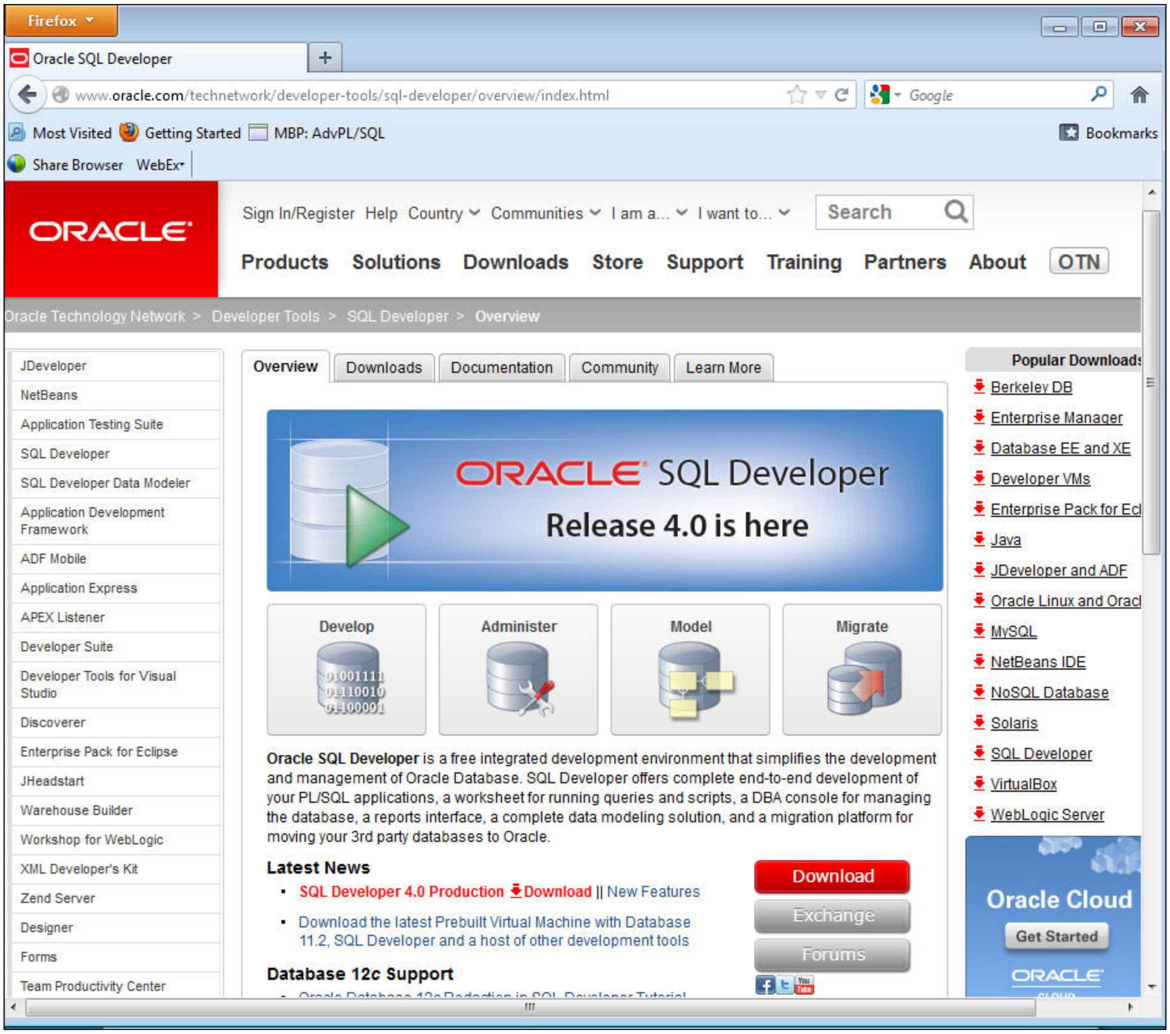
   Bookmark the page for future access.
3. Access the SQL Developer tutorials at: http://apex.oracle.com/pls/apex/f?p=44785:2:514976529400::NO:2,RIR,CIR:P2_GROUP_ID,P2_PRODUCT_ID:1000,2040

Solution 1-1: 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, by referring to "Appendix B: Using SQL Developer."
2. Access the online Oracle SQL Developer Home page at: http://www.oracle.com/technology/products/database/sql_developer/index.html

   The Oracle SQL Developer Home page appears:



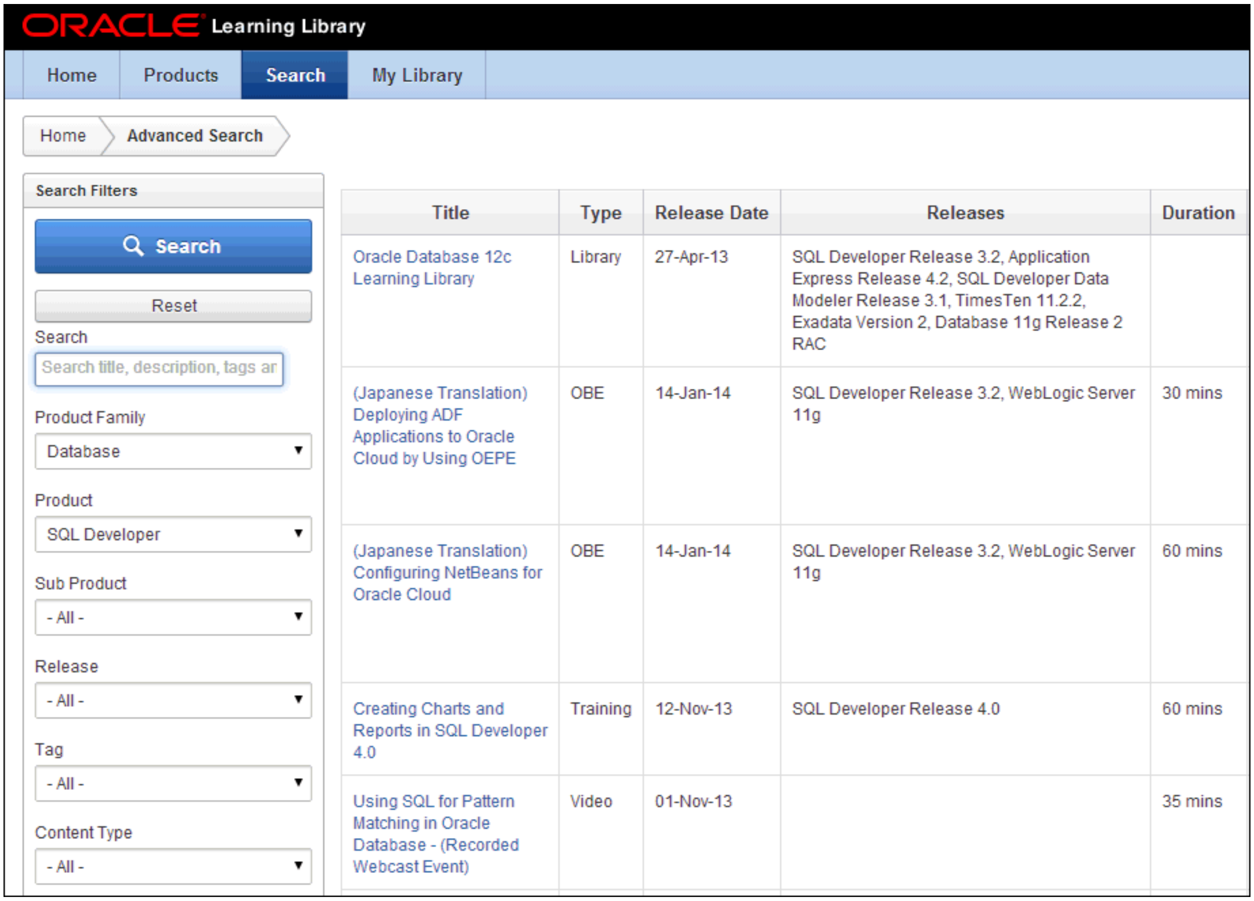
Oracle SQL Developer is a free integrated development environment that simplifies the development and management of Oracle Database. SQL Developer offers complete end-to-end development of your PL/SQL applications, a worksheet for running queries and scripts, a DBA console for managing the database, a reports interface, a complete data modeling solution, and a migration platform for moving your 3rd party databases to Oracle.

Latest News

- SQL Developer 4.0 Production Download || New Features
- Download the latest Prebuilt Virtual Machine with Database 11.2, SQL Developer and a host of other development tools

Database 12c Support

3. Access the SQL Developer tutorials at: http://apex.oracle.com/pls/apex/f?p=44785:2:514976529400::NO:2,RIR,CIR:P2_GROUP_ID,P2_PRODUCT_ID:1000,2040



Search: Search title, description, tags ar

Product Family: Database

Product: SQL Developer

Sub Product: - All -

Release: - All -

Tag: - All -

Content Type: - All -

Title	Type	Release Date	Releases	Duration
Oracle Database 12c Learning Library	Library	27-Apr-13	SQL Developer Release 3.2, Application Express Release 4.2, SQL Developer Data Modeler Release 3.1, TimesTen 11.2.2, Exadata Version 2, Database 11g Release 2 RAC	
(Japanese Translation) Deploying ADF Applications to Oracle Cloud by Using OEPE	OBE	14-Jan-14	SQL Developer Release 3.2, WebLogic Server 11g	30 mins
(Japanese Translation) Configuring NetBeans for Oracle Cloud	OBE	14-Jan-14	SQL Developer Release 3.2, WebLogic Server 11g	60 mins
Creating Charts and Reports in SQL Developer 4.0	Training	12-Nov-13	SQL Developer Release 4.0	60 mins
Using SQL for Pattern Matching in Oracle Database - (Recorded Webcast Event)	Video	01-Nov-13		35 mins

Practice 1-2: Creating and Using the New SQL Developer Database Connections

Overview

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

Tasks

1. Start SQL Developer.
2. Create a database connection by using the following information:
   a. Connection Name: `sh_connection`
   b. Username: `sh`
   c. Password: `sh`
   d. Hostname: `localhost`
   e. Port: `1521`
   f. SID: `orcl`
3. Test the new connection. If the Status is Success, use this new connection to connect to the database.
   a. Click the Test button in the “New / Select Database Connection” window. If the Status is Success, click the Connect button.
4. Similarly, create a new database connection named `hr_connection`.
   a. Connection Name: `hr_connection`
   b. Username: `hr`
   c. Password: `hr`
   d. Hostname: `localhost`
   e. Port: `1521`
   f. SID: `orcl`

   Note: Repeat step 3 to test the new `hr_connection` connection.
5. Similarly, create a new database connection named `oe_connection`.
   a. Connection Name: `oe_connection`
   b. Username: `oe`
   c. Password: `oe`
   d. Hostname: `localhost`
   e. Port: `1521`
   f. SID: `orcl`

   Note: Repeat step 3 to test the new `oe_connection` connection.
6. Repeat step 5 to create and test a new database connection named `sys_connection`. Enter `sys as sysdba` as the database connection username and `oracle` as the password.

   Note: Repeat step 3 to test the new `sys_connection` connection.

Solution 1-2: Creating and Using a New Oracle SQL Developer Database Connection

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

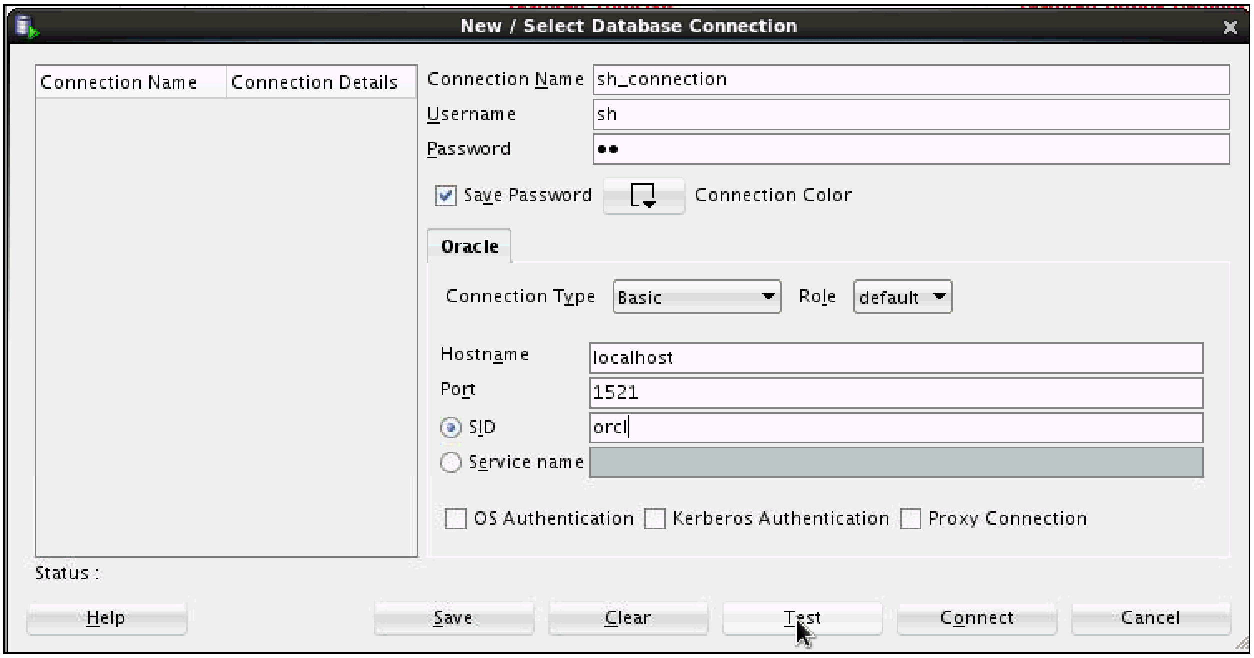
1. Start Oracle SQL Developer.

   Click the Oracle SQL Developer icon on your desktop.

2. Create a database connection by using the following information:
   a. Connection Name: `sh_connection`
   b. Username: `sh`
   c. Password: `sh`
   d. Hostname: `localhost`
   e. Port: `1521`
   f. SID: `orcl`

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

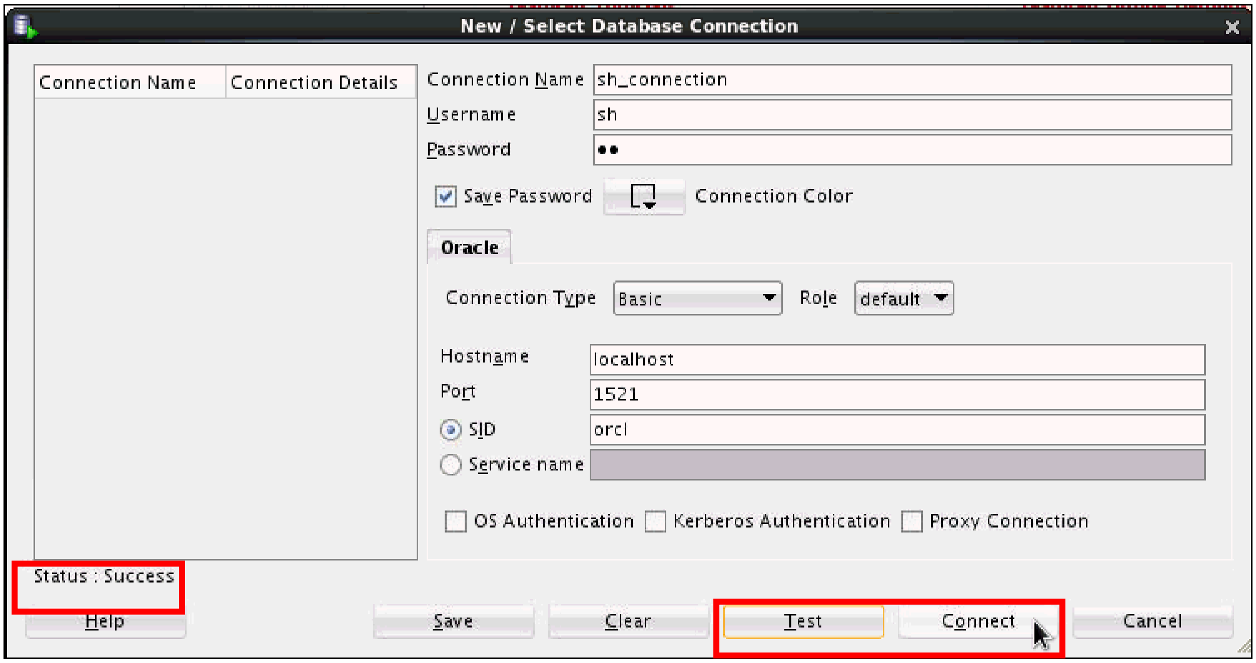
Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Enter the username, password, host name, and service name with the appropriate information, as provided above. The following is a sample of the newly created database connection for the `SH` schema using a local connection:



Connection Name: sh_connection
Username: sh
Password: ••
☑ Save Password
Connection Color
Oracle
Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
● SID: orcl
○ Service name
□ OS Authentication □ Kerberos Authentication □ Proxy Connection
Status :

3. Test the new connection. If the Status is Success, use this new connection to connect to the database.

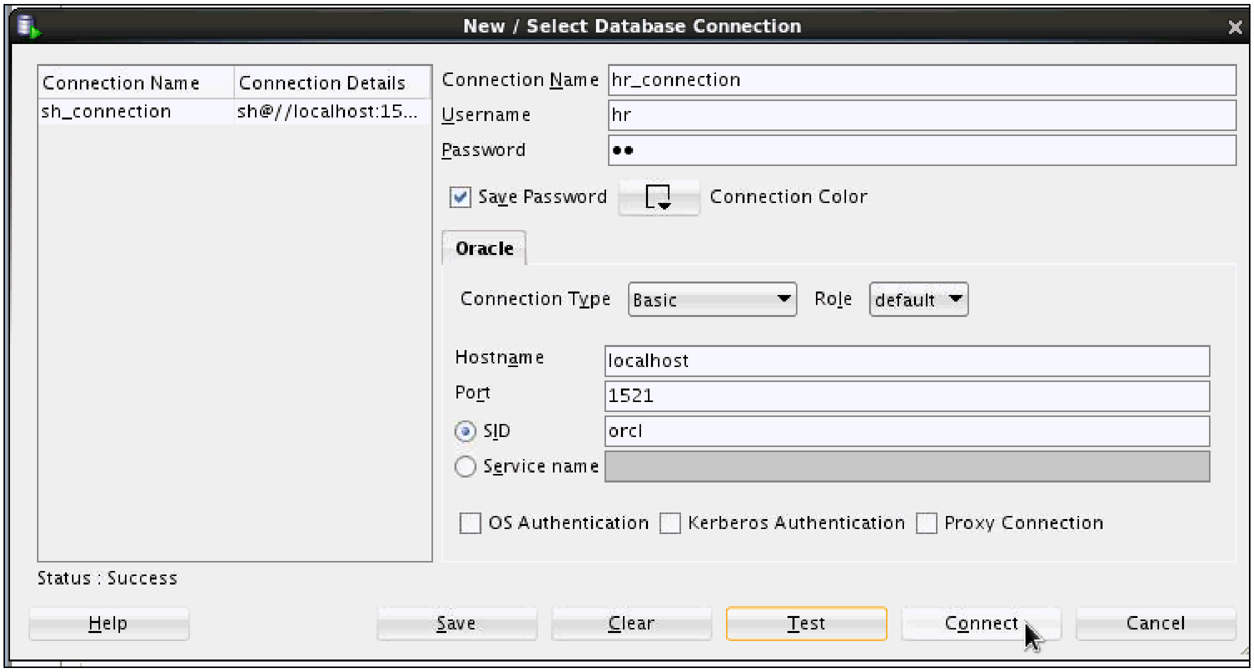
   a. Click the Test button in the “New / Select Database Connection” window. If the Status is Success, click the Connect button.



Connection Name: sh_connection
Username: sh
Password: ••
☑ Save Password
Connection Color
Oracle
Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
● SID: orcl
○ Service name
□ OS Authentication □ Kerberos Authentication □ Proxy Connection
Status : Success

4. Similarly, create a new database connection named `hr_connection`.
   a. Connection Name: `hr_connection`
   b. Username: `hr`
   c. Password: `hr`
   d. Hostname: `localhost`
   e. Port: `1521`
   f. SID: `orcl`

Note: Repeat step 3 to test the new `hr_connection` connection.



New / Select Database Connection

Connection Name	Connection Details
sh_connection	sh@//localhost:15...

Connection Name: hr_connection
Username: hr
Password: ••
☑ Save Password
Connection Color

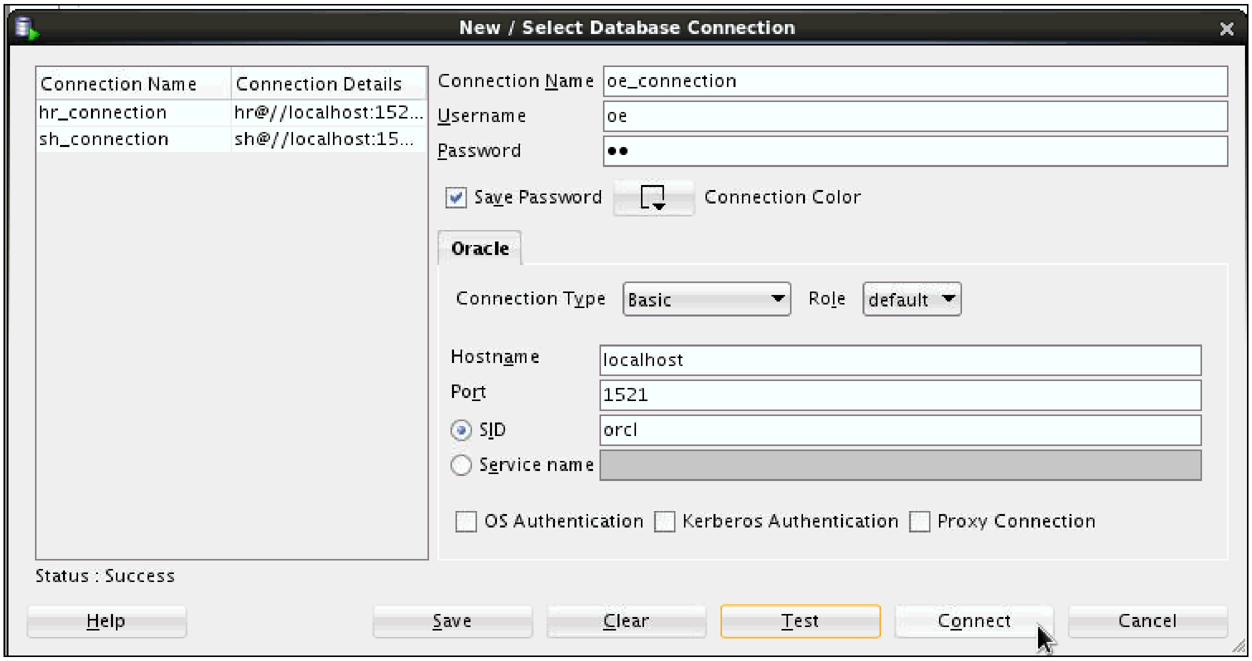
Oracle

Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
● SID: orcl
○ Service name:
□ OS Authentication □ Kerberos Authentication □ Proxy Connection

Status : Success

5. Similarly, create a new database connection named `oe_connection`.
   a. Connection Name: `oe_connection`
   b. Username: `oe`
   c. Password: `oe`
   d. Hostname: `localhost`
   e. Port: `1521`
   f. SID: `orcl`.

Note: Repeat step 3 to test the new `oe_connection` connection.



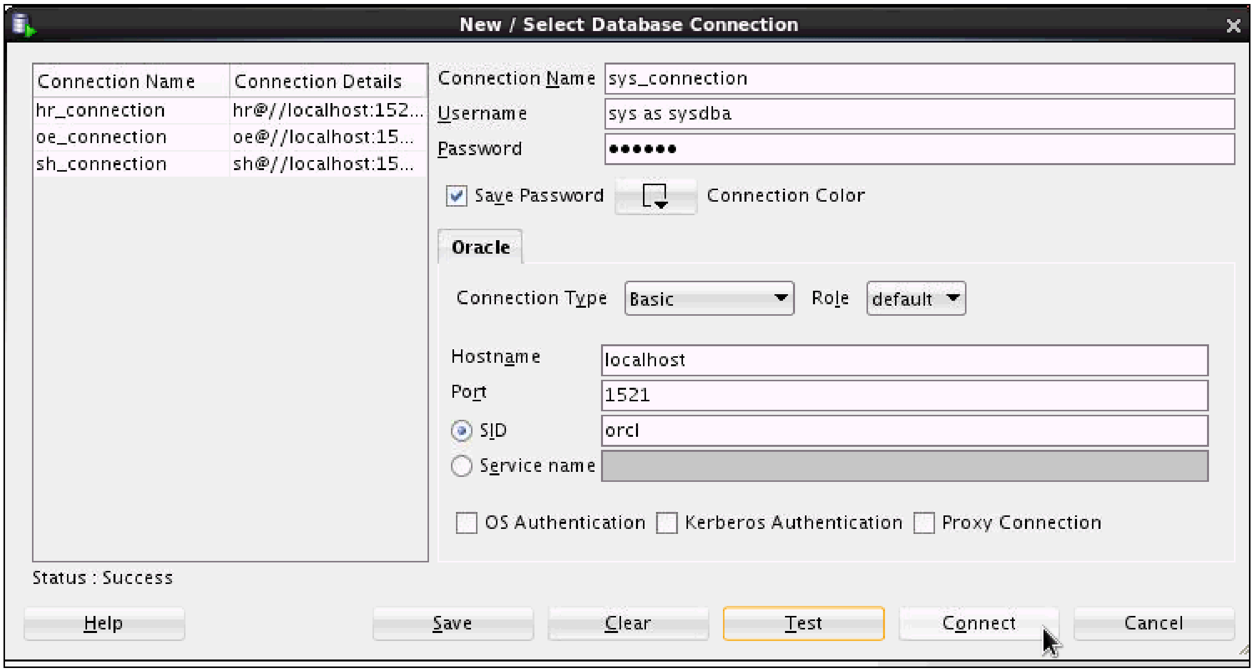
Connection Name	Connection Details
hr_connection	hr@//localhost:152...
sh_connection	sh@//localhost:15...

