

Oracle Database 12c: Develop PL/SQL Program Units

Activity Guide

D80170GC11
Edition 1.1
July 2014
D87349

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Disclaimer

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

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

Restricted Rights Notice

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

U.S. GOVERNMENT RIGHTS

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

Trademark Notice

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

Author

Dimpi Rani Sarmah

Technical Contributors and Reviewers

Wayne Abbott, Madhavi Siddireddy, Laszlo Czinkoczki, Nancy Greenberg, KimSeong Loh, Miyuki Osato, Swarnapriya Shridhar

This book was published using: Oracle Tutor

Table of Contents

Practices for Lesson 1: Introduction.....	1-1
Practices for Lesson 1.....	1-2
Practice 1-1: Creating and Using a New SQL Developer Database Connection.....	1-3
Solution 1-1: Creating and Using a New SQL Developer Database Connection.....	1-4
Practice 1-2: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block	1-6
Solution 1-2: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block	1-7
Practice 1-3: Setting Some SQL Developer Preferences	1-11
Solution 1-3: Setting Some SQL Developer Preferences	1-12
Practices for Lesson 2: Creating Procedures	2-1
Practices for Lesson 2.....	2-2
Practice 2-1: Creating, Compiling, and Calling Procedures	2-3
Solution 2-1: Creating, Compiling, and Calling Procedures	2-5
Practices for Lesson 3: Creating Functions and Debugging Subprograms.....	3-1
Practices for Lesson 3.....	3-2
Practice 3-1: Creating Functions.....	3-3
Solution 3-1: Creating Functions.....	3-5
Practice 3-2: Introduction to the SQL Developer Debugger	3-10
Solution 3-2: Introduction to the SQL Developer Debugger	3-11
Practices for Lesson 4: Creating Packages	4-1
Practices for Lesson 4.....	4-2
Practice 4-1: Creating and Using Packages	4-3
Solution 4-1: Creating and Using Packages	4-5
Practices for Lesson 5: Working with Packages.....	5-1
Practices for Lesson 5.....	5-2
Practice 5-1: Working with Packages	5-3
Solution 5-1: Working with Packages	5-6
Practices for Lesson 6: Using Oracle-Supplied Packages in Application Development.....	6-1
Practices for Lesson 6.....	6-2
Practice 6-1: Using the UTL_FILE Package	6-3
Solution 6-1: Using the UTL_FILE Package	6-4
Practices for Lesson 7: Using Dynamic SQL	7-1
Practices for Lesson 7.....	7-2
Practice 7-1: Using Native Dynamic SQL	7-3
Solution 7-1: Using Native Dynamic SQL	7-5
Practices for Lesson 8: Design Considerations for PL/SQL Code	8-1
Practices for Lesson 8.....	8-2
Practice 8-1: Using Bulk Binding and Autonomous Transactions.....	8-3
Solution 8-1: Using Bulk Binding and Autonomous Transactions.....	8-5
Practices for Lesson 9: Creating Triggers	9-1
Practices for Lesson 9.....	9-2
Practice 9-1: Creating Statement and Row Triggers	9-3
Solution 9-1: Creating Statement and Row Triggers	9-5
Practices for Lesson 10: Creating Compound, DDL, and Event Database Triggers	10-1
Practices for Lesson 10.....	10-2
Practice 10-1: Managing Data Integrity Rules and Mutating Table Exceptions.....	10-3

Solution 10-1: Managing Data Integrity Rules and Mutating Table Exceptions.....	10-6
Practices for Lesson 11: Using the PL/SQL Compiler	11-1
Practices for Lesson 11.....	11-2
Practice 11-1: Using the PL/SQL Compiler Parameters and Warnings	11-3
Solution 11-1: Using the PL/SQL Compiler Parameters and Warnings	11-4
Practices for Lesson 12: Managing Dependencies	12-1
Practices for Lesson 12.....	12-2
Practice 12-1: Managing Dependencies in Your Schema.....	12-3
Solution 12-1: Managing Dependencies in Your Schema.....	12-4
Additional Practices 1.....	13-1
Additional Practices 1.....	13-2
Practice 1-1: Creating a New SQL Developer Database Connection.....	13-3
Solution 1-1: Creating a New SQL Developer Database Connection.....	13-4
Practice 1-2: Adding a New Job to the JOBS Table.....	13-6
Solution 1-2: Adding a New Job to the JOBS Table.....	13-7
Practice 1-3: Adding a New Row to the JOB_HISTORY Table.....	13-9
Solution 1-3: Adding a New Row to the JOB_HISTORY Table.....	13-10
Practice 1-4: Updating the Minimum and Maximum Salaries for a Job	13-13
Solution 1-4: Updating the Minimum and Maximum Salaries for a Job	13-14
Practice 1-5: Monitoring Employees Salaries	13-17
Solution 1-5: Monitoring Employees Salaries	13-18
Practice 1-6: Retrieving the Total Number of Years of Service for an Employee	13-22
Solution 1-6: Retrieving the Total Number of Years of Service for an Employee	13-23
Practice 1-7: Retrieving the Total Number of Different Jobs for an Employee	13-26
Solution 1-7: Retrieving the Total Number of Different Jobs for an Employee	13-27
Practice 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions	13-29
Solution 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions	13-30
Practice 1-9: Creating a Trigger to Ensure that the Employees' Salaries Are Within the Acceptable Range.....	13-36
Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range	13-37
Additional Practices 2.....	14-1
Additional Practices 2.....	14-2
Practice 2-1: Creating the VIDEO_PKG Package.....	14-4
Solution 2-1: Creating the VIDEO_PKG Package.....	14-6

Practices for Lesson 1: Introduction

Chapter 1

Practices for Lesson 1

Overview

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

If you miss a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice. The solutions for these practices can be found in “Activity Guide: Practices and Solutions.”

Practice 1-1: Creating and Using a New SQL Developer Database Connection

Overview

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

Tasks

1. Start up SQL Developer by using the user ID and password that are provided to you by the instructor, such as `ora61`.
2. Create a database connection by using the following information:
 - a. Connection Name: `MyDBConnection`
 - b. Username: `ora61`
 - c. Password: `ora61`
 - d. Hostname: Enter the host name for your PC, or let the default `localhost` remain.
 - e. Port: `1521`
 - f. SID: `ORCL`
3. Test the new connection. If the Status shows as Success, connect to the database by using this new connection. Click the Test button in the New>Select Database Connection window. If the status shows as Success, click Connect.

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

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

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

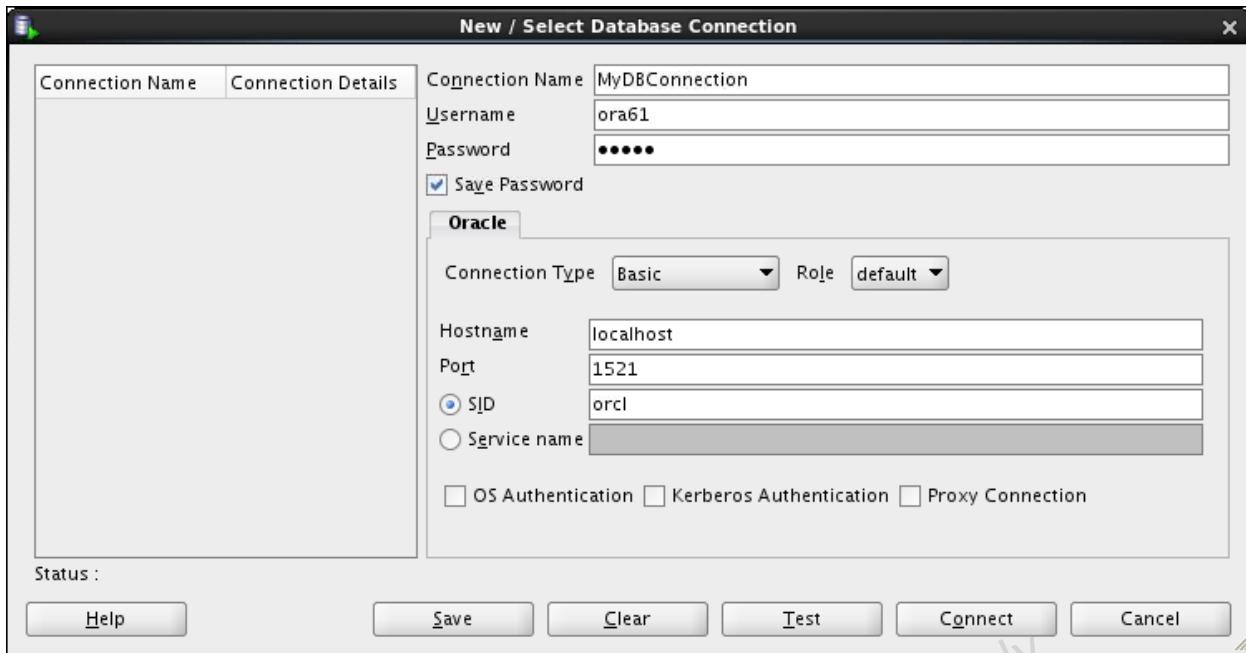
Click the SQL Developer icon on your desktop.



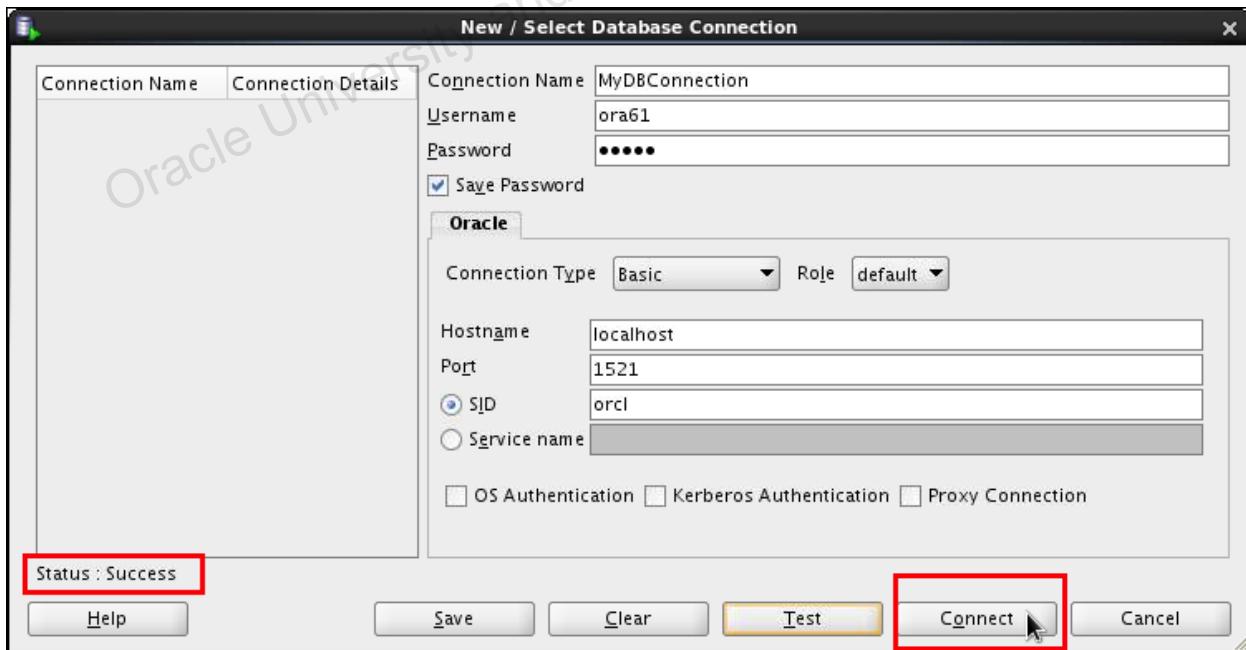
2. Create a database connection by using the following information:
 - a. Connection Name: MyDBConnection
 - b. Username: ora61
 - c. Password: ora61
 - d. Select *Save Password* checkbox to save the connection
 - e. Hostname: Enter the host name for your PC or alternatively mention localhost
 - f. Port: 1521
 - g. SID: ORCL

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

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



3. Test the new connection. If the Status shows as Success, connect to the database by using this new connection:
 - a. Click the Test button in the New/Select Database Connection window. If the status is Success, click the Connect button.



Practice 1-2: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block

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 and display its data.
 - a. Expand the MyDBConnection connection by clicking the plus sign next to it.
 - b. Expand the Tables icon by clicking the plus sign next to it.
 - c. Display the structure of the EMPLOYEES table.
2. Browse the EMPLOYEES table and display its data.
3. Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than \$10,000. Use both the Execute Statement (or press the F9 key) and the Run Script icon (or press the F5 key) icons to execute the SELECT statement. Review the results of both methods of executing the SELECT statements in the appropriate tabs.

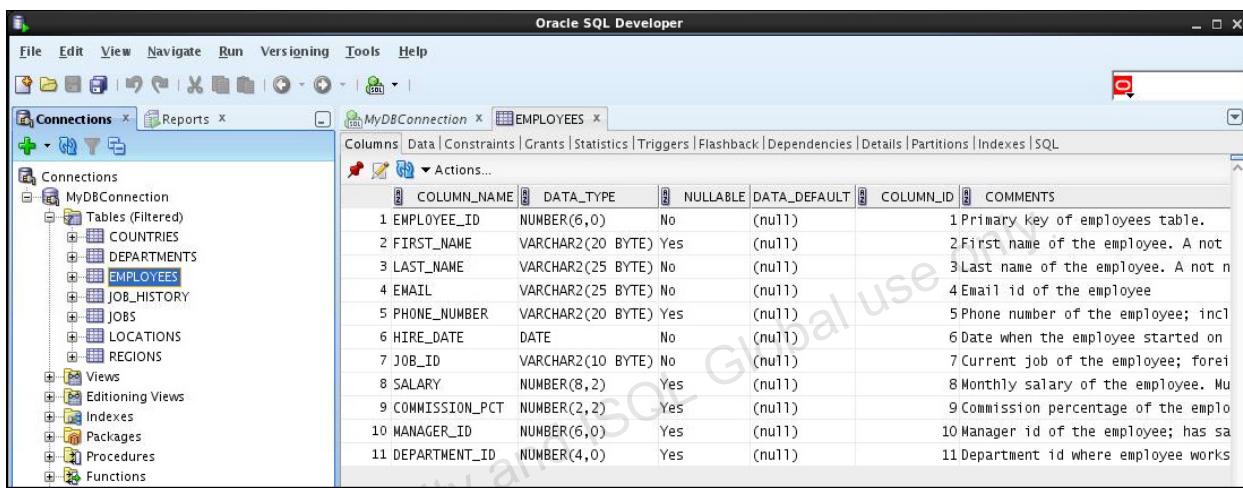
Note: Take a few minutes to familiarize yourself with the data, or consult Appendix A, which provides the description and data for all the tables in the HR schema that you will use in this course.
4. Create and execute a simple anonymous block that outputs “Hello World.”
 - a. Enable SET SERVEROUTPUT ON to display the output of the DBMS_OUTPUT package statements.
 - b. Use the SQL Worksheet area to enter the code for your anonymous block.
 - c. Click the Run Script (or press the F5 key) icon to run the anonymous block.

Solution 1-2: Browsing Your Schema Tables and Creating and Executing a Simple Anonymous Block

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

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

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

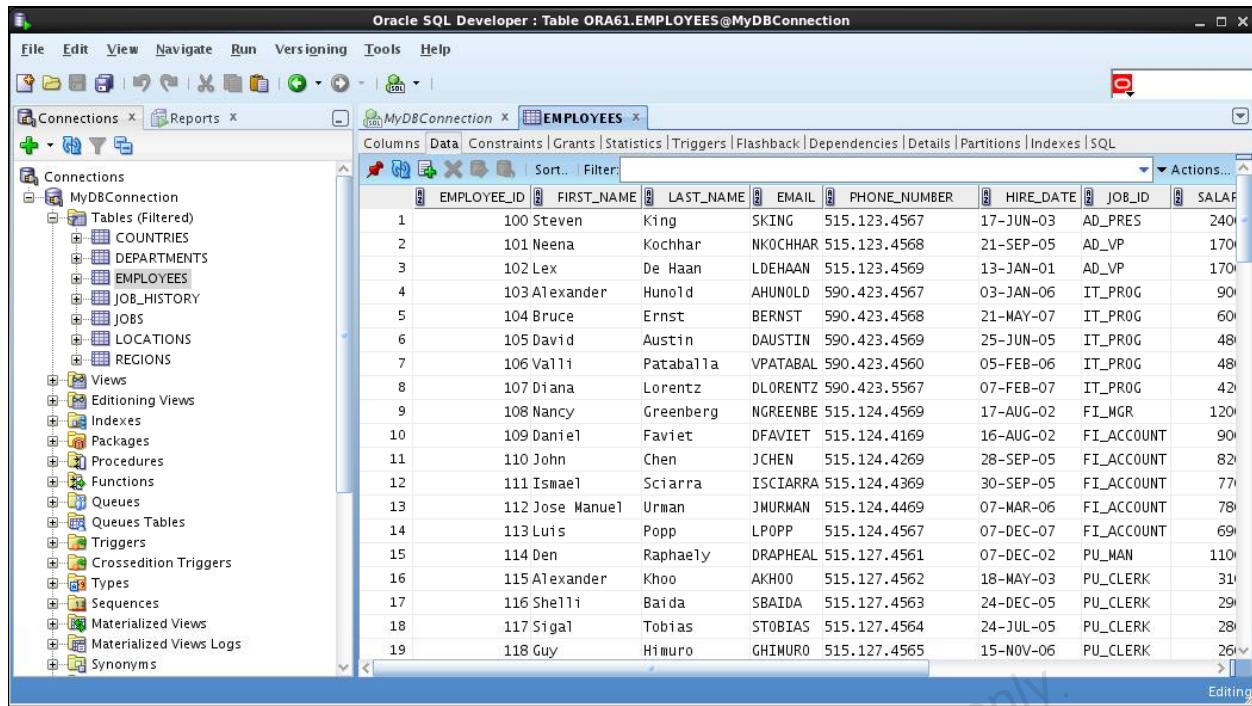


The screenshot shows the Oracle SQL Developer interface. On the left, the Connections pane shows a connection named 'MyDBConnection' with several tables listed under 'Tables (Filtered)'. The 'EMPLOYEES' table is selected. On the right, the main workspace shows the 'EMPLOYEES' table structure in the 'Columns' tab. The table has 11 columns with the following details:

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	Primary key of employees table.
2	FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	First name of the employee. A not
3	LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	Last name of the employee. A not n
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 employee; incl
6	HIRE_DATE	DATE	No	(null)	6	Date when the employee started on
7	JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	Current job of the employee; forei
8	SALARY	NUMBER(8,2)	Yes	(null)	8	Monthly salary of the employee. Mu
9	COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	Commission percentage of the emplo
10	MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	Manager id of the employee; has sa
11	DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	Department id where employee works

2. Browse the EMPLOYEES table and display its data.

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



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

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

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

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

Open the sol_01.sql file in the /home/oracle/labs/plpu/solns directory by using one of the following two methods:

- On the Files tab, select (or navigate to) the script file that you want to open.
- Double-click the file name to open. The code of the script file is displayed in the SQL Worksheet area. Uncomment and select the solution for Task 3.
- To run the code, click the Run Script (or press F5) icon on the SQL Worksheet toolbar.

Alternatively, you can also:

- Select Open from the File menu. The Open dialog box is displayed.
- In the Open dialog box, select (or navigate to) the script file that you want to open.

- c. Click Open. The code of the script file is displayed in the SQL Worksheet area. Uncomment and select the solution for Task 3.
- d. To run the code, click the Run Script icon (or press F5) on the SQL Worksheet toolbar.

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

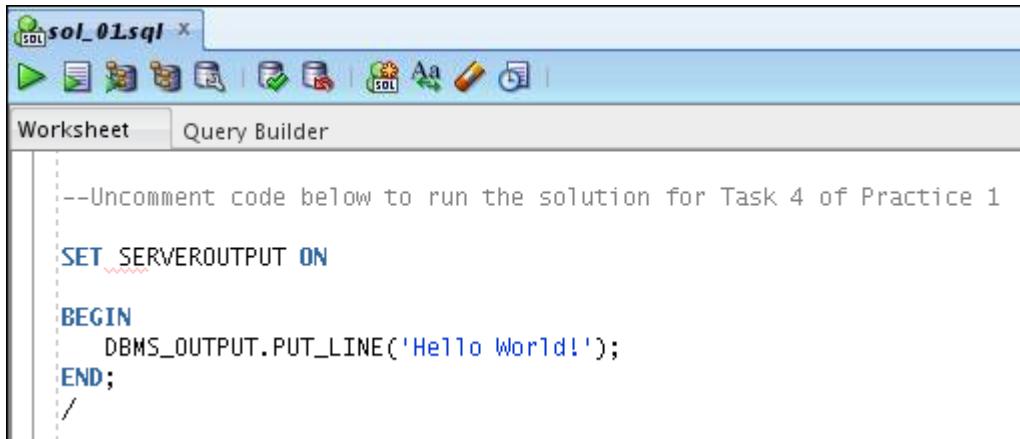
```
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE SALARY > 10000;
```

	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000
4	Greenberg	12008
5	Raphaely	11000
6	Russell	14000
7	Partners	13500
8	Errazuriz	12000
9	Cambrault	11000
10	Zlotkey	10500
11	Vishney	10500
12	Ozer	11500
13	Abel	11000
14	Hartstein	13000
15	Higgins	12008

- 4. Create and execute a simple anonymous block that outputs “Hello World.”
 - a. Enable SET SERVEROUTPUT ON to display the output of the DBMS_OUTPUT package statements.
- Enter the following command in the SQL Worksheet area, and then click the Run Script icon (or press F5).

```
SET SERVEROUTPUT ON
```

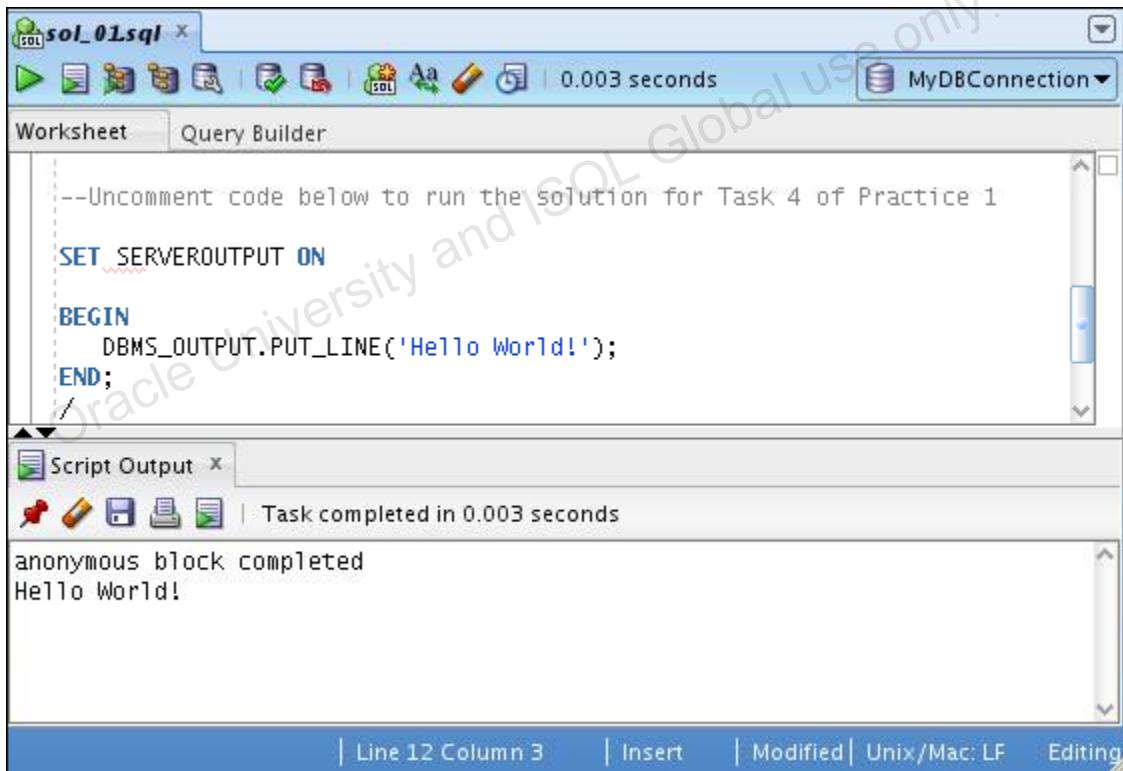
- b. Use the SQL Worksheet area to enter the code for your anonymous block. Open the `sol_01.sql` file in the `/home/oracle/labs/plpu/solns` directory and uncomment and select the code under Task 4. The code is displayed as follows:



The screenshot shows the Oracle SQL Worksheet interface. The title bar says "sol_01sql". The main window has tabs for "Worksheet" and "Query Builder", with "Worksheet" selected. The code area contains the following SQL:

```
--Uncomment code below to run the solution for Task 4 of Practice 1
SET SERVEROUTPUT ON
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
/
```

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



The screenshot shows the Oracle SQL Worksheet interface with the "Script Output" tab selected. The title bar says "sol_01sql". The "MyDBConnection" connection is selected. The code area is identical to the previous screenshot. The "Script Output" tab shows the results of the execution:

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

Below the output, status information is shown: "Task completed in 0.003 seconds". The bottom status bar indicates "Line 12 Column 3", "Insert", "Modified", "Unix/Mac: LF", and "Editing".

Practice 1-3: Setting Some SQL Developer Preferences

Overview

In this practice, you set some SQL Developer preferences.

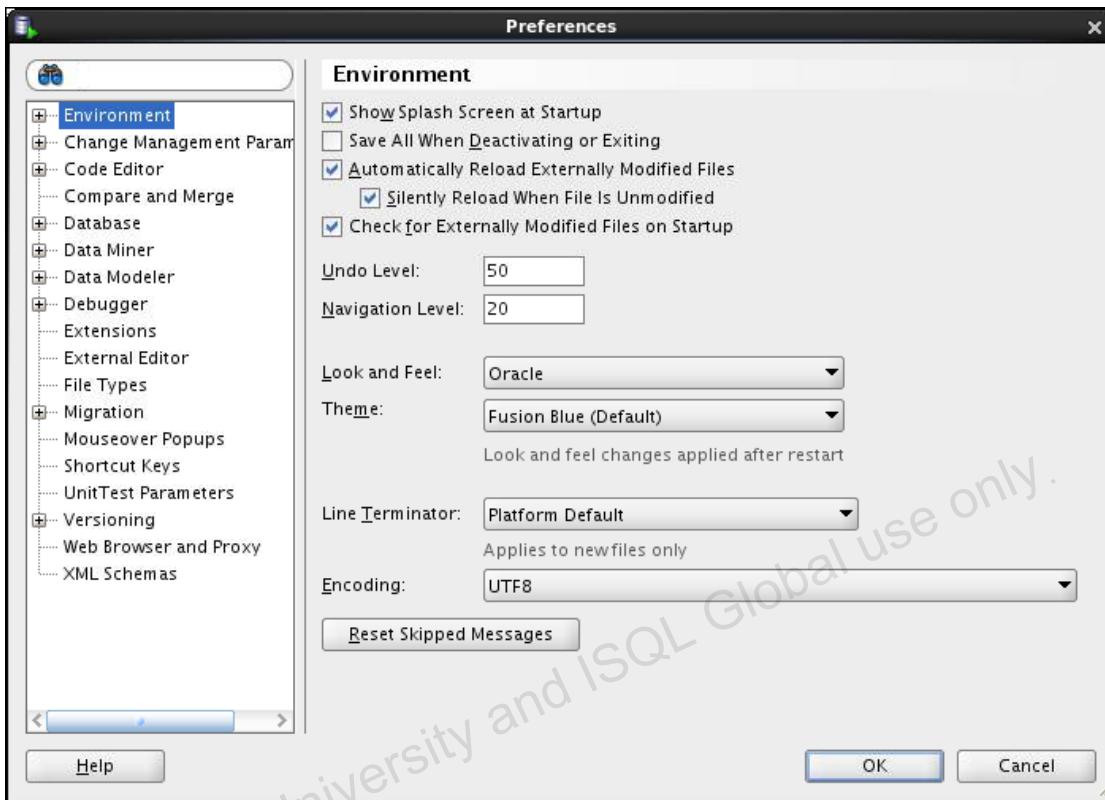
Tasks

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

Solution 1-3: Setting Some SQL Developer Preferences

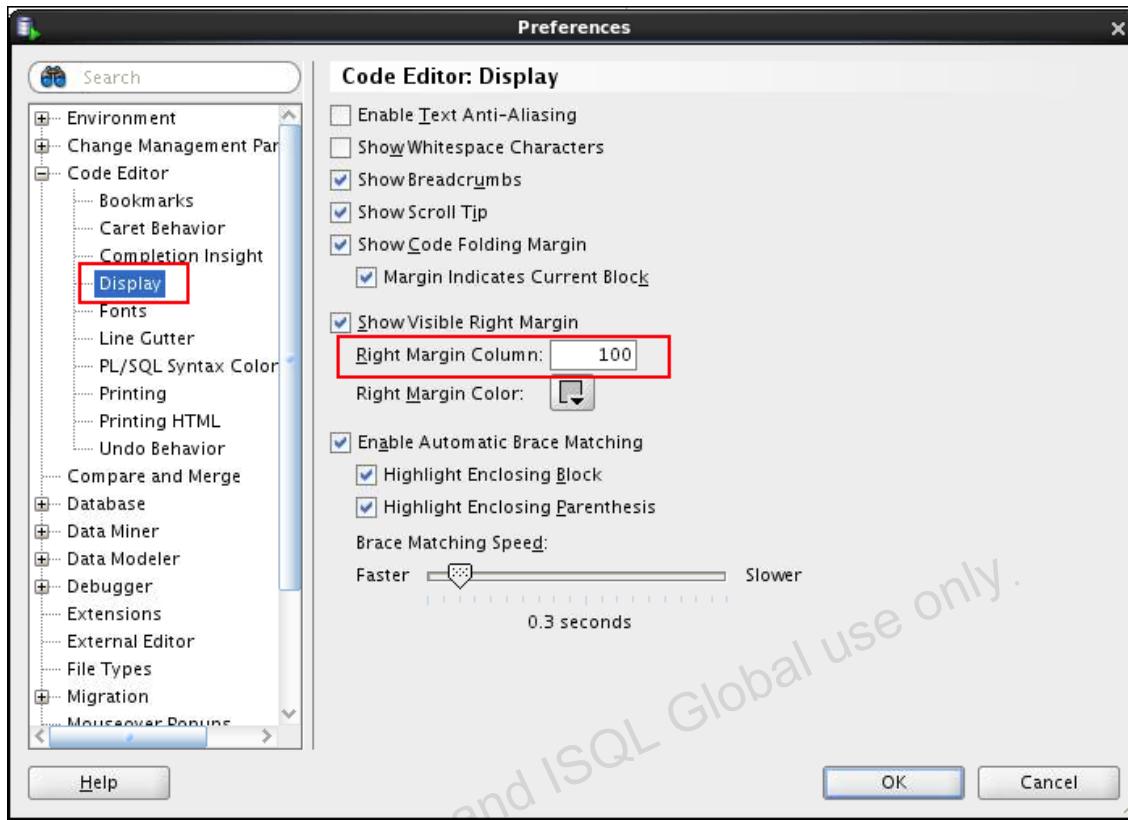
In this practice, you set some SQL Developer preferences.

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

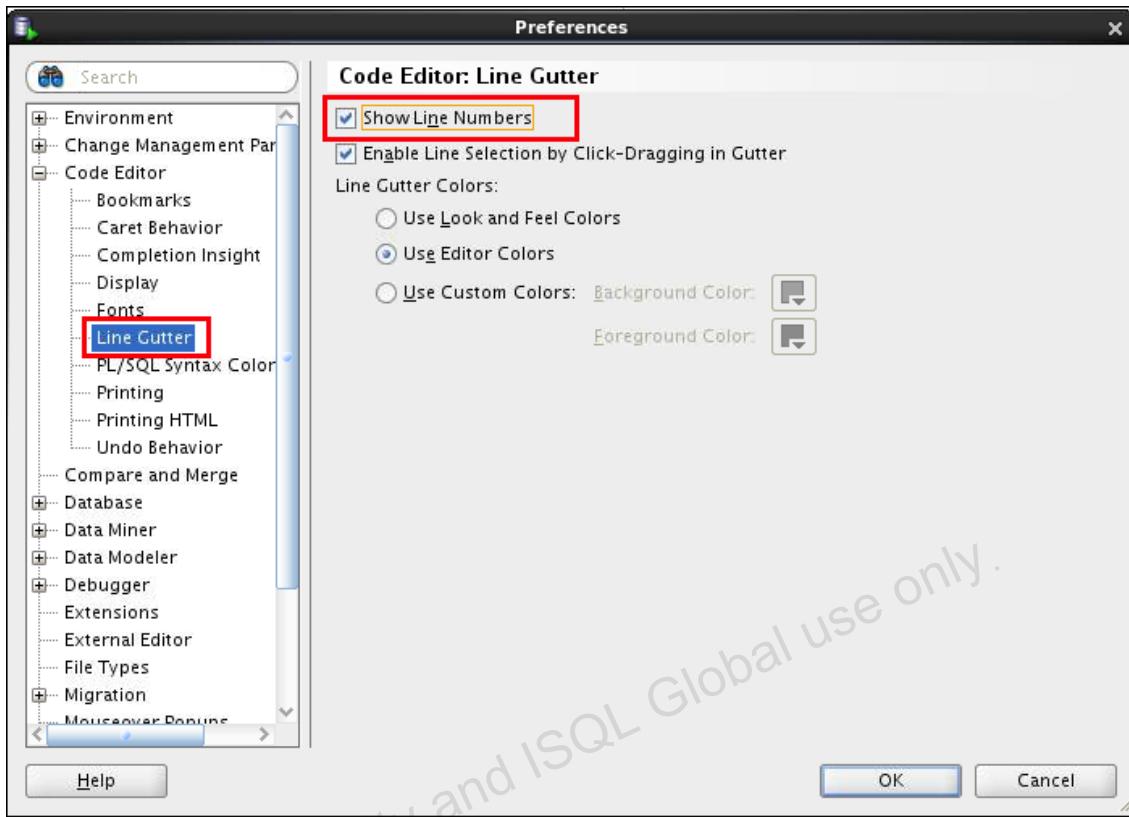


2. Expand the Code Editor option, and then click the Display option to display the “Code Editor: Display” section. The “Code Editor: Display” section contains general options for the appearance and behavior of the code editor.

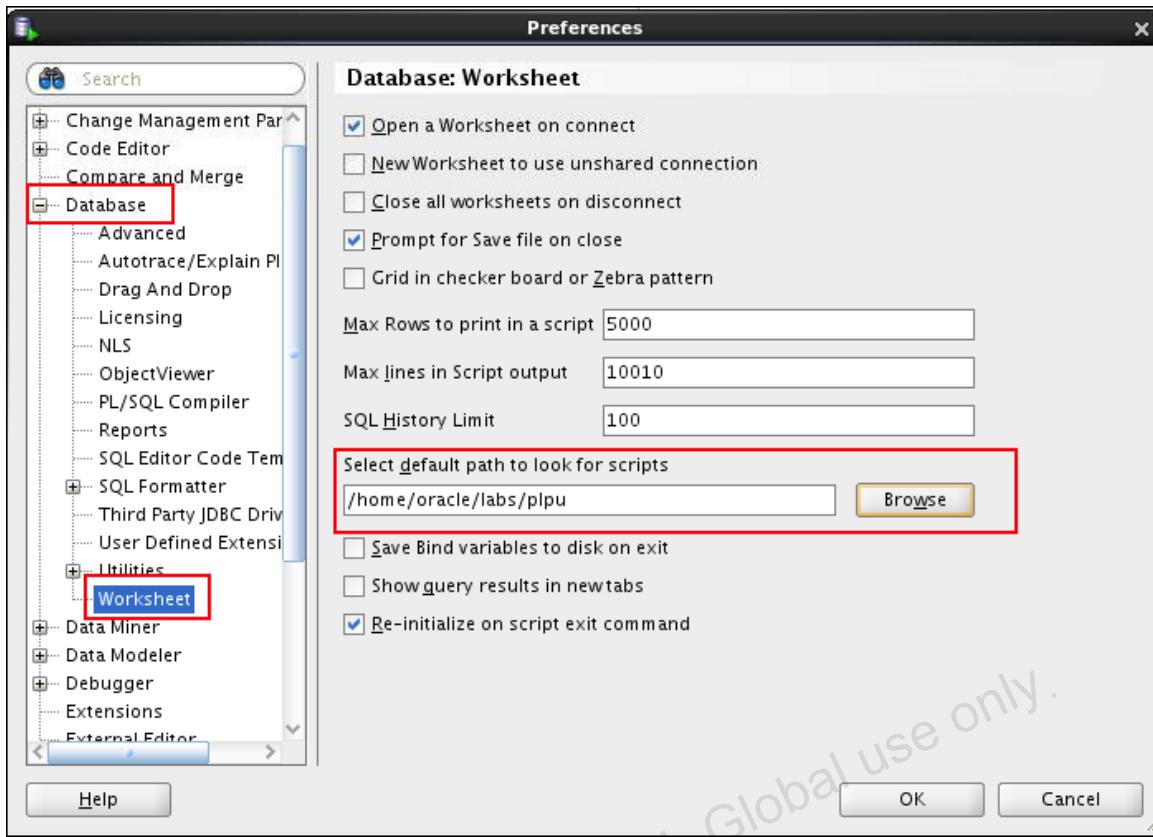
- a. Enter 100 in the Right Margin Column text box in the Show Visible Right Margin section. This renders a right margin that you can set to control the length of lines of code.



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



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



4. Click OK to accept your changes and to exit the Preferences window.
5. Familiarize yourself with the labs directory on the /home/oracle/labs/plpu directory.
 - a. Click the **Files** tab (next to the **Connections** tab).
 - b. Navigate to the /home/oracle/labs/plpu directory.
 - c. How many subdirectories do you see in the labs directory?
 - d. Navigate through the directories, and open a script file without executing the code.

Unauthorized reproduction or distribution prohibited. Copyright 2012, Oracle and/or its affiliates.

Oracle University and ISQL Global use only.

Practices for Lesson 2: Creating Procedures

Chapter 2

Practices for Lesson 2

Overview

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

Note:

1. Before starting this practice, execute the
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_02.sql` script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 2-1: Creating, Compiling, and Calling Procedures

Overview

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

Note: Execute `cleanup_02.sql` from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following task.

Task

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

Note: You can create the procedure (and other objects) by entering the code in the SQL Worksheet area, and then click the Run Script (F5) icon. This creates and compiles the procedure. To find out whether or not the procedure has any errors, click the procedure name in the procedure node, and then select Compile from the pop-up menu.
 - b. Invoke the procedure with `IT_DBA` as the job ID and `Database Administrator` as the job title. Query the `JOBS` table and view the results.

anonymous block completed		MIN_SALARY	MAX_SALARY
JOB_ID	JOB_TITLE		
IT_DBA	Database Administrator		

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

anonymous block completed		MIN_SALARY	MAX_SALARY
JOB_ID	JOB_TITLE		
IT_DBA	Data Administrator		

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

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

3. Create a procedure called DEL_JOB to delete a job from the JOBS table.
- Create a procedure called DEL_JOB to delete a job. Include the necessary exception-handling code if no job is deleted.
 - Invoke the procedure using the job ID IT_DBA. Query the JOBS table and view the results.

```
anonymous block completed  
no rows selected
```

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

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

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

```
v_salary  
----  
8000  
  
v_job  
-----  
ST_MAN
```

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

Solution 2-1: Creating, Compiling, and Calling Procedures

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

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

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

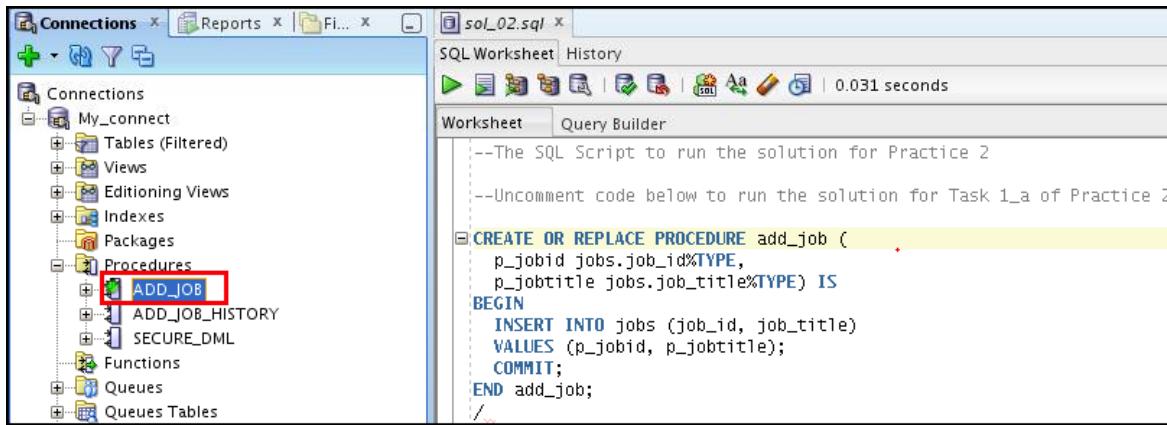
Open the `sol_02.sql` file in the `/home/oracle/labs/plpu/solns` directory. Uncomment and select the code for task 1_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "sol_02.sql". The toolbar includes icons for Run Script (highlighted), Save, Undo, Redo, Copy, Paste, Find, and others. The connection dropdown says "MyDBConnection". The main area has tabs for "Worksheet" and "Query Builder", with "Worksheet" selected. The code in the worksheet pane is:

```
4
5 CREATE OR REPLACE PROCEDURE add_job (
6   p_jobid jobs.job_id%TYPE,
7   p_jobtitle jobs.job_title%TYPE) IS
8 BEGIN
9   INSERT INTO jobs (job_id, job_title)
10  VALUES (p_jobid, p_jobtitle);
11  COMMIT;
12 END add_job;
13 /
```

The "Script Output" pane at the bottom shows the message "PROCEDURE ADD_JOB compiled" and "Task completed in 0.025 seconds".

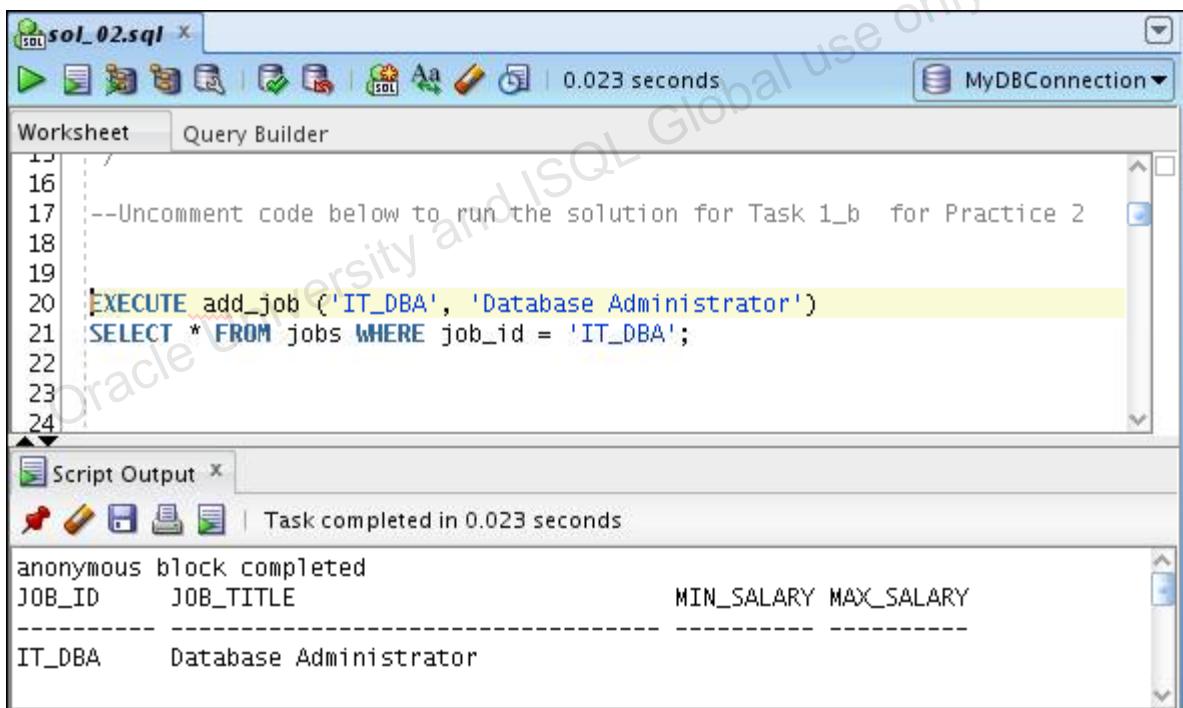
To view the newly created procedure, click the Procedures node in the Object Navigator. If the newly created procedure is not displayed, right-click the Procedures node, and then select Refresh from the shortcut menu. The new procedure is displayed as follows:



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

Execute the code for Task 1_b from `sol_02.sql` script. The code and the result are displayed as follows:

Note: Be sure to comment the previous code before uncommenting the next set of code.



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

Run the code for Task 1_c from `sol_02.sql` script. The code and the result are displayed as follows:

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

The screenshot shows a SQL Worksheet window titled 'sol_02.sql'. The code in the worksheet is:

```
26 --Uncomment code below to run the solution for Task 1_c  for Practice 2
27
28
29 EXECUTE add_job ('ST_MAN', 'Stock Manager')
30
31
```

The 'Script Output' pane shows the following error message:

```
Error starting at line 30 in command:
EXECUTE add_job ('ST_MAN', 'Stock Manager')
Error report:
ORA-00001: unique constraint (ORA61.JOB_ID_PK) violated
ORA-06512: at "ORA61.ADD_JOB", line 5
ORA-06512: at line 1
00001. 00000 -  "unique constraint (%s.%s) violated"
*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.
        For Trusted Oracle configured in DBMS MAC mode, you may see
        this message if a duplicate entry exists at a different level.
*Action: Either remove the unique restriction or do not insert the key.
```

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

Run the code for Task 2_a from the sol_02.sql script. The code and the result are displayed as follows:

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

- Connections:** MyDBConnection
- Script:** sol_02.sql
- Worksheet:** The code for the procedure is listed:

```
33 --Uncomment code below to run the solution for Task 2_a  for Practice 2
34
35
36
37 CREATE OR REPLACE PROCEDURE upd_job (
38   p_jobid IN jobs.job_id%TYPE,
39   p_jobtitle IN jobs.job_title%TYPE) IS
40 BEGIN
41   UPDATE jobs
42     SET job_title = p_jobtitle
43   WHERE job_id = p_jobid;
44   IF SQL%NOTFOUND THEN
45     RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
46   END IF;
47 END upd_job;
48 /
49
50 */
```
- Output:** Script Output shows the message "PROCEDURE UPD_JOB compiled" and "Task completed in 0.049 seconds".

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

Run the code for Task 2_b from sol_02.sql script. The code and the result are displayed as follows:

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

- Connections:** MyDBConnection
- Script:** sol_02.sql
- Worksheet:** The code executed is:

```
1 EXECUTE upd_job ('IT_DBA', 'Data Administrator')
2 SELECT * FROM jobs WHERE job_id = 'IT_DBA';
```
- Output:** Script Output shows "anonymous block completed". The Query Result shows the following data:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		

Query Result 1 shows the same data.

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

Run the code for Task 2_c from sol_02.sql script. The code and the result are displayed as follows:

```
MyDBConnection X sol_02.sql X
SQL Worksheet History
Worksheet Query Builder
59
60
61
62 --Uncomment code below to run the solution for Task 2_c for Practice 2
63
64
65
66 EXECUTE upd_job ('IT_WEB', 'Web Master')
67 SELECT * FROM jobs WHERE job_id = 'IT_WEB';
68
69
70
```

Script Output X Query Result X

SQL | All Rows Fetched: 0 in 0.003 seconds

JOB_ID	JOB_TITLE	MIN_SA...	MAX_SA...
--------	-----------	-----------	-----------

3. Create a procedure called DEL_JOB to delete a job from the JOBS table.
 - a. Create a procedure called DEL_JOB to delete a job. Include the necessary exception-handling code if no job is deleted.

Run the code for Task 3_a from sol_02.sql script. The code and the result are displayed as follows:

```
SQL Worksheet History
Worksheet Query Builder
70
71 --Uncomment code below to run the solution for Task 3_a for Practice 2
72
73 CREATE OR REPLACE PROCEDURE del_job (p_jobid jobs.job_id%TYPE) IS
74 BEGIN
75   DELETE FROM jobs
76   WHERE job_id = p_jobid;
77   IF SQL%NOTFOUND THEN
78     RAISE_APPLICATION_ERROR(-20203, 'No jobs deleted.');
79   END IF;
80 END DEL_JOB;
81 /
```

Script Output X

Task completed in 0.034 seconds

PROCEDURE DEL_JOB compiled

- b. To invoke the procedure and then query the JOBS table, uncomment and select the code under task 3_b in the /home/oracle/labs/plpu/solns/sol_02.sql script. Click the Run Script icon (or press F9) icon on the SQL Worksheet toolbar to invoke the procedure. Click the Query Result tab to see the code and the result displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "MyDBConnection x sol_02.sql x". The main area is a "Worksheet" tab showing the following SQL code:

```
86
87 --Uncomment code below to run the solution for Task 3_b for Practice 2
88
89
90 EXECUTE del_job ('IT_DBA')
91 SELECT * FROM jobs WHERE job_id = 'IT_DBA';
92
93
```

Below the worksheet, there is a "Script Output" tab showing the results of the query:

```
All Rows Fetched: 0 in 0.001 seconds
```

At the bottom, there is a toolbar with icons for Script, Output, and SQL, and a results grid showing columns: JOB_ID, JOB_TITLE, MIN_SA..., MAX_SA... with no data present.

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

To invoke the procedure and then query the JOBS table, uncomment and select the code under task 3_c in the /home/oracle/labs/plpu/solns/sol_02.sql script. Click the Run Script (F5) icon on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The title bar says "MyDBConnection X sol_02.sql X". The toolbar includes icons for Run, Save, Undo, Redo, and others. The status bar at the top right says "1.94599998 seconds". Below the toolbar, tabs for "Worksheet" and "Query Builder" are visible, with "Worksheet" selected. The main workspace contains the following SQL code:

```
95
96
97 --Uncomment code below to run the solution for Task 3_c for Practice 2
98
99 EXECUTE del_job ('IT_WEB')
100
101
102
```

Below the workspace is a "Script Output" window with the following error message:

```
Error starting at line 99 in command:
EXECUTE del_job ('IT_WEB')
Error report:
ORA-20203: No jobs deleted.
ORA-06512: at "ORA61.DEL_JOB", line 6
ORA-06512: at line 1
```

4. Create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.
- Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Remove syntax errors, if any, and then recompile the code.
Uncomment and select the code for Task 4_a from the sol_02.sql script. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The top menu bar includes 'SQL Worksheet' and 'History'. Below the menu is a toolbar with various icons. The main workspace is titled 'Worksheet' and contains the following PL/SQL code:

```
99  
100 --Uncomment code below to run the solution for Task 4_a for Practice 2  
101  
102 CREATE OR REPLACE PROCEDURE get_employee  
103     (p_empid IN employees.employee_id%TYPE,  
104      p_sal    OUT employees.salary%TYPE,  
105      p_job    OUT employees.job_id%TYPE) IS  
106 BEGIN  
107     SELECT salary, job_id  
108     INTO p_sal, p_job  
109     FROM employees  
110     WHERE employee_id = p_empid;  
111 END get_employee;  
112 /  
113
```

The code at line 112 is highlighted with a yellow background. Below the workspace is a 'Script Output' window with the message 'Task completed in 0.048 seconds' and the output 'PROCEDURE GET_EMPLOYEE compiled'.

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



- b. Execute the procedure using host variables for the two OUT parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120. **Uncomment and select the code under Task 4_b from sol_02.sql script. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:**

The screenshot shows the Oracle SQL Worksheet interface. The top window is titled 'sol_02.sql' and contains the following PL/SQL code:

```
--Uncomment code below to run the solution for Task 4_b for Practice 2

VARIABLE v_salary NUMBER
VARIABLE v_job    VARCHAR2(15)
EXECUTE get_employee(120, :v_salary, :v_job)
PRINT v_salary v_job
```

The bottom window is titled 'Script Output' and displays the results of the execution:

```
anonymous block completed
V_SALARY
-----
8000

V_JOB
-----
ST_MAN
```

A watermark reading "Oracle University and ISQL Global use only." is diagonally across the bottom of the screenshot.