Connection Name: oe_connection
Username: oe
Password: ●●
☑ Save Password
Connection Color
Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
● SID: orcl
○ Service name
□ OS Authentication □ Kerberos Authentication □ Proxy Connection

Status : Success

6. Repeat step 5 to create and test a new database connection named `sys_connection`. Enter `sys as sysdba` as the database connection username and `oracle` as the password.

   Note: Repeat step 3 to test the new `sys_connection` connection.



New / Select Database Connection

Connection Name	Connection Details
hr_connection	hr@//localhost:152...
oe_connection	oe@//localhost:15...
sh_connection	sh@//localhost:15...

Connection Name: sys_connection

Username: sys as sysdba

Password: ●●●●●●

☑ Save Password

Connection Color

Oracle

Connection Type: Basic

Role: default

Hostname: localhost

Port: 1521

● SID: orcl

○ Service name

□ OS Authentication □ Kerberos Authentication □ Proxy Connection

Status : Success

Practice 1-3: Browsing the HR, SH, and OE Schema Tables

Overview

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

Tasks

1. Browse the structure of the EMPLOYEES table in the HR schema.
   a. Expand the hr_connection 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 income is greater than $10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the SELECT statement. Review the results of both methods of executing the SELECT statements on the appropriate tabs.

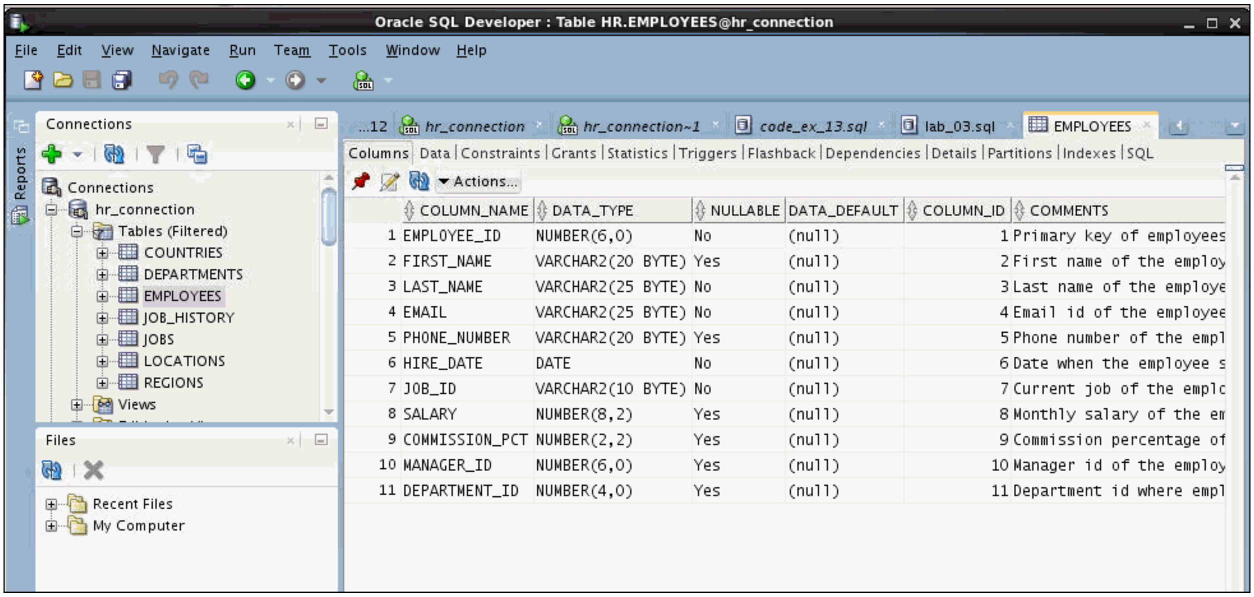
   Note: Take a few minutes to familiarize yourself with the data, or consult “Appendix A: Table Descriptions and Data,” which provides the description and data for all tables in the HR, SH, and OE schemas that you will use in this course.
4. Create and execute a simple anonymous block that outputs “Hello 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 icon (F5) to run the anonymous block.
5. Browse the structure of the SALES table in the SH schema and display its data.
   a. Double-click the sh_connection connection.
   b. Expand the Tables icon by clicking the plus sign next to it.
   c. Display the structure of the SALES table.
   d. Browse the SALES table and display its data.
6. Browse the structure of the ORDERS table in the OE schema and display its data.
   a. Double-click the oe_connection database connection.
   b. Expand the Tables icon by clicking the plus sign next to it.
   c. Display the structure of the ORDERS table.
   d. Browse the ORDERS table and display its data.

Solution 1-3: Browsing the HR, SH, and OE Schema Tables

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

1. Browse the structure of the EMPLOYEES table in the HR schema.
   a. Expand the hr_connection 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.

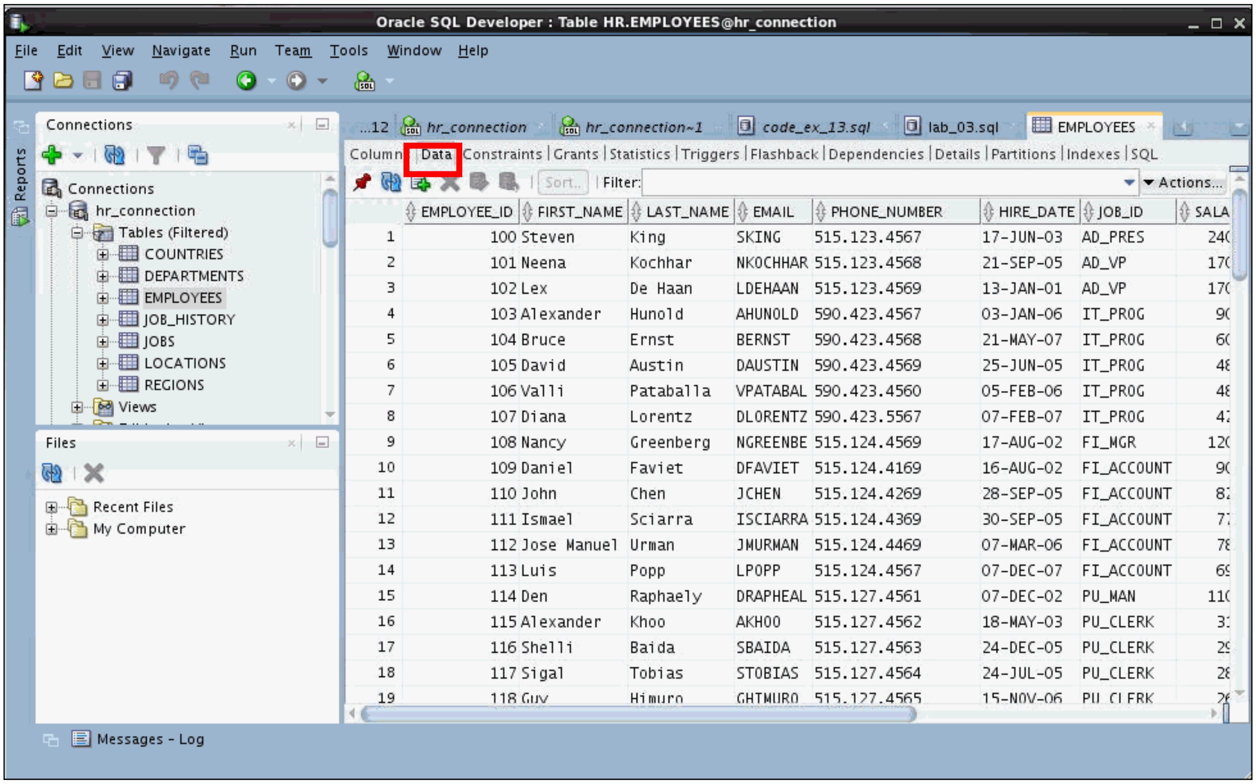
      Double-click the EMPLOYEES table. The Columns tab displays the columns in the EMPLOYEES table:



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	Primary key of employees
2	FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	First name of the employ
3	LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	Last name of the employe
4	EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	Email id of the employee
5	PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	Phone number of the empl
6	HIRE_DATE	DATE	No	(null)	6	Date when the employee s
7	JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	Current job of the emplc
8	SALARY	NUMBER(8,2)	Yes	(null)	8	Monthly salary of the em
9	COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	Commission percentage of
10	MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	Manager id of the employ
11	DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	Department id where empl

2. Browse the `EMPLOYEES` table and display its data.

   To display the employee data, click the Data tab. The `EMPLOYEES` table data is displayed:



	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	240
2	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-05	AD_VP	170
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-01	AD_VP	170
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-06	IT_PROG	90
5	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-07	IT_PROG	60
6	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-05	IT_PROG	48
7	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-06	IT_PROG	48
8	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-07	IT_PROG	42
9	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-02	FI_MGR	120
10	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-02	FI_ACCOUNT	90
11	110	John	Chen	JCHEN	515.124.4269	28-SEP-05	FI_ACCOUNT	82
12	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-05	FI_ACCOUNT	77
13	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-06	FI_ACCOUNT	78
14	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-07	FI_ACCOUNT	69
15	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-02	PU_MAN	110
16	115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-03	PU_CLERK	31
17	116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-05	PU_CLERK	29
18	117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-05	PU_CLERK	28
19	118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-06	PU_CLERK	26

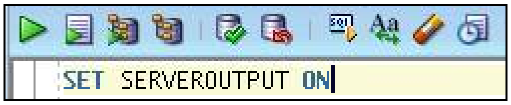
3. Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than $10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements on the appropriate tabs.

   Note: Take a few minutes to familiarize yourself with the data, or consult “Appendix B: Table Descriptions and Data,” which provides the description and data for all tables in the `HR`, `SH`, and `OE` schemas that you will use in this course.

   Display the SQL Worksheet by using either of the following two methods: Select Tools > SQL Worksheet, or click the Open SQL Worksheet icon. The Select Connection window appears. Enter the following statement in the SQL Worksheet:

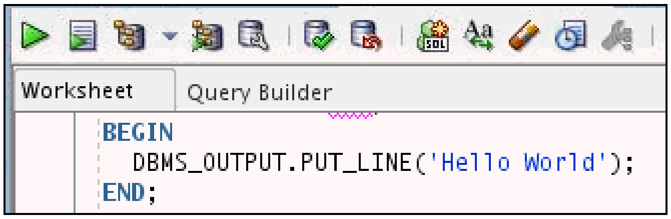
```
SELECT *
FROM employees
WHERE SALARY > 10000;
```

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.



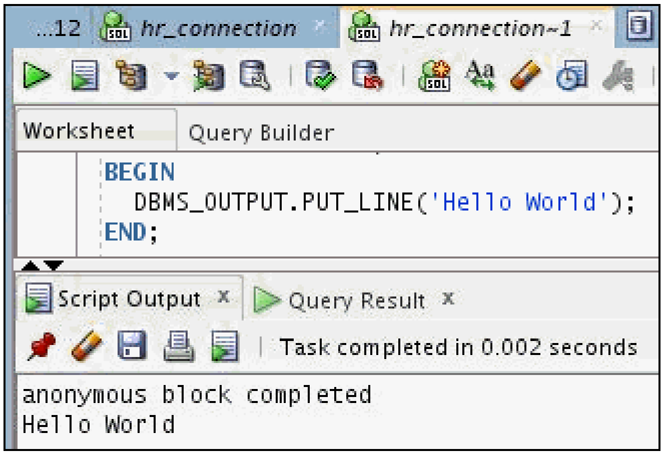
```
SET SERVEROUTPUT ON
```

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



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

   c. Click the Run Script icon (F5) to run the anonymous block.
   The Script Output tab displays the output of the anonymous block:



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

Task completed in 0.002 seconds

```
anonymous block completed
Hello World
```

5. Browse the structure of the `SALES` table in the `SH` schema and display its data.
   a. Double-click the `sh_connection` database connection.
   b. Expand the Tables icon by clicking the plus sign next to it.
   c. Display the structure of the `SALES` table.
   d. Browse the `SALES` table and display its data.

   Double-click the `SALES` table. The `Columns` tab displays the columns in the `SALES` table. To display the sales data, click the Data tab. The `SALES` table data is displayed.

6. Browse the structure of the `ORDERS` table in the `OE` schema and display its data.
   a. Double-click the `oe_connection` database connection.
   b. Expand the Tables icon by clicking the plus sign next to it.
   c. Display the structure of the `ORDERS` table.
   d. Browse the `ORDERS` table and display its data.

   Double-click the `ORDERS` table. The `Columns` tab displays the columns in the `ORDERS` table. To display the order data, click the Data tab. The `ORDERS` table data is displayed.

Practice 1- 4: Accessing the Oracle Database 12*c* Release 1 Online Documentation Library

Overview

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

Tasks

1. Access the Oracle Database 12*c* Release documentation webpage at: http://www.oracle.com/pls/db121/homepage
2. Bookmark the page for future access.
3. Display the complete list of books available for Oracle Database 12*c*, Release 1.
4. Make a note of the following documentation references that you will use in this course:
    - Advanced Application Developer's Guide
    - New Features Guide
    - PL/SQL Language Reference
    - Oracle Database Reference
    - Oracle Database Concepts
    - SQL Developer User's Guide
    - SQL Language Reference Guide
    - SQL*Plus User's Guide and Reference

Solution 1-4: Accessing the Oracle Database 12c, Release 1 Online Documentation Library

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

1. Access the Oracle Database 12*c* Release documentation webpage at: http://www.oracle.com/pls/db121/homepage
2. Bookmark the page for future access.
3. Display the complete list of books available for Oracle Database 12*c*, Release 1.
4. Make a note of the following documentation references that you will use in this course as needed:
    - *Oracle Database 2 Day + Data Warehousing Guide*
    - *Oracle Database Data Warehousing Guide*
    - *Oracle Database SQL Developer User's Guide*
    - *Oracle Database Reference*
    - *Oracle Database New Features Guide*
    - *Oracle Database SQL Language Reference*
    - *SQL*Plus User's Guide and Reference*
    - *Oracle Database SQLJ Developer's Guide and Reference*
    - *Oracle Database Concepts*
    - *Oracle Database Sample Schemas*

Practices for Lesson 2: PL/SQL Programming Concepts: Review

Chapter 2

Practices for Lesson 2: Overview

Lesson Overview

In this practice, you test and review your PL/SQL knowledge. This knowledge is necessary as a baseline for the subsequent chapters to build upon.

Practice 2-1: PL/SQL Knowledge Quiz

Overview

In this practice, you will answer some questions to check your understanding of the basic PL/SQL concepts.

Task

Review each question carefully, and write your answers on a sheet of paper.

Note: If you do not know an answer, go on to the next question.

PL/SQL Basics

1. Which are the four key areas of the basic PL/SQL block? What happens in each area?
2. What is a variable and where is it declared?
3. What is a constant and where is it declared?
4. What are the different modes for parameters and what does each mode do?
5. How does a function differ from a procedure?
6. Which are the two main components of a PL/SQL package?
   a. In what order are they defined?
   b. Are both required?
7. How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL*Plus?
8. What is a record?
9. What is an index by table?
10. How are loops implemented in PL/SQL?
11. How is branching logic implemented in PL/SQL?

Cursor Basics

12. What is an explicit cursor?
13. Where do you define an explicit cursor?
14. Name the five steps for using an explicit cursor.
15. What is the syntax used to declare a cursor?
16. What does the `FOR UPDATE` clause do within a cursor definition?
17. Which command opens an explicit cursor?
18. Which command closes an explicit cursor?
19. Name five implicit actions that a cursor FOR loop provides.

20. Describe what the following cursor attributes do:
    - `cursor_name%ISOPEN`
    - `cursor_name%FOUND`
    - `cursor_name%NOTFOUND`
    - `cursor_name%ROWCOUNT`

Exceptions

21. An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?
22. An exception handler is mandatory within a PL/SQL subprogram. (True/False)
23. What syntax do you use in the exception handler area of a subprogram?
24. How do you code for a `NO_DATA_FOUND` error?
25. Name three types of exceptions.
26. To associate an exception identifier with an Oracle error code, what `pragma` do you use and where?
27. How do you explicitly raise an exception?
28. What types of exceptions are implicitly raised?
29. What does the `RAISE_APPLICATION_ERROR` procedure do?

Dependencies

30. Which objects can a procedure or function directly reference?
31. Which are the two statuses that a schema object can have and where are they recorded?
32. The Oracle server automatically recompiles invalid procedures when they are called from the same ______. To avoid compile problems with remote database calls, you can use the ________ model instead of the timestamp model.
33. Which data dictionary contains information on direct dependencies?
34. What script would you run to create the `deptree` and `ideptree` views?
35. What does the `deptree_fill` procedure do and what are the arguments that you must provide?

Oracle-Supplied Packages

36. What does the `DBMS_OUTPUT` package do?
37. How do you write “This procedure works.” from within a PL/SQL program by using `DBMS_OUTPUT`?
38. What does `dbms_sql` do and how does this compare with native dynamic SQL?

Solution 2-1: PL/SQL Knowledge Quiz

Solution

In this practice, you will answer some questions to check your understanding of the basic PL/SQL concepts.

Task

Review each question carefully, and write your answers on a sheet of paper.

Note: If you do not know an answer, go on to the next question.

PL/SQL Basics

1. What are the four key areas of the basic PL/SQL block? What happens in each area?
    - **Header section: Names the program unit and identifies it as a procedure, function, or package; also identifies any parameters that the code may use**
    - **Declarative section: Area used to define variables, constants, cursors, and exceptions; starts with the keyword IS or AS**
    - **Executable section: Main processing area of the PL/SQL program; starts with the keyword BEGIN**
    - **Exception handler section: Optional error handling section; starts with the keyword EXCEPTION**
2. What is a variable and where is it declared?

    Variables are used to store data during PL/SQL block execution. You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the `NOT NULL` constraint on the variable.
    Syntax: `variable_name datatype[(size)][:= initial_value];`
3. What is a constant and where is it declared?

    Constants are variables that never change. Constants are declared and assigned a value in the declarative section, before the executable section.
    Syntax: `constant_name CONSTANT datatype[(size)] := initial_value;`
4. What are the different modes for parameters and what does each mode do?

    There are three parameter modes: `IN`, `OUT`, and `IN OUT`. `IN` is the default and it means that a value is passed into the subprogram. `OUT` mode indicates that the subprogram is passing a value generated in the subprogram out to the calling environment. `IN OUT` mode means that a value is passed into the subprogram. The subprogram may change the value and pass the value out to the calling environment.
5. How does a function differ from a procedure?

    A function must execute a `RETURN` statement that returns a value. Functions are called differently than procedures. They are called as an expression embedded within another command. Procedures are called as statements.

6. What are the two main components of a PL/SQL package?

   The package body and the package specification

   a. In what order are they defined?

   First the package specification, and then the package body

   b. Are both required?

   No, only a package specification is required. A specification can exist without a body, but a body cannot exist as valid without the specification.

7. How does the syntax of a `SELECT` statement used within a PL/SQL block differ from a `SELECT` statement issued in SQL*Plus?

   The `INTO` clause is required with a `SELECT` statement that is in a PL/SQL subprogram.

8. What is a record?

   A record is a composite type that has internal components, which can be manipulated individually. Use the `RECORD` data type to treat related but dissimilar data as a logical unit.

9. What is an index-by table?

   Index-by tables are a data structure declared in a PL/SQL block. It is similar to an array and comprises two components: the index and the data field. The data field is a column of a scalar or record data type, which stores the `INDEX BY` table elements.

10. How are loops implemented in PL/SQL?

    Looping constructs are used to repeat a statement or sequence of statements multiple times. PL/SQL has three looping constructs:

    - **Basic loops that perform repetitive actions without overall conditions**
    - **`FOR` loops that perform iterative control of actions based on a count**
    - **`WHILE` loops that perform iterative control of actions based on a condition**

11. How is branching logic implemented in PL/SQL?

    You can change the logical flow of statements within the PL/SQL block with a number of control structures. Branching logic is implemented within PL/SQL by using the conditional `IF` statement or `CASE` expressions.

Cursor Basics

12. What is an explicit cursor?

    The Oracle server uses work areas, called private SQL areas, to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. Use explicit cursors to individually process each row returned by a multiple-row `SELECT` statement.

13. Where do you define an explicit cursor?

    A cursor is defined in the declarative section.

14. Name the five steps for using an explicit cursor.

    Declare, Open, Fetch, Test for existing rows, and Close

15. What is the syntax used to declare a cursor?

    ```
    CURSOR cursor_name IS SELECT_statement
    ```

16. What does the FOR UPDATE clause do within a cursor definition?

    The FOR UPDATE clause locks the rows selected in the SELECT statement definition of the cursor.

17. What command opens an explicit cursor?

```
OPEN cursor_name;
```

18. What command closes an explicit cursor?

```
CLOSE cursor_name;
```

19. Name five implicit actions that a cursor FOR loop provides.

    Declares a record structure to match the select list of the cursor; opens the cursor; fetches from the cursor; exits the loop when the fetch returns no row; and closes the cursor

20. Describe what the following cursor attributes do:

    - **%ISOPEN: Returns a Boolean value indicating whether the cursor is open**
    - **%FOUND: Returns a Boolean value indicating whether the last fetch returned a value**
    - **%NOTFOUND: Returns a Boolean value indicating whether the last fetch did not return a value**
    - **%ROWCOUNT: Returns an integer indicating the number of rows fetched so far**

Exceptions

21. An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?

    Control is passed to the exception handler. If the exception is handled in the inner block, processing continues to the outer block. If the exception is not handled in the inner block, an exception is raised in the outer block and control is passed to the exception handler of the outer block. If neither the inner nor the outer block traps the exception, the program ends unsuccessfully.

22. An exception handler is mandatory within a PL/SQL subprogram. (True/False)

    False

23. What syntax do you use in the exception handler area of a subprogram?

```
EXCEPTION
       WHEN named_exception THEN
            statement[s];
            WHEN others THEN
            statement[s];
END;
```

24. How do you code for a NO_DATA_FOUND error?

```
EXCEPTION
       WHEN no_data_found THEN
            statement[s];
END;
```

25. Name three types of exceptions.

    User-defined, Oracle server predefined, and Oracle server non-predefined

26. To associate an exception identifier with an Oracle error code, what `pragma` do you use and where?

    Use the `PRAGMA EXCEPTION_INIT` and place the `PRAGMA EXCEPTION_INIT` in the declarative section.

27. How do you explicitly raise an exception?

    Use the `RAISE` statement or the `RAISE_APPLICATION_ERROR` procedure.

28. What types of exceptions are implicitly raised?

    All Oracle server exceptions (predefined and non-predefined) are automatically raised.

29. What does the `RAISE_APPLICATION_ERROR` procedure do?

    It enables you to issue user-defined error messages from subprograms.

Dependencies

30. Which objects can a procedure or function directly reference?

    Table, view, sequence, procedure, function, package specification, object specification, and collection type

31. What are the two statuses that a schema object can have and where are they recorded?

    The `user_objects` dictionary view contains a column called status. Its values are `VALID` and `INVALID`.

32. The Oracle server automatically recompiles invalid procedures when they are called from the same ______. To avoid compile problems with remote database calls, you can use the ________ model instead of the timestamp model.

    database
    signature

33. Which data dictionary contains information on direct dependencies?

    `user_dependencies`

34. Which script do you run to create the views `deptree` and `ideptree`?

    The `utldtree.sql` script

35. What does the `deptree_fill` procedure do and what are the arguments that you must provide?

    The `deptree_fill` procedure populates the `deptree` and `ideptree` views to display a tabular representation of all dependent objects, direct and indirect. You pass the object type, object owner, and object name to the `deptree_fill` procedure.

Oracle-Supplied Packages

36. What does the DBMS_OUTPUT package do?

    The DBMS_OUTPUT package enables you to send messages from stored procedures, packages, and triggers.

37. How do you write "This procedure works." from within a PL/SQL program by using DBMS_OUTPUT?

    ```
    DBMS_OUTPUT.PUT_LINE('This procedure works.');
    ```

38. What does dbms_sql do and how does it compare with native dynamic SQL?

    dbms_sql enables you to embed dynamic data manipulation language (DML), data definition language (DDL), and data control language (DCL) statements within a PL/SQL program. Native dynamic SQL allows you to place dynamic SQL statements directly into PL/SQL blocks. Native dynamic SQL in PL/SQL is easier to use than dbms_sql, requires much less application code, and performs better.

Practices for Lesson 3: Designing PL/SQL Code

Chapter 3

Practices for Lesson 3: Overview

Lesson Overview

In this practice, you determine the output of a PL/SQL code snippet and modify the snippet to improve the performance. Next, you implement subtypes and use cursor variables to pass values to and from a package.

Note: Files used in the practices are found in the `/labs` folder. Additionally, solution scripts are provided for each question and are located in the `/soln` folder. Your instructor will provide you with the exact location of these files. Connect as `OE` to perform the steps.

Practice 3-1: Designing PL/SQL Code

Overview

This practice covers the following topics:

- Determining the output of a PL/SQL block
- Improving the performance of a PL/SQL block
- Implementing subtypes
- Using cursor variables

Use OE connection to complete this practice.