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

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

The screenshot shows the Oracle SQL Developer interface. In the top-left corner, there's a connection named "MyDBConnection" and a tab labeled "sol_02.sql". Below the tabs, the "SQL Worksheet" tab is selected. The worksheet pane contains the following PL/SQL code:

```
--Uncomment code below to run the solution for Task 4_c for Practice 2
VARIABLE v_salary NUMBER
VARIABLE v_job    VARCHAR2(15)
EXECUTE get_employee(300, :v_salary, :v_job)
```

In the bottom-right pane, titled "Script Output", the results of the execution are shown. It indicates a task completed in 0.991 seconds and then displays an error message:

```
Error starting at line 131 in command:
EXECUTE get_employee(300, :v_salary, :v_job)
Error report:
ORA-01403: no data found
ORA-06512: at "ORA61.GET_EMPLOYEE", line 6
ORA-06512: at line 1
01403. 00000 -  "no data found"
*Cause:
*Action:
```

Practices for Lesson 3: Creating Functions and Debugging Subprograms

Chapter 3

Practices for Lesson 3

Overview

In practice 3-1, you create, compile, and use the following:

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

In practice 3-2, you are introduced to the basic functionality of the SQL Developer debugger:

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_03.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 3-1: Creating Functions

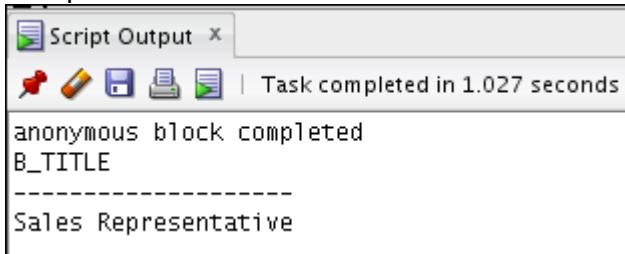
Overview

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

Note: Execute `cleanup_03.sql` from
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

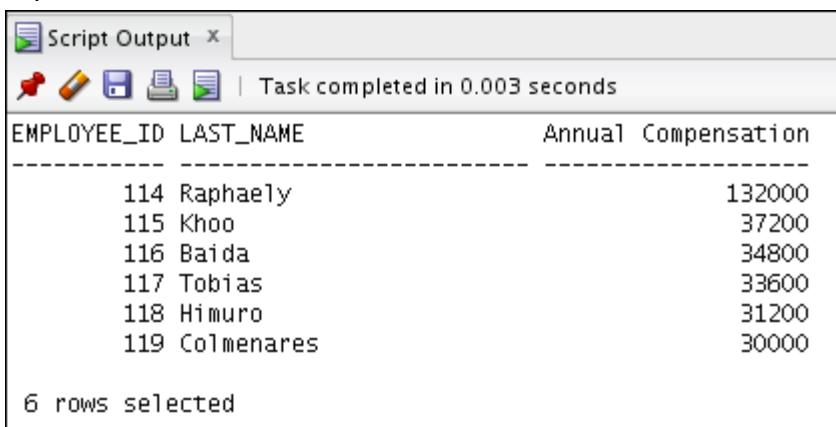
1. Create and invoke the `GET_JOB` function to return a job title.
 - a. Create and compile a function called `GET_JOB` to return a job title.
 - b. Create a `VARCHAR2` host variable called `b_title`, allowing a length of 35 characters. Invoke the function with job ID `SA REP` to return the value in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



```
Script Output X
anonymous block completed
B_TITLE
-----
Sales Representative
```

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

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



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

6 rows selected

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

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

Solution 3-1: Creating Functions

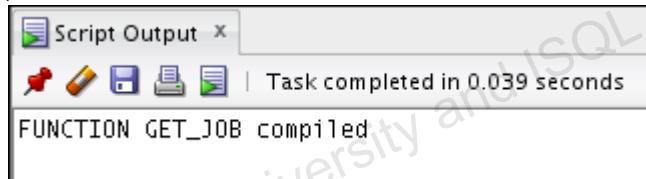
In this practice, you create, compile, and use stored functions and a procedure.

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

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

Open the /home/oracle/labs/plpu/sols/sol_03.sql script. Uncomment and select the code under Task 1_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

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



Note: If you encounter an “access control list (ACL) error” while executing this step, please perform the following workaround:

- a. Open SQL*Plus.
- b. Connect as SYSDBA.
- c. Execute the following code:

```
BEGIN
  DBMS_NETWORK_ACL_ADMIN.APPEND_HOST_ACE(
    host => '127.0.0.1',
    ace => xs$ace_type(privilege_list => xs$name_list('jdwp'),
      principal_name => 'ora61',
      principal_type => xs_acl.ptype_db));
END;
/
```

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

Uncomment and select the code under Task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

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

```
anonymous block completed
B_TITLE
-----
Sales Representative
```

Note: Be sure to add the comments back to the previous code before executing the next set of code. Alternatively, you can select the complete code before using the Run Script icon (or press F5) to execute it.

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

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

Uncomment and select the code under Task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

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

```
FUNCTION GET_ANNUAL_COMP compiled
```

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

Uncomment and select the code under Task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the function. The code and the result are displayed as follows:

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

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

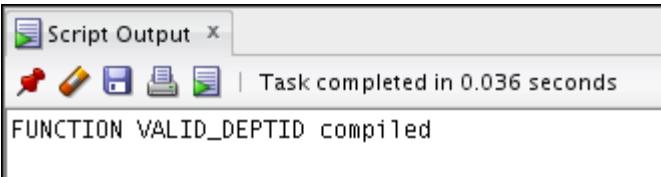
6 rows selected

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

Uncomment and select the code under Task 3_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create the function. The code and the result are displayed as follows:

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

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



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

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

Uncomment and select the code under Task 3_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. The code and the result are displayed as follows:

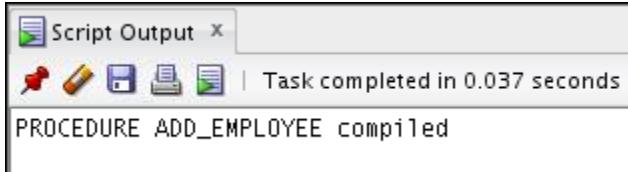
```
CREATE OR REPLACE PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name  employees.last_name%TYPE,
    p_email      employees.email%TYPE,
    p_job        employees.job_id%TYPE          DEFAULT 'SA_REP',
    p_mgr        employees.manager_id%TYPE       DEFAULT 145,
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

```
p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm, p_deptid);  
ELSE  
    RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try  
again.');
```

END IF;

END add_employee;

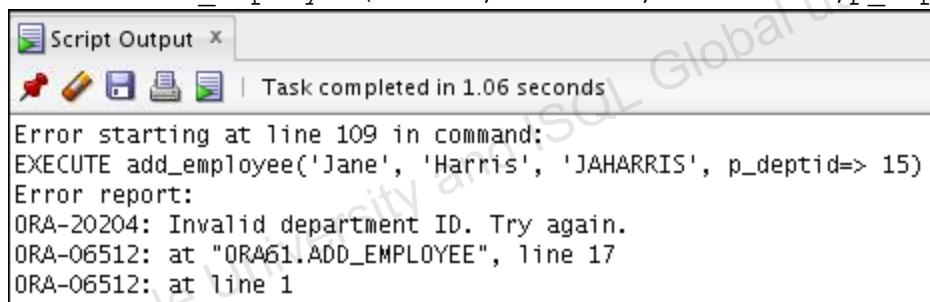
/



- c. Call ADD_EMPLOYEE for the name 'Jane Harris' in department 15, leaving other parameters with their default values. What is the result?

Uncomment and select the code under Task 3_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

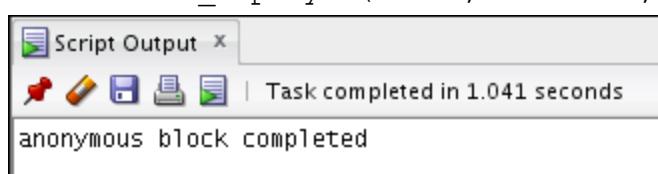
```
EXECUTE add_employee('Jane','Harris','JAHARRIS',p_deptid=> 15)
```



- d. Add another employee named Joe Harris in department 80, leaving the remaining parameters with their default values. What is the result?

Uncomment and select the code under Task 3_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the procedure. The code and the result are displayed as follows:

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



Practice 3-2: Introduction to the SQL Developer Debugger

Overview

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

Tasks

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

Solution 3-2: Introduction to the SQL Developer Debugger

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

1. Enable SERVEROUTPUT.

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

```
SET SERVEROUTPUT ON
```

2. Run the solution under Task 2 of practice 3-2 to create the emp_list procedure. Examine the code of the procedure and compile the procedure. Why do you get the compiler error?

Uncomment and select the code under Task 2 of Practice 3-2. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. The codex and the result are displayed as follows:

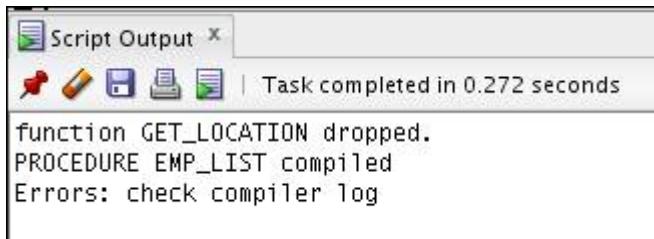
```
DROP FUNCTION get_location;

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

```

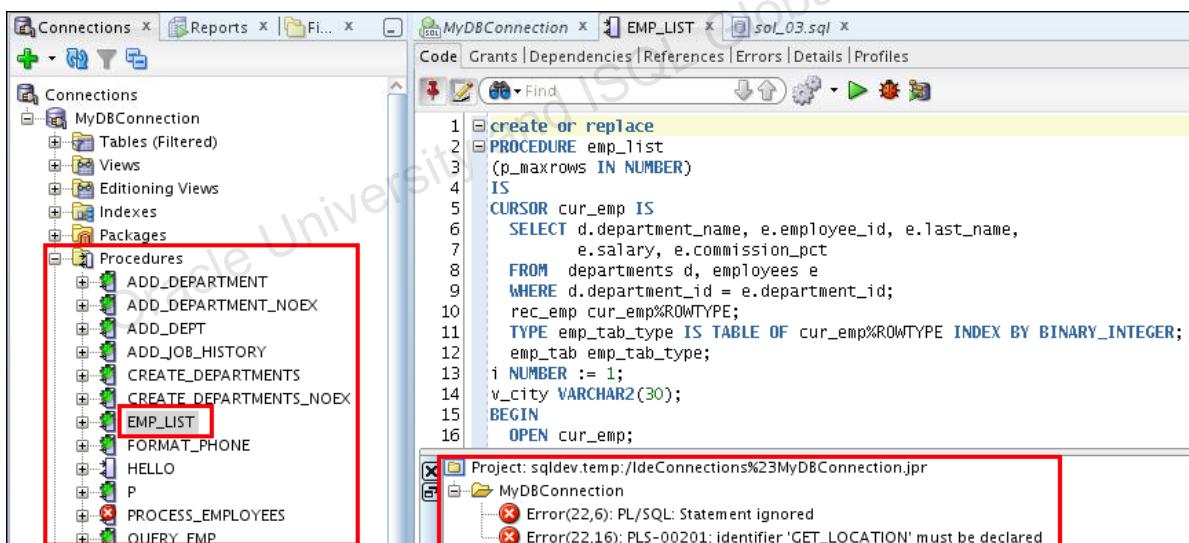
FOR j IN REVERSE 1..i LOOP
    DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
END LOOP;
END emp_list;
/

```



Note: You may expect an error message in the output if the `get_location` function does not exist. If the function exists, you get the above output.

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



- Run the solution under Task 3 of practice 3-2 to create the `get_location` function. Examine the code of the function, compile the function, and then correct any errors, if any.

Uncomment and select the code under Task 3 of Practice 3-2. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. The codex and the result are displayed as follows:

```

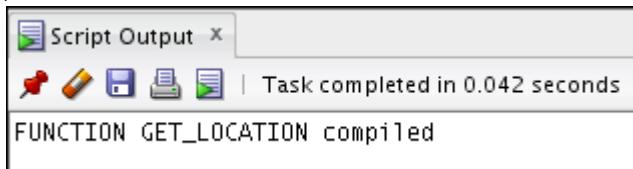
CREATE OR REPLACE FUNCTION get_location
( p_deptname IN VARCHAR2 ) RETURN VARCHAR2
AS
    v_loc_id NUMBER;

```

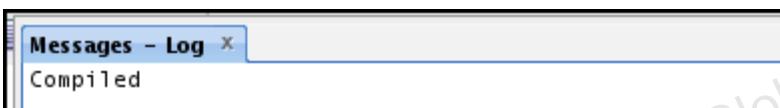
```

    v_city      VARCHAR2(30);
BEGIN
    SELECT d.location_id, l.city INTO v_loc_id, v_city
    FROM departments d, locations l
    WHERE upper(department_name) = upper(p_deptname)
    and d.location_id = l.location_id;
    RETURN v_city;
END GET_LOCATION;
/

```



4. Recompile the emp_list procedure. The procedure should compile successfully.
To recompile the procedure, right-click the procedure's name, and then select Compile from the shortcut menu.



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

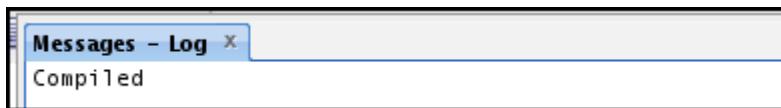
6. Add four breakpoints to the emp_list procedure to the following lines of code:
 - a. OPEN cur_emp;
 - b. WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
 - c. v_city := get_location (rec_emp.department_name);
 - d. CLOSE cur_emp;

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

```
1  create or replace
2  PROCEDURE emp_list
3      (p_maxrows IN NUMBER)
4  IS
5      CURSOR cur_emp IS
6          SELECT d.department_name, e.employee_id, e.last_name,
7              e.salary, e.commission_pct
8          FROM departments d, employees e
9          WHERE d.department_id = e.department_id;
10     rec_emp cur_emp%ROWTYPE;
11     TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY
12     emp_tab emp_tab_type;
13     i NUMBER := 1;
14     v_city VARCHAR2(30);
15
16     BEGIN
17         OPEN cur_emp;
18         FETCH cur_emp INTO rec_emp;
19         emp_tab(i) := rec_emp;
20         WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
21             i := i + 1;
22             FETCH cur_emp INTO rec_emp;
23             emp_tab(i) := rec_emp;
24             v_city := get_location (rec_emp.department_name);
25             dbms_output.put_line('Employee ' || rec_emp.last_name ||
26                                 ' works in ' || v_city );
27         END LOOP;
28         CLOSE cur_emp;
29         FOR j IN REVERSE 1..i LOOP
30             DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
31         END LOOP;
32     END emp_list;
```

7. Compile the `emp_list` procedure for debugging.

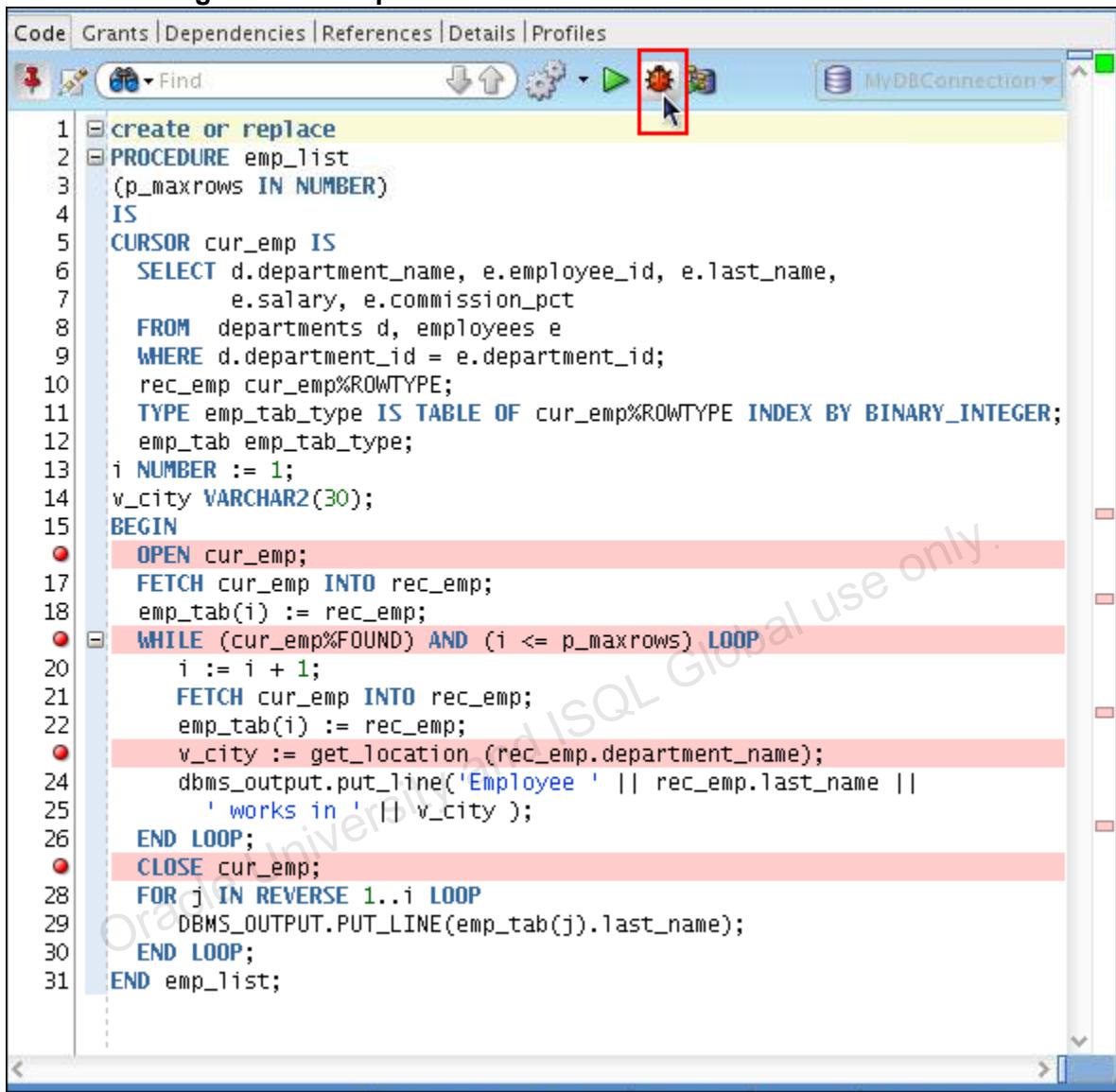
Click the “Compile for Debug” icon on the procedure’s toolbar and you get the output as shown below:



Note: If you get any warnings, it is expected. The two warnings are because the `PLSQL_DEBUG` parameter was deprecated in Oracle Database 11g, while SQL Developer is still using that parameter.

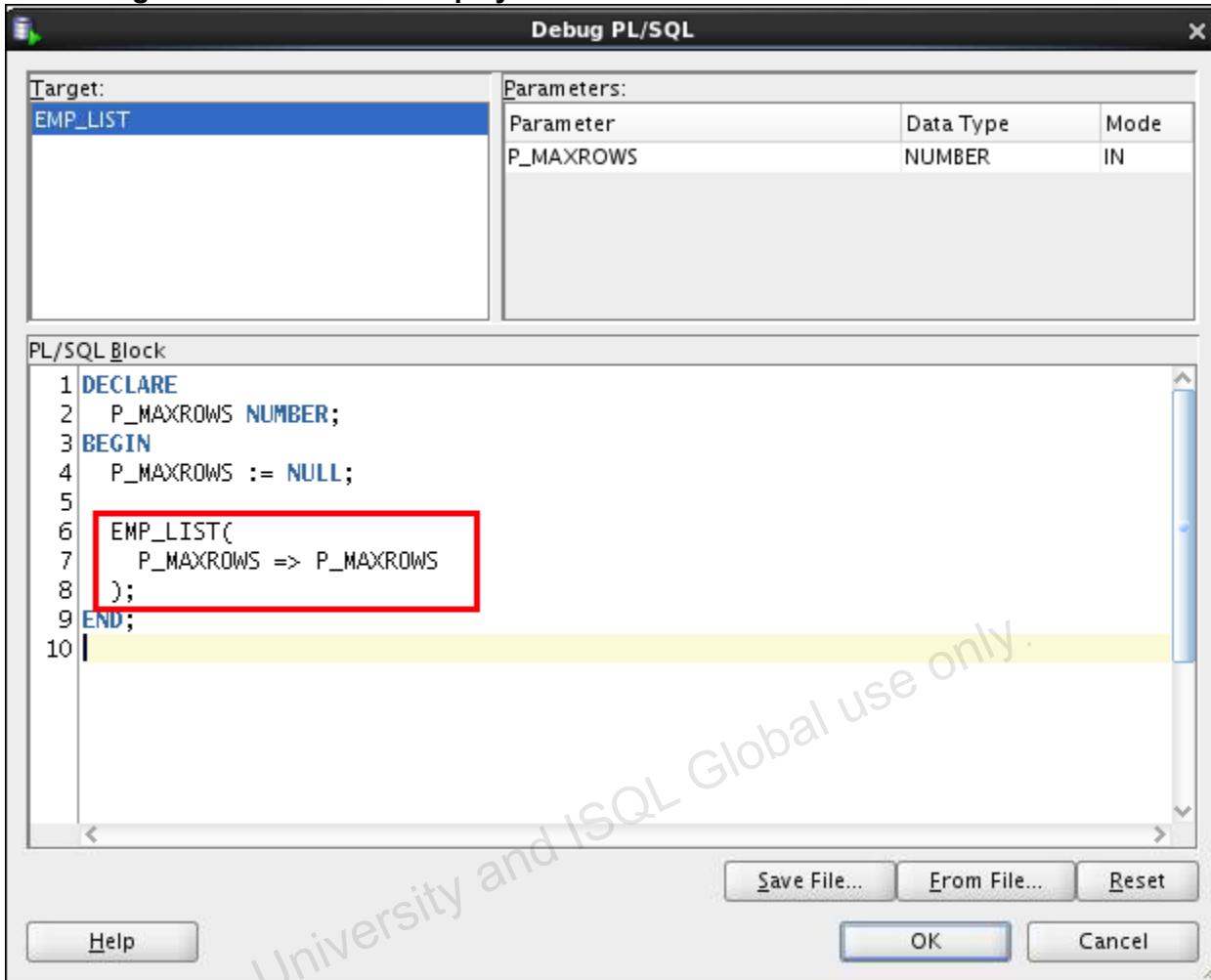
8. Debug the procedure.

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



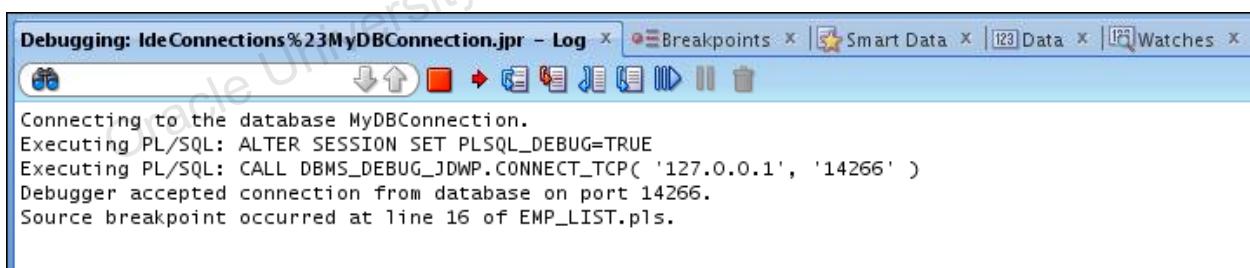
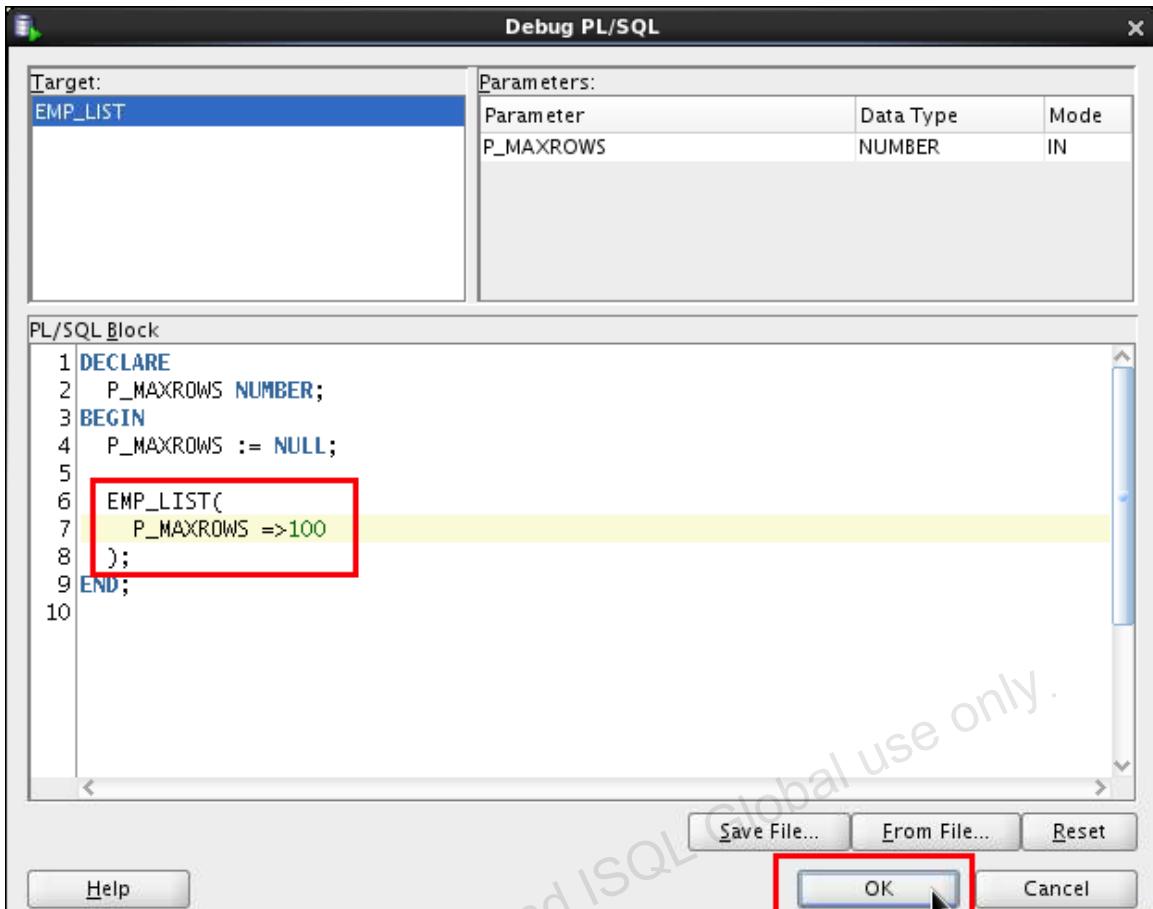
```
Code Grants Dependencies References Details Profiles
Find MyDBConnection
1 create or replace
2 PROCEDURE emp_list
3 (p_maxrows IN NUMBER)
4 IS
5 CURSOR cur_emp IS
6   SELECT d.department_name, e.employee_id, e.last_name,
7         e.salary, e.commission_pct
8   FROM departments d, employees e
9   WHERE d.department_id = e.department_id;
10  rec_emp cur_emp%ROWTYPE;
11  TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
12  emp_tab emp_tab_type;
13  i NUMBER := 1;
14  v_city VARCHAR2(30);
15 BEGIN
16  OPEN cur_emp;
17  FETCH cur_emp INTO rec_emp;
18  emp_tab(i) := rec_emp;
19  WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20    i := i + 1;
21    FETCH cur_emp INTO rec_emp;
22    emp_tab(i) := rec_emp;
23    v_city := get_location(rec_emp.department_name);
24    dbms_output.put_line('Employee ' || rec_emp.last_name ||
25      ' works in ' || v_city );
26  END LOOP;
27  CLOSE cur_emp;
28  FOR j IN REVERSE 1..i LOOP
29    DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30  END LOOP;
31 END emp_list;
```

The Debug PL/SQL window is displayed as follows:



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

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



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

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

The screenshot shows the Oracle Database IDE's Data tab. The variables listed are P_MAXROWS, REC_EMP, and EMP_TAB. P_MAXROWS has a value of 100 and a type of NUMBER. REC_EMP is expanded to show fields: COMMISSION_PCT (NULL), DEPARTMENT_NAME (NULL), EMPLOYEE_ID (NULL), LAST_NAME (NULL), and SALARY (NULL). It also has a Rowtype value. EMP_TAB is shown as an indexed table with one row (I) containing a value of 1 and a type of NUMBER. V_CITY is NULL and has a type of VARCHAR2(30). The status bar at the bottom says "Debugging".

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

11. Use the Step Into debug option to step in to each line of code in `emp_list` and go through the while loop only once.
Press F7 to step into the code only once.
12. Examine the value of the variables on the Data tab. What are the values assigned to `REC_EMP` and `EMP_TAB`?

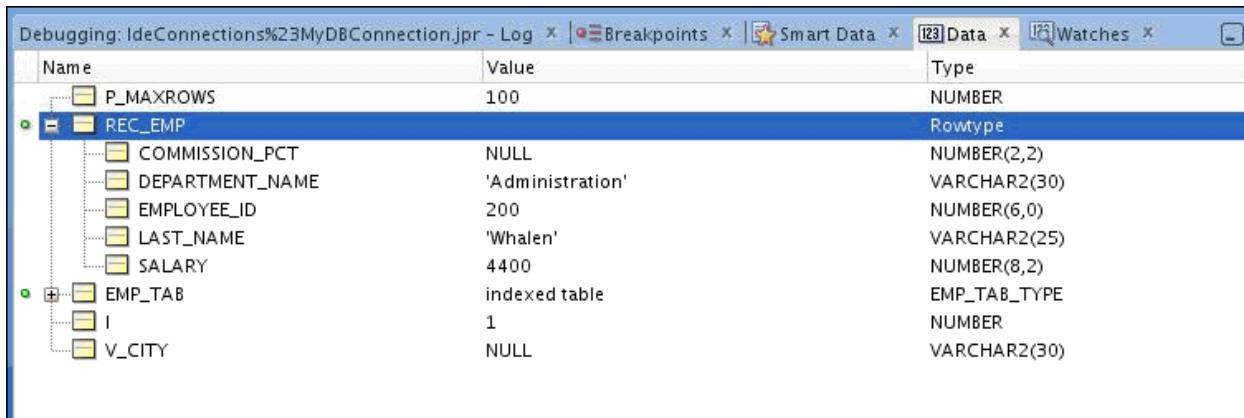
Note that when the line `FETCH cur_emp INTO rec_emp;` is executed, `rec_emp` is initialized as shown below:

The screenshot shows the Oracle Database IDE's Data tab after the fetch operation. The variables are the same as before, but their values have changed. P_MAXROWS is still 100. REC_EMP now contains data: DEPARTMENT_NAME is 'Administration', EMPLOYEE_ID is 200, LAST_NAME is 'Whalen', and SALARY is 4400. It still has a Rowtype value. EMP_TAB is expanded to show a single row _values with a value of 1 and a type of EMP_TAB_TYPE element[0]. I is 1 and V_CITY is NULL. The status bar at the bottom says "Debugging".

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

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

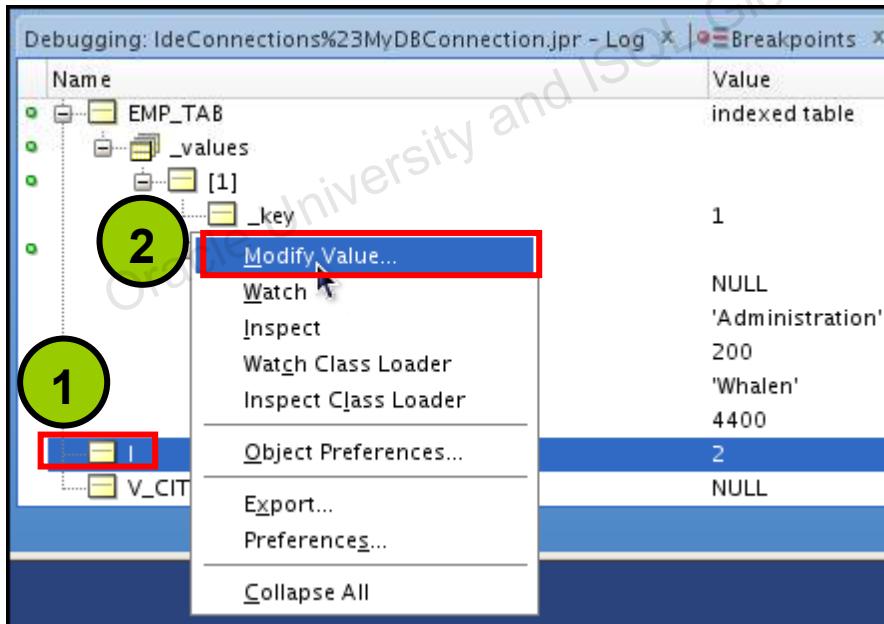
When the line `emp_tab(i) := rec_emp;` is executed, `emp_tab` is initialized to `rec_emp` as shown below:

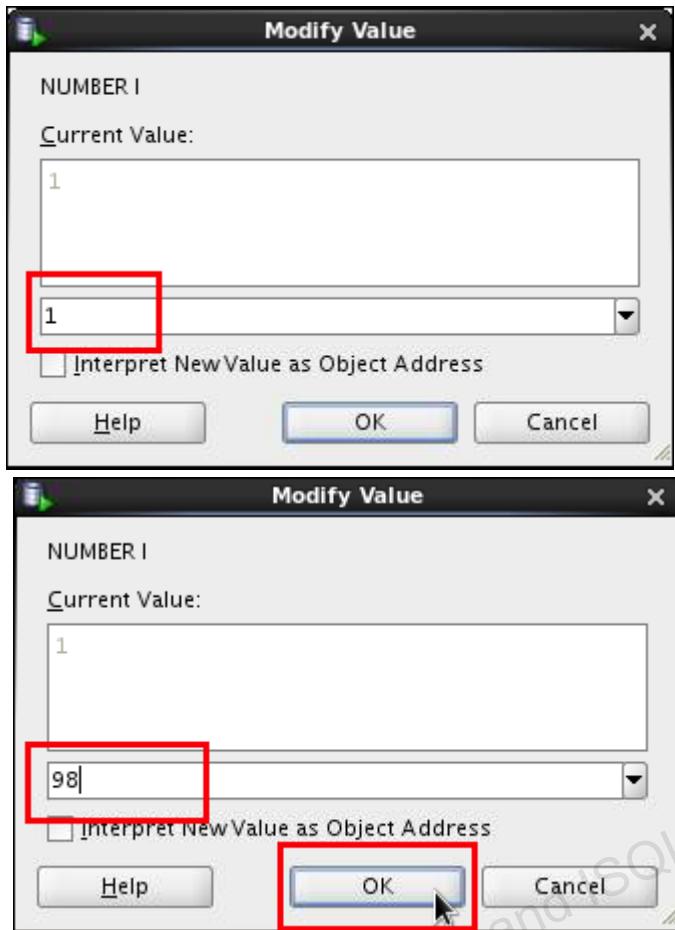


Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
COMMISSION_PCT	NULL	NUMBER(2,2)
DEPARTMENT_NAME	'Administration'	VARCHAR2(30)
EMPLOYEE_ID	200	NUMBER(6,0)
LAST_NAME	'Whalen'	VARCHAR2(25)
SALARY	4400	NUMBER(8,2)
EMP_TAB	indexed table	EMP_TAB_TYPE
I	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

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

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





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

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

```
Exception breakpoint occurred at line 29 of EMP_LIST.pls.  
$0oracle.EXCEPTION_ORA_1403:  
ORA-01403: no data found  
ORA-06512: at "ORA61.EMP_LIST", line 28  
ORA-06512: at line 6  
Executing PL/SQL: CALL DBMS_DEBUG_JDWP.DISCONNECT()  
Employee Hartstein works in Toronto  
Employee Fay works in Toronto  
Employee Raphaelly works in Seattle  
Employee Khoo works in Seattle  
Khoo  
Raphaelly  
Fay  
Hartstein  
Process exited.  
Disconnecting from the database MyDBConnection.  
Debugger disconnected from database.
```

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

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

Practices for Lesson 4: Creating Packages

Chapter 4

Practices for Lesson 4

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_04.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 4-1: Creating and Using Packages

Overview

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

Note: Execute `cleanup_04.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

1. Create a package specification and body called `JOB_PKG`, containing a copy of your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures as well as your `GET_JOB` function.
Note: Use the code from your previously saved procedures and functions when creating the package. You can copy the code in a procedure or function, and then paste the code into the appropriate section of the package.
 - a. Create the package specification including the procedures and function headings as public constructs.
 - b. Create the package body with the implementations for each of the subprograms.
 - c. Delete the following stand-alone procedures and function you just packaged using the Procedures and Functions nodes in the Object Navigation tree:
 - 1) The `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures
 - 2) The `GET_JOB` function
 - d. Invoke your `ADD_JOB` package procedure by passing the values `IT_SYSAN` and `SYSTEMS ANALYST` as parameters.
 - e. Query the `JOBS` table to see the result.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst	10000	15000

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

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

Solution 4-1: Creating and Using Packages

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

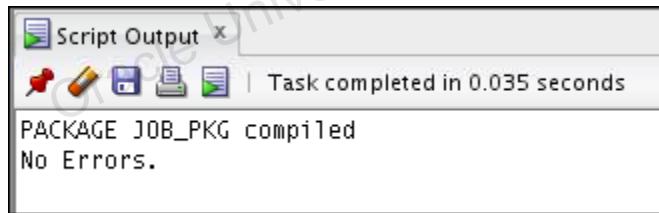
1. Create a package specification and body called `JOB_PKG`, containing a copy of your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures as well as your `GET_JOB` function.

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

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

Open the `/home/oracle/labs/plpu/solns/sol_04.sql` script. Uncomment and select the code under Task 1_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package specification. The code and the result are displayed as follows:

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



- b. Create the package body with the implementations for each of the subprograms. **Uncomment and select the code under Task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package body. The code and the result are displayed as follows:**

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

```
END add_job;

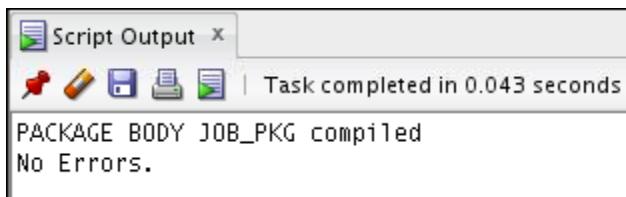
PROCEDURE del_job (p_jobid jobs.job_id%TYPE) IS
BEGIN
    DELETE FROM jobs
    WHERE job_id = p_jobid;
    IF SQL%NOTFOUND THEN
        RAISE_APPLICATION_ERROR(-20203, 'No jobs deleted.');
    END IF;
END DEL_JOB;

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

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

END job_pkg;
/
```

SHOW ERRORS



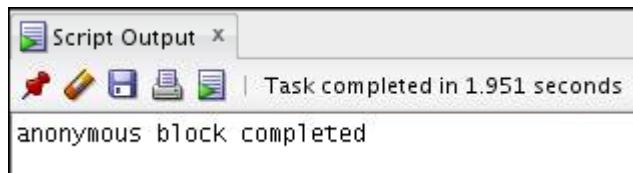
- c. Delete the following stand-alone procedures and functions you just packaged by using the Procedures and Functions nodes in the Object Navigation tree:
- 1) The ADD_JOB, UPD_JOB, and DEL_JOB procedures
 - 2) The GET_JOB function

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

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

Uncomment and select the code under Task 1_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

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



- e. Query the JOBS table to see the result.

Uncomment and select the code under Task 1_e. Click the Run Script icon (or press F5) or the Execute Statement (or press F9) on the SQL Worksheet toolbar to query the JOBS table. The code and the result are displayed as follows:

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

Script Output			
Task completed in 0.003 seconds			
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst		

2. Create and invoke a package that contains private and public constructs.
- a. Create a package specification and a package body called EMP_PKG that contains the following procedures and function that you created earlier:
 - 1) ADD_EMPLOYEE procedure as a public construct
 - 2) GET_EMPLOYEE procedure as a public construct
 - 3) VALID_DEPTID function as a private construct

Uncomment and select the code under Task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

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

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

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

```

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

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

```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. It contains the following text:

```

Script Output x
PACKAGE EMP_PKG compiled
No Errors.
PACKAGE BODY EMP_PKG compiled
No Errors.

```

The status bar at the bottom of the window indicates: 'Task completed in 0.07 seconds'.

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

Uncomment and select the code under Task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

Note: You must complete step 3-2-a before performing this step. If you didn't complete step 3-2-a, run the code under Task 2_a first.

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

The screenshot shows the 'Script Output' window with the following content:

```
Script Output x
Task completed in 1.03 seconds
Error starting at line 165 in command:
EXECUTE emp_pkg.add_employee('Jane', 'Harris', 'JAHARRIS', p_deptid => 15)
Error report:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA61.EMP_PKG", line 32
ORA-06512: at line 1
```

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

Uncomment and select the code under Task 2_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

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

The screenshot shows the 'Script Output' window with the following content:

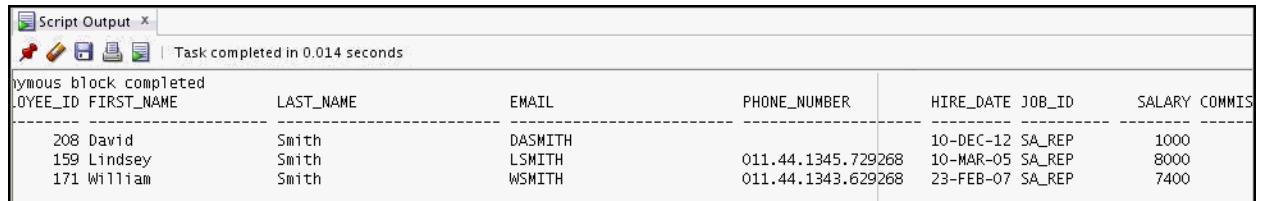
```
Script Output
Task completed in 0.125 seconds
anonymous block completed
```

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

Uncomment and select the code under Task 2_d. Click the Run Script icon (or press F5) or the Execute Statement icon (or press F9), while making sure the cursor is on any of the SELECT statement code, on the SQL Worksheet toolbar to query the EMPLOYEES table. The code and the result (Execute Statement icon) are displayed as follows:

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

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



A screenshot of the Oracle SQL Developer interface showing the 'Script Output' window. The window title is 'Script Output X'. It displays a message 'Task completed in 0.014 seconds' and a successful anonymous block completion message. Below this, there is a table output showing employee data from the 'EMP' table. The columns are: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, and COMMISSION_PCT. The data rows are:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
208	David	Smith	DASMITH		10-DEC-12	SA_REP	1000	
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-05	SA_REP	8000	
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-07	SA_REP	7400	

Practices for Lesson 5: Working with Packages

Chapter 5

Practices for Lesson 5

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_05.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 5-1: Working with Packages

Overview

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

Note: Execute `cleanup_05.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

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

- 1) Implement the first GET_EMPLOYEE function to query an employee using the employee's ID.
 - 2) Implement the second GET_EMPLOYEE function to use the equality operator on the value supplied in the p_family_name parameter.
 - d. Click Run Script to re-create and compile the package.
 - e. Add a utility procedure PRINT_EMPLOYEE to the EMP_PKG package as follows:
 - 1) The procedure accepts an EMPLOYEES%ROWTYPE as a parameter.
 - 2) The procedure displays the following for an employee on one line, using the DBMS_OUTPUT package:
 - department_id
 - employee_id
 - first_name
 - last_name
 - job_id
 - salary
 - f. Click the Run Script icon (or press F5) to create and compile the package.
 - g. Use an anonymous block to invoke the EMP_PKG.GET_EMPLOYEE function with an employee ID of 100 and family name of 'Joplin'. Use the PRINT_EMPLOYEE procedure to display the results for each row returned.
3. Because the company does not frequently change its departmental data, you can improve performance of your EMP_PKG by adding a public procedure, INIT_DEPARTMENTS, to populate a private PL/SQL table of valid department IDs. Modify the VALID_DEPTID function to use the private PL/SQL table contents to validate department ID values.

Note: The code under Task 3 contains the solution for steps a, b, and c.

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

```
PROCEDURE init_departments;
```

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


```
TYPE boolean_tab_type IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tab_type;
```
 - 2) Use the department_id column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of TRUE. Enter the INIT_DEPARTMENTS procedure declaration at the end of the package body (right after the print_employees procedure) as follows:


```
PROCEDURE init_departments IS
BEGIN
```

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

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

```
BEGIN
    init_departments;
END;
```
- d. Click the Run Script icon (or press F5) to create and compile the package.
4. Change the VALID_DEPTID validation processing function to use the private index-by table of department IDs.
 - a. Modify the VALID_DEPTID function to perform its validation by using the index-by table of department ID values. Click the Run Script icon (or press F5) to create the package. Compile the package.
 - b. Test your code by calling ADD_EMPLOYEE using the name James Bond in department 15. What happens?
 - c. Insert a new department. Specify 15 for the department ID and 'Security' for the department name. Commit and verify the changes.
 - d. Test your code again, by calling ADD_EMPLOYEE using the name James Bond in department 15. What happens?
 - e. Execute the EMP_PKG.INIT_DEPARTMENTS procedure to update the internal index-by table with the latest departmental data.
 - f. Test your code by calling ADD_EMPLOYEE by using the employee name James Bond, who works in department 15. What happens?
 - g. Delete employee James Bond and department 15 from their respective tables, commit the changes, and refresh the department data by invoking the EMP_PKG.INIT_DEPARTMENTS procedure. Make sure you enter SET SERVEROUTPUT ON first.
5. Reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.
 - Edit the package specification and reorganize subprograms alphabetically. Click Run Script to re-create the package specification. Compile the package specification. What happens?
 - Edit the package body and reorganize all subprograms alphabetically. Click Run Script to re-create the package specification. Re-compile the package specification. What happens?
 - Correct the compilation error using a forward declaration in the body for the appropriate subprogram reference. Click Run Script to re-create the package, and then recompile the package. What happens?

Solution 5-1: Working with Packages

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

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

Open the `/home/oracle/labs/plpu/sols/sol_05.sql` file. Uncomment and select the code under Task 1_a. The code is displayed as follows:

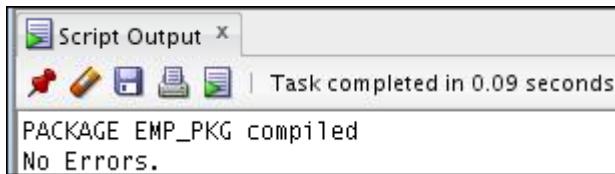
```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE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

  -- New overloaded add_employee

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

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

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



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

Uncomment and select the code under Task 1_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows (the newly added code is highlighted in bold face text in the code box below):

```

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

  -- New overloaded add_employee

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

  -- End of the spec of the new overloaded add_employee

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

```

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

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

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

```

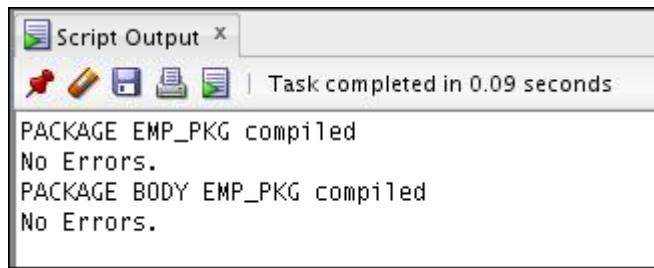
```
END add_employee;

-- New overloaded add_employee procedure

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

-- End declaration of the overloaded add_employee procedure

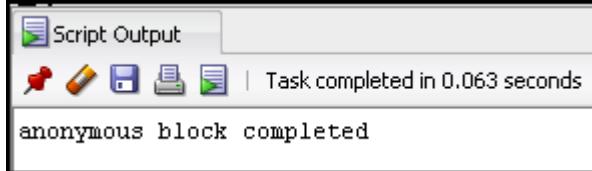
PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE) IS
BEGIN
    SELECT salary, job_id
    INTO p_sal, p_job
    FROM employees
    WHERE employee_id = p_empid;
END get_employee;
END emp_pkg;
/
SHOW ERRORS
```



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

Uncomment and select the code under Task 1_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedure. The code and the result are displayed as follows:

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



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

Uncomment and select the code under Task 1_e. Click anywhere on the SELECT statement, and then click the Execute Statement icon (or press F5) on the SQL Worksheet toolbar to execute the query. The code and the result are displayed as follows:

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

Script Output X										
Task completed in 0.005 seconds										
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
209	Samuel	Joplin	SJOPLIN		18-NOV-12	SA_REP	1000	0	145	30

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

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

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

Uncomment and select the code under Task 2_a.

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

```

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

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

-- New overloaded get_employees functions specs starts here:

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

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

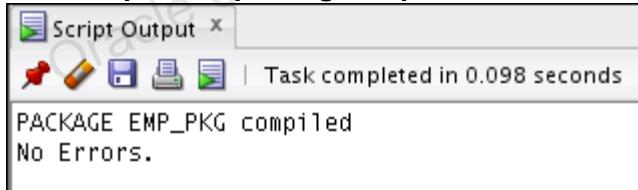
-- New overloaded get_employees functions specs ends here.

END emp_pkg;
/
SHOW ERRORS

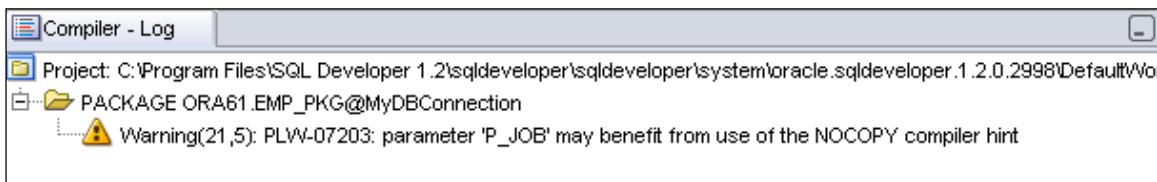
```

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

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



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



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

Uncomment and select the code under Task 2_c. The newly added functions are highlighted in the following code box.

```

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

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

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

    -- New overloaded get_employees functions specs starts here:

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

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

    -- New overloaded get_employees functions specs ends here.

END emp_pkg;

```

```

/
SHOW ERRORS

-- package body

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

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

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

```

```
        Try again.') ;
      END IF;
END add_employee;

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

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

-- New get_employee function declaration starts here

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

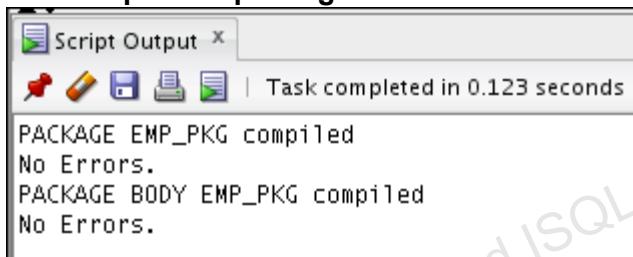
FUNCTION get_employee(p_family_name
employees.last_name%type)
  return employees%rowtype IS
  rec_emp employees%rowtype;
```

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

-- New overloaded get_employee function declaration ends here

END emp_pkg;
/
SHOW ERRORS
```

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



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

Uncomment and select the code under Task 2_e. The newly added code is highlighted in the following code box.

```
-- Package SPECIFICATION
```

```
CREATE OR REPLACE PACKAGE emp_pkg IS
PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
```

```

p_job employees.job_id%TYPE DEFAULT 'SA_REP',
p_mgr employees.manager_id%TYPE DEFAULT 145,
p_sal employees.salary%TYPE DEFAULT 1000,
p_commc employees.commission_pct%TYPE DEFAULT 0,
p_deptid employees.department_id%TYPE DEFAULT 30);

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

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

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

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

-- New print_employee print_employee procedure spec

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

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

```

```

WHEN NO_DATA_FOUND THEN
  RETURN FALSE;
END valid_deptid;

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

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

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

```

```

BEGIN
  SELECT salary, job_id
  INTO p_sal, p_job
  FROM employees
  WHERE employee_id = p.empid;
END get_employee;

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

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

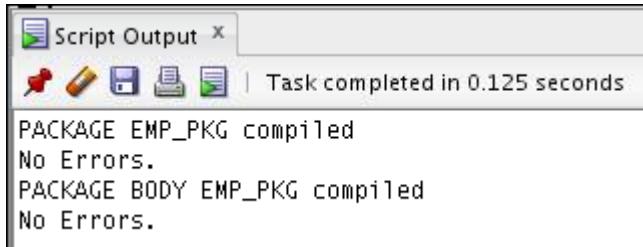
-- New print_employees procedure declaration.

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

END emp_pkg;
/
SHOW ERRORS

```

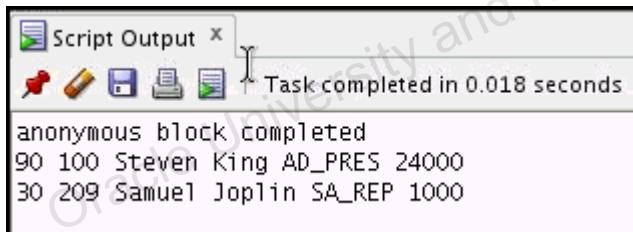
- f. Click the Run Script icon (or press F5) to create and compile the package.
Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create and compile the package.



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

Uncomment and select the code under Task 2_g.