Task

1. Determine the output of the following code snippet given in Task 1 of the lab_03.sql file.

```
SET SERVEROUTPUT ON
BEGIN
    UPDATE orders SET order_status = order_status;
    FOR v_rec IN ( SELECT order_id FROM orders )
    LOOP
          IF SQL%ISOPEN THEN
             DBMS_OUTPUT.PUT_LINE('TRUE -' || SQL%ROWCOUNT);
          ELSE
             DBMS_OUTPUT.PUT_LINE('FALSE -' || SQL%ROWCOUNT);
          END IF;
    END LOOP;
END;
/
```

2. Modify the following code snippet in Task 2 of the lab_03.sql file to make better use of the FOR UPDATE clause and improve the performance of the program.

```
DECLARE
    CURSOR cur_update
    IS SELECT * FROM customers
    WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
    FOR v_rec IN cur_update
    LOOP
          IF v_rec IS NOT NULL
          THEN
               UPDATE customers
               SET credit_limit = credit_limit + 200
               WHERE customer_id = v_rec.customer_id;
          END IF;
    END LOOP;
END;
/
```

3. Create a package specification that defines a subtype that can be used for the `warranty_period` field of the `product_information` table. Name this package `mytypes`. The type holds the month and year for a warranty period.
4. Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer ID, first name, phone numbers, credit limit, and email address. Both subroutines should use the cursor variable to return the necessary details.

Solution 3-1: Designing PL/SQL Code

This practice covers the following topics:

- Determining the output of a PL/SQL block
- Improving the performance of a PL/SQL block
- Implementing subtypes
- Using cursor variables

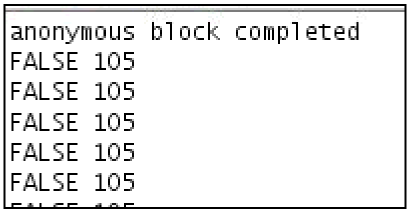
Use the `OE` connection to complete this practice.

1. Determine the output of the following code snippet in Task 1 of the `lab_03.sql` file.

   Connect as `OE`.

```
SET SERVEROUTPUT ON
BEGIN
  UPDATE orders SET order_status = order_status;
  FOR v_rec IN ( SELECT order_id FROM orders )
    LOOP
      IF SQL%ISOPEN THEN
        DBMS_OUTPUT.PUT_LINE('TRUE -' || SQL%ROWCOUNT);
      ELSE
        DBMS_OUTPUT.PUT_LINE('FALSE -' || SQL%ROWCOUNT);
      END IF;
    END LOOP;
END;
/
```

   Execute the code from the `lab_03.sql` file. It will show `FALSE – 105` for each row fetched.



```
anonymous block completed
FALSE 105
FALSE 105
FALSE 105
FALSE 105
FALSE 105
```

2. Modify the following code snippet in Task 2 of the `lab_03.sql` file to make better use of the `FOR UPDATE` clause and improve the performance of the program.

```
DECLARE
  CURSOR cur_update
    IS SELECT * FROM customers
    WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
    LOOP
      IF v_rec IS NOT NULL THEN
        UPDATE customers
          SET credit_limit = credit_limit + 200
          WHERE customer_id = v_rec.customer_id;
      END IF;
    END LOOP;
END;
/
```

Modify Task 2 in the `lab_03.sql` file to add the `'CURRENT OF'` clause:

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
    LOOP
      UPDATE customers
      SET credit_limit = credit_limit + 200
      WHERE CURRENT OF cur_update;
    END LOOP;
END;
/
```

```
anonymous block completed
```

Alternatively, you can execute the code from Task 2 of the `sol_03.sql` file.

3. Create a package specification that defines a subtype that can be used for the `warranty_period` field of the `product_information` table. Name this package `mytypes`. The type holds the month and year for a warranty period.

```
CREATE OR REPLACE PACKAGE mytypes
IS
  TYPE typ_warranty
    IS RECORD (month POSITIVE, year PLS_INTEGER);
  SUBTYPE warranty IS typ_warranty; -- based on RECORD type
END mytypes;
/
```

```
PACKAGE MYTYPES compiled
```

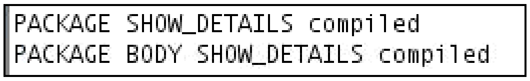
Alternatively, you can execute the code from Task 3 of the `sol_03.sql` file.

4. Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer ID, first name, phone numbers, credit limit, and email address.
   Both the subroutines should use the cursor variable to return the necessary details.

```
CREATE OR REPLACE PACKAGE show_details AS

TYPE rt_order IS REF CURSOR RETURN orders%ROWTYPE;

TYPE typ_cust_rec IS RECORD
  (cust_id NUMBER(6), cust_name VARCHAR2(20),
   custphone customers.phone_numbers%TYPE,
   credit NUMBER(9,2), cust_email VARCHAR2(30));
TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;

PROCEDURE get_order(p_orderid IN NUMBER, p_cv_order IN OUT
rt_order);
```

```
PROCEDURE get_cust(p_custid IN NUMBER, p_cv_cust IN OUT
rt_cust);
END show_details;
/

CREATE OR REPLACE PACKAGE BODY show_details AS
PROCEDURE get_order
  (p_orderid IN NUMBER, p_cv_order IN OUT rt_order)
IS
BEGIN
  OPEN p_cv_order FOR
    SELECT * FROM orders
      WHERE order_id = p_orderid;
-- CLOSE p_cv_order
END get_order;

PROCEDURE get_cust
  (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)
IS
BEGIN
  OPEN p_cv_cust FOR
    SELECT customer_id, cust_first_name, phone_numbers,
credit_limit,
           cust_email FROM customers
      WHERE customer_id = p_custid;
-- CLOSE p_cv_cust
```

```
END get_cust;
END;
/
```



```
PACKAGE SHOW_DETAILS compiled
PACKAGE BODY SHOW_DETAILS compiled
```

Alternatively, you can execute the code from Task 4 of the `sol_03.sql` file.

Practice 3-2: Designing Using the ACCESSIBLE BY Clause

Overview

In this practice, you will learn to use the ACCESSIBLE BY clause.

Use the HR and OE connections as instructed in the task.

Task

1. Examine the code given in Task 1 of practice 3-2 in the lab_03.sql file:

   Use the HR connection to execute the function TAX.

```
CREATE OR REPLACE FUNCTION TAX(P_AMOUNT IN NUMBER)
RETURN NUMBER
ACCESSIBLE BY (depts,oe.tax_amount)
IS
M  NUMBER;
BEGIN
IF P_AMOUNT <8000 THEN
M:=0.08;
ELSIF P_AMOUNT <18000 THEN
M:=0.25;
ELSE
M:=0.31;
END IF;
RETURN P_AMOUNT*M;
END;
/
```

2. Grant privilege to the OE user to execute the function TAX.
3. Create the depts procedure, which returns the employee details of the given dept_id.
4. Create another procedure depts2 and execute as the HR user. Examine the output.
5. Create a procedure to execute the TAX function from the OE connection. Observe the output.

   You should get an error when you try to access the restricted program unit.

Solution 3-2: Designing Using the ACCESSIBLE BY Clause

In this practice, you will learn to use the ACCESSIBLE BY clause.

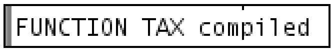
Use the HR and OE connections as instructed in the task.

Task

1. Examine the code given in Task 1 of practice 3-2 in the lab_03.sql file:

   Use the HR connection to execute the function TAX.

```
CREATE OR REPLACE FUNCTION TAX(P_AMOUNT IN NUMBER)
RETURN NUMBER
ACCESSIBLE BY (depts,oe.tax_amount)
IS
M  NUMBER;
BEGIN
IF P_AMOUNT <8000 THEN
M:=0.08;
ELSIF P_AMOUNT <18000 THEN
M:=0.25;
ELSE
M:=0.31;
END IF;
RETURN P_AMOUNT*M;
END;
/
```

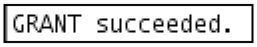


```
FUNCTION TAX compiled
```

   The TAX function uses the ACCESSIBLE BY clause to provide access of the TAX function to the depts procedure owned by HR, and to depts2 owned by the OE user.

2. Grant privilege to the OE user to execute the function TAX.

```
-- Use HR connection
GRANT EXECUTE ON TAX TO OE;
```

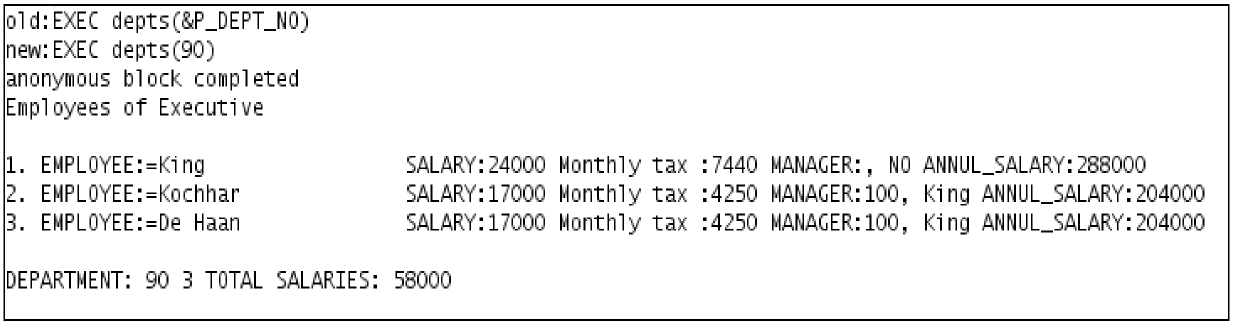


```
GRANT succeeded.
```

   Alternatively, you can run the code from Task 2 of practice 3-2 in sol_03.sql.

3. Use the HR connection and create the depts procedure, which returns the employee details of the given dept_id.

```
CREATE OR REPLACE PROCEDURE depts(P_DEPTNO
EMPLOYEES.DEPARTMENT_ID%TYPE

DEFAULT NULL)
IS
CURSOR C_EMP(C_DEPTNO EMPLOYEES.DEPARTMENT_ID%TYPE) IS
SELECT EMPLOYEE_ID,LAST_NAME,SALARY,MANAGER_ID,
       12*SALARY*(1+NVL(COMMISSION_PCT,0)) ANN_SAL
FROM EMPLOYEES WHERE DEPARTMENT_ID=NVL(C_DEPTNO,DEPARTMENT_ID);
R C_EMP%ROWTYPE;
SUMMARY NUMBER;
V_DEPT_NAME DEPARTMENTS.DEPARTMENT_NAME%TYPE;
MANAGER_NAME EMPLOYEES.LAST_NAME%TYPE;
BEGIN
SUMMARY:=0;
DBMS_OUTPUT.ENABLE(100000);
SELECT DEPARTMENT_NAME INTO V_DEPT_NAME
FROM DEPARTMENTS WHERE DEPARTMENT_ID=P_DEPTNO;
DBMS_OUTPUT.PUT_LINE('Employees of '||V_DEPT_NAME||chr(10));
OPEN C_EMP(P_DEPTNO);
LOOP
  FETCH C_EMP INTO R ;
  EXIT WHEN C_EMP%NOTFOUND;
  SUMMARY:=SUMMARY+R.SALARY;
  IF R.MANAGER_ID IS NOT NULL THEN
  SELECT LAST_NAME INTO MANAGER_NAME FROM EMPLOYEES
  WHERE EMPLOYEE_ID=R.MANAGER_ID;
  ELSE
  MANAGER_NAME:='NO';
  END IF;
  DBMS_OUTPUT.PUT_LINE(C_EMP%ROWCOUNT||'. EMPLOYEE:='||rpad
(R.LAST_NAME,20,' ')||
  ' SALARY:'||R.SALARY||' Monthly tax :'||tax(r.salary)||
  ' MANAGER:'||R.MANAGER_ID||', '|| MANAGER_NAME||'
ANNUL_SALARY:'||

R.ANN_SAL);
END LOOP;
DBMS_OUTPUT.PUT_LINE(CHR(10)||'DEPARTMENT: '||P_DEPTNO||'
'||C_EMP

%ROWCOUNT
   ||' TOTAL SALARIES: '||SUMMARY);
CLOSE C_EMP;
EXCEPTION
WHEN no_data_found THEN
DBMS_OUTPUT.PUT_LINE('No department !');

END depts;
/
SET SERVEROUTPUT ON;
EXEC depts(&P_DEPT_NO);
```



```
old:EXEC depts(&P_DEPT_NO)
new:EXEC depts(90)
anonymous block completed
Employees of Executive

1. EMPLOYEE:=King                 SALARY:24000 Monthly tax :7440 MANAGER:, NO ANNUL_SALARY:288000
2. EMPLOYEE:=Kochhar              SALARY:17000 Monthly tax :4250 MANAGER:100, King ANNUL_SALARY:204000
3. EMPLOYEE:=De Haan              SALARY:17000 Monthly tax :4250 MANAGER:100, King ANNUL_SALARY:204000

DEPARTMENT: 90 3 TOTAL SALARIES: 58000
```

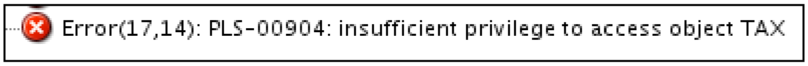
Alternatively, you can run the solution from Task 3 of practice 3-2 in `sol_03.sql`.

4. Create another procedure `depts2` and execute as the `HR` user. Examine the output.

```
CREATE OR REPLACE PROCEDURE depts2(p_deptno
employees.department_id%type:=90)
AUTHID CURRENT_USER IS
v_max_sal NUMBER;
BEGIN
SELECT max(tax(salary)) INTO v_max_sal
FROM EMPLOYEES
WHERE department_id=p_deptno;
DBMS_OUTPUT.PUT_LINE
('The Maximum Tax Value In Department('||P_DEPTNO||') Is:
'||V_MAX_SAL);
END depts2;
/
```

When the `HR` user wants to create a `depts2` procedure with the above-mentioned code, Oracle produces an error message as shown below.

This is because, the `depts2` procedure was not entitled to refer to the `TAX` function.



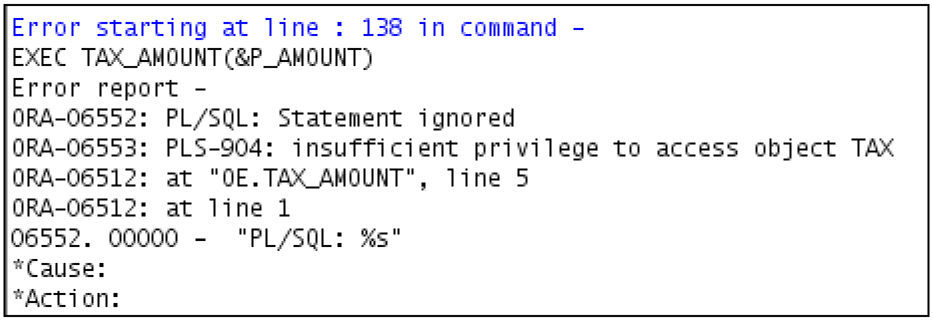
```
Error(17,14): PLS-00904: insufficient privilege to access object TAX
```

Alternatively, you can run the solution from Task 4 of practice 3-2 in `sol_03.sql`.

5. Create a procedure to execute the `TAX` function from the `OE` connection. Observe the output:

   Use the `OE` connection.

```
create or replace PROCEDURE OE.tax_amount(P_AMOUNT NUMBER)
authid current_user is
v_tax number;
begin
select hr.tax(p_amount) into v_tax
from dual;

dbms_output.put_line
('The tax value ('||p_amount||') is: '||v_tax);
end tax_amount;
/

SET SERVEROUTPUT ON;
EXEC TAX_AMOUNT(&P_AMOUNT);
/
```



```
Error starting at line : 138 in command -
EXEC TAX_AMOUNT(&P_AMOUNT)
Error report -
ORA-06552: PL/SQL: Statement ignored
ORA-06553: PLS-904: insufficient privilege to access object TAX
ORA-06512: at "OE.TAX_AMOUNT", line 5
ORA-06512: at line 1
06552. 00000 -  "PL/SQL: %s"
*Cause:
*Action:
```

You will get an error when you try to access the restricted program unit.

Alternatively, you can run the solution from Task 5 of practice 3-2 in `sol_03.sql`.

Practices for Lesson 4: Overview of Collections

Chapter 4

Practices for Lesson 4: Overview

Lesson Overview

In this practice, you analyze collections for common errors, and create a collection.

Use the OE schema for this practice.

Practice 4-1: Analyzing Collections

Overview

In this practice, you create a nested table collection and use PL/SQL code to manipulate the collection.

Use the OE connection for this practice.

Task

1. Examine the following definitions. Run task 1 of the lab_04.sql script to create these objects.

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder (  -- create database table
     ordid     NUMBER(5),
     supplier  NUMBER(5),
     requester NUMBER(4),
     ordered   DATE,
     items     typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```

2. The following code generates an error. Run task 2 of the lab_04.sql script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO TABLE (SELECT items
                   FROM   pOrder
                   WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```

   a. Why does the error occur?
   b. How can you fix the error?

3. Examine the following code, which produces an error. Which line causes the error, and how do you fix it?

   Note: You can run task 3 of the `lab_04.sql` script to view the error output.

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ;
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```

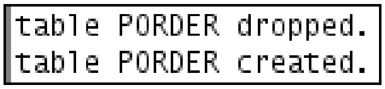
Solution 4-1: Analyzing Collections

In this practice, you create a nested table collection and use PL/SQL code to manipulate the collection.

Use the OE connection to complete the practice.

1. Examine the following definitions. Run task 1 of the lab_04.sql script to create these objects.

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
CREATE TABLE pOrder (  -- create database table
     ordid     NUMBER(5),
     supplier  NUMBER(5),
     requester NUMBER(4),
     ordered   DATE,
     items     typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```

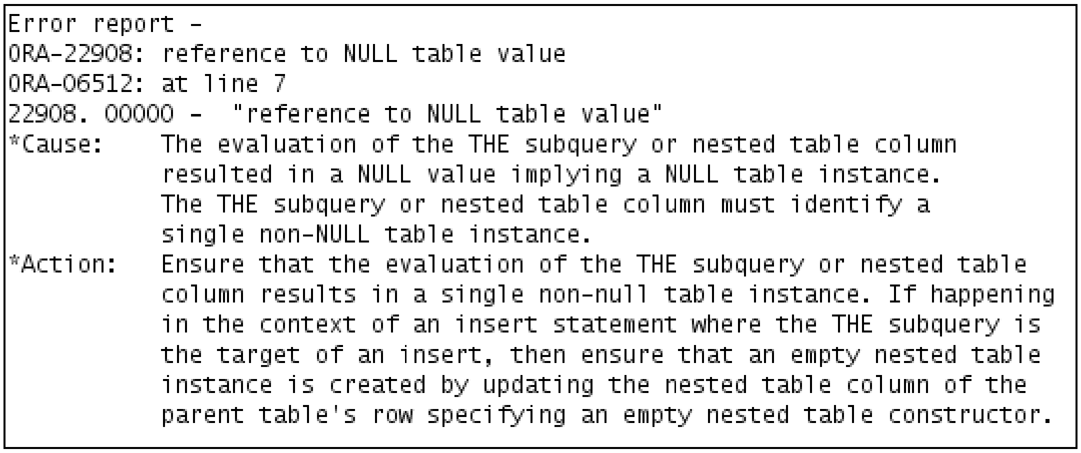


```
table PORDER dropped.
table PORDER created.
```

Note: The drop command will throw an error if the table pOrder does not exist.

2. The following code generates an error. Run task 2 of the lab_04.sql script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO TABLE (SELECT items
                    FROM   pOrder
                    WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```



```
Error report -
ORA-22908: reference to NULL table value
ORA-06512: at line 7
22908. 00000 -  "reference to NULL table value"
*Cause:    The evaluation of the THE subquery or nested table column
           resulted in a NULL value implying a NULL table instance.
           The THE subquery or nested table column must identify a
           single non-NULL table instance.
*Action:   Ensure that the evaluation of the THE subquery or nested table
           column results in a single non-null table instance. If happening
           in the context of an insert statement where the THE subquery is
           the target of an insert, then ensure that an empty nested table
           instance is created by updating the nested table column of the
           parent table's row specifying an empty nested table constructor.
```

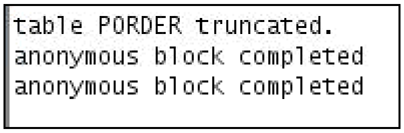
a. Why does the error occur?

The error "`ORA-22908: reference to NULL table value`" results from setting the table columns to NULL.

b. How can you fix the error?

You should always use a nested table's default constructor to initialize it:

```
TRUNCATE TABLE pOrder;

-- A better approach is to avoid setting the table
-- column to NULL, and instead, use a nested table's
-- default constructor to initialize
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE,
            typ_item_nst(typ_item(99, 129.00)));
END;
/

BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, null);
  -- Once the nested table is set to null, use the update
  -- update statement
  UPDATE pOrder
    SET items = typ_item_nst(typ_item(99, 129.00))
    WHERE ordid = 1000;
END;
/
```



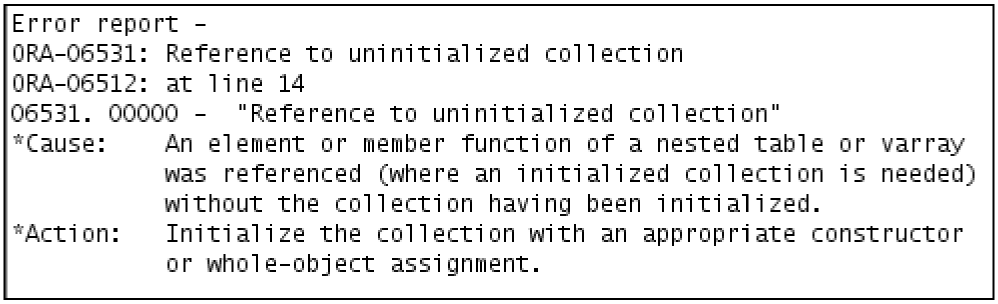
```
table PORDER truncated.
anonymous block completed
anonymous block completed
```

Alternatively, run the code from task 2 of `sol_04.sql`.

3. Examine the following code. This code produces an error. Which line causes the error, and how do you fix it?

   Note: You can run task 3 of the `lab_04.sql` script to view the error output.

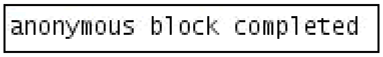
```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ;
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```



```
Error report -
ORA-06531: Reference to uninitialized collection
ORA-06512: at line 14
06531. 00000 -  "Reference to uninitialized collection"
*Cause:    An element or member function of a nested table or varray
           was referenced (where an initialized collection is needed)
           without the collection having been initialized.
*Action:   Initialize the collection with an appropriate constructor
           or whole-object assignment.
```

This causes an `ORA-06531: Reference to uninitialized collection` error. To fix it, initialize the `v_am` variable by using the same technique as the others:

```
DECLARE
  TYPE credit_card_typ
  IS VARRAY(100) OF VARCHAR2(30);

  v_mc   credit_card_typ := credit_card_typ();
  v_visa credit_card_typ := credit_card_typ();
  v_am   credit_card_typ := credit_card_typ();
  v_disc credit_card_typ := credit_card_typ();
  v_dc   credit_card_typ := credit_card_typ();

BEGIN
  v_mc.EXTEND;
  v_visa.EXTEND;
  v_am.EXTEND;
  v_disc.EXTEND;
  v_dc.EXTEND;
END;
/
```



```
anonymous block completed
```

Alternatively, run the code from task 3 of `sol_04.sql`.

Practices for Lesson 5: Using Collections

Chapter 5

Practices for Lesson 5: Overview

Lesson Overview

In this practice, you write a PL/SQL package to manipulate the collection.

Use the OE schema for this practice.

Practice 5-1: Using Collections

Overview

In this practice, you use PL/SQL code to manipulate the collection.

Assumptions

Practice 04 is completed.

Task

Implement a nested table column in the `CUSTOMERS` table and write PL/SQL code to manipulate the nested table.

Use the `OE` schema to complete the tasks.

1. Create a nested table to hold credit card information.
   a. Create an object type called `typ_cr_card`. It should have the following specification:

      ```
      card_type  VARCHAR2(25)
      card_num   NUMBER
      ```

   b. Create a nested table type called `typ_cr_card_nst`, which is a table of `typ_cr_card`.
   c. Add a column called `credit_cards` to the `CUSTOMERS` table. Make this column a nested table of type `typ_cr_card_nst`. You can use the following syntax:

      ```
      ALTER TABLE customers ADD
      (credit_cards typ_cr_card_nst)
         NESTED TABLE credit_cards STORE AS c_c_store_tab;
      ```

2. Create a PL/SQL package that manipulates the `credit_cards` column in the `CUSTOMERS` table.
   a. Open the `lab_05.sql` file. It contains the package specification and part of the package body. Examine task 2 of `lab_05.sql`.
   b. Complete the following code so that the package:
      - Inserts credit card information (the credit card name and number for a specific customer)
      - Displays credit card information in an unnested format

      ```
      CREATE OR REPLACE PACKAGE credit_card_pkg
      IS
        PROCEDURE update_card_info
          (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
      VARCHAR2);
        PROCEDURE display_card_info
          (p_cust_id NUMBER);
      END credit_card_pkg;  -- package spec
      /

      CREATE OR REPLACE PACKAGE BODY credit_card_pkg
      IS
      ```

```
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
 -- cards exist, add more

 -- fill in code here

    ELSE -- no cards for this customer, construct one

 -- fill in code here

    END IF;
  END update_card_info;


  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;

 -- fill in code here to display the nested table
 -- contents

  END display_card_info;
END credit_card_pkg;  -- package body
/
```

3. Test your package with the following statements and compare the output:

```
EXECUTE credit_card_pkg.display_card_info(120)

EXECUTE credit_card_pkg.update_card_info
    (120, 'Visa', 11111111)

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

EXECUTE credit_card_pkg.display_card_info(120)

EXECUTE credit_card_pkg.update_card_info
    (120, 'MC', 2323232323)

EXECUTE credit_card_pkg.update_card_info
    (120, 'DC', 4444444)

EXECUTE credit_card_pkg.display_card_info(120)
```

Alternatively, you can execute the code of task 3 from `lab_05.sql`.

4. Write a `SELECT` statement against the `credit_cards` column to unnest the data. Use the `TABLE` expression. Use SQL*Plus.
For example, if the `SELECT` statement returns:

```
SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;
```

```
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
--------------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111), TYP_CR_CARD('MC',
2323232323), TYP_CR_CARD('DC', 4444444))
```

Rewrite it using the `TABLE` expression so that the results look like the following:

```
-- Use the table expression so that the result is:
CUSTOMER_ID CUST_LAST_NAME  CARD_TYPE        CARD_NUM
----------- --------------- ------------- -----------
        120 Higgins         Visa             11111111
        120 Higgins         MC             2323232323
        120 Higgins         DC                4444444
```

Alternatively, you can execute the code of task 4 from `lab_05.sql`.