```
SET SERVEROUTPUT ON
BEGIN
    emp_pkg.print_employee(emp_pkg.get_employee(100));
    emp_pkg.print_employee(emp_pkg.get_employee('Joplin'));
END;
/
```



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

Note: The code under Task 3 contains the solutions for steps a, b, and c.

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

```
PROCEDURE init_departments;
```

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

- 1) Declare the valid_departments variable and its type definition

boolean_tab_type before all procedures in the body. Enter the following at the beginning of the package body:

```

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

```

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

```

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

```

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

```

BEGIN
    init_departments;
END;

```

Uncomment and select the code under Task 3. The newly added code is highlighted in the following code box:

```
-- Package SPECIFICATION
```

```

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

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

PROCEDURE get_employee(
    p.empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,

```

```
p_job OUT employees.job_id%TYPE);  
  
FUNCTION get_employee(p_emp_id employees.employee_id%type)  
    return employees%rowtype;  
  
FUNCTION get_employee(p_family_name employees.last_name%type)  
    return employees%rowtype;  
  
-- New procedure init_departments spec  
PROCEDURE init_departments;  
  
PROCEDURE print_employee(p_rec_emp employees%rowtype);  
  
END emp_pkg;  
/  
SHOW ERRORS  
  
-- Package BODY  
  
CREATE OR REPLACE PACKAGE BODY emp_pkg IS  
  
    -- New type  
    TYPE boolean_tab_type IS TABLE OF BOOLEAN  
        INDEX BY BINARY_INTEGER;  
    valid_departments boolean_tab_type;  
  
    FUNCTION valid_deptid(p_deptid IN  
        departments.department_id%TYPE) RETURN BOOLEAN IS  
        v_dummy PLS_INTEGER;  
    BEGIN  
        SELECT 1  
        INTO v_dummy  
        FROM departments  
        WHERE department_id = p_deptid;  
        RETURN TRUE;  
    EXCEPTION  
        WHEN NO_DATA_FOUND THEN  
            RETURN FALSE;  
    END valid_deptid;  
  
    PROCEDURE add_employee(  
        p_emp_id IN employees.employee_id%TYPE,  
        p_job IN employees.job_id%TYPE,  
        p_salary IN employees.salary%TYPE,  
        p_hire_date IN employees.hire_date%TYPE,  
        p_department_id IN departments.department_id%TYPE,  
        p_manager_id IN employees.employee_id%TYPE,  
        p_job_id IN jobs.job_id%TYPE);  
    BEGIN  
        employees%ROWTYPE e;  
        e.employee_id := p_emp_id;  
        e.last_name := p_family_name;  
        e.first_name := p_given_name;  
        e.job_id := p_job;  
        e.hire_date := p_hire_date;  
        e.salary := p_salary;  
        e.department_id := p_department_id;  
        e.manager_id := p_manager_id;  
        e.job_id := p_job_id;  
        employees%append(e);  
    END add_employee;  
  
    PROCEDURE update_employee(p_employee_id IN employees.employee_id%TYPE,  
        p_new_first_name IN employees.first_name%TYPE,  
        p_new_last_name IN employees.last_name%TYPE);  
    BEGIN  
        employees%ROWTYPE e;  
        e := employees(p_employee_id);  
        e.first_name := p_new_first_name;  
        e.last_name := p_new_last_name;  
        employees%update(e);  
    END update_employee;  
  
    PROCEDURE delete_employee(p_employee_id IN employees.employee_id%TYPE);  
    BEGIN  
        employees%ROWTYPE e;  
        e := employees(p_employee_id);  
        employees%delete(e);  
    END delete_employee;  
  
    PROCEDURE print_all_employees;  
    BEGIN  
        FOR i IN employees.FIRST..employees.LAST  
        LOOP  
            DBMS_OUTPUT.PUT_LINE(employees(i).first_name || ' ' ||  
                employees(i).last_name || ' (' || employees(i).job_id || ')');  
        END LOOP;  
    END print_all_employees;
```

```

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

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

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

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

```

```

        WHERE employee_id = p_empid;
END get_employee;

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

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

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

-- New init_departments procedure declaration.

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

```

```

-- call the new init_departments procedure.

BEGIN
    init_departments;
END emp_pkg;
/
SHOW ERRORS

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

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

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

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

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

--New procedure init_departments spec

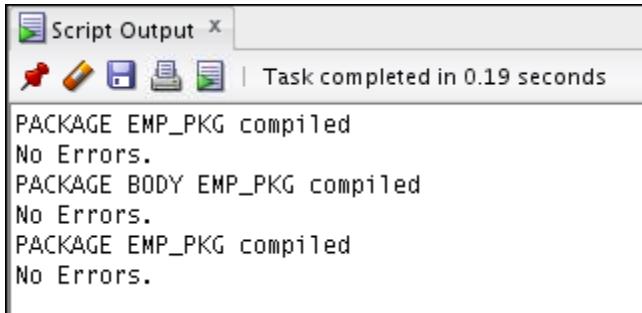
PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

```

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



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the following text:

```
PACKAGE EMP_PKG compiled
No Errors.
PACKAGE BODY EMP_PKG compiled
No Errors.
PACKAGE EMP_PKG compiled
No Errors.
```

4. Change the VALID_DEPTID validation processing function to use the private PL/SQL table of department IDs.
 - a. Modify the VALID_DEPTID function to perform its validation by using the PL/SQL table of department ID values. Click the Run Script icon (or press F5) to create and compile the package.

Uncomment and select the code under Task 4_a. Click the Run Script icon (or press F5) to create and compile the package. The newly added code is highlighted in the following code box.

```
-- Package SPECIFICATION

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

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

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

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

```

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

-- New procedure init_departments spec
PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS

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

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

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

```

```

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

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

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

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

```

```

        RETURN rec_emp;
END;

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

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

-- New init_departments procedure declaration.

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

-- call the new init_departments procedure.
BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

The screenshot shows a 'Script Output' window with the following text:
PACKAGE EMP_PKG compiled
No Errors.
PACKAGE BODY EMP_PKG compiled
No Errors.

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

Uncomment and select the code under Task 4_b.

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

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

The screenshot shows a 'Script Output' window with the following error message:
Error starting at line 788 in command:
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
Error report:
ORA-20204: Invalid department ID.
Try again.
ORA-06512: at "ORA61.EMP_PKG", line 34
ORA-06512: at "ORA61.EMP_PKG", line 46
ORA-06512: at line 1

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

Uncomment and select the code under Task 4_c. The code and result are displayed as follows:

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

The screenshot shows a 'Script Output' window with the following output:
1 rows inserted.
committed.

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

Uncomment and select the code under Task 4_d. The code and the result are displayed as follows:

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

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output X'. Below it are icons for redo, undo, save, and print. A status bar at the bottom right says 'Task completed in 1.925 seconds'. The main area contains the following text:
Error starting at line 802 in command:
EXECUTE emp_pkg.add_employee('James', 'Bond', 15)
Error report:
ORA-20204: Invalid department ID.
Try again.
ORA-06512: at "ORA61.EMP_PKG", line 34
ORA-06512: at "ORA61.EMP_PKG", line 46
ORA-06512: at line 1

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

- e. Execute the `EMP_PKG.INIT_DEPARTMENTS` procedure to update the index-by table with the latest departmental data.

Uncomment and select the code under Task 4_e. The code and result are displayed as follows:

```
EXECUTE EMP_PKG.INIT_DEPARTMENTS
```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output'. Below it are icons for redo, undo, save, and print. A status bar at the bottom right says 'Task completed in 0.016 seconds'. The main area contains the text:
anonymous block completed

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

Uncomment and select the code under Task 4_f. The code and the result are displayed as follows.

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

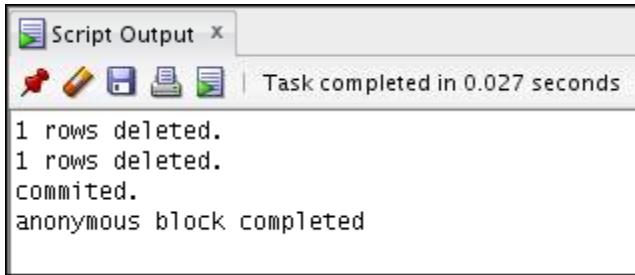
The row is finally inserted because the department 15 record exists in the database and the package's PL/SQL index-by table, due to invoking `EMP_PKG.INIT_DEPARTMENTS`, which refreshes the package state data.

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output'. Below it are icons for redo, undo, save, and print. A status bar at the bottom right says 'Task completed in 0.016 seconds'. The main area contains the text:
anonymous block completed

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

Open Uncomment and select the code under Task 4_g. The code and the result are displayed as follows.

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



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

Uncomment and select the code under Task 5_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create and compile the package. The code and the result are displayed as follows. The package's specification subprograms are already in an alphabetical order.

```
CREATE OR REPLACE PACKAGE emp_pkg IS

-- the package spec is already in an alphabetical order.

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

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

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

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

```

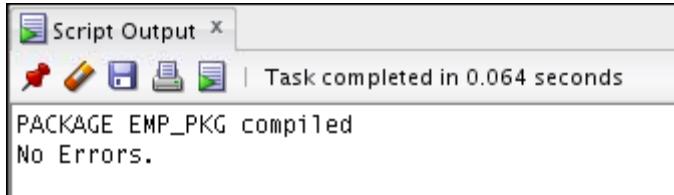
FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype;

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS

```



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

Uncomment and select the code under Task 5_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create the package. The code and the result are displayed as follows.

```

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

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

```

```

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

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

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

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

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

```

```

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

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

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

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

BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

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

```

Script Output x
| Task completed in 0.063 seconds
PACKAGE BODY EMP_PKG compiled
Errors: check compiler log
16/8      PLS-00313: 'VALID_DEPTID' not declared in this scope
16/5      PL/SQL: Statement ignored

```

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

Uncomment and select the code under Task 5_c. The function's forward declaration is highlighted in the code box below. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create and compile the package. The code and the result are displayed as follows.

```

-- Package BODY

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

  -- forward declaration of valid_deptid

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

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

```

```

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

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

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

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

```

```

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

-- New alphabetical location of function init_departments.

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

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

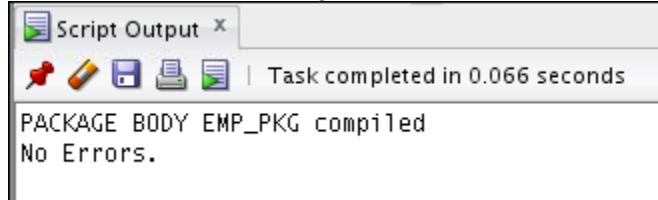
-- New alphabetical location of function valid_deptid.

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

```

```
BEGIN  
    init_departments;  
END emp_pkg;  
  
/  
SHOW ERRORS
```

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



Practices for Lesson 6: Using Oracle-Supplied Packages in Application Development

Chapter 6

Practices for Lesson 6

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_06.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 6-1: Using the UTL_FILE Package

Overview

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

Note: Execute `cleanup_06.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

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

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

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

Solution 6-1: Using the UTL_FILE Package

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

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

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

Open the file in the /home/oracle/labs/plpu/sols/sol_06.sql script.

Uncomment and select the code under Task 1.

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

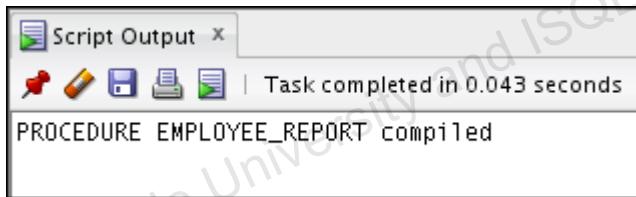
CREATE OR REPLACE PROCEDURE employee_report(
  p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
  f UTL_FILE.FILE_TYPE;
  CURSOR cur_avg IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                     FROM employees inner
             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

```

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

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

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



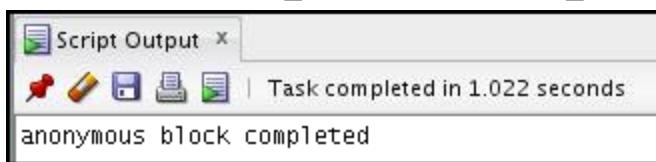
2. Invoke the procedure using the following as arguments:

- Use REPORTS_DIR as the alias for the directory object as the first parameter.
- Use sal_rpt61.txt as the second parameter.

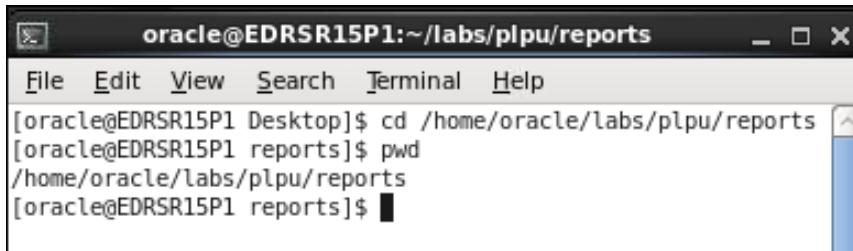
Uncomment and select the code under Task 2. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to execute the procedure. The result is shown below. Ensure that the external file and the database are on the same PC.

-- For example, if you are student ora61, use 61 as a prefix

```
EXECUTE employee_report ('REPORTS_DIR', 'sal_rpt61.txt')
```



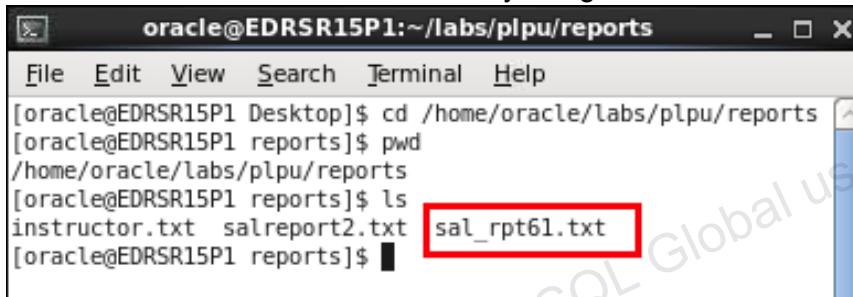
3. View the generated output text file as follows:
 - a. Double-click the **Terminal** icon on your desktop. The **Terminal** window is displayed.
 - b. At the \$ prompt, change to the `/home/oracle/labs/plpu/reports` directory that contains the generated output file, `sal_rpt61.txt` using the `cd` command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



A screenshot of a terminal window titled "oracle@EDRSR15P1:~/labs/plpu/reports". The window has a menu bar with File, Edit, View, Search, Terminal, and Help. The main area shows the command line history:
[oracle@EDRSR15P1 Desktop]\$ cd /home/oracle/labs/plpu/reports
[oracle@EDRSR15P1 reports]\$ pwd
/home/oracle/labs/plpu/reports
[oracle@EDRSR15P1 reports]\$

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

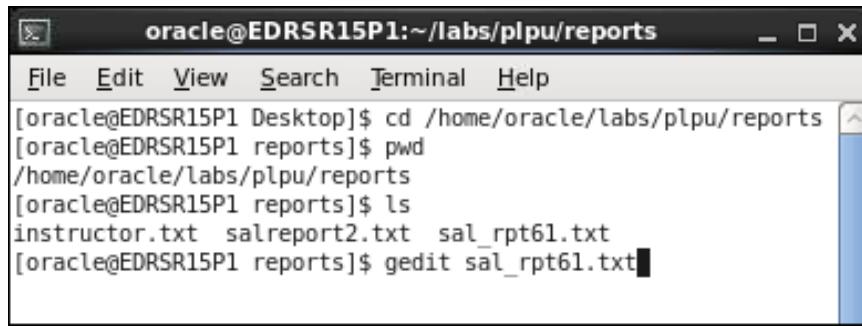
- c. List the contents of the current directory using the `ls` command as follows:



A screenshot of a terminal window titled "oracle@EDRSR15P1:~/labs/plpu/reports". The window has a menu bar with File, Edit, View, Search, Terminal, and Help. The main area shows the command line history:
[oracle@EDRSR15P1 Desktop]\$ cd /home/oracle/labs/plpu/reports
[oracle@EDRSR15P1 reports]\$ pwd
/home/oracle/labs/plpu/reports
[oracle@EDRSR15P1 reports]\$ ls
instructor.txt salreport2.txt **sal_rpt61.txt**
[oracle@EDRSR15P1 reports]\$

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

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



A screenshot of a terminal window titled "oracle@EDRSR15P1:~/labs/plpu/reports". The window has a menu bar with File, Edit, View, Search, Terminal, and Help. The terminal session shows the following commands:

```
[oracle@EDRSR15P1 Desktop]$ cd /home/oracle/labs/plpu/reports
[oracle@EDRSR15P1 reports]$ pwd
/home/oracle/labs/plpu/reports
[oracle@EDRSR15P1 reports]$ ls
instructor.txt salreport2.txt sal_rpt61.txt
[oracle@EDRSR15P1 reports]$ gedit sal_rpt61.txt
```

Oracle University and ISQL Global use only.

The screenshot shows a Windows Notepad window with the title bar 'sal_rpt61.txt'. The content of the window is a report titled 'Employees who earn more than average salary: REPORT GENERATED ON 19-NOV-12'. The report lists employee names, their employee ID (e.g., 20, 30, 50, 60, 80, 90, 100, 110), and their salaries (\$13,000.00 to \$24,000.00). The report concludes with '*** END OF REPORT ***'.

Employee Name	Employee ID	Salary
Hartstein	20	\$13,000.00
Raphaely	30	\$11,000.00
Ladwig	50	\$3,600.00
Rajs	50	\$3,500.00
Sarchand	50	\$4,200.00
Bull	50	\$4,100.00
Chung	50	\$3,800.00
Dilly	50	\$3,600.00
Bell	50	\$4,000.00
Everett	50	\$3,900.00
Mourgos	50	\$5,800.00
Vollman	50	\$6,500.00
Kaufling	50	\$7,900.00
Fripp	50	\$8,200.00
Weiss	50	\$8,000.00
Hunold	60	\$9,000.00
Ernst	60	\$6,000.00
Russell	80	\$14,000.00
Partners	80	\$13,500.00
Errazuriz	80	\$12,000.00
Cambrault	80	\$11,000.00
Zlotkey	80	\$10,500.00
Tucker	80	\$10,000.00
Bernstein	80	\$9,500.00
Hall	80	\$9,000.00
King	80	\$10,000.00
Sully	80	\$9,500.00
McEwen	80	\$9,000.00
Vishney	80	\$10,500.00
Greene	80	\$9,500.00
Ozer	80	\$11,500.00
Bloom	80	\$10,000.00
Fox	80	\$9,600.00
Abel	80	\$11,000.00
Hutton	80	\$8,800.00
Taylor	80	\$10,406.00
King	90	\$24,000.00
Faviet	100	\$9,000.00
Greenberg	100	\$12,008.00
Higgins	110	\$12,008.00

*** END OF REPORT ***

Note: The output may slightly vary based on the data in the employees table.

Practices for Lesson 7: Using Dynamic SQL

Chapter 7

Practices for Lesson 7

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_07.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 7-1: Using Native Dynamic SQL

Overview

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

Note: Execute `cleanup_07.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

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

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

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

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

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

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

- d. Describe the `MY_CONTACTS` table structure.

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

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

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

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

- h. Execute the `UPD_ROW` procedure with the following row data:

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

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

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

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

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

Solution 7-1: Using Native Dynamic SQL

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

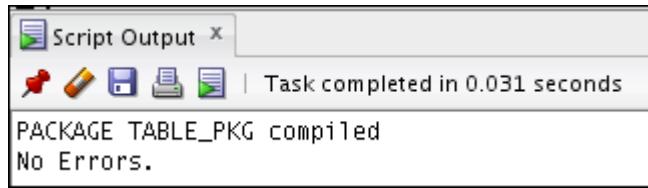
1. Create a package called `TABLE_PKG` that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. The subprograms should manage optional default parameters with NULL values.

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

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

Open the `/home/oracle/labs/plpu/solns/sol_07.sql` script. Uncomment and select the code under Task 1_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package specification. The code and the result are displayed as follows:

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



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

Uncomment and select the code under Task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package specification. The code and the result are shown below.

```

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

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

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

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

```

```

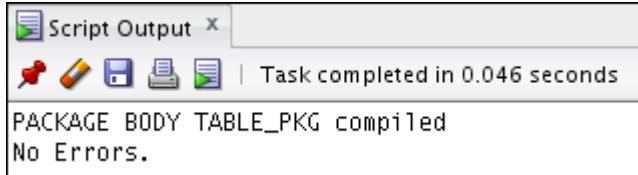
        execute(v_stmt);
END;

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

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

END table_pkg;
/
SHOW ERRORS

```



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

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

Uncomment and select the code under Task 1_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create the package specification. The code and the results are displayed as follows:

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

```
Script Output X
| Task completed in 0.993 seconds
anonymous block completed
CREATE TABLE my_contacts (id number(4), name varchar2(40))
```

- d. Describe the MY_CONTACTS table structure.

```
DESCRIBE my_contacts
```

The result is displayed as follows:

```
Script Output X
| Task completed in 0.529 seconds
DESCRIBE my_contacts
Name Null Type
-----
ID      NUMBER(4)
NAME    VARCHAR2(40)
```

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

```
SET SERVEROUTPUT ON

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

Uncomment and select the code under Task 1_e. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to execute the script.

```
Script Output X
| Task completed in 0.038 seconds
anonymous block completed
INSERT INTO my_contacts (id, name) VALUES (1,'Lauran Serhal')
INSERT INTO my_contacts (id, name) VALUES (2,'Nancy')
INSERT INTO my_contacts (id,name) VALUES (3,'Sunitha Patel')
INSERT INTO my_contacts (id,name) VALUES (4,'Valli Pataballa')
```

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

The code and the results are displayed as follows:

The screenshot shows the Oracle SQL Developer interface. The top menu bar has 'Worksheet' and 'Query Builder' tabs. The main workspace contains the following SQL code:

```
--Uncomment code below to run the solution for Task 1_f of Practice 7

SELECT *
FROM my_contacts;
```

Below the workspace is a 'Script Output' window with a toolbar. It displays the message: 'Task completed in 0.005 seconds'. The output pane shows the results of the query:

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

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

The code and the results are displayed as follows:

The screenshot shows the Oracle SQL Developer interface. The top menu bar has 'Worksheet' and 'Query Builder' tabs. The main workspace contains the following PL/SQL code:

```
--Uncomment code below to run the solution for Task 1_g of Practice 7

SET SERVEROUTPUT ON
EXECUTE table_pkg.del_row('my_contacts', 'id=3')
```

Below the workspace is a 'Script Output' window with a toolbar. It displays the message: 'Task completed in 0.564 seconds'. The output pane shows the results of the package execution:

```
anonymous block completed
DELETE FROM my_contacts WHERE id=3
```

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

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

The code and the results are displayed as follows:

A screenshot of the Oracle SQL Developer interface. The top toolbar shows various icons and the text "1.00600004 seconds". Below the toolbar, the "Worksheet" tab is selected. In the worksheet pane, there is a comment line: "--Uncomment code below to run the solution for Task 1_h of Practice 7". Below it, the code is executed: `SET SERVEROUTPUT ON` and `EXEC table_pkg.upd_row('my_contacts','name=''Nancy Greenberg''','id=2')`. The "Script Output" tab is selected at the bottom, showing the message "Task completed in 1.006 seconds". The output pane displays the result of the update: "anonymous block completed" followed by the SQL command: `UPDATE my_contacts SET name='Nancy Greenberg' WHERE id=2`.

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

The code and the results are displayed as follows:

A screenshot of the Oracle SQL Developer interface. The top toolbar shows various icons and the text "0.002 seconds". Below the toolbar, the "Worksheet" tab is selected. In the worksheet pane, there is a comment line: "--Uncomment code below to run the solution for Task 1_i of Practice 7". Below it, the query is executed: `SELECT * FROM my_contacts;`. The "Script Output" tab is selected at the bottom, showing the message "Task completed in 0.002 seconds". The output pane displays the results of the query in a table format:

ID	NAME
1	Lauran Serhal
2	Nancy Greenberg
4	Valli Pataballa

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

The code and the results are displayed as follows:

The screenshot shows the Oracle SQL Worksheet interface. The top toolbar has various icons for running scripts, saving, and zooming, with a status bar showing "0.002 seconds". Below the toolbar, tabs for "Worksheet" and "Query Builder" are visible, with "Worksheet" selected. The main workspace contains the following PL/SQL code:

```
--Uncomment code below to run the solution for Task 1_j of Practice 7

EXECUTE table_pkg.remove('my_contacts')
DESCRIBE my_contacts
```

Below the workspace is a "Script Output" window with the following log:

```
anonymous block completed
DROP TABLE my_contacts

DESCRIBE my_contacts
ERROR:
-----
ERROR: object MY_CONTACTS does not exist
```

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

Uncomment and select the code under Task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package specification. The code and the results are shown below.

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

The screenshot shows the Oracle SQL Worksheet interface. The main area displays the following PL/SQL code:

```
--Uncomment code below to run the solution for Task 2_a of Practice 7
CREATE OR REPLACE PACKAGE compile_pkg IS
    PROCEDURE make(p_name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS
```

Below the code, the "Script Output" window shows the results of the compilation:

```
PACKAGE COMPILE_PKG compiled
No Errors.
```

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

Uncomment and select the code under Task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package body. The code and the results are displayed as follows:

```
CREATE OR REPLACE PACKAGE BODY compile_pkg IS

    PROCEDURE execute(p_stmt VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE(p_stmt);
        EXECUTE IMMEDIATE p_stmt;
    END;

    FUNCTION get_type(p_name VARCHAR2) RETURN VARCHAR2 IS
```

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

```

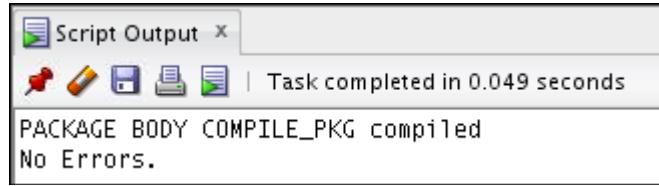
v_proc_type VARCHAR2(30) := NULL;
BEGIN

    -- The ROWNUM = 1 is added to the condition
    -- to ensure only one row is returned if the
    -- name represents a PACKAGE, which may also
    -- have a PACKAGE BODY. In this case, we can
    -- only compile the complete package, but not
    -- the specification or body as separate
    -- components.

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

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

```

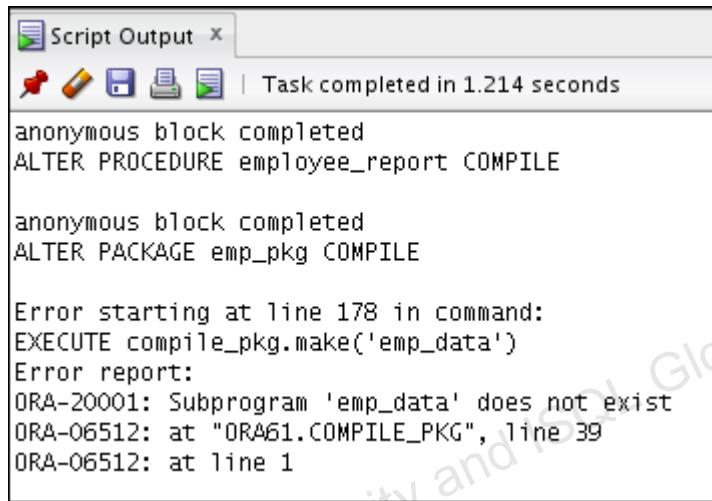


- c. Use the `COMPILE_PKG.MAKE` procedure to compile the following:

- 1) The `EMPLOYEE_REPORT` procedure
- 2) The `EMP_PKG` package
- 3) A nonexistent object called `EMP_DATA`

Uncomment and select the code under task 2_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to execute the package's procedure. The result is shown below.