Solution 5-1: Using Collections

In this practice, you use PL/SQL code to manipulate the collection.

1. Create a nested table to hold credit card information.

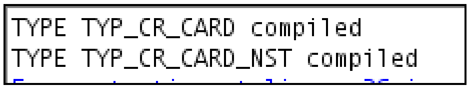
   a. Create an object type called `typ_cr_card`. It should have the following specification:

   ```
   card_type VARCHAR2(25)
   card_num  NUMBER
   ```

   ```
   CREATE TYPE typ_cr_card AS OBJECT  --create object
    (card_type  VARCHAR2(25),
     card_num   NUMBER);
   /
   ```

   b. Create a nested table type called `typ_cr_card_nst`, which is a table of `typ_cr_card`.

   ```
   CREATE TYPE typ_cr_card_nst -- define nested table type
     AS TABLE OF typ_cr_card;
   /
   ```



```
TYPE TYP_CR_CARD compiled
TYPE TYP_CR_CARD_NST compiled
```

   Add a column called `credit_cards` to the `CUSTOMERS` table.

   Make this column a nested table of type `typ_cr_card_nst`.

   You can use the following syntax:

   ```
   ALTER TABLE customers ADD
   credit_cards typ_cr_card_nst
      NESTED TABLE credit_cards STORE AS c_c_store_tab;
   ```

   Alternatively, you can run the solution of 1_a, 1_b, and 1_c from `sol_05.sql`.

2. Create a PL/SQL package that manipulates the `credit_cards` column in the `CUSTOMERS` table.

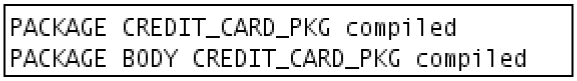
   a. Open the `lab_05.sql` file. It contains the package specification and part of the package body.

   b. Here is the complete code of the package that inserts the credit card information and displays the credit card information in an unnested format

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

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

   CREATE OR REPLACE PACKAGE BODY credit_card_pkg
   ```

```
IS

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


  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
          DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type || ' ');
```

```
        DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
      END LOOP;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
  END display_card_info;
END credit_card_pkg;  -- package body
/
```



```
PACKAGE CREDIT_CARD_PKG compiled
PACKAGE BODY CREDIT_CARD_PKG compiled
```

Alternatively, you can execute the solution of task 2 from `sol_05.sql`.

3. Test your package with the following statements and compare the output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info
    (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info
    (120, 'MC', 2323232323)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info
    (120, 'DC', 4444444)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
```

```
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
PL/SQL procedure successfully completed.
```

Alternatively, you can execute the code of task 3 from `lab_05.sql`.

4. Write a `SELECT` statement against the `credit_cards` column to unnest the data. Use the `TABLE` expression. Use SQL*Plus.

   For example, if the `SELECT` statement returns:

```
SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;
```

```
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
---------------------------------------------------------
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111), TYP_CR_CARD('MC',
2323232323), TYP_CR_CARD('DC', 4444444))
```

   Rewrite it using the `TABLE` expression so that the results look like the following:

```
-- Use the table expression so that the result is:
CUSTOMER_ID CUST_LAST_NAME  CARD_TYPE        CARD_NUM
----------- --------------- ------------- -----------
        120 Higgins         Visa             11111111
        120 Higgins         MC             2323232323
        120 Higgins         DC                4444444
```

```
SELECT c1.customer_id, c1.cust_last_name, c2.*
FROM   customers c1, TABLE(c1.credit_cards) c2
WHERE  customer_id = 120;
```

Alternatively, you can execute the code of task 4 from `lab_05.sql`.

Practices for Lesson 6: Manipulating Large Objects

Chapter 6

Practices for Lesson 6: Overview

Lesson Overview

This practice covers the following topics:

- Creating object types of the `CLOB` and `BLOB` data types
- Creating a table with the `LOB` data types as columns
- Using the `DBMS_LOB` package to populate and interact with the `LOB` data
- Setting up the environment for `LOB`s

Practice 6-1: Working with LOBs

Overview

In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS_LOB package to populate the table and manipulate the data.

Use the OE connection to complete this practice.

Task

1. Create a table called PERSONNEL. The table should contain the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
LAST_NAME	VARCHAR2	35
REVIEW	CLOB	N/A
PICTURE	BLOB	N/A

2. Insert two rows into the PERSONNEL table, one each for employee 2034 (whose last name is Allen) and employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide NULL as the value for the BLOB.
3. Examine and execute the /home/oracle/labs/labs/lab_06.sql script. The script creates a table named REVIEW_TABLE. This table contains the annual review information for each employee. The script also contains two statements to insert review details about two employees.
4. Update the PERSONNEL table.
   a. Populate the CLOB for the first row by using this subquery in an UPDATE statement:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2034;
```

   b. Populate the CLOB for the second row by using PL/SQL and the DBMS_LOB package. Use the following SELECT statement to provide a value for the LOB locator:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2035;
```

5. Create a procedure that adds a locator to a binary file in the PICTURE column of the PRODUCT_INFORMATION table. The binary file is a picture of the product. The image files are named after the product IDs. You must load an image file locator into all rows in the Printers category (CATEGORY_ID = 12) in the PRODUCT_INFORMATION table.
   a. Create a DIRECTORY object called PRODUCT_PIC that references the location of the binary. These files are available in the /home/oracle/Labs/DATA_FILES/PRODUCT_PIC folder.
   b. Add the image column to the PRODUCT_INFORMATION table.

c. Create a PL/SQL procedure called `load_product_image` that uses `DBMS_LOB.FILEEXISTS` to test whether the product picture file exists. If the file exists, set the `BFILE` locator for the file in the `PICTURE` column; otherwise, display a message that the file does not exist. Use the `DBMS_OUTPUT` package to report file size information about each image associated with the `PICTURE` column.

d. Invoke the procedure by passing the name of the `PRODUCT_PIC` directory object as a string literal parameter value.

e. Check the `LOB` space usage of the `PRODUCT_INFORMATION` table. Use the `/home/oracle/labs/labs/lab_06.sql` file to create the procedure and execute it.

Solution 6-1: Working with LOBs

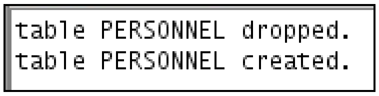
In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS_LOB package to populate the table and manipulate the data.

Use your OE connection.

1. Create a table called PERSONNEL. The table should contain the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
LAST_NAME	VARCHAR2	35
REVIEW	CLOB	N/A
PICTURE	BLOB	N/A

```
DROP TABLE personnel
/
CREATE TABLE personnel
(id NUMBER(6) constraint personnel_id_pk PRIMARY KEY,
 last_name VARCHAR2(35),
 review CLOB,
 picture BLOB);
```



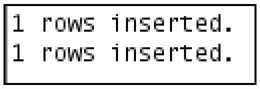
```
table PERSONNEL dropped.
table PERSONNEL created.
```

The drop command would return an error if the PERSONNEL table is not already created.

Alternatively, you can run the solution for task 1 from sol_06.sql.

2. Insert two rows into the PERSONNEL table, one each for employee 2034 (whose last name is Allen) and employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide NULL as the value for the BLOB.

```
INSERT INTO  personnel
VALUES (2034, 'Allen', empty_clob(), NULL);

INSERT INTO  personnel
VALUES (2035, 'Bond', empty_clob(), NULL);
```

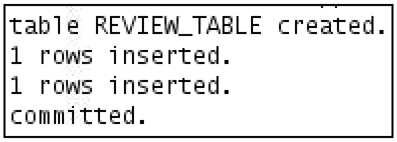


```
1 rows inserted.
1 rows inserted.
```

Alternatively, you can run the solution for task 2 from sol_06.sql.

3. Examine and execute task 3 located in the /home/oracle/labs/labs/lab_06.sql script. The script creates a table named REVIEW_TABLE. This table contains annual review information for each employee. The script also contains two statements to insert review details for two employees.

```
CREATE TABLE review_table (
 employee_id number,
 ann_review  VARCHAR2(2000));

INSERT INTO review_table
VALUES (2034,
       'Very good performance this year. '||
       'Recommended to increase salary by $500');

INSERT INTO review_table
VALUES (2035,
       'Excellent performance this year. '||
       'Recommended to increase salary by $1000');

COMMIT;
```



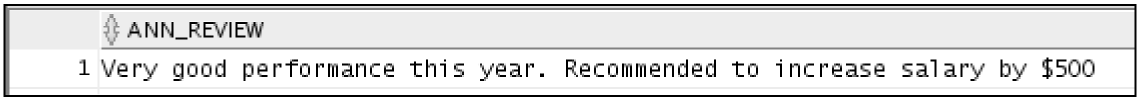
```
table REVIEW_TABLE created.
1 rows inserted.
1 rows inserted.
committed.
```

4. Update the PERSONNEL table.

   a. Populate the CLOB for the first row by using the following subquery in an UPDATE statement:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2034;
```

```
UPDATE personnel
 SET review = (SELECT ann_review
               FROM   review_table
               WHERE  employee_id = 2034)
 WHERE last_name = 'Allen';
```



```
     ANN_REVIEW
   1 Very good performance this year. Recommended to increase salary by $500
```

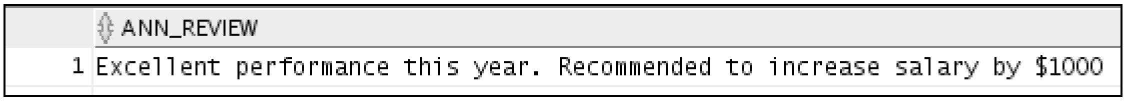
```
1 rows updated.
```

   b. Populate the CLOB for the second row by using PL/SQL and the DBMS_LOB package. Use the following SELECT statement to provide a value for the LOB locator:

```
SELECT ann_review
```

```
FROM   review_table
WHERE  employee_id = 2035;
```

```
UPDATE personnel
 SET review = (SELECT ann_review
              FROM   review_table
              WHERE  employee_id = 2035)
 WHERE last_name = 'Bond';
```



```
  ANN_REVIEW
1 Excellent performance this year. Recommended to increase salary by $1000
```

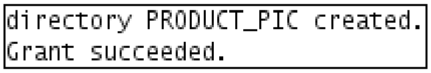
```
1 rows updated.
```

Alternatively, you can run the solution for task 4_a, 4_b from `sol_06.sql`.

5. Create a procedure that adds a locator to a binary file to the `PICTURE` column of the `PRODUCT_INFORMATION` table. The binary file is a picture of the product. The image files are named after the product IDs. You must load an image file locator into all rows in the Printers category (`CATEGORY_ID = 12`) in the `PRODUCT_INFORMATION` table.

   a. Create a `DIRECTORY` object called `PRODUCT_PIC` that references the location of the binary file. These files are available in the `/home/oracle/labs/DATA_FILES/PRODUCT_PIC` folder.

```
CREATE OR REPLACE DIRECTORY product_pic AS
'/home/oracle/labs/DATA_FILES/PRODUCT_PIC';
GRANT READ on DIRECTORY product_pic TO OE;
```



```
directory PRODUCT_PIC created.
Grant succeeded.
```

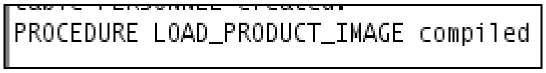
   b. Add the image column to the `PRODUCT_INFORMATION` table by using:
   `ALTER TABLE product_information ADD (picture BFILE);`

```
table PRODUCT_INFORMATION altered.
```

   c. Create a PL/SQL procedure called `load_product_imag`e that uses `DBMS_LOB.FILEEXISTS` to test whether the product picture file exists. If the file exists, set the BFILE locator for the file in the `PICTURE` column; otherwise, display a message that the file does not exist. Use the `DBMS_OUTPUT` package to report file size information for each image associated with the `PICTURE` column.
   (Alternatively, use the code snippet contained in task 5c of the `lab_06.sql` file.)

```
CREATE OR REPLACE PROCEDURE load_product_image
(p_dir IN VARCHAR2)
IS
  v_file          BFILE;
  v_filename      VARCHAR2(40);
  v_rec_number    NUMBER;
  v_file_exists   BOOLEAN;
  CURSOR product_csr IS
```

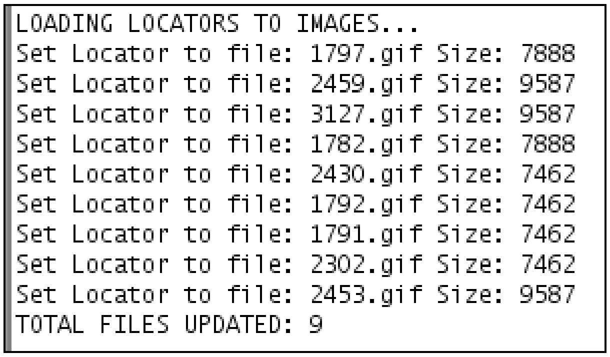
```
    SELECT product_id
    FROM product_information
    WHERE category_id = 12
    FOR UPDATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('LOADING LOCATORS TO IMAGES...');
  FOR rec IN product_csr
  LOOP
    v_filename := rec.product_id || '.gif';
    v_file := BFILENAME(p_dir, v_filename);
    v_file_exists := (DBMS_LOB.FILEEXISTS(v_file) = 1);
    IF v_file_exists THEN
     DBMS_LOB.FILEOPEN(v_file);
     UPDATE product_information
       SET picture = v_file
       WHERE CURRENT OF product_csr;
     DBMS_OUTPUT.PUT_LINE('Set Locator to file: '|| v_filename
       || ' Size: ' || DBMS_LOB.GETLENGTH(v_file));
     DBMS_LOB.FILECLOSE(v_file);
     v_rec_number := product_csr%ROWCOUNT;
    ELSE
     DBMS_OUTPUT.PUT_LINE('File ' || v_filename ||
      ' does not exist');
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('TOTAL FILES UPDATED: ' ||
                        v_rec_number);
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_LOB.FILECLOSE(v_file);
      DBMS_OUTPUT.PUT_LINE('Error: '|| to_char(SQLCODE) ||
        SQLERRM);
END load_product_image;
/
```



```
PROCEDURE LOAD_PRODUCT_IMAGE compiled
```

d. Invoke the procedure by passing the name of the `PRODUCT_PIC` directory object as a string literal parameter value.

```
SET SERVEROUTPUT ON
EXECUTE load_product_image('PRODUCT_PIC');
```



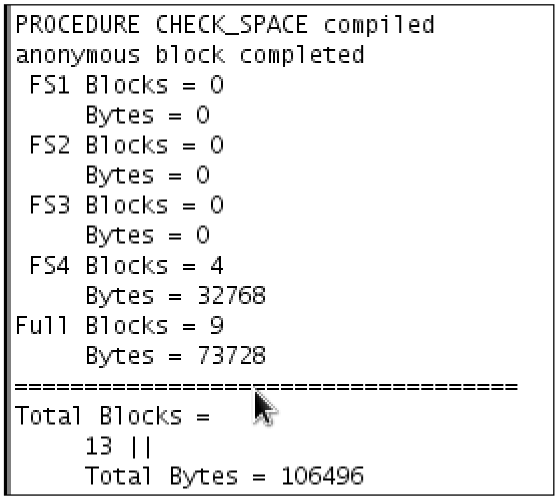
```
LOADING LOCATORS TO IMAGES...
Set Locator to file: 1797.gif Size: 7888
Set Locator to file: 2459.gif Size: 9587
Set Locator to file: 3127.gif Size: 9587
Set Locator to file: 1782.gif Size: 7888
Set Locator to file: 2430.gif Size: 7462
Set Locator to file: 1792.gif Size: 7462
Set Locator to file: 1791.gif Size: 7462
Set Locator to file: 2302.gif Size: 7462
Set Locator to file: 2453.gif Size: 9587
TOTAL FILES UPDATED: 9
```

Alternatively, you can run the code for the solutions 5_a, 5_b, 5_c, 5_d from `sol_06.sql`.

e. Check the `LOB` space usage of the `PRODUCT_INFORMATION` table. Use the `/home/oracle/labs/labs/lab_06.sql` file to create the procedure and execute it.

```
CREATE OR REPLACE PROCEDURE check_space
IS
  l_fs1_bytes NUMBER;
  l_fs2_bytes NUMBER;
  l_fs3_bytes NUMBER;
  l_fs4_bytes NUMBER;
  l_fs1_blocks NUMBER;
  l_fs2_blocks NUMBER;
  l_fs3_blocks NUMBER;
  l_fs4_blocks NUMBER;
  l_full_bytes NUMBER;
  l_full_blocks NUMBER;
  l_unformatted_bytes NUMBER;
  l_unformatted_blocks NUMBER;
BEGIN
  DBMS_SPACE.SPACE_USAGE(
    segment_owner      => 'OE',
    segment_name       => 'PRODUCT_INFORMATION',
    segment_type       => 'TABLE',
    fs1_bytes          => l_fs1_bytes,
    fs1_blocks         => l_fs1_blocks,
    fs2_bytes          => l_fs2_bytes,
    fs2_blocks         => l_fs2_blocks,
    fs3_bytes          => l_fs3_bytes,
    fs3_blocks         => l_fs3_blocks,
    fs4_bytes          => l_fs4_bytes,
    fs4_blocks         => l_fs4_blocks,
    full_bytes         => l_full_bytes,
    full_blocks        => l_full_blocks,
    unformatted_blocks => l_unformatted_blocks,
    unformatted_bytes  => l_unformatted_bytes
   );
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
     Bytes = '||l_fs1_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
     Bytes = '||l_fs2_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
     Bytes = '||l_fs3_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
     Bytes = '||l_fs4_bytes);
  DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
     Bytes = '||l_full_bytes);
  DBMS_OUTPUT.PUT_LINE('===============================');
  DBMS_OUTPUT.PUT_LINE('Total Blocks =
     '||to_char(l_fs1_blocks + l_fs2_blocks +
     l_fs3_blocks + l_fs4_blocks + l_full_blocks)||  ' ||
     Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
     + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/
```

```
set serveroutput on
execute check_space;
```



```
PROCEDURE CHECK_SPACE compiled
anonymous block completed
 FS1 Blocks = 0
     Bytes = 0
 FS2 Blocks = 0
     Bytes = 0
 FS3 Blocks = 0
     Bytes = 0
 FS4 Blocks = 4
     Bytes = 32768
Full Blocks = 9
     Bytes = 73728
====================================
Total Blocks =
     13 ||
     Total Bytes = 106496
```

Practices for Lesson 7: Using Advanced Interface Methods

Chapter 7

Practices for Lesson 7: Overview

Lesson Overview

In this practice, you write two PL/SQL programs: One program calls an external C routine, and the other calls a Java routine.

Practice 7-1: Using Advanced Interface Methods

Overview

In this practice, you will execute programs to interact with C routines and Java code.

Use the `OE` connection.

Task

An external C routine definition is created for you. The `.c` file is stored in the `/home/oracle/labs/labs` directory. This function returns the tax amount based on the total sales figure that is passed to the function as a parameter. The `.c` file is named `calc_tax.c`. The function is defined as:

```
#include <ctype.h>
int calc_tax(int n)
{
int tax;
tax = (n*8)/100;
return(tax);

}
```

1. A shared library file called `calc_tax.so` was created for you. Copy the file from the `/home/oracle/labs/labs` directory into your `/u01/app/oracle/product/12.1.0/dbhome_1/bin` directory.
2. Connect to the `sys` connection, and create the alias library object. Name the library object `c_code` and define its path as:

```
CREATE OR REPLACE LIBRARY c_code
AS '/u01/app/oracle/product/12.1.0/dbhome_1/bin/calc_tax.so';
/
```

3. Grant the execute privilege on the library to the `OE` user by executing the following command:

```
GRANT EXECUTE ON c_code TO OE;
```

4. Publish the external C routine. As the `OE` user, create a function named `call_c`. This function has one numeric parameter and it returns a binary integer. Identify the `AS LANGUAGE`, `LIBRARY`, and `NAME` clauses of the function.
5. Create a procedure to call the `call_c` function that was created in the previous step. Name this procedure `C_OUTPUT`. It has one numeric parameter. Include a `DBMS_OUTPUT.PUT_LINE` statement so that you can view the results returned from your C function.
6. Set the `SERVEROUTPUT ON` and execute the `C_OUTPUT` procedure.

Calling Java from PL/SQL

A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (4 digits followed by a space). The name of the `.class` file is `FormatCreditCardNo.class`. The method is defined as:

```
public class FormatCreditCardNo
{
public static final void formatCard(String[] cardno)
{
int count=0, space=0;
String oldcc=cardno[0];
String[] newcc= {""};
while (count<16)
{
newcc[0]+= oldcc.charAt(count);
space++;
if (space ==4)
{  newcc[0]+=" "; space=0;  }
count++;
}
cardno[0]=newcc [0];
}
}
```

7. Load the `.java` source file.
8. Publish the Java class method by defining a PL/SQL procedure named `CCFORMAT`. This procedure accepts one `IN OUT` parameter.
   Use the following definition for the `NAME` parameter:

   ```
   NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
   ```

9. Execute the Java class method. Define one SQL*Plus or Oracle SQL Developer variable, initialize it, and use the `EXECUTE` command to execute the `CCFORMAT` procedure.
   Your output should match the `PRINT` output as shown here:

```
VARIABLE x VARCHAR2(20)
EXECUTE :x := '1234567887654321'
EXECUTE ccformat(:x)
PRINT x
```

```
anonymous block completed
anonymous block completed
X
--------------------
1234 5678 8765 4321  1234 5678 1234 5678
```

Solution 7-1: Using Advanced Interface Methods

In this practice, you will execute programs to interact with C routines and Java code.

Use the `OE` connection.

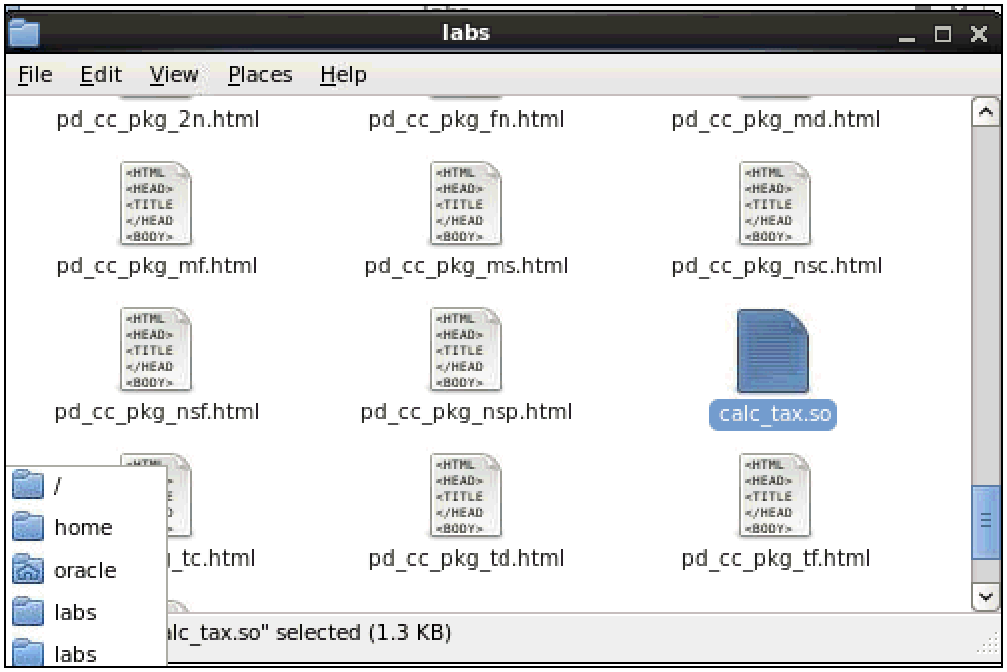
Using External C Routines

An external C routine definition is created for you. The `.c` file is stored in the `/home/oracle/labs/labs` directory. This function returns the tax amount based on the total sales figure that is passed to the function as a parameter. The `.c` file is named `calc_tax.c`. The function is defined as:

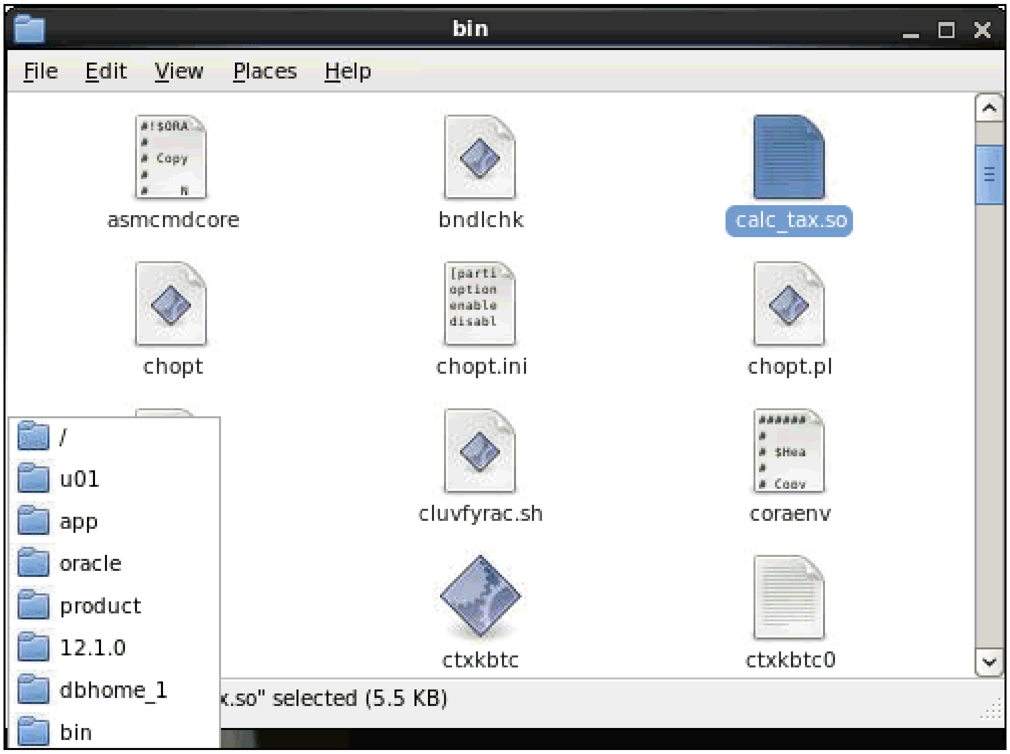
```
#include <ctype.h>
int calc_tax(int n)
{

int tax;
tax = (n*8)/100;
return(tax);

}
```

1. A shared library file called `calc_tax.so` was created for you. Copy the file from the `/home/oracle/labs/labs` directory into your `/u01/app/oracle/product/12.1.0/dbhome_1/bin` directory.

   Open the `/home/oracle/labs/labs` directory. Select `calc_tax.so`. Select Edit > Copy.

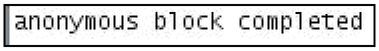


   Navigate to the `/u01/app/oracle/product/12.1.0/dbhome_1/bin` folder. Right-click the `BIN` directory and select Paste from the shortcut menu.



2. Connect to the `sys` connection, and create the alias library object. Name the library object `c_code` and define its path as:

```
-- Use SYS connection
CREATE OR REPLACE LIBRARY c_code
AS '/u01/app/oracle/product/12.1.0/dbhome_1/bin/calc_tax.so';
/
```

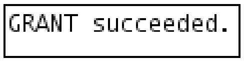


```
anonymous block completed
```

Alternatively, you can run the solution for task 2 from `sol_07.sql`.

3. Grant the execute privilege on the library to the `OE` user by executing the following command:

```
-- Use SYS connection
GRANT EXECUTE ON c_code TO OE;
```

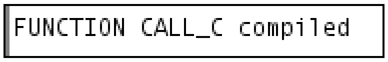


```
GRANT succeeded.
```

Alternatively, you can run the solution for task 3 from `sol_07.sql`.

4. Publish the external C routine. As the OE user, create a function named call_c. This function has one numeric parameter and it returns a binary integer. Identify the AS LANGUAGE, LIBRARY, and NAME clauses of the function.

```
-- Use OE Connection
CREATE OR REPLACE FUNCTION call_c
(x BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
LIBRARY sys.c_code
NAME "calc_tax";
/
```

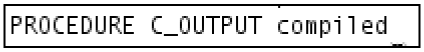


```
FUNCTION CALL_C compiled
```

Alternatively, you can run the solution for task 4 from sol_07.sql.

5. Create a procedure to call the call_c function created in the previous step. Name this procedure C_OUTPUT. It has one numeric parameter. Include a DBMS_OUTPUT.PUT_LINE statement so that you can view the results returned from your C function.

```
-- Use OE connection
CREATE OR REPLACE PROCEDURE c_output
  (p_in IN BINARY_INTEGER)
IS
  i BINARY_INTEGER;
BEGIN
  i := call_c(p_in);
  DBMS_OUTPUT.PUT_LINE('The total tax is: ' || i);
END c_output;
/
```

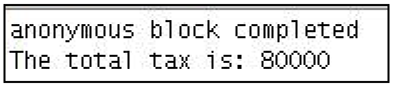


```
PROCEDURE C_OUTPUT compiled
```

Alternatively, you can run the solution for task 5 from sol_07.sql.

6. Set SERVEROUTPUT ON and execute the C_OUTPUT procedure.

```
SET SERVEROUTPUT ON

EXECUTE c_output(1000000)
```



```
anonymous block completed
The total tax is: 80000
```

Alternatively, you can run the solution for task 6 from sol_07.sql.

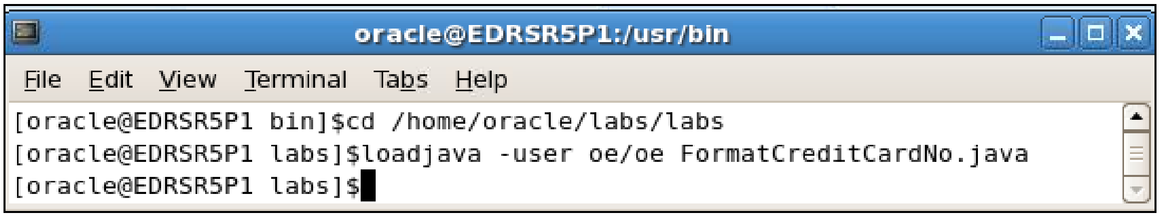
Calling Java from PL/SQL

A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (four digits followed by a space). The name of the `.class` file is `FormatCreditCardNo.class`. The method is defined as:

```
public class FormatCreditCardNo
{
public static final void formatCard(String[] cardno)
{
int count=0, space=0;
String oldcc=cardno[0];
String[] newcc= {""};
while (count<16)
{
newcc[0]+= oldcc.charAt(count);
space++;
if (space ==4)
{  newcc[0]+=" "; space=0;  }
count++;
}
cardno[0]=newcc [0];
}
}
```

7. Load the `.java` source file.

   You can execute the individual commands from the Linux terminal window.



```
[oracle@EDRSR5P1 bin]$cd /home/oracle/labs/labs
[oracle@EDRSR5P1 labs]$loadjava -user oe/oe FormatCreditCardNo.java
[oracle@EDRSR5P1 labs]$
```

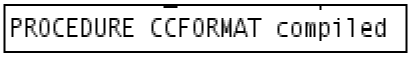
Alternatively, you can copy and paste the commands in the Linux terminal for task 7 from `sol_07.sql`.

8. Publish the Java class method by defining a PL/SQL procedure named `CCFORMAT`. This procedure accepts one `IN OUT` parameter.
   Use the following definition for the `NAME` parameter:

   Use the `OE` connection.

   ```
   NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
   ```

   ```
   CREATE OR REPLACE PROCEDURE ccformat
   (x IN OUT VARCHAR2)
   AS LANGUAGE JAVA
   NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
   /
   ```

   ```
   PROCEDURE CCFORMAT compiled
   ```

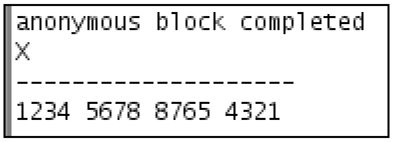
   Alternatively, you can run the solution for task 8 from `sol_07.sql`.

9. Execute the Java class method. Define one SQL*Plus or Oracle SQL Developer variable, initialize it, and use the `EXECUTE` command to execute the `CCFORMAT` procedure. Your output should match the `PRINT` output shown here:

   Use the `OE` connection.

   ```
     EXECUTE ccformat(:x);

   X
   --------------------
   1234 5678 8765 4321
   ```

   ```
   VARIABLE x VARCHAR2(20)
   EXECUTE :x := '1234567887654321'

   EXECUTE ccformat(:x)

   PRINT x
   ```

   ```
   anonymous block completed
   X
   --------------------
   1234 5678 8765 4321
   ```

   Alternatively, you can run the solution for task 9 from `sol_07.sql`.

Practices for Lesson 8: Performance and Tuning

Chapter 8

Practices for Lesson 8: Overview

Lesson Overview

In this practice, you measure and examine performance and tuning, and you tune some of the code that you created for the OE application.

- Break a previously built subroutine into smaller executable sections.
- Pass collections into subroutines.
- Add error handling for BULK INSERT.

Practice 8-1: Performance and Tuning

Overview

In this practice, you will tune a PL/SQL code and include bulk binds to improve performance.

Task

Writing Better Code

1. Open the `lab_08.sql` file and examine the package given in task 1. The package body is shown here:

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

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

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
      WHERE customer_id = p_cust_id;

  IF v_card_info.EXISTS(1) THEN  -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type,
                                      p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id;
    ELSE   -- no cards for this customer yet, construct one
```

```
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
            (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id;
    END IF;
  END update_card_info;

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

This code needs to be improved. The following issues exist in the code:

- The local variables use the `INTEGER` data type.
- The same `SELECT` statement is run in the two procedures.
- The same `IF v_card_info.EXISTS(1) THEN` statement is in the two procedures.

Using Efficient Data Types

2. To improve the code, make the following modifications:
   a. Change the local `INTEGER` variables to use a more efficient data type.
   b. Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2);
   PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/
```

   c. Have the function return `TRUE` if the customer has credit cards. The function should return `FALSE` if the customer does not have credit cards. Pass an uninitialized nested table into the function. The function places the credit card information into this uninitialized parameter.

3. Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info –
        (120, 'AM EX', 55555555555)

EXECUTE credit_card_pkg.display_card_info(120)
```

4. You must modify the `UPDATE_CARD_INFO` procedure to return information (by using the `RETURNING` clause) about the credit cards being updated. Assume that this information will be used by another application developer in your team, who is writing a graphical reporting utility on customer credit cards.
   a. Open the `lab_08.sql` file. It contains the code as modified in step 2.
   b. Modify the code to use the `RETURNING` clause to find information about the rows that are affected by the `UPDATE` statements.
   c. You can test your modified code with the following procedure (contained in task 4_c of `lab_08.sql`):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
  v_card_info typ_cr_card_nst;
BEGIN
  credit_card_pkg.update_card_info
    (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
```

```
/
```

d. Test your code with the following statements that are set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)

SELECT credit_cards FROM customers WHERE customer_id = 125;
```

Collecting Exception Information

5. Now you test exception handling with the `SAVE EXCEPTIONS` clause.

a. Run the statement from task 5_a of the `lab_08.sql` file to create a test table:

```
CREATE TABLE card_table
(accepted_cards VARCHAR2(50) NOT NULL);
```

b. Open the `lab_08.sql` file and run task 5_b:

```
DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
   'insert into card_table (accepted_cards) values ( :the_card)'
    USING v_cards(j);
END;
/
```

c. Note the output:____

d. Open the `lab_08.sql` file and run task 5_d:

```
    DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
  bulk_errors EXCEPTION;
  PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
   'insert into card_table (accepted_cards) values ( :the_card)'
    USING v_cards(j);
 EXCEPTION
  WHEN  bulk_errors THEN
    FOR j IN 1..sql%bulk_exceptions.count
  LOOP
    Dbms_Output.Put_Line (
      TO_CHAR( sql%bulk_exceptions(j).error_index ) || ':
      ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );
  END LOOP;
END;
/
```

e. Note the output:____

f. Why is the output different?

Timing Performance of `SIMPLE_INTEGER` and `PLS_INTEGER`

6. Now you compare the performance between the `PLS_INTEGER` and `SIMPLE_INTEGER` data types with native compilation:

   a. Run task 6_a from the `lab_08.sql` file to create a testing procedure that contains conditional compilation:

```
CREATE OR REPLACE PROCEDURE p
IS
  t0       NUMBER :=0;
  t1       NUMBER :=0;

 $IF $$Simple $THEN
  SUBTYPE My_Integer_t IS                     SIMPLE_INTEGER;
  My_Integer_t_Name CONSTANT VARCHAR2(30) := 'SIMPLE_INTEGER';
 $ELSE
  SUBTYPE My_Integer_t IS                     PLS_INTEGER;
  My_Integer_t_Name CONSTANT VARCHAR2(30) := 'PLS_INTEGER';
 $END

 v00  My_Integer_t := 0;     v01  My_Integer_t := 0;
 v02  My_Integer_t := 0;     v03  My_Integer_t := 0;
 v04  My_Integer_t := 0;     v05  My_Integer_t := 0;

 two      CONSTANT My_Integer_t := 2;
 lmt      CONSTANT My_Integer_t := 100000000;
```

```
BEGIN
  t0 := DBMS_UTILITY.GET_CPU_TIME();
  WHILE v01 < lmt LOOP
    v00 := v00 + Two;
    v01 := v01 + Two;
    v02 := v02 + Two;
    v03 := v03 + Two;
    v04 := v04 + Two;
    v05 := v05 + Two;
  END LOOP;

  IF v01 <> lmt OR v01 IS NULL THEN
    RAISE Program_Error;
  END IF;

  t1 := DBMS_UTILITY.GET_CPU_TIME();
  DBMS_OUTPUT.PUT_LINE(
    RPAD(LOWER($$PLSQL_Code_Type), 15)||
    RPAD(LOWER(My_Integer_t_Name), 15)||
    TO_CHAR((t1-t0), '9999')||' centiseconds');
END p;
```

b. Open the `lab_08.sql` file and run task 6_b:

```
ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = NATIVE PLSQL_CCFlags = 'simple:true'
REUSE SETTINGS;

EXECUTE p()

ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = native PLSQL_CCFlags = 'simple:false'
REUSE SETTINGS;

EXECUTE p()
```

c. Note the output:____

d. Explain the output.

Solution 8-1: Performance and Tuning

In this practice, you will tune a PL/SQL code and include bulk binds to improve performance.

Writing Better Code

1. Open the `lab_08.sql` file and examine the package (the package body is as follows) in task 1:

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

  PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS


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


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

This code needs to be improved. The following issues exist in the code:

- The local variables use the `INTEGER` data type.
- The same `SELECT` statement is run in the two procedures.
- The same `IF v_card_info.EXISTS(1) THEN` statement is in the two procedures.

Using Efficient Data Types

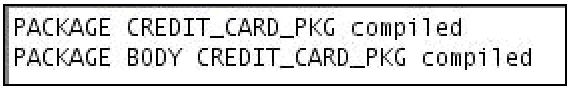
2. To improve the code, make the following modifications:
   a. Change the local `INTEGER` variables to use a more efficient data type.
   b. Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS

  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2);
   PROCEDURE display_card_info
    (p_cust_id NUMBER);

END credit_card_pkg;  -- package spec
/
```

   c. Have the function return `TRUE` if the customer has credit cards. The function should return `FALSE` if the customer does not have credit cards. Pass an uninitialized nested table into the function. The function places the credit card information into this uninitialized parameter.

```
-- note: If you did not complete lesson 5 practice, you will
need
-- to run solution scripts for tasks 1_a, 1_b, 1_c from
sol_05.sql
-- in order to have the supporting structures in place.

CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2);
  PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
```

```
IS

  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN
  IS
    v_card_info_exists BOOLEAN;
  BEGIN
    SELECT credit_cards
      INTO p_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF p_card_info.EXISTS(1) THEN
      v_card_info_exists := TRUE;
    ELSE
      v_card_info_exists := FALSE;
    END IF;
    RETURN v_card_info_exists;
  END cust_card_info;

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
-- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id;
    ELSE   -- no cards for this customer yet, construct one
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
            (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id;
    END IF;
  END update_card_info;
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
          DBMS_OUTPUT.PUT('Card Type: ' ||
            v_card_info(idx).card_type || ' ');
        DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
             v_card_info(idx).card_num );
      END LOOP;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
  END display_card_info;
END credit_card_pkg;  -- package body
/
```

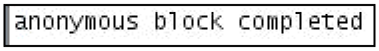


```
PACKAGE CREDIT_CARD_PKG compiled
PACKAGE BODY CREDIT_CARD_PKG compiled
```

Alternatively, run the code from task 2_c of `sol_08.sql`.

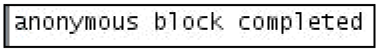
3. Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info –
        (120, 'AM EX', 55555555555)
```

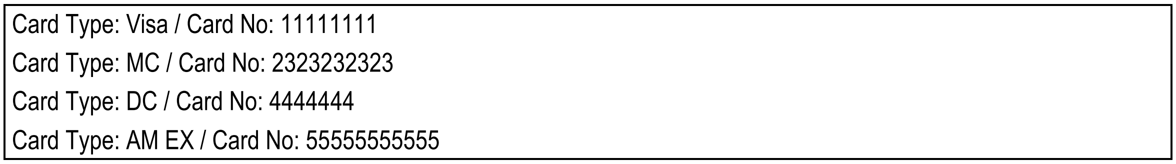


```
anonymous block completed
```

```
EXECUTE credit_card_pkg.display_card_info(120)
```



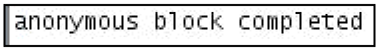
```
anonymous block completed
```



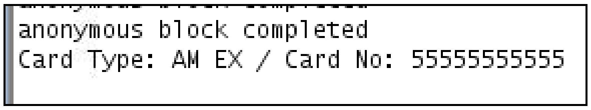
```
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
Card Type: AM EX / Card No: 55555555555
```

Note: If you did not complete Practice 5, your results will be:

```
EXECUTE credit_card_pkg.display_card_info(120)
```



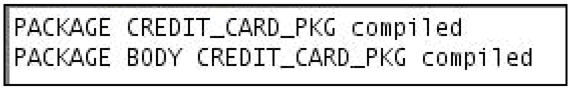
```
anonymous block completed
```



```
anonymous block completed
Card Type: AM EX / Card No: 55555555555
```

4. You must modify the `UPDATE_CARD_INFO` procedure to return information (by using the `RETURNING` clause) about the credit cards being updated. Assume that this information will be used by another application developer on your team, who is writing a graphical reporting utility on customer credit cards.
   a. Open the `lab_08.sql` file. It contains the code in task 4_a as modified in step 2.
   b. Modify the code to use the `RETURNING` clause to find information about the rows that are affected by the `UPDATE` statements.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst);
  PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN
  IS
    v_card_info_exists BOOLEAN;
  BEGIN
    SELECT credit_cards
      INTO p_card_info
      FROM customers
      WHERE customer_id = p_cust_id;
    IF p_card_info.EXISTS(1) THEN
      v_card_info_exists := TRUE;
    ELSE
      v_card_info_exists := FALSE;
    END IF;
    RETURN v_card_info_exists;
  END cust_card_info;

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN

    IF cust_card_info(p_cust_id, v_card_info) THEN
  -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET  credit_cards = v_card_info
        WHERE customer_id = p_cust_id
        RETURNING credit_cards INTO o_card_info;
    ELSE   -- no cards for this customer yet, construct one
      UPDATE customers
        SET  credit_cards = typ_cr_card_nst
            (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id
        RETURNING credit_cards INTO o_card_info;
    END IF;
  END update_card_info;

  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
          DBMS_OUTPUT.PUT('Card Type: ' ||
            v_card_info(idx).card_type || ' ');
        DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
            v_card_info(idx).card_num );
      END LOOP;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
  END display_card_info;
END credit_card_pkg;  -- package body
/
```

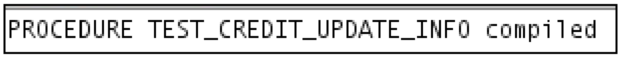


```
PACKAGE CREDIT_CARD_PKG compiled
PACKAGE BODY CREDIT_CARD_PKG compiled
```

Alternatively, run the code from task 4_b of `sol_08.sql`.

c. You can test your modified code with the following procedure (contained in task 4_c of `lab_08.sql`):

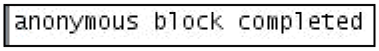
```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
  v_card_info typ_cr_card_nst;
BEGIN
  credit_card_pkg.update_card_info
    (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```



```
PROCEDURE TEST_CREDIT_UPDATE_INFO compiled
```

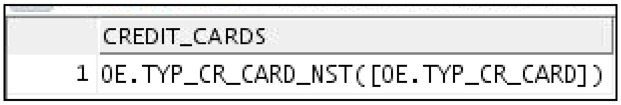
d. Test your code with the following statements that are set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
```



```
anonymous block completed
```

```
SELECT credit_cards FROM customers WHERE customer_id = 125;
```



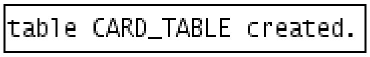
```
     CREDIT_CARDS
   1 OE.TYP_CR_CARD_NST([OE.TYP_CR_CARD])
```

Collecting Exception Information

5. Now you test exception handling with the `SAVE EXCEPTIONS` clause.

   a. Run task 5_a of the `lab_08.sql` file to create a test table:

```
CREATE TABLE card_table
(accepted_cards VARCHAR2(50) NOT NULL);
```

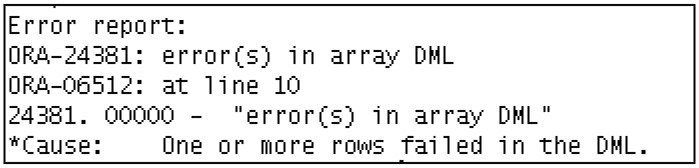


```
table CARD_TABLE created.
```

   b. Open the `lab_08.sql` file and run task 5_b:

```
DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
   'insert into card_table (accepted_cards) values
   ( :the_card)'
    USING v_cards(j);
/
END;
/
```

   c. Note the output:



```
Error report:
ORA-24381: error(s) in array DML
ORA-06512: at line 10
24381. 00000 -  "error(s) in array DML"
*Cause:    One or more rows failed in the DML.
```

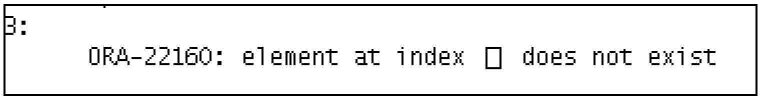
   This returns an "Error in Array DML (at line 10)," which is not very informative. The cause of this error: One or more rows failed in the DML.

   d. Open the `lab_08.sql` file and run task 5_d:

```
DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
  bulk_errors EXCEPTION;
  PRAGMA exception_init (bulk_errors, -24381 );
```

```
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
   'insert into card_table (accepted_cards) values (
    :the_card)'
    USING v_cards(j);
 EXCEPTION
  WHEN  bulk_errors THEN
    FOR j IN 1..sql%bulk_exceptions.count
  LOOP
    Dbms_Output.Put_Line (
      TO_CHAR( sql%bulk_exceptions(j).error_index ) || ':
      ' || SQLERRM(-sql%bulk_exceptions(j).error_code) );
  END LOOP;
END;
/
```

e. Note the output:



```
3:
     ORA-22160: element at index □ does not exist
```

f. Why is the output different?

The PL/SQL block raises the exception 22160 when it encounters an array element that was deleted. The exception is handled and the block is completed successfully.

Timing Performance of `SIMPLE_INTEGER` and `PLS_INTEGER`

6. Now you compare the performance between the `PLS_INTEGER` and `SIMPLE_INTEGER` data types with native compilation:

a. Run task 6_a of `lab_08.sql` to create a testing procedure that contains conditional compilation:

```
CREATE OR REPLACE PROCEDURE p
IS
  t0       NUMBER :=0;
  t1       NUMBER :=0;

 $IF $$Simple $THEN
  SUBTYPE My_Integer_t IS                     SIMPLE_INTEGER;
  My_Integer_t_Name CONSTANT VARCHAR2(30) := 'SIMPLE_INTEGER';
 $ELSE
  SUBTYPE My_Integer_t IS                     PLS_INTEGER;
  My_Integer_t_Name CONSTANT VARCHAR2(30) := 'PLS_INTEGER';
 $END

 v00  My_Integer_t := 0;     v01  My_Integer_t := 0;
 v02  My_Integer_t := 0;     v03  My_Integer_t := 0;
 v04  My_Integer_t := 0;     v05  My_Integer_t := 0;

 two      CONSTANT My_Integer_t := 2;
 lmt      CONSTANT My_Integer_t := 100000000;

BEGIN
  t0 := DBMS_UTILITY.GET_CPU_TIME();
  WHILE v01 < lmt LOOP
    v00 := v00 + Two;
    v01 := v01 + Two;
    v02 := v02 + Two;
    v03 := v03 + Two;
    v04 := v04 + Two;
    v05 := v05 + Two;
  END LOOP;

  IF v01 <> lmt OR v01 IS NULL THEN
    RAISE Program_Error;
  END IF;

  t1 := DBMS_UTILITY.GET_CPU_TIME();
  DBMS_OUTPUT.PUT_LINE(
    RPAD(LOWER($$PLSQL_Code_Type), 15)||
    RPAD(LOWER(My_Integer_t_Name), 15)||
    TO_CHAR((t1-t0), '9999')||' centiseconds');
END p;
/
```

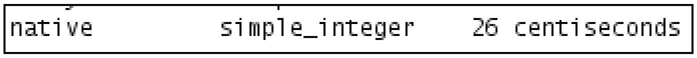
b. Open the `lab_08.sql` file and run task 6_b:

```
ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = NATIVE PLSQL_CCFlags = 'simple:true'
REUSE SETTINGS;

EXECUTE p()

ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = native PLSQL_CCFlags = 'simple:false'
```

```
REUSE SETTINGS;

EXECUTE p()
```

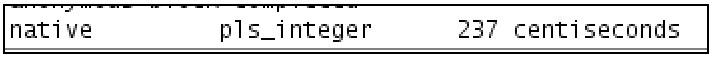
c. Note the output:

First run:



```
native        simple_integer    26 centiseconds
```

Second run:



```
native        pls_integer      237 centiseconds
```

d. Explain the output.

`SIMPLE_INTEGER` runs much faster in this scenario. If you can use the `SIMPLE_INTEGER` data type, it can improve performance.

Practices for Lesson 9: Improving Performance with Caching

Chapter 9

Practices for Lesson 9: Overview

Lesson Overview

In this practice, you implement SQL query result caching and PL/SQL result function caching. You run scripts to measure the cache memory values, manipulate queries and functions to turn caching on and off, and then examine cache statistics.

Practice 9-1: Improving Performance with Caching

Overview

In this practice, you examine the Explain Plan for a query, add the `RESULT_CACHE` hint to the query, and reexamine the Explain Plan results. You also execute some code and modify the code so that PL/SQL result caching is turned on.

Use the `OE` connection to complete this practice.

Task

Examining SQL and PL/SQL Result Caching

1. Use SQL Developer to connect to the `OE` schema. Examine the Explain Plan for the following query, which is found in the `lab_09.sql` file. To view the Explain Plan, click the Execute Explain Plan button on the toolbar in the Code Editor window.