```
SET SERVEROUTPUT ON
EXECUTE compile_pkg.make('employee_report')
EXECUTE compile_pkg.make('emp_pkg')
EXECUTE compile_pkg.make('emp_data')
```



The screenshot shows the 'Script Output' window from the Oracle SQL Worksheet. It contains the following text:

```
Script Output x
| Task completed in 1.214 seconds
anonymous block completed
ALTER PROCEDURE employee_report COMPILE

anonymous block completed
ALTER PACKAGE emp_pkg COMPILE

Error starting at line 178 in command:
EXECUTE compile_pkg.make('emp_data')
Error report:
ORA-20001: Subprogram 'emp_data' does not exist
ORA-06512: at "ORA61.COMPILE_PKG", line 39
ORA-06512: at line 1
```

Practices for Lesson 8: Design Considerations for PL/SQL Code

Chapter 8

Practices for Lesson 8

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_08.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 8-1: Using Bulk Binding and Autonomous Transactions

Overview

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

Note: Execute `cleanup_08.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

1. Update the `EMP_PKG` package with a new procedure to query employees in a specified department.
 - a. In the package specification:
 - 1) Declare a `get_employees` procedure with a parameter called `dept_id`, which is based on the `employees.department_id` column type
 - 2) Define a nested PL/SQL type as a `TABLE OF EMPLOYEES%ROWTYPE`
 - b. In the package body:
 - 1) Define a private variable called `emp_table` based on the type defined in the specification to hold employee records
 - 2) Implement the `get_employees` procedure to bulk fetch the data into the table
 - c. Create a new procedure in the specification and body, called `show_employees`, which does not take arguments. The procedure displays the contents of the private PL/SQL table variable (if any data exists). Use the `print_employee` procedure that you created in an earlier practice. To view the results, click the Enable DBMS Output icon on the DBMS Output tab in SQL Developer, if you have not already done so.
 - d. Enable `SERVERTOUTPUT`. Invoke the `emp_pkg.get_employees` procedure for department 30, and then invoke `emp_pkg.show_employees`. Repeat this for department 60.
2. Your manager wants to keep a log whenever the `add_employee` procedure in the package is invoked to insert a new employee into the `EMPLOYEES` table.
 - a. First, load and execute the code under Task 2_a from the `/home/oracle/labs/plpu/solns/sol_08.sql` script to create a log table called `LOG_NEWEMP`, and a sequence called `log_newemp_seq`.
 - b. In the `EMP_PKG` package body, modify the `add_employee` procedure, which performs the actual `INSERT` operation. Add a local procedure called `audit_newemp` as follows:
 - 1) The `audit_newemp` procedure must use an autonomous transaction to insert a log record into the `LOG_NEWEMP` table.
 - 2) Store the `USER`, the current time, and the new employee name in the log table row.
 - 3) Use `log_newemp_seq` to set the `entry_id` column.

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

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

Solution 8-1: Using Bulk Binding and Autonomous Transactions

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

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

Open the /home/oracle/labs/plpu/solns/sol_08.sql script. Uncomment and select the code under Task 1_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the specification. The code and the results are displayed as follows. The newly added code is highlighted in bold letters in the code box below.

```

CREATE OR REPLACE PACKAGE emp_pkg IS

  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

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

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

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

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

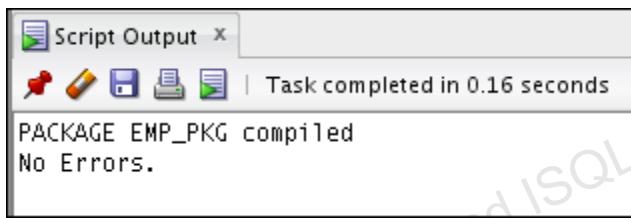
FUNCTION get_employee(p_family_name employees.last_name%type)
    return employees%rowtype;

PROCEDURE get_employees(p_dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

END emp_pkg;
/
SHOW ERRORS
```



- b. In the package body:
- 1) Define a private variable called `emp_table` based on the type defined in the specification to hold employee records
 - 2) Implement the `get_employees` procedure to bulk fetch the data into the table
- Uncomment and select the code under Task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package body. The code and the results are shown below. The newly added code is highlighted in bold letters in the code box below.**

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

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

```

        RETURN valid_departments.exists(p_deptid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;

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

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

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

```

```

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

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

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

-- New get_employees procedure.

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

PROCEDURE init_departments IS

```

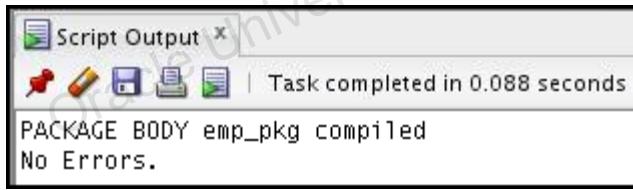
```

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

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

BEGIN
    init_departments;
END emp_pkg;
/
SHOW ERRORS

```



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

Uncomment and select the code under Task 1_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create and compile the package with the new procedure. The code and the results are shown below.

```
-- Package SPECIFICATION

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

```

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

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

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

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

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

PROCEDURE get_employees(p_dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN

```

```

INDEX BY BINARY_INTEGER;

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

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

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

```

```

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

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

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

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

PROCEDURE init_departments IS
BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP

```

```

        valid_departments(rec.department_id) := TRUE;
    END LOOP;
END;

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

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

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

BEGIN
    init_departments;
END emp_pkg;

/
SHOW ERRORS

```

The screenshot shows the 'Script Output' window from the Oracle SQL Worksheet. It displays the following text:
PACKAGE EMP_PKG compiled
No Errors.
PACKAGE BODY EMP_PKG compiled
No Errors.

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

Uncomment and select the code under Task 1_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to invoke the package's procedures. The code and the results are shown below:

```
SET SERVEROUTPUT ON

EXECUTE emp_pkg.get_employees(30)
EXECUTE emp_pkg.show_employees

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

The screenshot shows the 'Script Output' window from the Oracle SQL Worksheet. It displays the following output:
anonymous block completed
anonymous block completed
Employees in Package table
30 209 Samuel Joplin SA_REP 1000
30 114 Den Raphaely PU_MAN 11000
30 115 Alexander Khoo PU_CLERK 3100
30 116 Shelli Baida PU_CLERK 2900
30 117 Sigal Tobias PU_CLERK 2800
30 118 Guy Himuro PU_CLERK 2600
30 119 Karen Colmenares PU_CLERK 2500

anonymous block completed
anonymous block completed
Employees in Package table
60 103 Alexander Hunold IT_PROG 9000
60 104 Bruce Ernst IT_PROG 6600
60 105 David Austin IT_PROG 4800
60 106 Valli Pataballa IT_PROG 4800
60 107 Diana Lorentz IT_PROG 4200

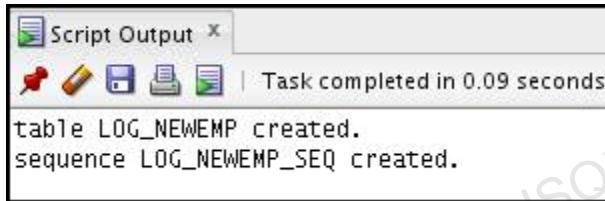
2. Your manager wants to keep a log whenever the add_employee procedure in the package is invoked to insert a new employee into the EMPLOYEES table.

- a. First, load and execute the code under Task 2_a from /home/oracle/labs/plpu/solns/sol_08.sql script to create a log table called LOG_NEWEMP, and a sequence called log_newemp_seq.

Uncomment and select the code under Task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

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

CREATE SEQUENCE log_newemp_seq;
```



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

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

Uncomment and select the code under Task 2_b. The newly added code is highlighted in bold letters in the following code box. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are displayed as follows:

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS

    TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;
```

```

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

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

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

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

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

PROCEDURE get_employees(p_dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    TYPE boolean_tab_type IS TABLE OF BOOLEAN

```

```

INDEX BY BINARY_INTEGER;

valid_departments boolean_tab_type;
emp_table          emp_tab_type;

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

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

-- New local procedure

PROCEDURE audit_newemp IS
PRAGMA AUTONOMOUS_TRANSACTION;
user_id VARCHAR2(30) := USER;
BEGIN
INSERT INTO log_newemp (entry_id, user_id, log_time,
name)
VALUES (log_newemp_seq.NEXTVAL, user_id,
sysdate,p_first_name||' '||p_last_name);
COMMIT;
END audit_newemp;

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

```

```

        RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID.
Try again.');
      END IF;
    END add_employee;

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

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

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

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

```

```

        FROM employees
        WHERE last_name = p_family_name;
        RETURN rec_emp;
    END;

-- New get_employees procedure.

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

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

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

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

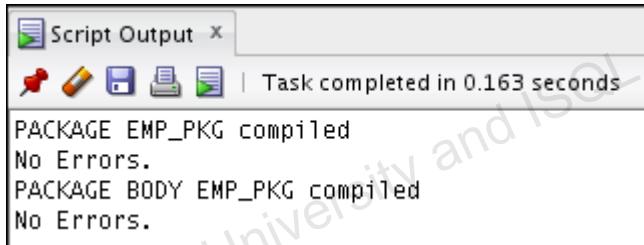
```

```

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

BEGIN
    init_departments;
END emp_pkg;
/
SHOW ERRORS

```



- c. Modify the add_employee procedure to invoke audit_emp before it performs the insert operation.

Uncomment and select the code under Task 2_c. The newly added code is highlighted in bold letters in the following code box. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS

    TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

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

```

```

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

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

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

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

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

PROCEDURE get_employees(p_dept_id
    employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p_rec_emp employees%rowtype);

PROCEDURE show_employees;

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

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

    valid_departments boolean_tab_type;

```

```

emp_table           emp_tab_type;

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

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

PROCEDURE audit_newemp IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    user_id VARCHAR2(30) := USER;
BEGIN
    INSERT INTO log_newemp (entry_id, user_id, log_time, name)
    VALUES (log_newemp_seq.NEXTVAL, user_id,
    sysdate,p_first_name||' '||p_last_name);
    COMMIT;
END audit_newemp;

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

PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,

```

```

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

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

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

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

```

```

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

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

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

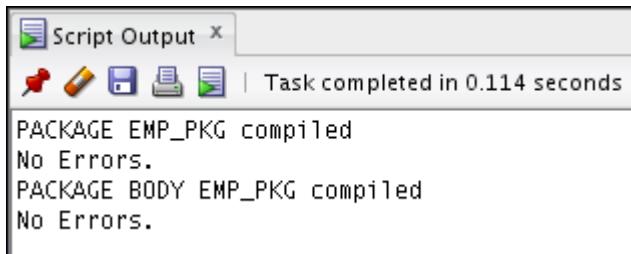
PROCEDURE show_employees IS
BEGIN
    IF emp_table IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('Employees in Package table');
        FOR i IN 1 .. emp_table.COUNT
        LOOP
            print_employee(emp_table(i));
        END LOOP;
    END IF;
END show_employees;

FUNCTION valid_deptid(p_deptid IN
departments.department_id%TYPE)
RETURN BOOLEAN IS
    v_dummy PLS_INTEGER;
BEGIN
    RETURN valid_departments.exists(p_deptid);

```

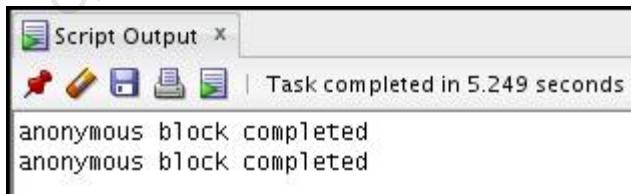
```
EXCEPTION
  WHEN NO_DATA_FOUND THEN

    RETURN FALSE;
END valid_deptid;
BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
```



- d. Invoke the add_employee procedure for these new employees: Max Smart in department 20 and Clark Kent in department 10. What happens?
Uncomment and select the code under Task 2_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are as follows.

```
EXECUTE emp_pkg.add_employee('Max', 'Smart', 20)
EXECUTE emp_pkg.add_employee('Clark', 'Kent', 10)
```



Both insert statements complete successfully. The log table has two log records as shown in the next step.

- e. Query the two EMPLOYEES records added, and the records in the LOG_NEWEMP table. How many log records are present?

Uncomment and select the code under Task 2_e. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are displayed as follows:

```
select department_id, employee_id, last_name, first_name
from employees
where last_name in ('Kent', 'Smart');
```

```
select * from log_newemp;
```

The screenshot shows the 'Script Output' window with the title bar 'Script Output'. Below it is a toolbar with icons for script, edit, file, and run. A message bar indicates 'Task completed in 0.011 seconds'. The main area displays two tables. The first table has columns 'DEPARTMENT_ID', 'EMPLOYEE_ID', 'LAST_NAME', and 'FIRST_NAME'. It contains two rows: one for employee 212 Kent (Clark) and one for employee 211 Smart (Max). The second table has columns 'ENTRY_ID', 'USER_ID', 'LOG_TIME', and 'NAME'. It also contains two rows: one for entry 1 by user ORA61 at 20-NOV-12 for Max Smart, and one for entry 2 by user ORA61 at 20-NOV-12 for Clark Kent.

DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	FIRST_NAME
10	212	Kent	Clark
20	211	Smart	Max

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA61	20-NOV-12	Max Smart
2	ORA61	20-NOV-12	Clark Kent

There are two log records, one for Smart and another for Kent.

- f. Execute a ROLLBACK statement to undo the insert operations that have not been committed. Use the same queries from step 2 e. as follows:
- 1) Use the first query to check whether the employee rows for Smart and Kent have been removed.
 - 2) Use the second query to check the log records in the LOG_NEWEMP table. How many log records are present? Why?

```
ROLLBACK;
```

The screenshot shows the 'Script Output' window with the title bar 'Script Output'. Below it is a toolbar with icons for script, edit, file, and run. A message bar indicates 'Task completed in 0.006 seconds'. The main area displays a single line of text: 'rollback complete.' followed by a blank table. The table has columns 'ENTRY_ID', 'USER_ID', 'LOG_TIME', and 'NAME'. It contains two rows: one for entry 1 by user ORA61 at 20-NOV-12 for Max Smart, and one for entry 2 by user ORA61 at 20-NOV-12 for Clark Kent.

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA61	20-NOV-12	Max Smart
2	ORA61	20-NOV-12	Clark Kent

The two employee records are removed (rolled back). The two log records remain in the log table because they were inserted using an autonomous transaction, which is unaffected by the rollback performed in the main transaction.

Practices for Lesson 9: Creating Triggers

Chapter 9

Practices for Lesson 9

Overview

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_09.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 9-1: Creating Statement and Row Triggers

Overview

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

Note: Execute `cleanup_09.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

1. The rows in the JOBS table store a minimum and maximum salary allowed for different `JOB_ID` values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.
 - a. Create a procedure called `CHECK_SALARY` as follows:
 - 1) The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.
 - 2) The procedure uses the job ID to determine the minimum and maximum salary for the specified job.
 - 3) If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>." Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.
 - b. Create a trigger called `CHECK_SALARY_TRG` on the `EMPLOYEES` table that fires before an `INSERT` or `UPDATE` operation on each row:
 - 1) The trigger must call the `CHECK_SALARY` procedure to carry out the business logic.
 - 2) The trigger should pass the new job ID and salary to the procedure parameters.
2. Test the `CHECK_SALARY_TRG` trigger using the following cases:
 - a. Using your `EMP_PKG.ADD_EMPLOYEE` procedure, add employee Eleanor Beh to department 30. What happens and why?
 - b. Update the salary of employee 115 to \$2,000. In a separate update operation, change the employee job ID to `HR_REP`. What happens in each case?
 - c. Update the salary of employee 115 to \$2,800. What happens?
3. Update the `CHECK_SALARY_TRG` trigger to fire only when the job ID or salary values have actually changed.
 - a. Implement the business rule using a `WHEN` clause to check whether the `JOB_ID` or `SALARY` values have changed.
 - Note:** Make sure that the condition handles the `NULL` in the `OLD.column_name` values if an `INSERT` operation is performed; otherwise, an insert operation will fail.
 - b. Test the trigger by executing the `EMP_PKG.ADD_EMPLOYEE` procedure with the following parameter values:
 - `p_first_name: 'Eleanor'`
 - `p_last_name: 'Beh'`

- p_Email: 'EBEH'
 - p_Job: 'IT_PROG'
 - p_Sal: 5000
- c. Update employees with the IT_PROG job by incrementing their salary by \$2,000. What happens?
 - d. Update the salary to \$9,000 for Eleanor Beh.
Hint: Use an UPDATE statement with a subquery in the WHERE clause. What happens?
 - e. Change the job of Eleanor Beh to ST_MAN using another UPDATE statement with a subquery. What happens?
4. You are asked to prevent employees from being deleted during business hours.
 - a. Write a statement trigger called DELETE_EMP_TRG on the EMPLOYEES table to prevent rows from being deleted during weekday business hours, which are from 9:00 AM to 6:00 PM.
 - b. Attempt to delete employees with JOB_ID of SA REP who are not assigned to a department.
Hint: This is employee Grant with ID 178.

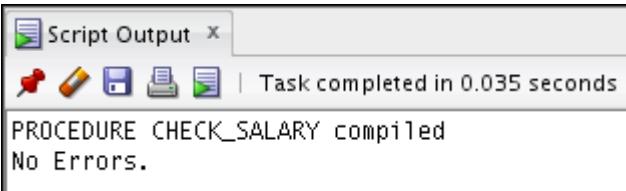
Solution 9-1: Creating Statement and Row Triggers

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.

1. The rows in the JOBS table store a minimum and maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall in the range allowed for their job type, for insert and update operations.
 - a. Create a procedure called CHECK_SALARY as follows:
 - 1) The procedure accepts two parameters, one for an employee's job ID string and the other for the salary.
 - 2) The procedure uses the job ID to determine the minimum and maximum salary for the specified job.
 - 3) If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message "Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>". Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.

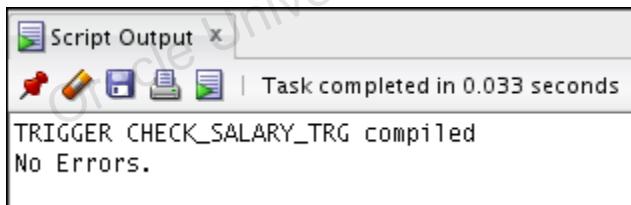
**Open sol_09.sql script from /home/oracle/labs/plpu/soln directory.
Uncomment and select the code under Task 1_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.**

```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN
  SELECT min_salary, max_salary INTO v_minsal, v_maxsal
  FROM jobs
  WHERE job_id = UPPER(p_the_job);
  IF p_the_salary NOT BETWEEN v_minsal AND v_maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $' || p_the_salary || '. ||
      'Salaries for job '|| p_the_job ||
      ' must be between $'|| v_minsal || ' and $' || v_maxsal);
  END IF;
END;
/
SHOW ERRORS
```



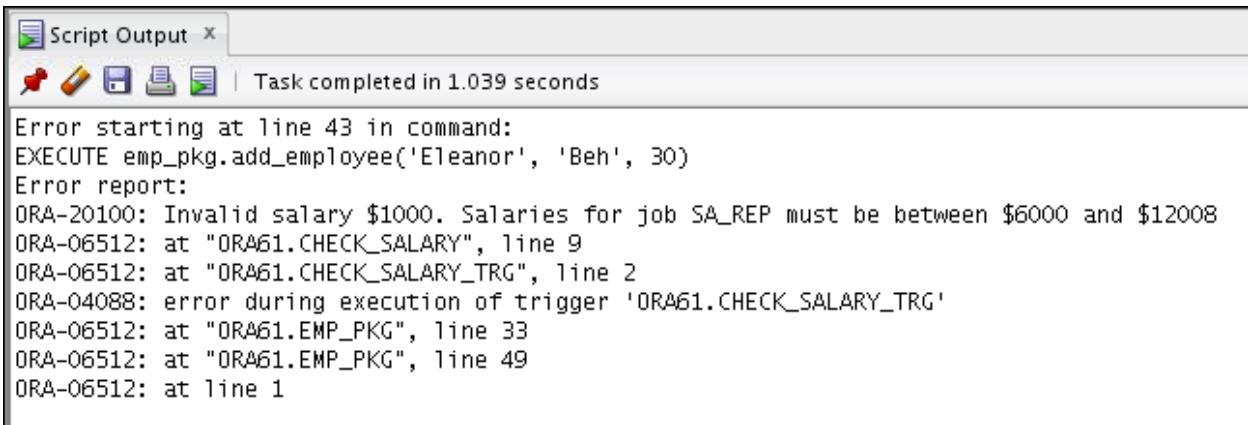
- b. Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row:
- 1) The trigger must call the CHECK_SALARY procedure to carry out the business logic.
 - 2) The trigger should pass the new job ID and salary to the procedure parameters.
- Uncomment and select the code under Task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.**

```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees
FOR EACH ROW
BEGIN
    check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```



2. Test the CHECK_SALARY_TRG trigger using the following cases:
- a. Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh to department 30. What happens and why?
- Uncomment and select the code under Task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.**

```
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)
```



The screenshot shows the 'Script Output' window from Oracle SQL Worksheet. The title bar says 'Script Output'. Below it is a toolbar with icons for Run, Stop, Save, Print, and Copy. A status bar at the bottom right says 'Task completed in 1.039 seconds'. The main area contains an error message:

```
Error starting at line 43 in command:  
EXECUTE emp_pkg.add_employee('Eleanor', 'Beh', 30)  
Error report:  
ORA-20100: Invalid salary $1000. Salaries for job SA_REP must be between $6000 and $12008  
ORA-06512: at "ORA61.CHECK_SALARY", line 9  
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2  
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'  
ORA-06512: at "ORA61.EMP_PKG", line 33  
ORA-06512: at "ORA61.EMP_PKG", line 49  
ORA-06512: at line 1
```

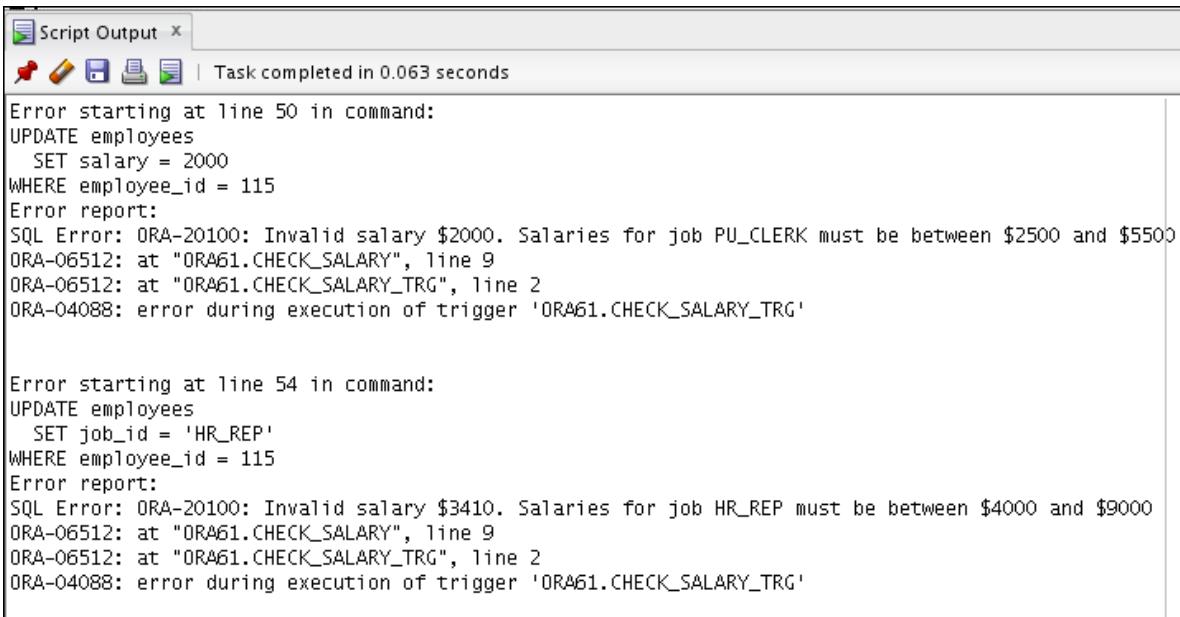
The trigger raises an exception because the EMP_PKG.ADD_EMPLOYEE procedure invokes an overloaded version of itself that uses the default salary of \$1,000 and a default job ID of SA_REP. However, the JOBS table stores a minimum salary of \$6,000 for the SA_REP type.

- b. Update the salary of employee 115 to \$2,000. In a separate update operation, change the employee job ID to HR_REP. What happens in each case?

Uncomment and select the code under Task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees  
SET salary = 2000  
WHERE employee_id = 115;
```

```
UPDATE employees  
SET job_id = 'HR_REP'  
WHERE employee_id = 115;
```



The screenshot shows the 'Script Output' window of the Oracle SQL Worksheet. It displays two separate update commands that failed due to trigger constraints. The first command attempts to set the salary of employee 115 to 2000, which fails because the salary must be between \$2500 and \$5500 for the PU_CLERK job. The second command attempts to change the job of employee 115 to HR_REP, which fails because the salary must be between \$4000 and \$9000 for the HR_REP job.

```
Error starting at line 50 in command:  
UPDATE employees  
  SET salary = 2000  
 WHERE employee_id = 115  
Error report:  
SQL Error: ORA-20100: Invalid salary $2000. Salaries for job PU_CLERK must be between $2500 and $5500  
ORA-06512: at "ORA61.CHECK_SALARY", line 9  
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2  
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'  
  
Error starting at line 54 in command:  
UPDATE employees  
  SET job_id = 'HR REP'  
 WHERE employee_id = 115  
Error report:  
SQL Error: ORA-20100: Invalid salary $3410. Salaries for job HR REP must be between $4000 and $9000  
ORA-06512: at "ORA61.CHECK_SALARY", line 9  
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2  
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
```

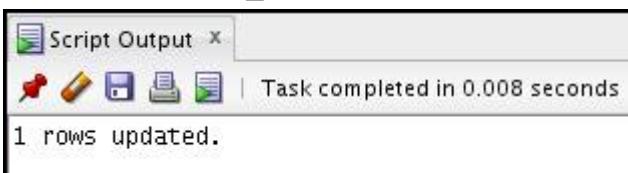
The first update statement fails to set the salary to \$2,000. The check salary trigger rule fails the update operation because the new salary for employee 115 is less than the minimum allowed for the PU_CLERK job ID.

The second update fails to change the employee's job because the current employee's salary of \$3,100 is less than the minimum for the new HR_REP job ID.

- c. Update the salary of employee 115 to \$2,800. What happens?

Uncomment and select the code under Task 2_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees  
  SET salary = 2800  
 WHERE employee_id = 115;
```



The screenshot shows the 'Script Output' window again, this time displaying a successful update. The command sets the salary of employee 115 to 2800, and the output shows '1 rows updated.', indicating the operation was successful.