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

2. Add the `RESULT_CACHE` hint to the query and reexamine the Explain Plan results.

```
SELECT /*+ result_cache */
       count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

   Examine the Explain Plan results, compared to the previous results.

3. The following code is used to generate a list of warehouse names for pick lists in applications. The `WAREHOUSES` table is fairly stable and is not modified often.

   Click the Run Script button to compile this code: (You can use the `lab_09.sql` file.)

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);
/
```

```
CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
  v_wh_names list_typ;
BEGIN
  SELECT warehouse_name
  BULK COLLECT INTO v_wh_names
  FROM   warehouses;
  RETURN v_wh_names;
  END get_warehouse_names;
```

4. Because the function is called frequently, and because the content of the data returned does not change frequently, this code is a good candidate for PL/SQL result caching.

   Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

Solution 9-1: Improving Performance with Caching

In this practice, you examine the Explain Plan for a query, add the `RESULT_CACHE` hint to the query, and reexamine the Explain Plan results. You also execute some code and modify the code so that PL/SQL result caching is turned on.

Use the `OE` connection.

Examining SQL and PL/SQL Result Caching

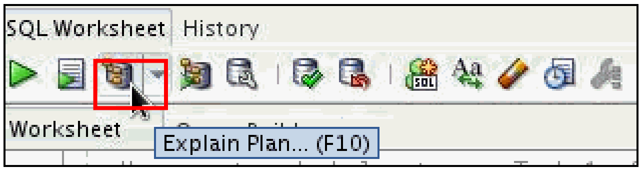
1. Use SQL Developer to connect to the `OE` schema. Examine the Explain Plan for the following query, which is found in the `lab_09.sql` file. To view the Explain Plan, click the Execute Explain Plan button on the toolbar in the Code Editor window.

   In Oracle SQL Developer, open the `lab_09.sql` file:

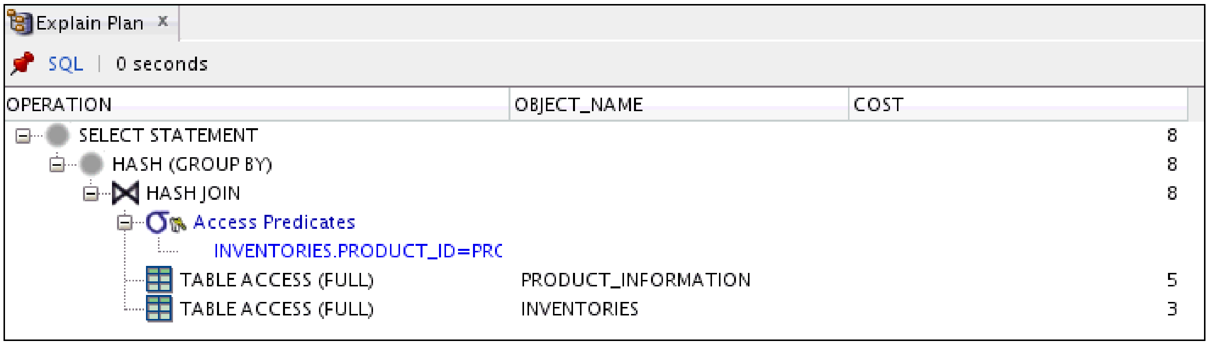
```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

   Select the `OE` connection.

   Click the Execute Explain Plan button on the toolbar and observe the results in the lower region:



   Results: SQL caching is not enabled and not visible in the Explain Plan.



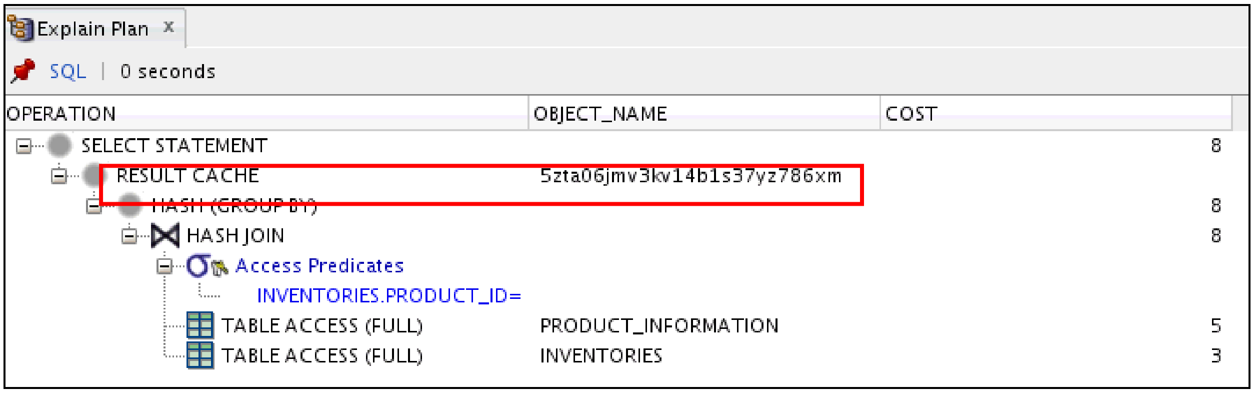
```
OPERATION                                  OBJECT_NAME              COST
SELECT STATEMENT                                                       8
  HASH (GROUP BY)                                                      8
    HASH JOIN                                                          8
      Access Predicates
        INVENTORIES.PRODUCT_ID=PRC
      TABLE ACCESS (FULL)                  PRODUCT_INFORMATION         5
      TABLE ACCESS (FULL)                  INVENTORIES                 3
```

2. Add the `RESULT_CACHE` hint to the query and reexamine the Explain Plan results.

```
SELECT /*+ result_cache */
       count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
FROM inventories natural join product_information
GROUP BY product_id, product_name;
```

Examine the Explain Plan results, compared to the previous Results.

Click the Execute Explain Plan button on the toolbar again, and compare the results in the lower region with the previous results:



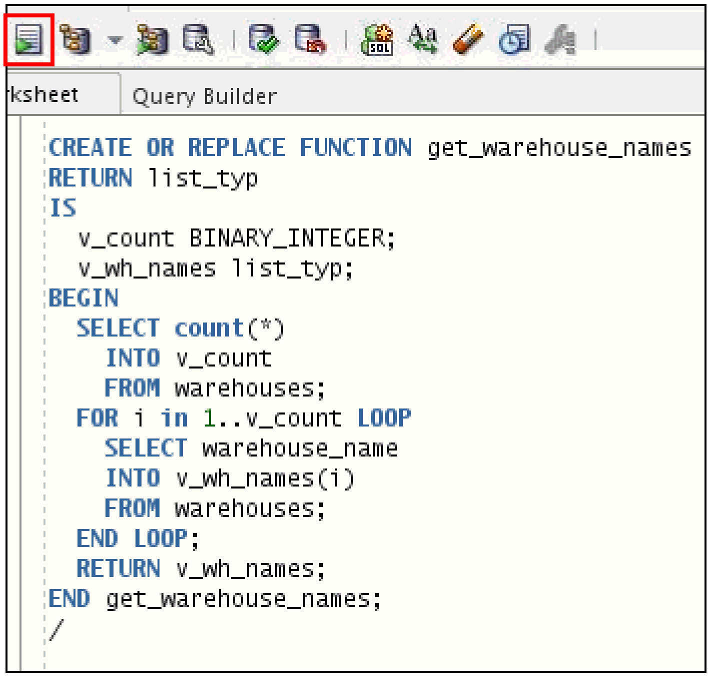
OPERATION	OBJECT_NAME	COST
SELECT STATEMENT		8
RESULT CACHE	5zta06jmv3kv14b1s37yz786xm	
HASH (GROUP BY)		8
HASH JOIN		8
Access Predicates		
INVENTORIES.PRODUCT_ID=		
TABLE ACCESS (FULL)	PRODUCT_INFORMATION	5
TABLE ACCESS (FULL)	INVENTORIES	3

Results: Note that result caching is used in the Explain Plan.

3. The following code is used to generate a list of warehouse names for pick lists in applications. The `WAREHOUSES` table is fairly stable and is not modified often.

   Click the Run Script button to compile this code: (You can use the `lab_09.sql` file.)

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);
/

CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
  v_count BINARY_INTEGER;
  v_wh_names list_typ;
BEGIN
  SELECT count(*)
    INTO v_count
    FROM warehouses;
  FOR i in 1..v_count LOOP
    SELECT warehouse_name
    INTO v_wh_names(i)
    FROM warehouses;
  END LOOP;
  RETURN v_wh_names;
END get_warehouse_names;
```



```
CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
  v_count BINARY_INTEGER;
  v_wh_names list_typ;
BEGIN
  SELECT count(*)
    INTO v_count
    FROM warehouses;
  FOR i in 1..v_count LOOP
    SELECT warehouse_name
    INTO v_wh_names(i)
    FROM warehouses;
  END LOOP;
  RETURN v_wh_names;
END get_warehouse_names;
/
```

Open `lab_09.sql`. Click the Run Script button. You have compiled the function without PL/SQL result caching.

4. Because the function is called frequently, and because the content of the data returned does not frequently change, this code is a good candidate for PL/SQL result caching.

   Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

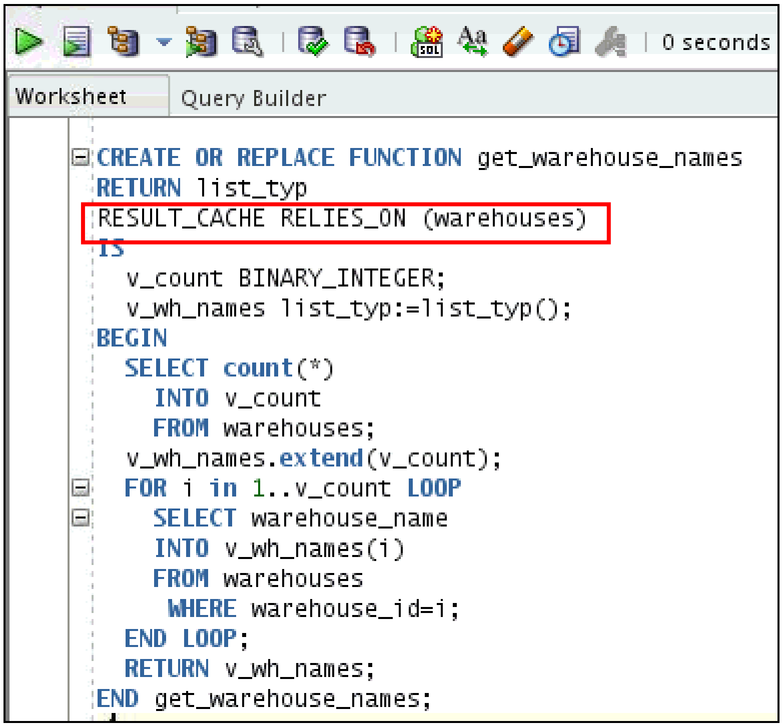
   Insert the following line after RETURN list_typ:

   `RESULT_CACHE RELIES_ON (warehouses)`

```
CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
RESULT_CACHE RELIES_ON (warehouses)
IS
  v_count BINARY_INTEGER;
  v_wh_names list_typ:=list_typ();
BEGIN
  SELECT count(*)
    INTO v_count
    FROM warehouses;
  v_wh_names.extend(v_count);
  FOR i in 1..v_count LOOP
    SELECT warehouse_name
    INTO v_wh_names(i)
```

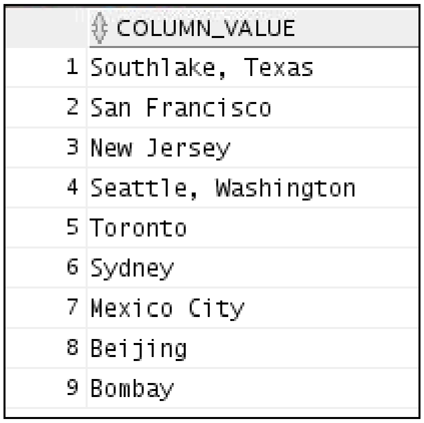
```
    FROM warehouses
    WHERE warehouse_id=i;
  END LOOP;
  RETURN v_wh_names;
END get_warehouse_names;
```

Click the Run Script button to recompile the code.



```
CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
RESULT_CACHE RELIES_ON (warehouses)
IS
  v_count BINARY_INTEGER;
  v_wh_names list_typ:=list_typ();
BEGIN
  SELECT count(*)
    INTO v_count
    FROM warehouses;
  v_wh_names.extend(v_count);
  FOR i in 1..v_count LOOP
    SELECT warehouse_name
    INTO v_wh_names(i)
    FROM warehouses
     WHERE warehouse_id=i;
  END LOOP;
  RETURN v_wh_names;
END get_warehouse_names;
```

```
SELECT * FROM TABLE(get_warehouse_names)
/
```



	COLUMN_VALUE
1	Southlake, Texas
2	San Francisco
3	New Jersey
4	Seattle, Washington
5	Toronto
6	Sydney
7	Mexico City
8	Beijing
9	Bombay

Alternatively, you can execute the solution for task 4 from sol_09.sql.

Practices for Lesson 10: Analyzing PL/SQL Code

Chapter 10

Practices for Lesson 10: Overview

Lesson Overview

In this practice, you will perform the following:

- Find coding information
- Use PL/Scope
- Use `DBMS_METADATA`

Practice 10-1: Analyzing PL/SQL Code

Overview

In this practice, you use PL/SQL and Oracle SQL Developer to analyze your code.

Use your OE connection.

Task

Finding Coding Information

1. Create the QUERY_CODE_PKG package to search your source code.

   Use the OE connection.

   a. Run task 1_a of the lab_10.sql script to create the QUERY_CODE_PKG package.

   b. Run the ENCAP_COMPLIANCE procedure to see which of your programs reference tables or views. (**Note:** Your results might differ slightly.)

   c. Run the FIND_TEXT_IN_CODE procedure to find all references to 'ORDERS'. (**Note:** Your results might differ slightly.)

   d. Use the SQL Developer Reports feature to find the same results for step C shown above.

Using PL/Scope

2. In the following steps, you use PL/Scope.

   Use the OE connection.

   a. Enable your session to collect identifiers.

   b. Recompile your CREDIT_CARD_PKG code.

   c. Verify that your PLSCOPE_SETTING is set correctly by issuing the following statement:

```
SELECT PLSCOPE_SETTINGS
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';
```

   d. Execute the following statement to create a hierarchical report on the identifier information about the CREDIT_CARD_PKG code. You can run task 2_d of the lab_10.sql script file.

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

3. Use DBMS_METADATA to find the metadata for the ORDER_ITEMS table.

   Use the OE connection.

   a. Create the GET_TABLE_MD function. You can run task 3_a of the lab_10.sql script.

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

   b. Issue the following statements to view the metadata generated from the GET_TABLE_MD function:

      You can run task 3_b of the lab_10.sql script.

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

   c. Generate an XML representation of the ORDER_ITEMS table by using the DBMS_METADATA.GET_XML function. Spool the output to a file named ORDER_ITEMS_XML.txt in the /home/oracle/labs folder.

   d. Verify that the ORDER_ITEMS_XML.txt file was created in the /home/oracle/labs folder.

Solution 10-1: Analyzing PL/SQL Code

In this practice, you use PL/SQL and Oracle SQL Developer to analyze your code.

Use your `OE` connection.

Finding Coding Information

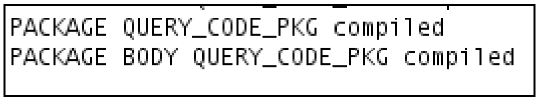
1. Create the `QUERY_CODE_PKG` package to search your source code.

   Use the `OE` connection.

   a. Run task 1_a of the `lab_10.sql` script to create the `QUERY_CODE_PKG` package.

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

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

  PROCEDURE encap_compliance IS
    SUBTYPE qualified_name_t IS VARCHAR2 (200);
    TYPE refby_rt IS RECORD (NAME qualified_name_t,
         referenced_by qualified_name_t );
    TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
    refby_aa refby_aat;
  BEGIN
```

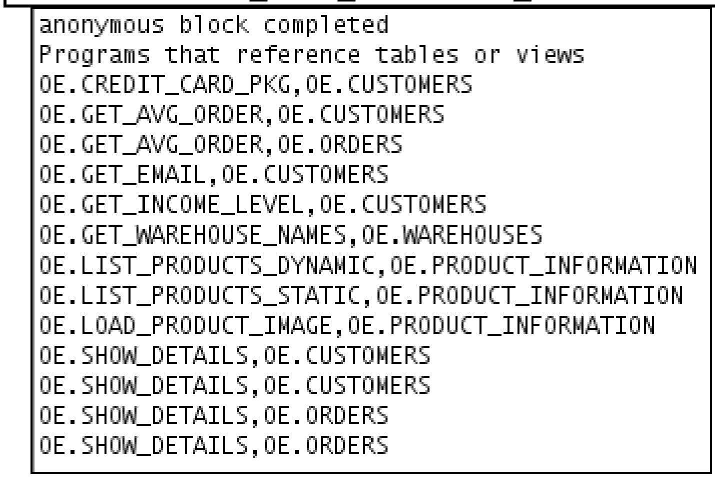
```
    SELECT owner || '.' || NAME refs_table
          , referenced_owner || '.' || referenced_name
          AS table_referenced
    BULK COLLECT INTO refby_aa
      FROM all_dependencies
      WHERE owner = USER
      AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
                   'PROCEDURE', 'FUNCTION')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



```
PACKAGE QUERY_CODE_PKG compiled
PACKAGE BODY QUERY_CODE_PKG compiled
```

b. Run the `ENCAP_COMPLIANCE` procedure to see which of your programs reference tables or views. (**Note:** Your results might differ slightly.)

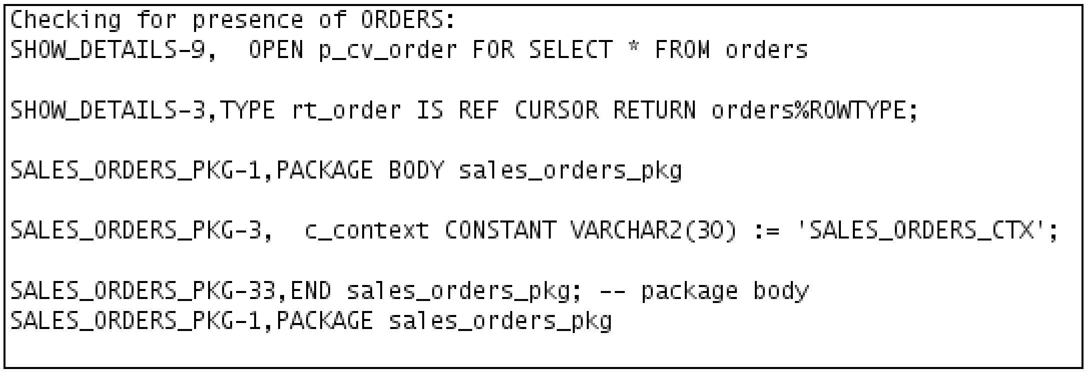
```
SET SERVEROUTPUT ON
EXECUTE query_code_pkg.encap_compliance
```



```
anonymous block completed
Programs that reference tables or views
OE.CREDIT_CARD_PKG,OE.CUSTOMERS
OE.GET_AVG_ORDER,OE.CUSTOMERS
OE.GET_AVG_ORDER,OE.ORDERS
OE.GET_EMAIL,OE.CUSTOMERS
OE.GET_INCOME_LEVEL,OE.CUSTOMERS
OE.GET_WAREHOUSE_NAMES,OE.WAREHOUSES
OE.LIST_PRODUCTS_DYNAMIC,OE.PRODUCT_INFORMATION
OE.LIST_PRODUCTS_STATIC,OE.PRODUCT_INFORMATION
OE.LOAD_PRODUCT_IMAGE,OE.PRODUCT_INFORMATION
OE.SHOW_DETAILS,OE.CUSTOMERS
OE.SHOW_DETAILS,OE.CUSTOMERS
OE.SHOW_DETAILS,OE.ORDERS
OE.SHOW_DETAILS,OE.ORDERS
```

c. Run the `FIND_TEXT_IN_CODE` procedure to find all references to `'ORDERS'`. (**Note:** Your results might differ slightly.)

```
SET SERVEROUTPUT ON
EXECUTE query_code_pkg.find_text_in_code('ORDERS')
```

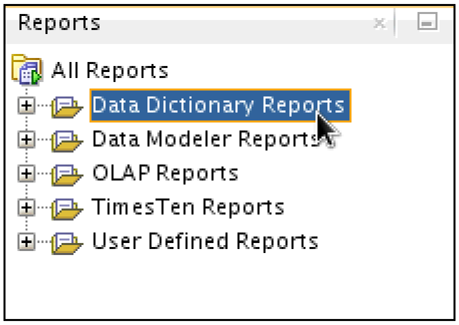


```
Checking for presence of ORDERS:
SHOW_DETAILS-9,  OPEN p_cv_order FOR SELECT * FROM orders

SHOW_DETAILS-3,TYPE rt_order IS REF CURSOR RETURN orders%ROWTYPE;

SALES_ORDERS_PKG-1,PACKAGE BODY sales_orders_pkg

SALES_ORDERS_PKG-3,  c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';

SALES_ORDERS_PKG-33,END sales_orders_pkg; -- package body
SALES_ORDERS_PKG-1,PACKAGE sales_orders_pkg
```

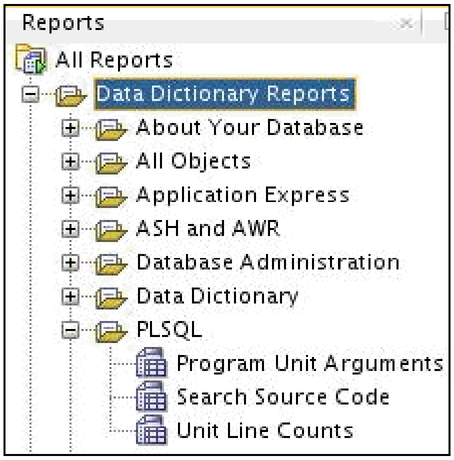
Alternatively, you can execute the solutions for tasks 1_b and 1_c from `sol_10.sql`.

d. Use the Oracle SQL Developer Reports feature to find the same results obtained in step c.

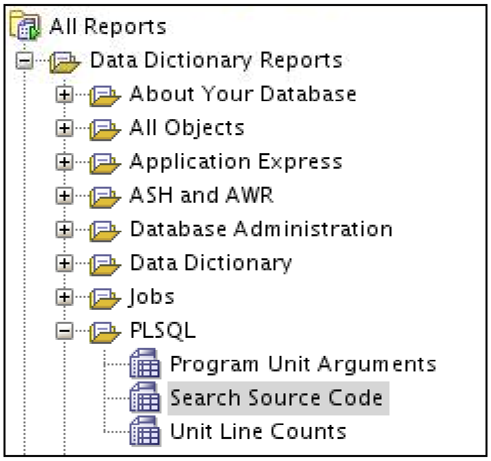
Navigate to the Reports tabbed page in Oracle SQL Developer.

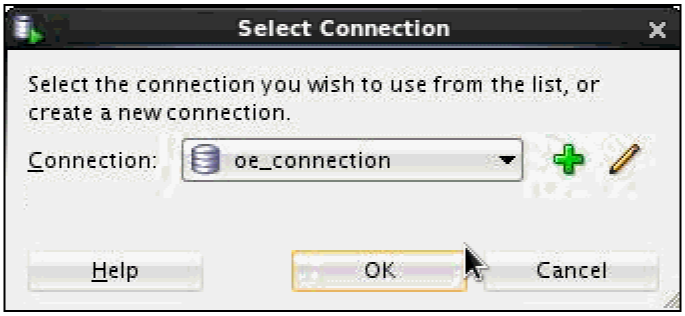


Expand the Data Dictionary Reports node and expand the PL/SQL node.



Select Search Source Code, and then select your `OE` connection and click OK.

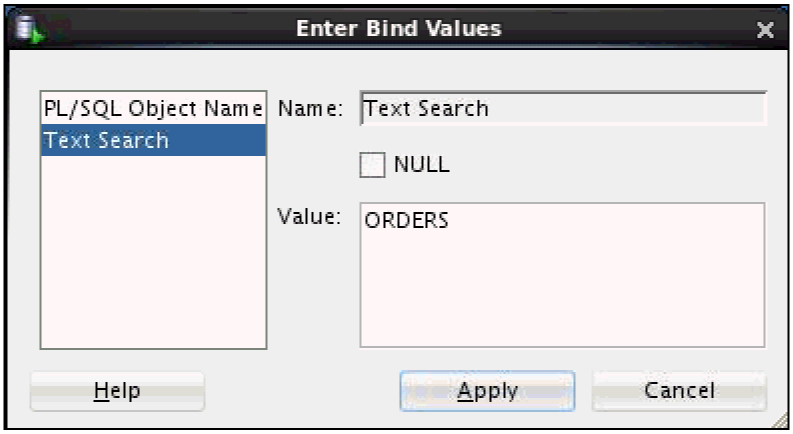




Select the connection you wish to use from the list, or create a new connection.

Connection: oe_connection

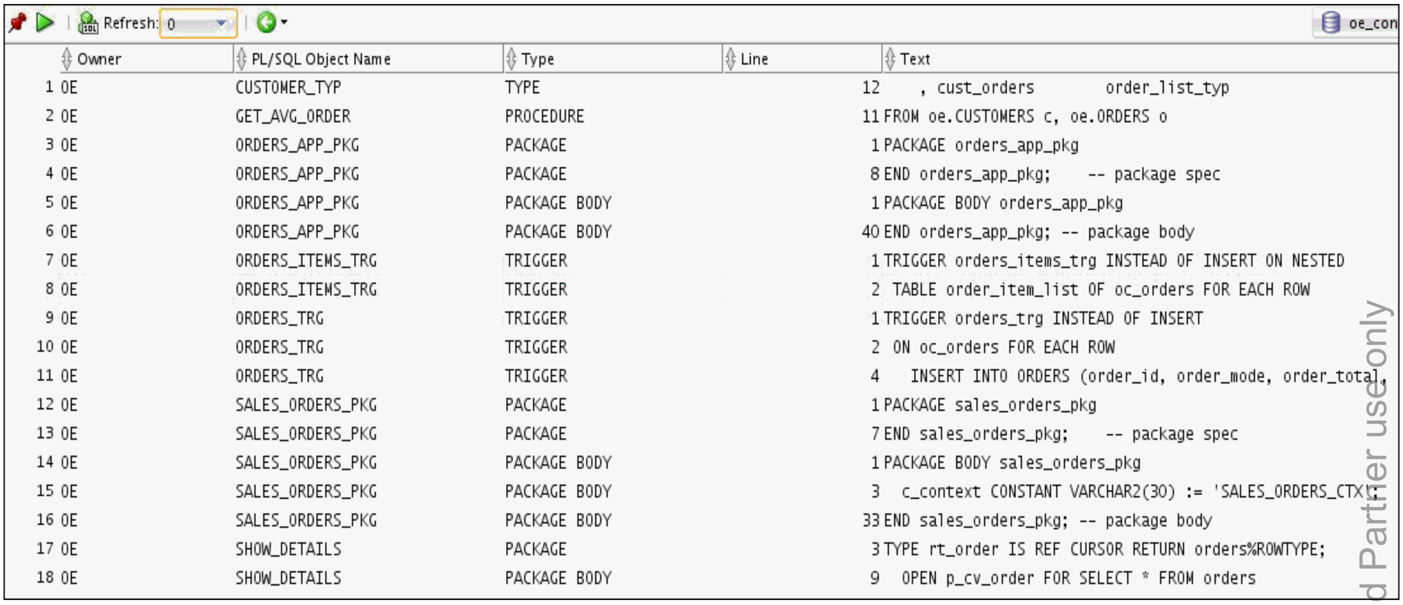
Select Text search and enter `ORDERS` in the Value: field. Click the Apply button.



Name: Text Search

□ NULL

Value: ORDERS



	Owner	PL/SQL Object Name	Type	Line	Text
1	OE	CUSTOMER_TYP	TYPE	12	, cust_orders order_list_typ
2	OE	GET_AVG_ORDER	PROCEDURE	11	FROM oe.CUSTOMERS c, oe.ORDERS o
3	OE	ORDERS_APP_PKG	PACKAGE	1	PACKAGE orders_app_pkg
4	OE	ORDERS_APP_PKG	PACKAGE	8	END orders_app_pkg; -- package spec
5	OE	ORDERS_APP_PKG	PACKAGE BODY	1	PACKAGE BODY orders_app_pkg
6	OE	ORDERS_APP_PKG	PACKAGE BODY	40	END orders_app_pkg; -- package body
7	OE	ORDERS_ITEMS_TRG	TRIGGER	1	TRIGGER orders_items_trg INSTEAD OF INSERT ON NESTED
8	OE	ORDERS_ITEMS_TRG	TRIGGER	2	TABLE order_item_list OF oc_orders FOR EACH ROW
9	OE	ORDERS_TRG	TRIGGER	1	TRIGGER orders_trg INSTEAD OF INSERT
10	OE	ORDERS_TRG	TRIGGER	2	ON oc_orders FOR EACH ROW
11	OE	ORDERS_TRG	TRIGGER	4	INSERT INTO ORDERS (order_id, order_mode, order_total,
12	OE	SALES_ORDERS_PKG	PACKAGE	1	PACKAGE sales_orders_pkg
13	OE	SALES_ORDERS_PKG	PACKAGE	7	END sales_orders_pkg; -- package spec
14	OE	SALES_ORDERS_PKG	PACKAGE BODY	1	PACKAGE BODY sales_orders_pkg
15	OE	SALES_ORDERS_PKG	PACKAGE BODY	3	c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
16	OE	SALES_ORDERS_PKG	PACKAGE BODY	33	END sales_orders_pkg; -- package body
17	OE	SHOW_DETAILS	PACKAGE	3	TYPE rt_order IS REF CURSOR RETURN orders%ROWTYPE;
18	OE	SHOW_DETAILS	PACKAGE BODY	9	OPEN p_cv_order FOR SELECT * FROM orders

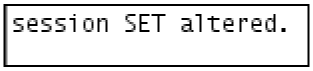
Note: The report content might vary based on the objects created by using your schema.

Using PL/Scope

Use the OE connection.

2. In the following steps, you use PL/Scope.
   a. Enable your session to collect identifiers.

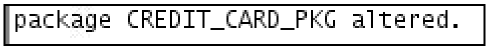
```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
```



```
session SET altered.
```

   b. Recompile your CREDIT_CARD_PKG code.

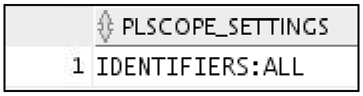
```
ALTER PACKAGE credit_card_pkg COMPILE;
```



```
package CREDIT_CARD_PKG altered.
```

   c. Verify that your PLSCOPE_SETTING is set correctly by issuing the following statement:

```
SELECT PLSCOPE_SETTINGS
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';
```

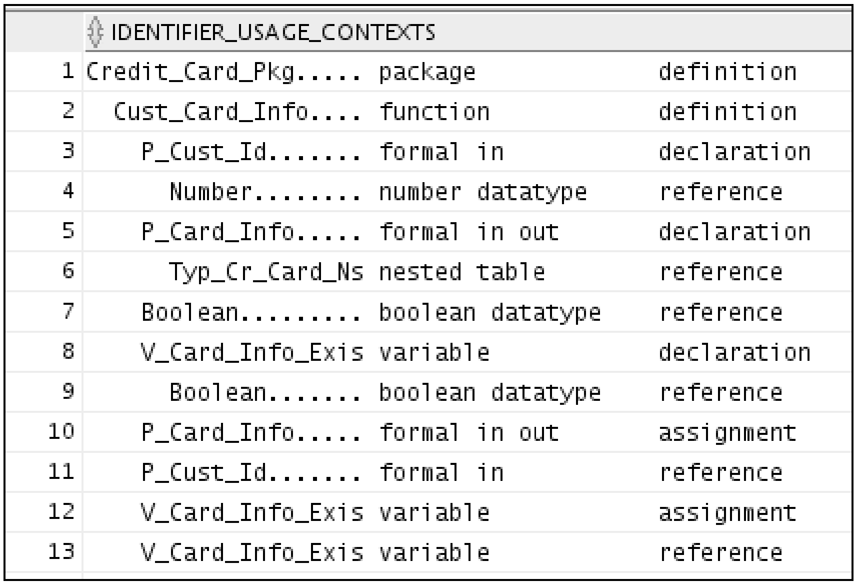


	PLSCOPE_SETTINGS
1	IDENTIFIERS:ALL

Alternatively, you can execute the solutions for tasks 2_a, 2_b, and 2_c from `sol_10.sql`.

   d. Execute the following statement to create a hierarchical report on the identifier information about the CREDIT_CARD_PKG code. You can run task 2_d of the lab_10.sql script file.

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



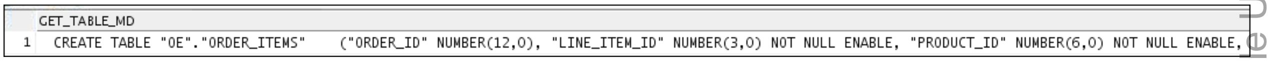
```
   IDENTIFIER_USAGE_CONTEXTS
 1 Credit_Card_Pkg..... package             definition
 2   Cust_Card_Info.... function            definition
 3     P_Cust_Id....... formal in           declaration
 4       Number........ number datatype     reference
 5     P_Card_Info..... formal in out       declaration
 6       Typ_Cr_Card_Ns nested table        reference
 7     Boolean......... boolean datatype    reference
 8     V_Card_Info_Exis variable            declaration
 9       Boolean....... boolean datatype    reference
10     P_Card_Info..... formal in out       assignment
11     P_Cust_Id....... formal in           reference
12     V_Card_Info_Exis variable            assignment
13     V_Card_Info_Exis variable            reference
```

3. Use `DBMS_METADATA` to find the metadata for the `ORDER_ITEMS` table.
   Use the `OE` connection.
   a. Create the `GET_TABLE_MD` function. You can run task 3_a of the `lab_10.sql` script.

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

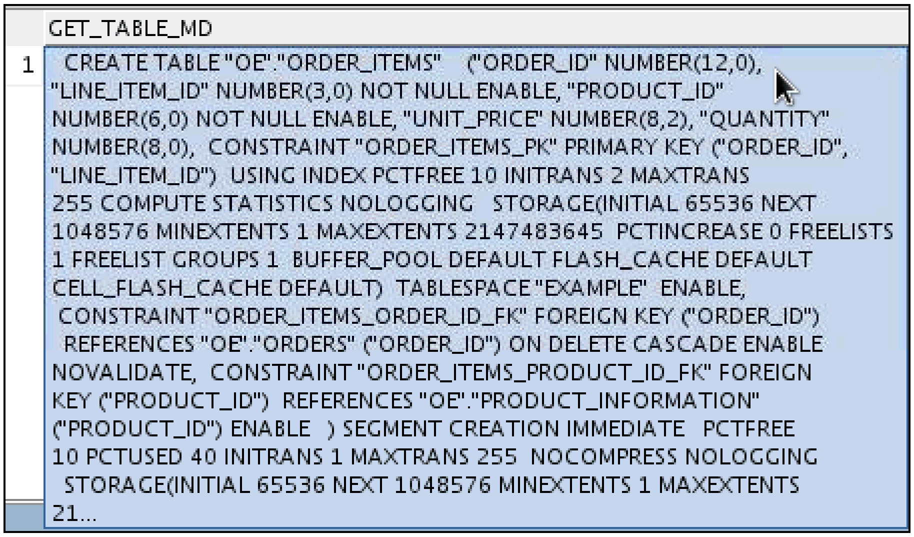
   b. Issue the following statements to view the metadata generated from the `GET_TABLE_MD` function:

```
set pagesize 0
set long  1000000

SELECT get_table_md FROM dual;
```



```
   GET_TABLE_MD
1  CREATE TABLE "OE"."ORDER_ITEMS"    ("ORDER_ID" NUMBER(12,0), "LINE_ITEM_ID" NUMBER(3,0) NOT NULL ENABLE, "PRODUCT_ID" NUMBER(6,0) NOT NULL ENABLE,
```

Move the cursor over the `get_table_md` column to see the detailed view of the record.

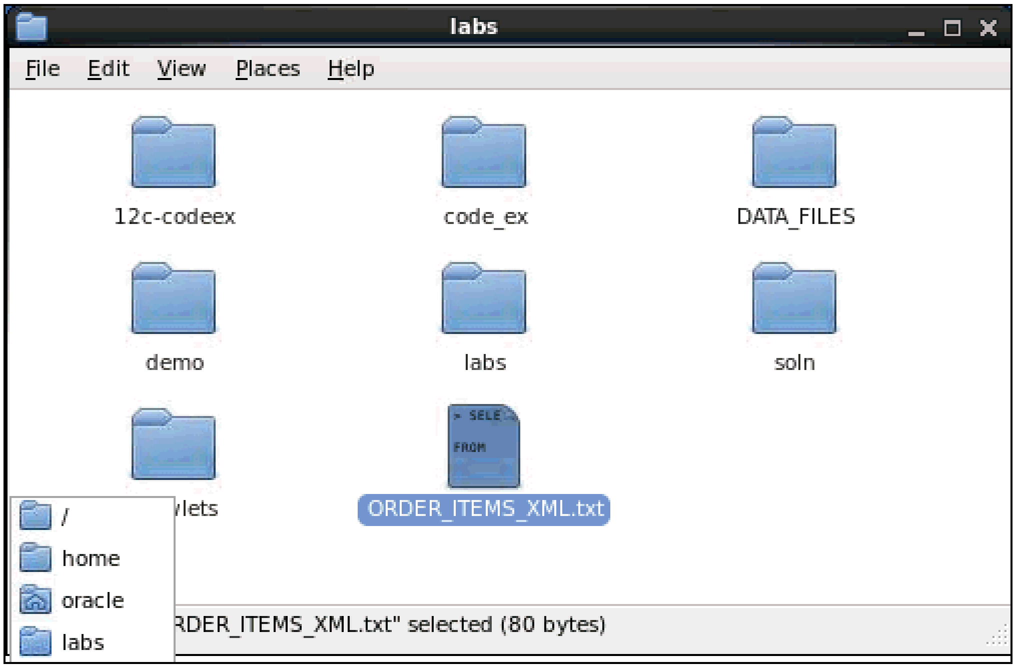


```
GET_TABLE_MD
1  CREATE TABLE "OE"."ORDER_ITEMS"   ("ORDER_ID" NUMBER(12,0),
"LINE_ITEM_ID" NUMBER(3,0) NOT NULL ENABLE, "PRODUCT_ID"
NUMBER(6,0) NOT NULL ENABLE, "UNIT_PRICE" NUMBER(8,2), "QUANTITY"
NUMBER(8,0),  CONSTRAINT "ORDER_ITEMS_PK" PRIMARY KEY ("ORDER_ID",
"LINE_ITEM_ID")  USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS
255 COMPUTE STATISTICS NOLOGGING   STORAGE(INITIAL 65536 NEXT
1048576 MINEXTENTS 1 MAXEXTENTS 2147483645  PCTINCREASE 0 FREELISTS
1 FREELIST GROUPS 1  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT
CELL_FLASH_CACHE DEFAULT)  TABLESPACE "EXAMPLE"  ENABLE,
 CONSTRAINT "ORDER_ITEMS_ORDER_ID_FK" FOREIGN KEY ("ORDER_ID")
  REFERENCES "OE"."ORDERS" ("ORDER_ID") ON DELETE CASCADE ENABLE
NOVALIDATE,  CONSTRAINT "ORDER_ITEMS_PRODUCT_ID_FK" FOREIGN
KEY ("PRODUCT_ID")  REFERENCES "OE"."PRODUCT_INFORMATION"
("PRODUCT_ID") ENABLE   ) SEGMENT CREATION IMMEDIATE   PCTFREE
10 PCTUSED 40 INITRANS 1 MAXTRANS 255  NOCOMPRESS NOLOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
21...
```

c. Generate an XML representation of the `ORDER_ITEMS` table by using the `DBMS_METADATA.GET_XML` function. Spool the output to a file named `ORDER_ITEMS_XML.txt` in the `/home/oracle/labs` folder.

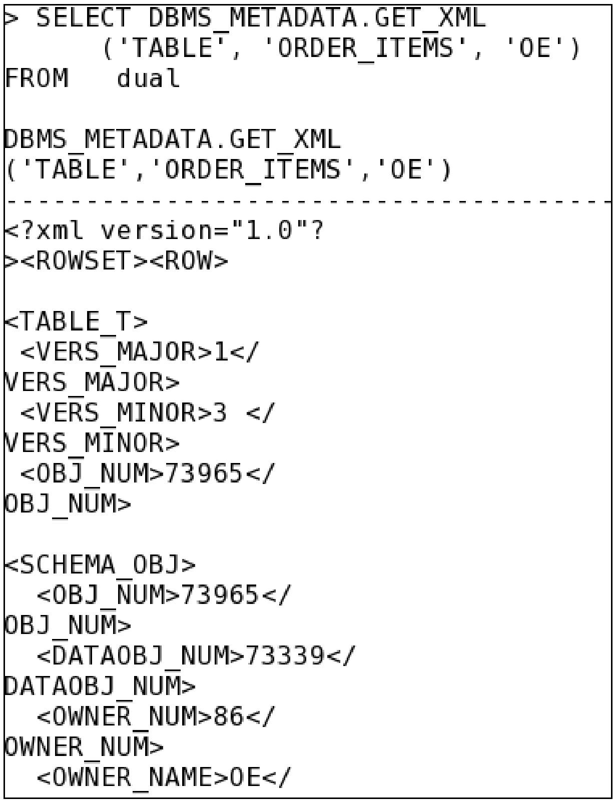
```
SPOOL /home/oracle/labs/ORDER_ITEMS_XML.txt

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

SPOOL OFF
```

d. Verify that the `ORDER_ITEMS_XML.txt` file was created in the `/home/oracle/labs` folder.



RDER_ITEMS_XML.txt" selected (80 bytes)

Open the file to verify its contents:



```
> SELECT DBMS_METADATA.GET_XML
      ('TABLE', 'ORDER_ITEMS', 'OE')
FROM   dual

DBMS_METADATA.GET_XML
('TABLE','ORDER_ITEMS','OE')
--------------------------------------
<?xml version="1.0"?
><ROWSET><ROW>

<TABLE_T>
 <VERS_MAJOR>1</
VERS_MAJOR>
 <VERS_MINOR>3 </
VERS_MINOR>
 <OBJ_NUM>73965</
OBJ_NUM>

<SCHEMA_OBJ>
  <OBJ_NUM>73965</
OBJ_NUM>
  <DATAOBJ_NUM>73339</
DATAOBJ_NUM>
  <OWNER_NUM>86</
OWNER_NUM>
  <OWNER_NAME>OE</
```

Practices for Lesson 11: Profiling and Tracing PL/SQL Code

Chapter 11

Practices for Lesson 11: Overview

Lesson Overview

In this practice, you write code to profile components in your application.

Practice 11-1: Profiling and Tracing PL/SQL Code

Overview

In this practice, you generate profiler data and analyze it.

Task

Use your OE connection.

1. Generate profiling data for your CREDIT_CARD_PKG.
   a. Re-create CREDIT_CARD_PKG by running the /home/oracle/labs/lab_11.sql script.

      You must identify the location of the profiler files. Create a DIRECTORY object to identify this information, and grant the necessary privileges. Use the SYS connection.
   b. Use DBMS_HPROF.START_PROFILING to start the profiler for your session.
   c. Run your CREDIT_CARD_PKG.UPDATE_CARD_INFO with the following data.

      ```
      credit_card_pkg.update_card_info
           (154, 'Discover', '123456789');
      ```
   d. Use DBMS_HPROF.STOP_PROFILING to stop the profiler.
2. Run the dbmshptab.sql script, located in the /u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin folder, to set up the profiler tables.
3. Use DBMS_HPROF.ANALYZE to analyze the raw data and write the information to the profiler tables.
   a. Get RUN_ID.
   b. Query the DBMSHP_RUNS table to find top-level information for RUN_ID that you retrieved.
   c. Query the DBMSHP_FUNCTION_INFO table to find information about each function profiled.
4. Use the plshprof command-line utility to generate simple HTML reports directly from the raw profiler data.
   a. Open a command window.
   b. Change the working directory to /home/oracle/labs/labs.
   c. Run the plshprof utility.
5. Open the report in your browser and review the data.

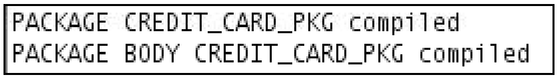
Solution 11-1: Profiling and Tracing PL/SQL Code

In this practice, you generate profiler data and analyze it.

Use your OE connection.

1. Generate profiling data for your CREDIT_CARD_PKG.
   a. Re-create CREDIT_CARD_PKG by running the /home/oracle/labs/labs/lab_11.sql script.

      Use the OE connection.

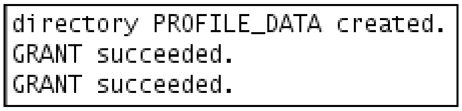


```
PACKAGE CREDIT_CARD_PKG compiled
PACKAGE BODY CREDIT_CARD_PKG compiled
```

   b. You must identify the location of the profiler files. Create a DIRECTORY object to identify this information, and grant the necessary privileges:

      Use the SYS connection.

```
CREATE DIRECTORY profile_data AS '/home/oracle/labs/labs';
GRANT READ, WRITE, EXECUTE ON DIRECTORY profile_data TO OE;
GRANT EXECUTE ON DBMS_HPROF TO OE;
```



```
directory PROFILE_DATA created.
GRANT succeeded.
GRANT succeeded.
```

   c. Use DBMS_HPROF.START_PROFILING to start the profiler for your session.

      Use the OE connection.

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

```
anonymous block completed
```

   d. Run your CREDIT_CARD_PKG.UPDATE_CARD_INFO with the following data.

```
credit_card_pkg.update_card_info
     (154, 'Discover', '123456789');
```

      Use the OE connection.

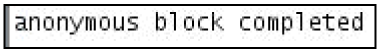
```
DECLARE
  v_card_info typ_cr_card_nst;
BEGIN
-- run application
  credit_card_pkg.update_card_info
     (154, 'Discover', '123456789');
END;
/
```

```
anonymous block completed
```

e. Use DBMS_HPROF.STOP_PROFILING to stop the profiler.

Use the OE connection.

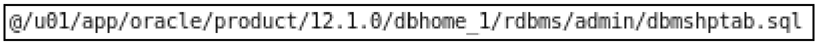
```
BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;
/
```



```
anonymous block completed
```

Alternatively, you can run the solutions for tasks 1_b, 1_c, 1_d, and 1_e from sol_11.sql.

2. Run the dbmshptab.sql script, located in the /u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin folder, to set up the profiler tables.



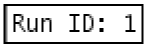
```
@/u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin/dbmshptab.sql
```

Alternatively, run the code from task 2 of sol_11.sql.

3. Use DBMS_HPROF.ANALYZE to analyze the raw data and write the information to the profiler tables.

a. Get RUN_ID.

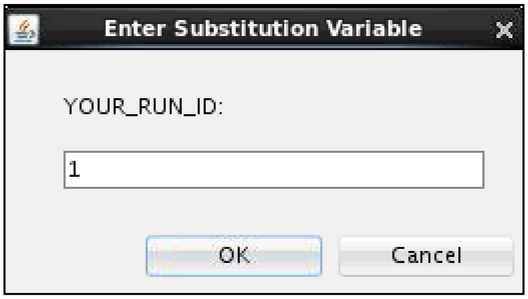
```
SET SERVEROUTPUT ON

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



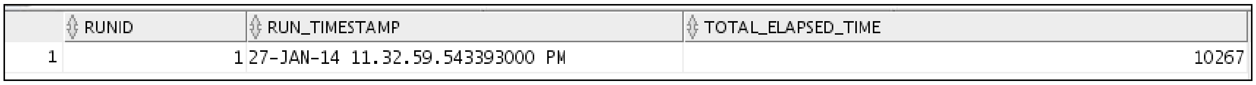
```
Run ID: 1
```

b. Query the DBMSHP_RUNS table to find top-level information for RUN_ID that you retrieved.

```
SET VERIFY OFF

SELECT runid, run_timestamp, total_elapsed_time
FROM dbmshp_runs
WHERE runid = &your_run_id;
```



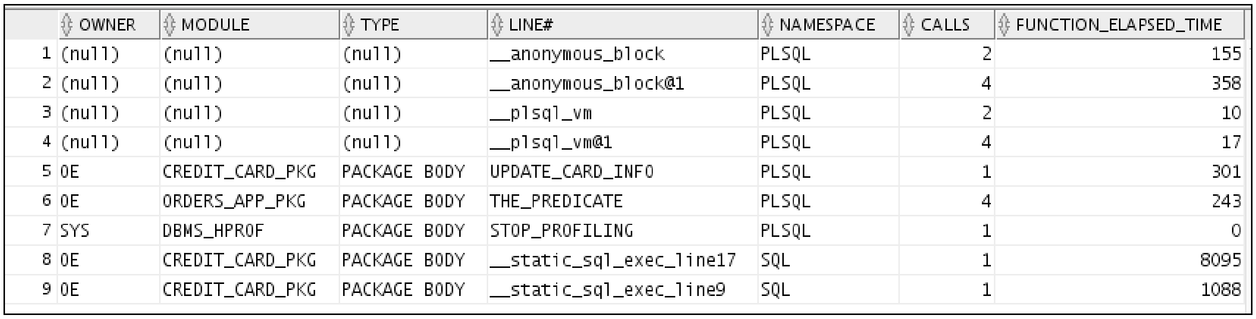
YOUR_RUN_ID: 1



	RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
1	1	27-JAN-14 11.32.59.543393000 PM	10267

c. Query the DBMSHP_FUNCTION_INFO table to find information about each function profiled.