```
1 rows updated.
```

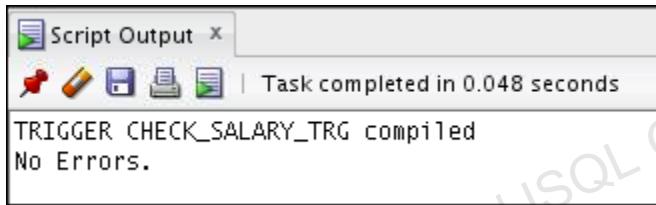
The update operation is successful because the new salary falls within the acceptable range for the current job ID.

3. Update the CHECK_SALARY_TRG trigger to fire only when the job ID or salary values have actually changed.
 - a. Implement the business rule using a WHEN clause to check whether the JOB_ID or SALARY values have changed.

Note: Make sure that the condition handles the NULL in the OLD.column_name values if an INSERT operation is performed; otherwise, an insert operation will fail.

Uncomment and select the code under Task 3_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

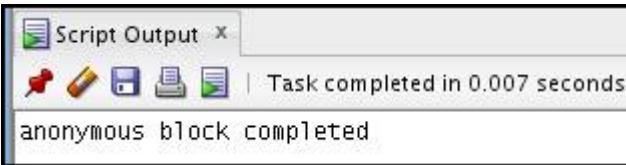
```
CREATE OR REPLACE TRIGGER check_salary_trg
BEFORE INSERT OR UPDATE OF job_id, salary
ON employees FOR EACH ROW
WHEN (new.job_id <> NVL(old.job_id,'?') OR
      new.salary <> NVL(old.salary,0))
BEGIN
    check_salary(:new.job_id, :new.salary);
END;
/
SHOW ERRORS
```



- b. Test the trigger by executing the EMP_PKG.ADD_EMPLOYEE procedure with the following parameter values:
- p_first_name: 'Eleanor'
 - p_last_name: 'Beh'
 - p_Email: 'EBEH'
 - p_Job: 'IT_PROG'
 - p_Sal: 5000

Uncomment and select the code under Task 3_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

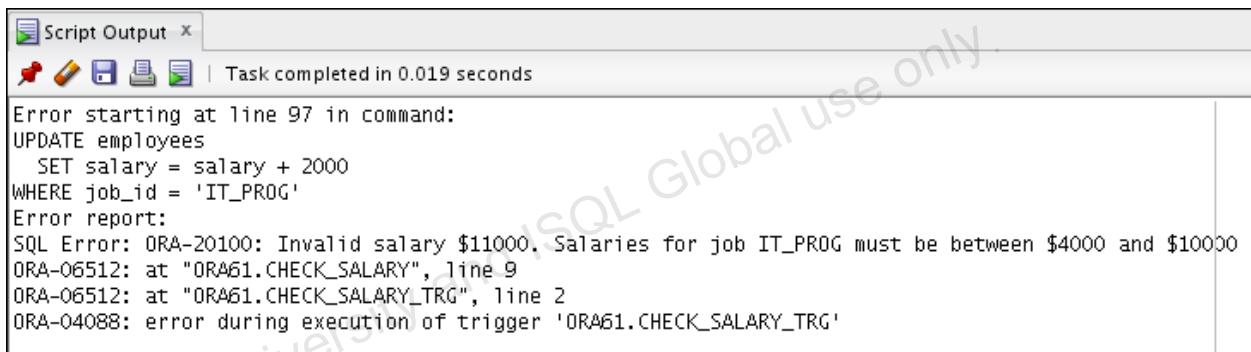
```
BEGIN
    emp_pkg.add_employee('Eleanor', 'Beh', 'EBEH',
                         p_job => 'IT_PROG', p_sal => 5000);
END;
/
```



- c. Update employees with the IT_PROG job by incrementing their salary by \$2,000. What happens?

Uncomment and select the code under Task 3_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
   SET salary = salary + 2000
 WHERE job_id = 'IT_PROG';
```



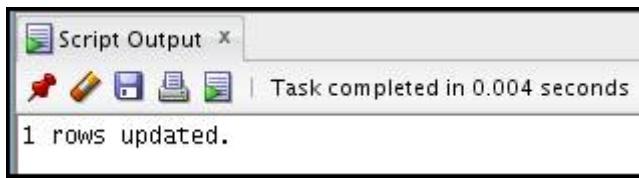
An employee's salary in the specified job type exceeds the maximum salary for that job type. No employee salaries in the IT_PROG job type are updated.

- d. Update the salary to \$9,000 for Eleanor Beh.

Hint: Use an UPDATE statement with a subquery in the WHERE clause. What happens?

Uncomment and select the code under Task 3_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
   SET salary = 9000
 WHERE employee_id = (SELECT employee_id
                        FROM employees
                       WHERE last_name = 'Beh');
```



- e. Change the job of Eleanor Beh to ST_MAN using another UPDATE statement with a subquery. What happens?

Uncomment and select the code under Task 3_e. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
UPDATE employees
  set job_id = 'ST_MAN'
WHERE employee_id = (SELECT employee_id
                      FROM employees
                     WHERE last_name = 'Beh');
```

The screenshot shows the 'Script Output' window of the Oracle SQL Worksheet. It displays the SQL command and its execution details:

```
Script Output X
Task completed in 0.019 seconds
Error starting at line 117 in command:
UPDATE employees
  set job_id = 'ST_MAN'
WHERE employee_id = (SELECT employee_id
                      FROM employees
                     WHERE last_name = 'Beh')
Error report:
SQL Error: ORA-20100: Invalid salary $9000. Salaries for job ST_MAN must be between $5500 and $8500
ORA-06512: at "ORA61.CHECK_SALARY", line 9
ORA-06512: at "ORA61.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY_TRG'
```

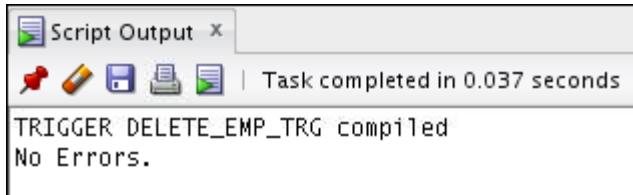
The maximum salary of the new job type is less than the employee's current salary; therefore, the update operation fails.

4. You are asked to prevent employees from being deleted during business hours.
- Write a statement trigger called DELETE_EMP_TRG on the EMPLOYEES table to prevent rows from being deleted during weekday business hours, which are from 9:00 AM to 6:00 PM.

Uncomment and select the code under Task 4_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER delete_emp_trg
BEFORE DELETE ON employees
DECLARE
  the_day VARCHAR2(3) := TO_CHAR(SYSDATE, 'DY');
  the_hour PLS_INTEGER := TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'));
BEGIN
  IF (the_hour BETWEEN 9 AND 18) AND (the_day NOT IN ('SAT', 'SUN')) THEN
    RAISE_APPLICATION_ERROR(-20150,
                           'Employee records cannot be deleted during the business');
```

hours of 9AM and 6PM') ;
END IF;
END;
/
SHOW ERRORS



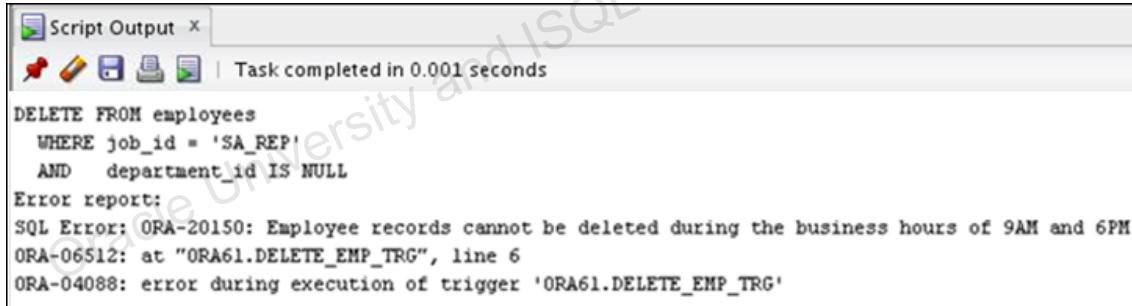
The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output'. Below it is a toolbar with icons for Run, Stop, Save, and others. The main area displays the message 'Task completed in 0.037 seconds' and 'TRIGGER DELETE_EMP_TRG compiled No Errors.'

- b. Attempt to delete employees with JOB_ID of SA REP who are not assigned to a department.

Hint: This is employee Grant with ID 178.

Uncomment and select the code under Task 4_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
DELETE FROM employees  
WHERE job_id = 'SA REP'  
AND department_id IS NULL;
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output'. Below it is a toolbar with icons for Run, Stop, Save, and others. The main area displays the SQL command and its execution results:

DELETE FROM employees
WHERE job_id = 'SA REP'
AND department_id IS NULL

Error report:
SQL Error: ORA-20150: Employee records cannot be deleted during the business hours of 9AM and 6PM
ORA-06512: at "ORA61.DELETE_EMP_TRG", line 6
ORA-04088: error during execution of trigger 'ORA61.DELETE_EMP_TRG'

Note: Depending on the current time on your host machine in the classroom, you may or may not be able to perform the delete operations. For example, in the screen capture above, the delete operation failed as it was performed outside the allowed business hours (based on the host machine time).

Practices for Lesson 10: Creating Compound, DDL, and Event Database Triggers

Chapter 10

Practices for Lesson 10

Overview

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_10.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 10-1: Managing Data Integrity Rules and Mutating Table Exceptions

Overview

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

Note: Execute `cleanup_10.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

1. Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salaries. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employees' salaries, the `CHECK_SALARY` trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.
 - a. Update your `EMP_PKG` package (that you last updated in the practice titled 'Creating Triggers') as follows:
 - 1) Add a procedure called `SET_SALARY` that updates the employees' salaries.
 - 2) The `SET_SALARY` procedure accepts the following two parameters: The job ID for those salaries that may have to be updated, and the new minimum salary for the job ID
 - b. Create a row trigger named `UPD_MINSALARY_TRG` on the JOBS table that invokes the `EMP_PKG.SET_SALARY` procedure, when the minimum salary in the JOBS table is updated for a specified job ID.
 - c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their `JOB_ID` is '`IT_PROG`'. Then, update the minimum salary in the JOBS table to increase it by \$1,000. What happens?
2. To resolve the mutating table issue, create a `JOBS_PKG` package to maintain in memory a copy of the rows in the JOBS table. Next, modify the `CHECK_SALARY` procedure to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, you must create a `BEFORE INSERT OR UPDATE` statement trigger on the EMPLOYEES table to initialize the `JOBS_PKG` package state before the `CHECK_SALARY` row trigger is fired.
 - a. Create a new package called `JOBS_PKG` with the following specification:

```

PROCEDURE initialize;
FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER;
FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(p_jobid VARCHAR2, min_salary

```

```

        NUMBER) ;
PROCEDURE set_maxsalary(p_jobid VARCHAR2, max_salary
        NUMBER) ;

```

- b. Implement the body of JOBS_PKG as follows:
 - 1) Declare a private PL/SQL index-by table called jobs_tab_type that is indexed by a string type based on the JOBS.JOB_ID%TYPE.
 - 2) Declare a private variable called jobstab based on the jobs_tab_type.
 - 3) The INITIALIZE procedure reads the rows in the JOBS table by using a cursor loop, and uses the JOB_ID value for the jobstab index that is assigned its corresponding row.
 - 4) The GET_MINSALARY function uses a p_jobid parameter as an index to the jobstab and returns the min_salary for that element.
 - 5) The GET_MAXSALARY function uses a p_jobid parameter as an index to the jobstab and returns the max_salary for that element.
 - 6) The SET_MINSALARY procedure uses its p_jobid as an index to the jobstab to set the min_salary field of its element to the value in the min_salary parameter.
 - 7) The SET_MAXSALARY procedure uses its p_jobid as an index to the jobstab to set the max_salary field of its element to the value in the max_salary parameter.
 - c. Copy the CHECK_SALARY procedure from Practice 9, Exercise 1a, and modify the code by replacing the query on the JOBS table with statements to set the local minsal and maxsal variables with values from the JOBS_PKG data by calling the appropriate GET_*SALARY functions. This step should eliminate the mutating trigger exception.
 - d. Implement a BEFORE INSERT OR UPDATE statement trigger called INIT_JOBPKG_TRG that uses the CALL syntax to invoke the JOBS_PKG.INITIALIZE procedure to ensure that the package state is current before the DML operations are performed.
 - e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the IT_PROG job type by 1,000 in the JOBS table. Follow this up with a query on the employees with the IT_PROG job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?
3. Because the CHECK_SALARY procedure is fired by CHECK_SALARY_TRG before inserting or updating an employee, you must check whether this still works as expected.
- a. Test this by adding a new employee using EMP_PKG.ADD_EMPLOYEE with the following parameters: ('Steve', 'Morse', 'SMORSE', and sal => 6500). What happens?
 - b. To correct the problem encountered when adding or updating an employee:
 - 1) Create a BEFORE INSERT OR UPDATE statement trigger called EMPLOYEE_INITJOBS_TRG on the EMPLOYEES table that calls the JOBS_PKG.INITIALIZE procedure.
 - 2) Use the CALL syntax in the trigger body.

- c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the EMPLOYEES table by displaying the employee ID, first and last names, salary, job ID, and department ID.

Solution 10-1: Managing Data Integrity Rules and Mutating Table Exceptions

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their jobs. You create a trigger for this rule. During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

1. Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salaries. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employees' salaries, the CHECK_SALARY trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.
 - a. Update your EMP_PKG package (that you last updated in Practice 9) as follows:
 - 1) Add a procedure called SET_SALARY that updates the employees' salaries.
 - 2) The SET_SALARY procedure accepts the following two parameters: The job ID for those salaries that may have to be updated, and the new minimum salary for the job ID

Open sol_10.sql script from /home/oracle/labs/plpu/soln directory. Uncomment and select the code under Task 1_a. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown as follows. The newly added code is highlighted in bold letters in the following code box.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS

  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

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

  PROCEDURE add_employee(
```

```

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

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

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

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

PROCEDURE get_employees(p.dept_id
employees.department_id%type);

PROCEDURE init_departments;

PROCEDURE print_employee(p.rec_emp employees%rowtype);

PROCEDURE show_employees;

-- New set_salary procedure

PROCEDURE set_salary(p.jobid VARCHAR2, p.min_salary NUMBER);

END emp_pkg;
/
SHOW ERRORS

-- Package BODY

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

    valid_departments boolean_tab_type;
    emp_table          emp_tab_type;

    FUNCTION valid_deptid(p.deptid IN
departments.department_id%TYPE)

```

```

        RETURN BOOLEAN;

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

    PROCEDURE audit_newemp IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        user_id VARCHAR2(30) := USER;
    BEGIN
        INSERT INTO log_newemp (entry_id, user_id, log_time, name)
        VALUES (log_newemp_seq.NEXTVAL, user_id,
        sysdate,p_first_name||' '||p_last_name);
        COMMIT;
    END audit_newemp;

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

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

```

```

BEGIN
    p_email := UPPER(SUBSTR(p_first_name, 1,
1) || SUBSTR(p_last_name, 1, 7));
    add_employee(p_first_name, p_last_name, p_email, p_deptid =>
p_deptid);
END;

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

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

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

PROCEDURE get_employees(p.dept_id
employees.department_id%type) IS
BEGIN
    SELECT * BULK COLLECT INTO emp_table
    FROM EMPLOYEES

```

```

        WHERE department_id = p_dept_id;
END;

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

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

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

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

```

```
-- New set_salary procedure
PROCEDURE set_salary(p_jobid VARCHAR2, p_min_salary NUMBER) IS
CURSOR cur_emp IS
  SELECT employee_id
  FROM employees
  WHERE job_id = p_jobid AND salary < p_min_salary;
BEGIN
  FOR rec_emp IN cur_emp
  LOOP
    UPDATE employees
      SET salary = p_min_salary
      WHERE employee_id = rec_emp.employee_id;
  END LOOP;
END set_salary;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. It contains the following text:

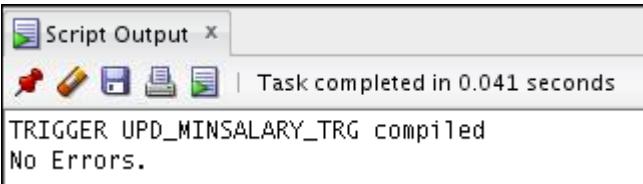
```
PACKAGE EMP_PKG compiled
No Errors.
PACKAGE BODY EMP_PKG compiled
No Errors.
```

Below the text, there is a status bar that says 'Task completed in 0.149 seconds'. The window has standard toolbar icons for copy, paste, save, etc.

- b. Create a row trigger named UPD_MINSALARY_TRG on the JOBS table that invokes the EMP_PKG.SET_SALARY procedure, when the minimum salary in the JOBS table is updated for a specified job ID.

Uncomment and select the code under Task 1_b. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER upd_minsalary_trg
AFTER UPDATE OF min_salary ON JOBS
FOR EACH ROW
BEGIN
  emp_pkg.set_salary(:new.job_id, :new.min_salary);
END;
/
SHOW ERRORS
```



- c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their `JOB_ID` is '`IT_PROG`'. Then, update the minimum salary in the `JOBS` table to increase it by \$1,000. What happens?

Uncomment and select the code under Task 1_c. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT employee_id, last_name, salary  
FROM employees  
WHERE job_id = 'IT_PROG';  
  
UPDATE jobs  
    SET min_salary = min_salary + 1000  
WHERE job_id = 'IT_PROG';
```

Script Output | Task completed in 0.044 seconds

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	7260
105	Austin	4800
106	Pataballa	4800
107	Lorentz	4200
214	Beh	9000

6 rows selected

Error starting at line 235 in command:

```
UPDATE jobs
   SET min_salary = min_salary + 1000
 WHERE job_id = 'IT_PROG'
Error report:
SQL Error: ORA-04091: table ORA61.JOB$ is mutating, trigger/function may not see it
ORA-06512: at "ORA61.CHECK_SALARY", line 5
ORA-06512: at "ORA61.SALARY_CHECK", line 1
ORA-04088: error during execution of trigger 'ORA61.SALARY_CHECK'
ORA-06512: at "ORA61.EMP_PKG", line 139
ORA-06512: at "ORA61.UPD_MINSALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA61.UPD_MINSALARY_TRG'
ORA-04091. 00000 - "table %s.%s is mutating, trigger/function may not see it"
*Cause: A trigger (or a user defined plsql function that is referenced in
this statement) attempted to look at (or modify) a table that was
in the middle of being modified by the statement which fired it.
*Action: Rewrite the trigger (or function) so it does not read that table.
```

The update of the min_salary column for job 'IT_PROG' fails because the UPD_MINSALARY_TRG trigger on the JOBS table attempts to update the employees' salaries by calling the EMP_PKG.SET_SALARY procedure. The SET_SALARY procedure causes the CHECK_SALARY_TRG trigger to fire (a cascading effect). The CHECK_SALARY_TRG calls the CHECK_SALARY procedure, which attempts to read the JOBS table data. While reading the JOBS table, the CHECK_SALARY procedure encounters the mutating table exception.

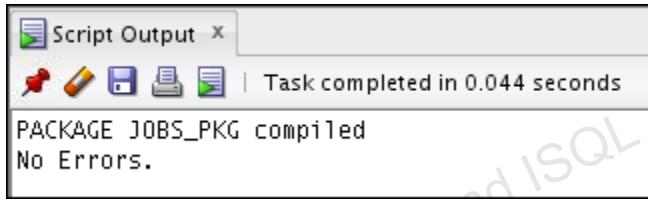
2. To resolve the mutating table issue, create a JOBS_PKG package to maintain in memory a copy of the rows in the JOBS table. Next, modify the CHECK_SALARY procedure to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, you must create a BEFORE INSERT OR UPDATE statement trigger on the EMPLOYEES table to initialize the JOBS_PKG package state before the CHECK_SALARY row trigger is fired.
 - a. Create a new package called JOBS_PKG with the following specification:

```
PROCEDURE initialize;
FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER;
FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(p_jobid VARCHAR2, min_salary
```

```
        NUMBER) ;
PROCEDURE set_maxsalary(p_jobid VARCHAR2, max_salary
        NUMBER) ;
```

Uncomment and select the code under Task 2_a, or copy and paste the following code in the SQL Worksheet area. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE PACKAGE jobs_pkg IS
    PROCEDURE initialize;
    FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER;
    FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER;
    PROCEDURE set_minsalary(p_jobid VARCHAR2, p_min_salary
        NUMBER);
    PROCEDURE set_maxsalary(p_jobid VARCHAR2, p_max_salary
        NUMBER);
END jobs_pkg;
/
SHOW ERRORS
```



- b. Implement the body of JOBS_PKG as follows:
- 1) Declare a private PL/SQL index-by table called `jobs_tab_type` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.
 - 2) Declare a private variable called `jobstab` based on the `jobs_tab_type`.
 - 3) The `INITIALIZE` procedure reads the rows in the `JOBS` table by using a cursor loop, and uses the `JOB_ID` value for the `jobstab` index that is assigned its corresponding row.
 - 4) The `GET_MINSalary` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `min_salary` for that element.
 - 5) The `GET_MAXSalary` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `max_salary` for that element.
 - 6) The `SET_MINSalary` procedure uses its `p_jobid` as an index to the `jobstab` to set the `min_salary` field of its element to the value in the `min_salary` parameter.
 - 7) The `SET_MAXSalary` procedure uses its `p_jobid` as an index to the `jobstab` to set the `max_salary` field of its element to the value in the `max_salary` parameter.

Uncomment and select the code under Task 2_b. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below. To compile the package's body, right-click the package's name or body in the Object Navigator tree, and then select Compile.

```
CREATE OR REPLACE PACKAGE BODY jobs_pkg IS
    TYPE jobs_tab_type IS TABLE OF jobs%rowtype
        INDEX BY jobs.job_id%type;
    jobstab jobs_tab_type;

    PROCEDURE initialize IS
    BEGIN
        FOR rec_job IN (SELECT * FROM jobs)
        LOOP
            jobstab(rec_job.job_id) := rec_job;
        END LOOP;
    END initialize;

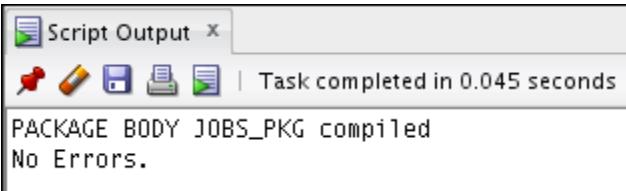
    FUNCTION get_minsalary(p_jobid VARCHAR2) RETURN NUMBER IS
    BEGIN
        RETURN jobstab(p_jobid).min_salary;
    END get_minsalary;

    FUNCTION get_maxsalary(p_jobid VARCHAR2) RETURN NUMBER IS
    BEGIN
        RETURN jobstab(p_jobid).max_salary;
    END get_maxsalary;

    PROCEDURE set_minsalary(p_jobid VARCHAR2, p_min_salary NUMBER)
    IS
    BEGIN
        jobstab(p_jobid).max_salary := p_min_salary;
    END set_minsalary;

    PROCEDURE set_maxsalary(p_jobid VARCHAR2, p_max_salary NUMBER)
    IS
    BEGIN
        jobstab(p_jobid).max_salary := p_max_salary;
    END set_maxsalary;

END jobs_pkg;
/
SHOW ERRORS
```



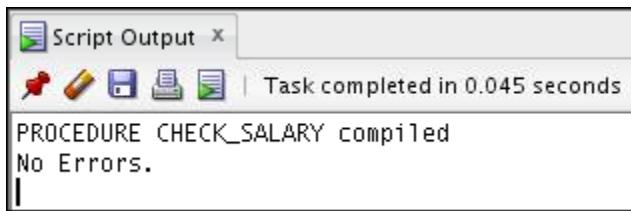
- c. Copy the CHECK_SALARY procedure from the practice titled “Creating Triggers,” Practice 9-1, and modify the code by replacing the query on the JOBS table with statements to set the local `minsal` and `maxsal` variables with values from the JOBS_PKG data by calling the appropriate `GET_*`SALARY functions. This step should eliminate the mutating trigger exception.

Uncomment and select the code under Task 2_c. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
    v_minsal jobs.min_salary%type;
    v_maxsal jobs.max_salary%type;
BEGIN

    -- Commented out to avoid mutating trigger exception on the
    -- JOBS table
    --SELECT min_salary, max_salary INTO v_minsal, v_maxsal
    --FROM jobs
    --WHERE job_id = UPPER(p_the_job);

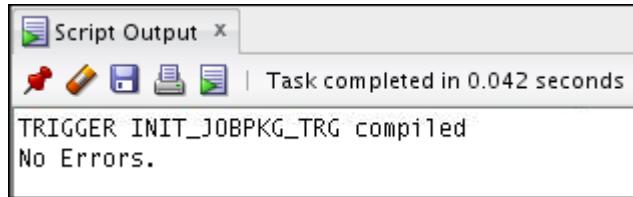
    v_minsal := jobs_pkg.get_minsalary(UPPER(p_the_job));
    v_maxsal := jobs_pkg.get_maxsalary(UPPER(p_the_job));
    IF p_the_salary NOT BETWEEN v_minsal AND v_maxsal THEN
        RAISE_APPLICATION_ERROR(-20100,
            'Invalid salary $'||p_the_salary||'. ||
            'Salaries for job '|| p_the_job ||
            ' must be between $'|| v_minsal ||' and $' || v_maxsal);
    END IF;
END;
/
SHOW ERRORS
```



- d. Implement a BEFORE INSERT OR UPDATE statement trigger called INIT_JOBPKG_TRG that uses the CALL syntax to invoke the JOBS_PKG.INITIALIZE procedure to ensure that the package state is current before the DML operations are performed.

Uncomment and select the code under Task 2_d. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE OR REPLACE TRIGGER init_jobpkg_trg
BEFORE INSERT OR UPDATE ON jobs
CALL jobs_pkg.initialize
/
SHOW ERRORS
```



- e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the IT_PROG job type by 1,000 in the JOBS table. Follow this up with a query on the employees with the IT_PROG job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?

Uncomment and select the code under Task 2_e. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

UPDATE jobs
SET min_salary = min_salary + 1000
WHERE job_id = 'IT_PROG';

SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';
```

```
Script Output x | Task completed in 0.066 seconds
EMPLOYEE_ID LAST_NAME          SALARY
-----
103 Hunold           9000
104 Ernst            7260
105 Austin           4800
106 Pataballa        4800
107 Lorentz          4200
214 Beh              9000

6 rows selected

1 rows updated.
EMPLOYEE_ID LAST_NAME          SALARY
-----
103 Hunold           9000
104 Ernst            7260
105 Austin           5000
106 Pataballa        5000
107 Lorentz          5000
214 Beh              9000

6 rows selected
```

The employees with last names Austin, Pataballa, and Lorentz have all had their salaries updated. No exception occurred during this process, and you implemented a solution for the mutating table trigger exception.

3. Because the CHECK_SALARY procedure is fired by CHECK_SALARY_TRG before inserting or updating an employee, you must check whether this still works as expected.
 - a. Test this by adding a new employee using EMP_PKG.ADD_EMPLOYEE with the following parameters: ('Steve', 'Morse', 'SMORSE', and sal => 6500). What happens?

Uncomment and select the code under Task 3_a. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

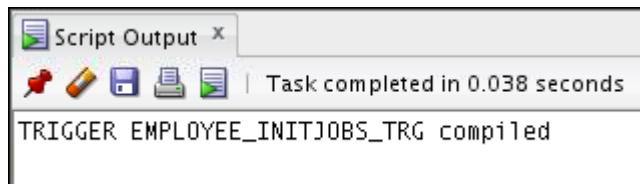
```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal => 6500)
```

```
Script Output x | Task completed in 1.013 seconds
anonymous block completed
```

- b. To correct the problem encountered when adding or updating an employee:
- 1) Create a BEFORE INSERT OR UPDATE statement trigger called EMPLOYEE_INITJOBS_TRG on the EMPLOYEES table that calls the JOBS_PKG.INITIALIZE procedure.
 - 2) Use the CALL syntax in the trigger body.

Uncomment and select the code under Task 3_b. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE TRIGGER employee_initjobs_trg  
BEFORE INSERT OR UPDATE OF job_id, salary ON employees  
CALL jobs_pkg.initialize  
/
```



- c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the EMPLOYEES table by displaying the employee ID, first and last names, salary, job ID, and department ID.

```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal  
=> 6500)  
/  
SELECT employee_id, first_name, last_name, salary, job_id,  
department_id  
FROM employees  
WHERE last_name = 'Morse';
```

Uncomment and select the code under Task 3_c. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

The screenshot shows the 'Script Output' window from the Oracle SQL Worksheet. The title bar says 'Script Output'. Below it is a toolbar with icons for Run, Stop, Save, Print, and Run Script (F5). A status bar at the bottom says 'Task completed in 0.004 seconds'. The main area displays an error message and a table output.