```
SELECT owner, module, type, function line#, namespace,
       calls, function_elapsed_time
FROM   dbmshp_function_info
WHERE  runid = 2;
```



	OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	FUNCTION_ELAPSED_TIME
1	(null)	(null)	(null)	__anonymous_block	PLSQL	2	155
2	(null)	(null)	(null)	__anonymous_block@1	PLSQL	4	358
3	(null)	(null)	(null)	__plsql_vm	PLSQL	2	10
4	(null)	(null)	(null)	__plsql_vm@1	PLSQL	4	17
5	OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	1	301
6	OE	ORDERS_APP_PKG	PACKAGE BODY	THE_PREDICATE	PLSQL	4	243
7	SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1	0
8	OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line17	SQL	1	8095
9	OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line9	SQL	1	1088

Alternatively, you can run the solutions for tasks 3_a, 3_b, and 3_c from `sol_11.sql`.

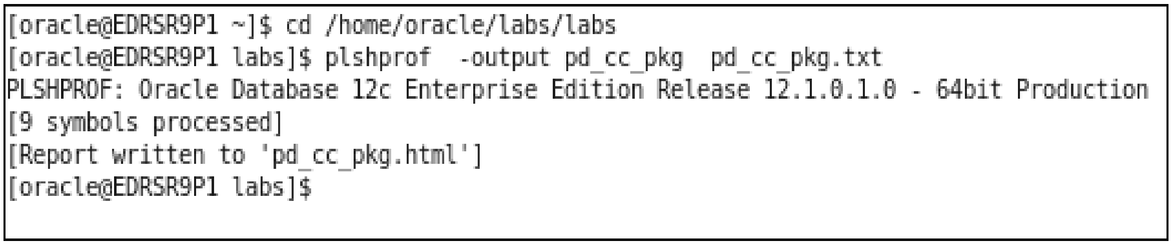
4. Use the plshprof command-line utility to generate simple HTML reports directly from the raw profiler data.

a. Open a command window.

b. Change the working directory to /home/oracle/labs/labs.

c. Run the plshprof utility.

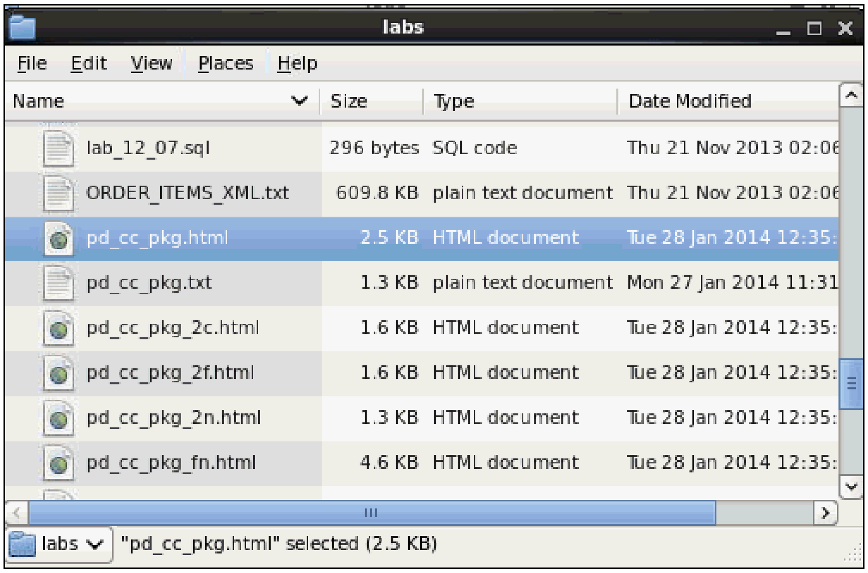
```
--at your command window, change your working directory to
/home/oracle/labs/labs
cd /home/oracle/labs/labs
plshprof  -output pd_cc_pkg  pd_cc_pkg.txt
```



```
[oracle@EDRSR9P1 ~]$ cd /home/oracle/labs/labs
[oracle@EDRSR9P1 labs]$ plshprof  -output pd_cc_pkg  pd_cc_pkg.txt
PLSHPROF: Oracle Database 12c Enterprise Edition Release 12.1.0.1.0 - 64bit Production
[9 symbols processed]
[Report written to 'pd_cc_pkg.html']
[oracle@EDRSR9P1 labs]$
```

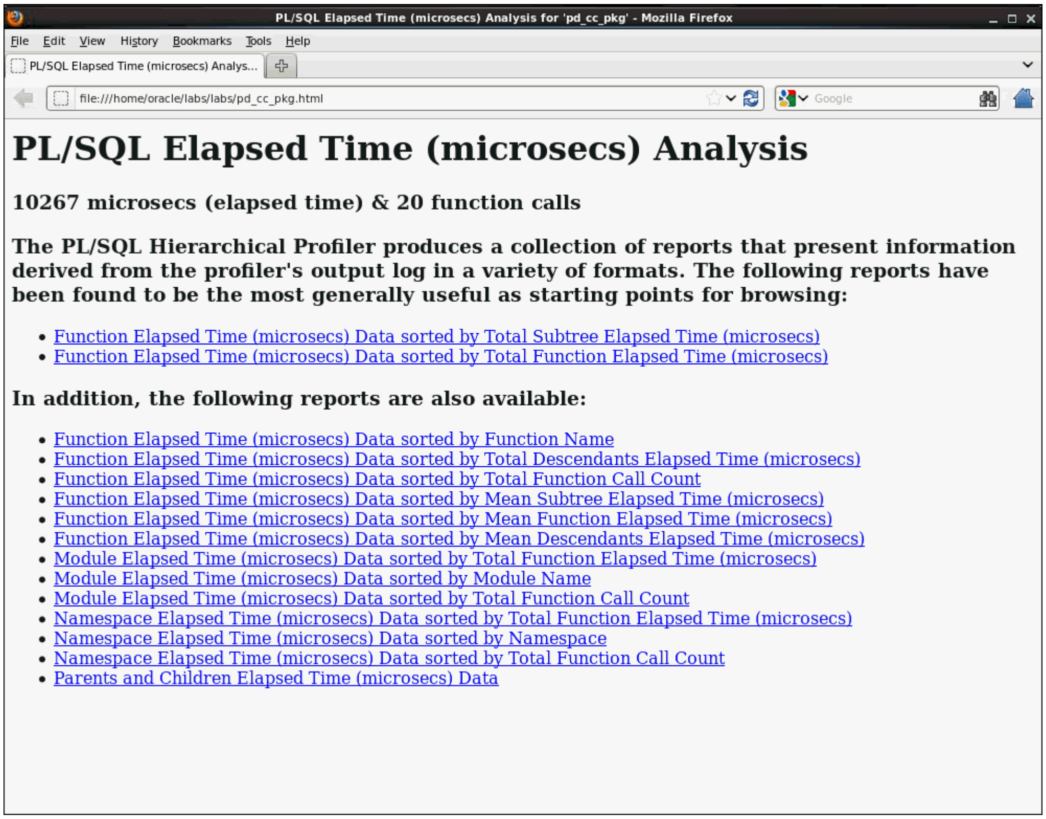
5. Open the report in your browser and review the data.

Navigate to the `/home/oracle/labs/labs` folder.



Name	Size	Type	Date Modified
lab_12_07.sql	296 bytes	SQL code	Thu 21 Nov 2013 02:06
ORDER_ITEMS_XML.txt	609.8 KB	plain text document	Thu 21 Nov 2013 02:06
pd_cc_pkg.html	2.5 KB	HTML document	Tue 28 Jan 2014 12:35:
pd_cc_pkg.txt	1.3 KB	plain text document	Mon 27 Jan 2014 11:31
pd_cc_pkg_2c.html	1.6 KB	HTML document	Tue 28 Jan 2014 12:35:
pd_cc_pkg_2f.html	1.6 KB	HTML document	Tue 28 Jan 2014 12:35:
pd_cc_pkg_2n.html	1.3 KB	HTML document	Tue 28 Jan 2014 12:35:
pd_cc_pkg_fn.html	4.6 KB	HTML document	Tue 28 Jan 2014 12:35:

labs "pd_cc_pkg.html" selected (2.5 KB)


PL/SQL Elapsed Time (microsecs) Analysis for 'pd_cc_pkg' - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
PL/SQL Elapsed Time (microsecs) Analys...
Google

file:///home/oracle/labs/labs/pd_cc_pkg.html

PL/SQL Elapsed Time (microsecs) Analysis

10267 microsecs (elapsed time) & 20 function calls

The PL/SQL Hierarchical Profiler produces a collection of reports that present information derived from the profiler's output log in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:

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

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Descendants Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Mean Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Function Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Descendants Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- Module Elapsed Time (microsecs) Data sorted by Module Name
- Module Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- Namespace Elapsed Time (microsecs) Data sorted by Namespace
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Parents and Children Elapsed Time (microsecs) Data

Practices for Lesson 12: Implementing Fine-Grained Access Control for VPD

Chapter 12

Practices for Lesson 12: Overview

Lesson Overview

In this practice, you:

- Create an application context
- Create a policy
- Create a logon trigger
- Implement a virtual private database
- Test the virtual private database

Practice 12-1: Implementing Fine-Grained Access Control for VPD

Overview

In this practice, you define an application context and security policy to implement the policy: "Sales Representatives can see only their own order information in the ORDERS table." You create sales representative IDs to test the success of your implementation.

Task

Examine the definition of the ORDERS table and the ORDER count for each sales representative:

```
DESCRIBE orders
```

```
Name                 Null?    Type
------------------ -------- -------------------------------
ORDER_ID             NOT NULL NUMBER(12)
ORDER_DATE           NOT NULL TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE                    VARCHAR2(8)
CUSTOMER_ID          NOT NULL NUMBER(6)
ORDER_STATUS                  NUMBER(2)
ORDER_TOTAL                   NUMBER(8,2)
SALES_REP_ID                  NUMBER(6)
PROMOTION_ID                  NUMBER(6)
```

```
SELECT sales_rep_id, count(*)
FROM   orders
GROUP BY sales_rep_id;
```

```
SALES_REP_ID   COUNT(*)
------------ ----------
         153          5
         154         10
         155          5
         156          5
         158          7
         159          7
         160          6
         161         13
         163         12
                     35
10 rows selected.
```

Note: Use SQL*Plus to complete the following steps.

1. Use your SYS connection. Examine and then run the lab_12.sql script.
   This script creates the sales representative ID accounts with appropriate privileges to access the database.

2. Set up an application context:
   a. Connect to the database as `SYS` before creating this context.
   b. Create an application context named `sales_orders_ctx`.
   c. Associate this context to `oe.sales_orders_pkg`.
3. Connect as `OE`.
   a. Examine this package specification:

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

   b. Create this package specification and the package body in the `OE` schema.
   c. When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';
```

   d. Use these constants in the `SET_APP_CONTEXT` procedure to set the application context to the current user.
4. Connect as `SYS` and define the policy.
   a. Use `DBMS_RLS.ADD_POLICY` to define the policy.
   b. Use these specifications for the parameter values:

```
object_schema   OE
object_name     ORDERS
policy_name     OE_ORDERS_ACCESS_POLICY
function_schema OE
policy_function SALES_ORDERS_PKG.THE_PREDICATE
statement_types SELECT, INSERT, UPDATE, DELETE
update_check    FALSE,
enable          TRUE);
```

5. Connect as `SYS` and create a logon trigger to implement fine-grained access control. Name the trigger `SET_ID_ON_LOGON`. This trigger causes the context to be set as each user is logged on.

6. Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         154         10
```

Note: During debugging, you may need to disable or remove some of the objects created for this lesson.

- If you need to disable the logon trigger, issue the following command:

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

- If you need to remove the policy that you created, issue the following command:

```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
'OE_ORDERS_ACCESS_POLICY')
```

Solution 12-1: Implementing Fine-Grained Access Control for VPD

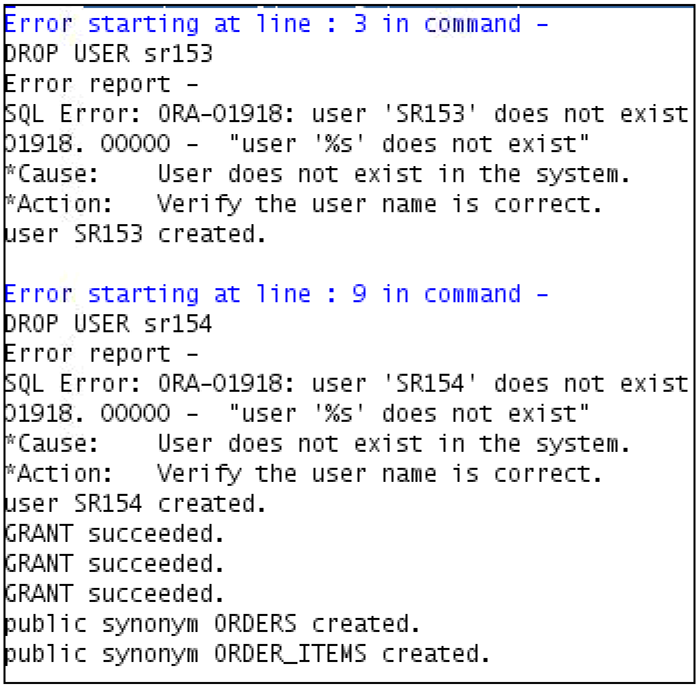
In this practice, you define an application context and security policy to implement the policy: “Sales representatives can see only their own order information in the ORDERS table.” You create sales representative IDs to test the success of your implementation. Examine the definition of the ORDERS table and the ORDER count for each sales representative.

Note: Use SQL*Plus to complete the following steps.

1. Use your SYS connection. Examine and then run the lab_12.sql script.
   This script creates the sales representative ID accounts with appropriate privileges to access the database.

```
DROP USER sr153;
CREATE USER sr153 IDENTIFIED BY oracle
 DEFAULT TABLESPACE USERS
 TEMPORARY TABLESPACE TEMP
 QUOTA UNLIMITED ON USERS;

DROP USER sr154;
CREATE USER sr154 IDENTIFIED BY oracle
 DEFAULT TABLESPACE USERS
 TEMPORARY TABLESPACE TEMP
 QUOTA UNLIMITED ON USERS;

GRANT create session
    , alter session
TO sr153, sr154;

GRANT SELECT, INSERT, UPDATE, DELETE ON
  oe.orders TO sr153, sr154;

GRANT SELECT, INSERT, UPDATE, DELETE ON
  oe.order_items TO sr153, sr154;

CREATE PUBLIC SYNONYM orders FOR oe.orders;
CREATE PUBLIC SYNONYM order_items FOR oe.order_items;
```

```
@lab_12.sql
```



```
Error starting at line : 3 in command -
DROP USER sr153
Error report -
SQL Error: ORA-01918: user 'SR153' does not exist
01918. 00000 -  "user '%s' does not exist"
*Cause:    User does not exist in the system.
*Action:   Verify the user name is correct.
user SR153 created.

Error starting at line : 9 in command -
DROP USER sr154
Error report -
SQL Error: ORA-01918: user 'SR154' does not exist
01918. 00000 -  "user '%s' does not exist"
*Cause:    User does not exist in the system.
*Action:   Verify the user name is correct.
user SR154 created.
GRANT succeeded.
GRANT succeeded.
GRANT succeeded.
public synonym ORDERS created.
public synonym ORDER_ITEMS created.
```

2. Set up an application context:
   a. Connect to the database as `SYS` before creating this context.
   b. Create an application context named `sales_orders_ctx`.
   c. Associate this context with the `oe.sales_orders_pkg`.

```
CREATE CONTEXT sales_orders_ctx
USING oe.sales_orders_pkg;
```

```
context SALES_ORDERS_CTX created.
```

Alternatively, run the code from task 2 of `sol_12.sql`.

3. Connect as `OE`.
   a. Examine this package specification:

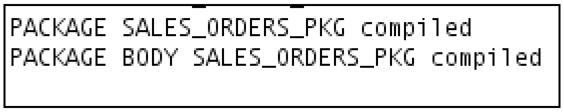
```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
 PROCEDURE set_app_context;
 FUNCTION the_predicate
   (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END sales_orders_pkg;    -- package spec
/
```

   b. Create this package specification, and then the package body in the `OE` schema.
   c. When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30)  := 'SALES_ORDERS_CTX';
c_attrib  CONSTANT VARCHAR2(30)  := 'SALES_REP';
```

   d. Use these constants in the `SET_APP_CONTEXT` procedure to set the application context to the current user.

```
CREATE OR REPLACE PACKAGE BODY sales_orders_pkg
IS
  c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
  c_attrib  CONSTANT VARCHAR2(30) := 'SALES_REP';

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

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

END sales_orders_pkg; -- package body
/
```



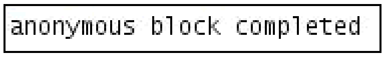
```
PACKAGE SALES_ORDERS_PKG compiled
PACKAGE BODY SALES_ORDERS_PKG compiled
```

Alternatively, run the code from task 3 of `sol_12.sql`.

4. Connect as `SYS` and define the policy.
   a. Use `DBMS_RLS.ADD_POLICY` to define the policy.
   b. Use the following specifications for the parameter values:

object_schema	OE
object_name	ORDERS
policy_name	OE_ORDERS_ACCESS_POLICY
function_schema	OE
policy_function	SALES_ORDERS_PKG.THE_PREDICATE
statement_types	SELECT, INSERT, UPDATE, DELETE
update_check	FALSE,
enable	TRUE

```
DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
   'OE',
   'ORDERS',
   'OE_ORDERS_ACCESS_POLICY',
   'OE',
   'SALES_ORDERS_PKG.THE_PREDICATE',
   'SELECT, INSERT, UPDATE, DELETE',
   FALSE,
   TRUE);
END;
/
```

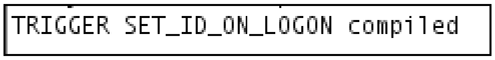


```
anonymous block completed
```

Alternatively, run the code from task 4 of `sol_12.sql`.

5. Connect as `SYS` and create a logon trigger to implement fine-grained access control. Name the trigger `SET_ID_ON_LOGON`. This trigger causes the context to be set as each user is logged on.

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

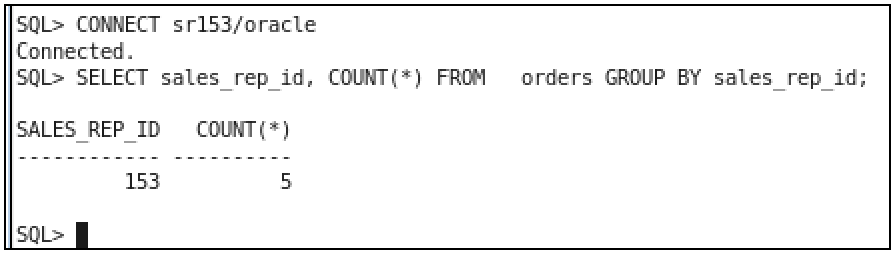


```
TRIGGER SET_ID_ON_LOGON compiled
```

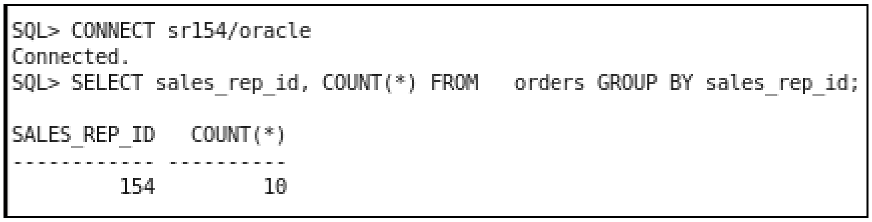
Alternatively, run the code from task 5 of `sol_12.sql`.

6. Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match the following:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         154         10
```



```
SQL> CONNECT sr153/oracle
Connected.
SQL> SELECT sales_rep_id, COUNT(*) FROM   orders GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5

SQL>
```



```
SQL> CONNECT sr154/oracle
Connected.
SQL> SELECT sales_rep_id, COUNT(*) FROM   orders GROUP BY sales_rep_id;

SALES_REP_ID   COUNT(*)
------------ ----------
         154         10
```

Note: During debugging, you may need to disable or remove some of the objects created for this lesson.

- If you need to disable the logon trigger, issue the following command:
  `ALTER TRIGGER set_id_on_logon DISABLE;`
- If you need to remove the policy that you created, issue the following command:
  ```
  EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS',-
  'OE_ORDERS_ACCESS_POLICY')
  ```

Alternatively, run the code from task 6 of `sol_12.sql`.

Practices for Lesson 13: Safeguarding Your Code Against SQL Injection Attacks

Chapter 13

Practices for Lesson 13: Overview

Lesson Overview

In this practice, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks

Overview

In this practice, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use the `OE` connection for this practice.

Task

1. Only code that is used in web applications is vulnerable to SQL injection attack.
   a. True
   b. False
2. Code that is most vulnerable to SQL injection attack contains: (Check all that apply.)
   a. Input parameters
   b. Dynamic SQL with bind arguments
   c. Dynamic SQL with concatenated input values
   d. Calls to exterior functions
3. By default, a stored procedure or SQL method executes with the privileges of the owner (definer's rights).
   a. True
   b. False
4. By using `AUTHID CURRENT_USER` in your code, you are: (Check all that apply.)
   a. Specifying that the code executes with invoker's rights
   b. Specifying that the code executes with the highest privilege level
   c. Eliminating any possible SQL injection vulnerability
   d. Not eliminating all possible SQL injection vulnerabilities
5. Match each attack surface reduction technique with an example of the technique.

Technique	Example
Executes code with minimal privileges	Specify appropriate parameter types
Lock the database	Revoke privileges from `PUBLIC`
Reduce arbitrary input	Use invoker's rights

6. Examine the following code. Run task 6 of the `lab_13.sql` script to create the procedure.

```
CREATE OR REPLACE PROCEDURE get_income_level (p_email VARCHAR2
DEFAULT NULL)
IS
  TYPE      cv_custtyp IS REF CURSOR;
  cv        cv_custtyp;
  v_income  customers.income_level%TYPE;
  v_stmt    VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT income_level FROM customers WHERE
             cust_email = ''' || p_email || '''';
  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
      FETCH cv INTO v_income;
      EXIT WHEN cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Income level is: '||v_income);
  END LOOP;
  CLOSE cv;

EXCEPTION WHEN OTHERS THEN
   dbms_output.PUT_LINE(sqlerrm);
   dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END get_income_level;
/
```

a. Execute the following statements and note the results.

```
exec get_income_level(‘Kris.Harris@DIPPER.EXAMPLE.COM’)

exec get_income_level('x'' union select username from all_users
where ''x''=''x')
```

b. Has SQL injection occurred?

7. Rewrite the code to protect against SQL injection. You can run step 7 of the `lab_13.sql` script to re-create the procedure.

a. Execute the following statements and note the results:

```
exec get_income_level(‘Kris.Harris@DIPPER.EXAMPLE.COM’)

exec get_income_level('x'' union select username from all_users
where ''x''=''x')
```

b. Has SQL injection occurred?

Solution 13-1: Safeguarding Your Code Against SQL Injection Attacks

In this practice, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use the `OE` connection for this practice.

Understanding SQL Injection

1. Only code used in web applications is vulnerable to SQL injection attack.
   b. **False**

2. Code that is most vulnerable to SQL injection attack contains: (Check all that apply.)
   c. **Dynamic SQL with concatenated input values**

3. By default, a stored procedure or SQL method executes with the privileges of the owner (definer's rights).
   a. **True**

4. By using `AUTHID CURRENT_USER` in your code, you are: (Check all that apply.)
   a. **Specifying that the code executes with invoker's rights**
   d. **Not eliminating all possible SQL injection vulnerabilities**

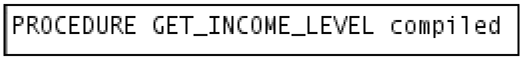
5. Match each attack surface reduction technique to an example of the technique.
   Technique: Example
   Executes code with minimal privileges: Use invoker's rights
   Lock the database: Revoke privileges from `PUBLIC`
   Reduce arbitrary input: Specify appropriate parameter types

Rewriting Code to Protect Against SQL Injection

6. Examine this code. Run task 6 in the `lab_13.sql` script to create the procedure.

```
CREATE OR REPLACE PROCEDURE get_income_level
  (p_email VARCHAR2 DEFAULT NULL)
IS
  TYPE      cv_custtyp IS REF CURSOR;
  cv        cv_custtyp;
  v_income  customers.income_level%TYPE;
  v_stmt    VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT income_level FROM customers WHERE
             cust_email = ''' || p_email || '''';

  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
```

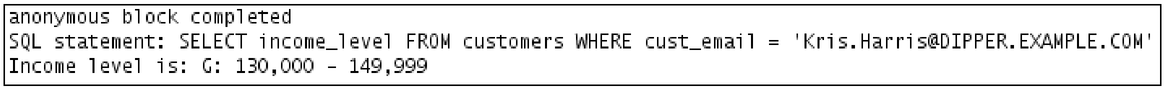
```
      FETCH cv INTO v_income;
      EXIT WHEN cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('Income level is: '||v_income);
  END LOOP;
  CLOSE cv;

EXCEPTION WHEN OTHERS THEN
   dbms_output.PUT_LINE(sqlerrm);
   dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END get_income_level;
/
```



```
PROCEDURE GET_INCOME_LEVEL compiled
```

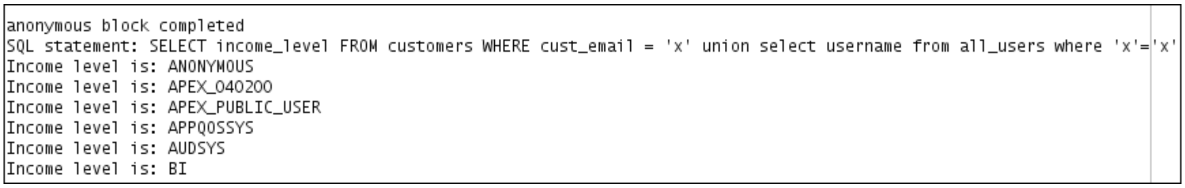
a. Execute the following statements and note the results.

```
SET SERVEROUTPUT ON
exec get_income_level(‘Kris.Harris@DIPPER.EXAMPLE.COM’)
```



```
anonymous block completed
SQL statement: SELECT income_level FROM customers WHERE cust_email = 'Kris.Harris@DIPPER.EXAMPLE.COM'
Income level is: G: 130,000 - 149,999
```

```
exec get_income_level('x'' union select username from all_users
where ''x''=''x')
```



```
anonymous block completed
SQL statement: SELECT income_level FROM customers WHERE cust_email = 'x' union select username from all_users where 'x'='x'
Income level is: ANONYMOUS
Income level is: APEX_040200
Income level is: APEX_PUBLIC_USER
Income level is: APPQOSSYS
Income level is: AUDSYS
Income level is: BI
```

Alternatively, you can execute the solution for task 6_a from `sol_13.sql`.

b. Has SQL injection occurred?

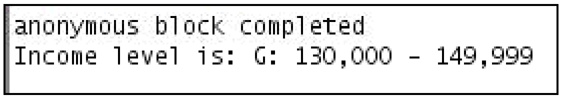
Yes, by using dynamic SQL constructed via concatenation of input values, you see all users in the database.

7. Rewrite the code to protect against SQL injection. You can run step `07` in the `lab_13.sql` script to re-create the procedure.

```
CREATE OR REPLACE
PROCEDURE get_income_level (p_email VARCHAR2 DEFAULT NULL)
AS
BEGIN
FOR i IN
  (SELECT income_level
   FROM customers
   WHERE cust_email = p_email)
  LOOP
      DBMS_OUTPUT.PUT_LINE('Income level is:
        '||i.income_level);
  END LOOP;
END get_income_level;
/
```

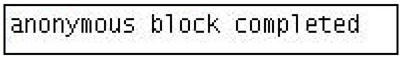
a. Execute the following statements and note the results.

```
SET SERVEROUTPUT ON
exec get_income_level('Kris.Harris@DIPPER.EXAMPLE.COM')
```



```
anonymous block completed
Income level is: G: 130,000 - 149,999
```

```
exec get_income_level('x'' union select username from all_users
where ''x''=''x')
```



```
anonymous block completed
```

Alternatively, you can execute the solution for task 7_a from `sol_13.sql`.

b. Has SQL injection occurred?

No