Error starting at line 384 in command:
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal => 6500)
Error report:
ORA-00001: unique constraint (ORA61.EMP_EMAIL_UK) violated
ORA-06512: at "ORA61.EMP_PKG", line 33
ORA-06512: at line 1
00001. 00000 - "unique constraint (%s.%s) violated"
*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.
For Trusted Oracle configured in DBMS MAC mode, you may see
this message if a duplicate entry exists at a different level.
*Action: Either remove the unique restriction or do not insert the key.
log_execution: Employee Inserted

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	DEPARTMENT_ID
215	Steve	Morse	6500	SA_REP	30

Practices for Lesson 11: Using the PL/SQL Compiler

Chapter 11

Practices for Lesson 11

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_11.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 11-1: Using the PL/SQL Compiler Parameters and Warnings

Overview

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

Note: Execute `cleanup_11.sql` script from `/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

1. Display the following information about compiler-initialization parameters by using the `USER_PLSQL_OBJECT_SETTINGS` data dictionary view. Note the settings for the `ADD_JOB_HISTORY` object.
Note: Click the Execute Statement icon (or press F9) to display the results on the Results tab.
 - a. Object name
 - b. Object type
 - c. The object's compilation mode
 - d. The compilation optimization level
2. Alter the `PLSQL_CODE_TYPE` parameter to enable native compilation for your session, and compile `ADD_DEPARTMENT`.
 - a. Execute the `ALTER SESSION` command to enable native compilation for the session.
 - b. Compile the `ADD_DEPARTMENT` procedure.
 - c. Rerun the code under Task 1 in `sol_11` script. Note the `PLSQL_CODE_TYPE` parameter.
 - d. Switch compilation to use interpreted compilation mode as follows:
3. Use the Tools > Preferences>Database > PL/SQL Compiler Options region to disable all compiler warnings categories.
4. Edit, examine, and execute the `lab_11_04.sql` script to create the `UNREACHABLE_CODE` procedure. Click the Run Script icon (or press F5) to create the procedure. Use the procedure name in the Navigation tree to compile the procedure.
5. What are the compiler warnings that are displayed in the Compiler – Log tab, if any?
6. Enable all compiler-warning messages for this session using the Preferences window.
7. Recompile the `UNREACHABLE_CODE` procedure using the Object Navigation tree. What compiler warnings are displayed, if any?
8. Use the `USER_ERRORS` data dictionary view to display the compiler-warning messages details as follows.
9. Create a script named `warning_msgs` that uses the `EXECUTE DBMS_OUTPUT` and the `DBMS_WARNING` packages to identify the categories for the following compiler-warning message numbers: 5050, 6075, and 7100. Enable `SERVERTOUTPUT` before running the script.

Solution 11-1: Using the PL/SQL Compiler Parameters and Warnings

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

- Display the following information about compiler-initialization parameters by using the `USER_PLSQL_OBJECT_SETTINGS` data dictionary view. Note the settings for the `ADD_JOB_HISTORY` object.

Note: Click the Execute Statement icon (or press F9) to display the results in the Results tab.

- Object name
- Object type
- The object's compilation mode
- The compilation optimization level

Uncomment and select the code under Task 1. Click the Execute Statement icon (or press F9) on the SQL Worksheet toolbar to run the query. The code and a sample of the results are shown below.

```
SELECT name, type, plsql_code_type as code_type,
       plsql_optimize_level as opt_lvl
  FROM user_plsql_object_settings;
```

NAME	TYPE	CODE_TYPE	OPT_LVL
1 ADD_COL	PROCEDURE	INTERPRETED	0
2 ADD_DEPARTMENT	PROCEDURE	INTERPRETED	0
3 ADD_DEPARTMENT_NOEX	PROCEDURE	INTERPRETED	0

Note: In this step, our focus is on the `ADD_DEPARTMENT` procedure. Please ignore the difference in screenshot, if any.

- Alter the `PLSQL_CODE_TYPE` parameter to enable native compilation for your session, and compile `ADD_JOB_HISTORY`.
 - Execute the `ALTER SESSION` command to enable native compilation for the session.

Uncomment and select the code under Task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

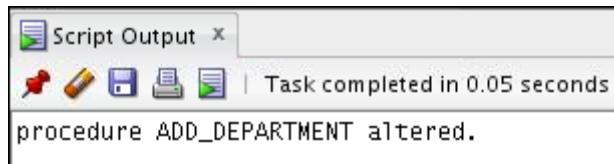
```
ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE';
```

Script Output
Task completed in 0.002 seconds
session SET altered.

- b. Compile the ADD_DEPARTMENT procedure.

Uncomment and select the code under Task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
ALTER PROCEDURE add_department COMPILE;
```



- c. Rerun the code under Task 1 from sol_11.sql script by clicking the Execute Statement icon (or pressing F9) on the SQL Worksheet. Note the PLSQL_CODE_TYPE parameter.

```
SELECT name, type, plsql_code_type as code_type,  
plsql_optimize_level as opt_lvl  
FROM user_plsql_object_settings;
```

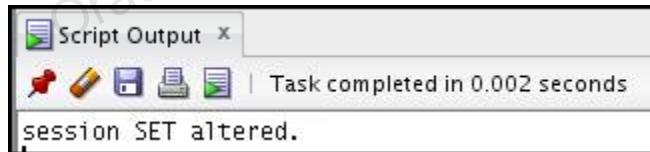
The screenshot shows a 'Query Result' window with a toolbar at the top containing icons for Run, Save, Print, and SQL. Below the toolbar, a message says 'Fetched 50 rows in 0.009 seconds'. The main area is a table with four columns: NAME, TYPE, CODE_TYPE, and OPT_LVL. The rows are numbered 1, 2, and 3. Row 2, which contains the data for 'ADD_DEPARTMENT', has a red border around it.

NAME	TYPE	CODE_TYPE	OPT_LVL
1 ADD_COL	PROCEDURE	INTERPRETED	0
2 ADD_DEPARTMENT	PROCEDURE	NATIVE	1
3 ADD_DEPARTMENT_NOEX	PROCEDURE	INTERPRETED	0

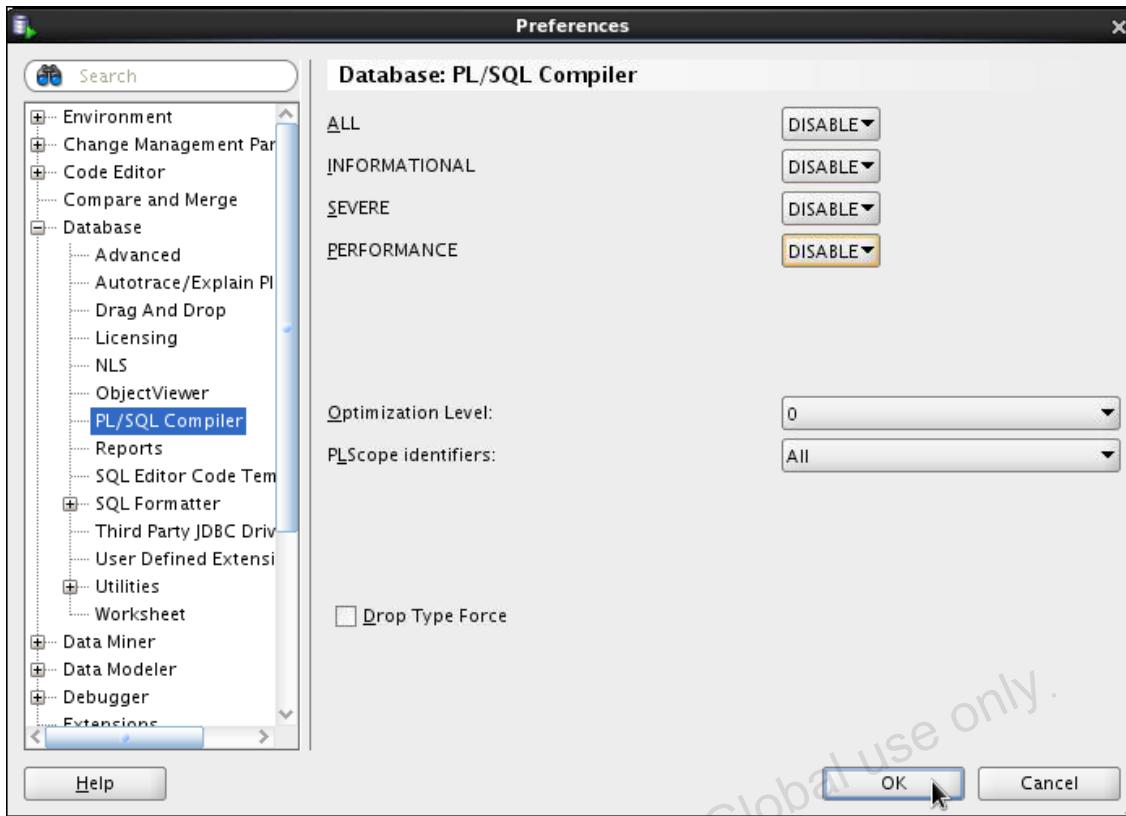
...

- d. Switch compilation to use interpreted compilation mode as follows:

```
ALTER SESSION SET PLSQL_CODE_TYPE = 'INTERPRETED';
```



3. Use the Tools > Preferences > Database > PL/SQL Compiler Options region to disable all compiler warnings categories.

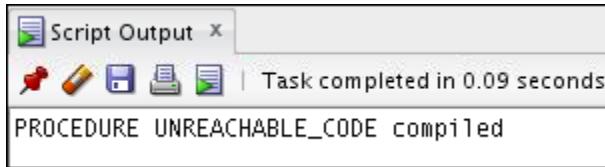


Select DISABLE for all four PL/SQL compiler warnings categories, and then click OK.

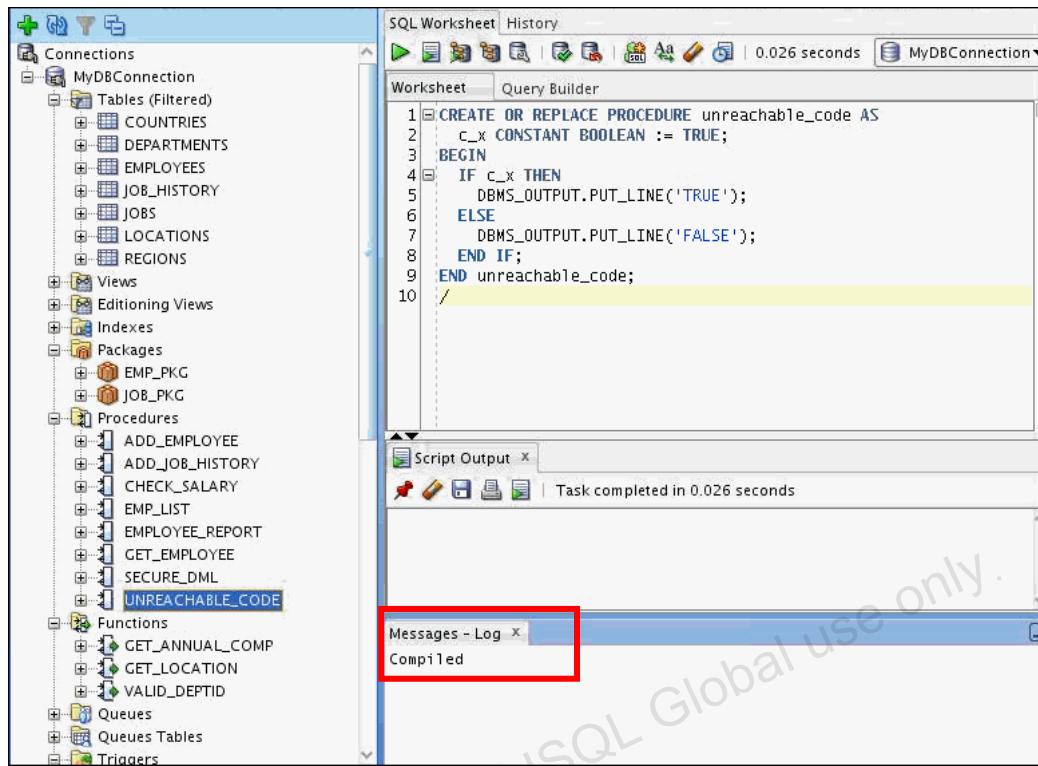
4. Edit, examine, and execute the `lab_11_04.sql` script to create the `UNREACHABLE_CODE` procedure. Click the Run Script icon (or press F5) to create and compile the procedure.

Uncomment and select the code under Task 4. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
CREATE OR REPLACE PROCEDURE unreachable_code AS
  c_x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF c_x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/
```



To view any compiler warning errors, right-click the procedure's name in the Procedures node in the Navigation tree, and then click Compile.



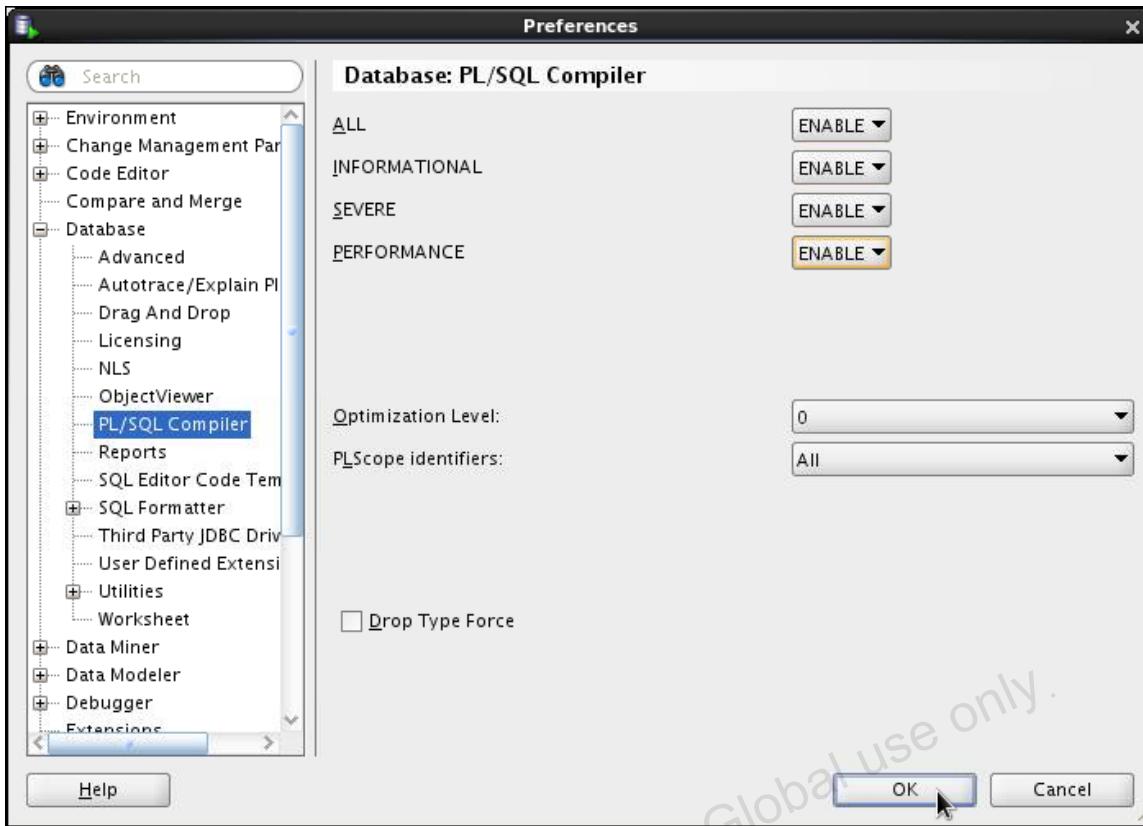
Note

- If the procedure is not displayed in the Navigation tree, click the Refresh icon on the Connections tab.
- Make sure your Messages – Log tab is displayed (select View > Log from the menu bar).

5. What are the compiler warnings that are displayed in the Compiler – Log tab, if any?

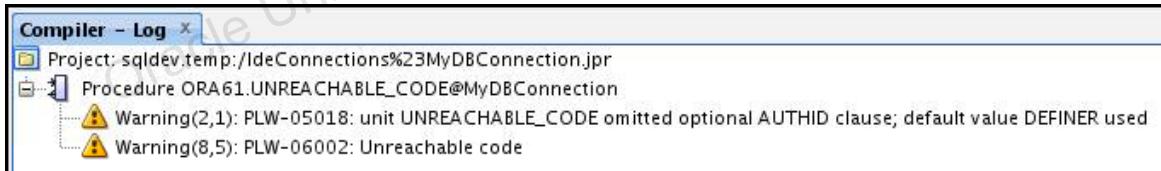
Note that the message on the Messages – Log tab is “Compiled” without any warning messages because you disabled the compiler warnings in step 3.

6. Enable all compiler-warning messages for this session using the Preferences window.
Select ENABLE for all four PL/SQL compiler warnings and then click OK.



7. Recompile the UNREACHABLE_CODE procedure using the Object Navigation tree. What compiler warnings are displayed, if any?

Right-click the procedure's name in the Object Navigation tree and select Compile. Note the messages displayed in the Compiler – Log tab.



Note: If you get the following two warnings in SQL Developer, it is expected in some versions of SQL Developer. If you do get the following warnings, it is because your version of SQL Developer still uses the Oracle 11g database deprecated PLSQL_DEBUG parameter.

Warning (1) :PLW-06015 :parameter PLSQL_DEBUG is deprecated ; use PLSQL_OPTIMIZE_LEVEL=1

Warning (1) :PLW-06013 :deprecated parameter PLSQL_DEBUG forces PLSQL_OPTIMIZE_LEVEL<=1

8. Use the USER_ERRORS data dictionary view to display the compiler-warning messages details as follows.

```
DESCRIBE user_errors
```

DESCRIBE user_errors		
Name	Null	Type
NAME	NOT NULL	VARCHAR2(128)
TYPE		VARCHAR2(12)
SEQUENCE	NOT NULL	NUMBER
LINE	NOT NULL	NUMBER
POSITION	NOT NULL	NUMBER
TEXT	NOT NULL	VARCHAR2(4000)
ATTRIBUTE		VARCHAR2(9)
MESSAGE_NUMBER		NUMBER

```
SELECT *
FROM user_errors;
```

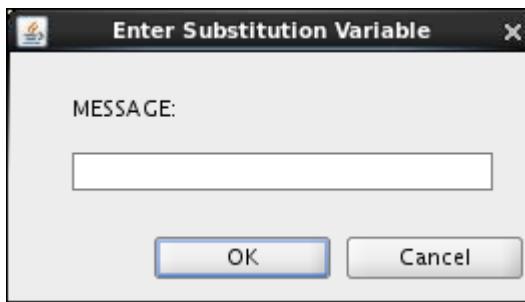
Query Result						
NAME	TYPE	SEQUENCE	LINE	POSITION	TEXT	ATTRIBUTE
1 UNREACHABLE_CODE PROCEDURE		1	1		1 PLW-05018: unit UNREACHABLE_CODE omitted optional... WARNING	5018
2 UNREACHABLE_CODE PROCEDURE		2	7		5 PLW-06002: Unreachable code	6002

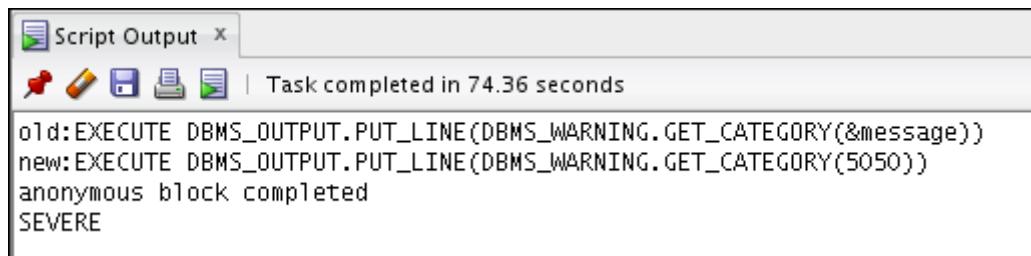
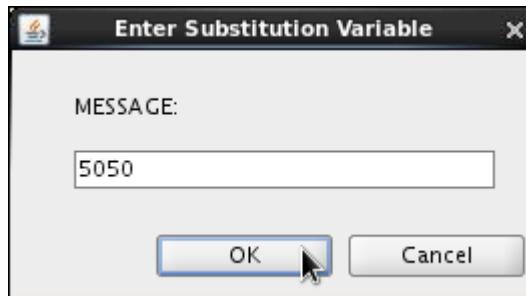
Note: The output was displayed on the Results tab because we used the F9 key to execute the SELECT statement. The results of the SELECT statement might be different depending on the amount of errors you had in your session.

9. Create a script named warning_msgs that uses the EXECUTE DBMS_OUTPUT and the DBMS_WARNING packages to identify the categories for the following compiler-warning message numbers: 5050, 6075, and 7100. Enable SERVEROUTPUT before running the script.

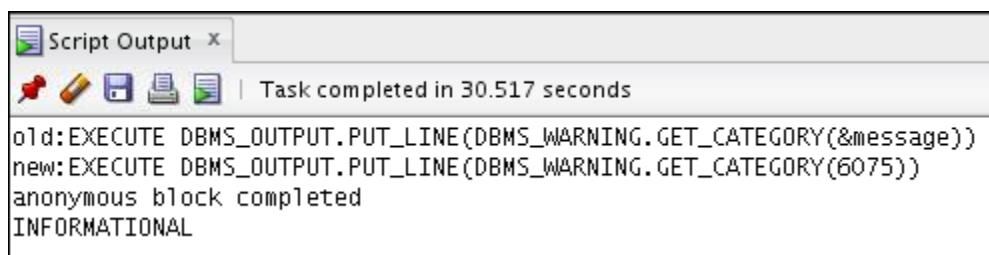
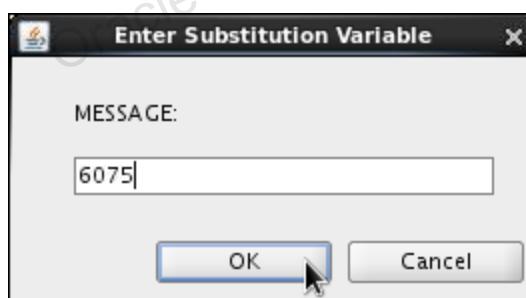
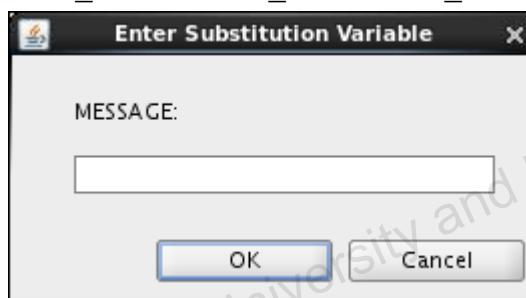
Uncomment and select the code under Task 9. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```

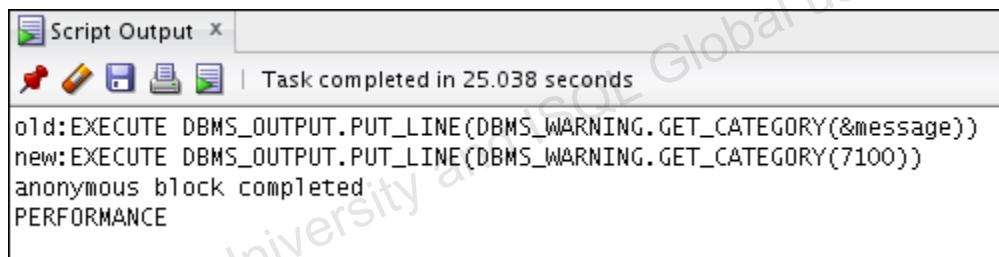
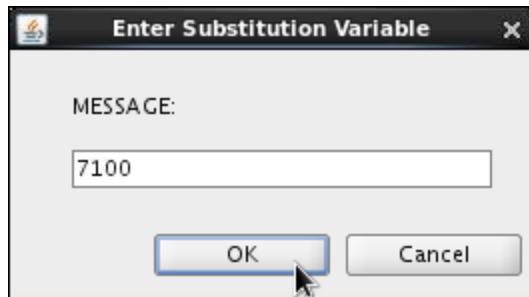
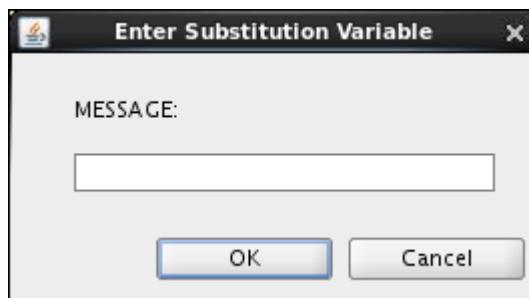




```
EXECUTE  
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```



```
EXECUTE  
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message)) ;
```



Unauthorized reproduction or distribution prohibited. Copyright 2012, Oracle and/or its affiliates.

Oracle University and ISQL Global use only.

Practices for Lesson 12: Managing Dependencies

Chapter 12

Practices for Lesson 12

Overview

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

Note:

1. Before starting this practice, execute
`/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_12.sql`
script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 12-1: Managing Dependencies in Your Schema

Overview

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

Note: Execute `cleanup_12.sql` script from

`/home/oracle/labs/plpu/code_ex/cleanup_scripts/` before performing the following tasks.

Task

1. Create a tree structure showing all dependencies involving your `add_employee` procedure and your `valid_deptid` function.

Note: Create `add_employee` procedure and `valid_deptid` function from Practice 3 of lesson titled “Creating Functions and Debugging Subprograms” before performing the tasks.

- a. Load and execute the `utldtree.sql` script, which is located in the `/home/oracle/labs/plpu/labs` directory.
- b. Execute the `deptree_fill` procedure for the `add_employee` procedure.
- c. Query the `IDEPTREE` view to see your results.
- d. Execute the `deptree_fill` procedure for the `valid_deptid` function.
- e. Query the `IDEPTREE` view to see your results.

If you have time, complete the following exercise:

2. Dynamically validate invalid objects.
 - a. Make a copy of your `EMPLOYEES` table, called `EMPS`.
 - b. Alter your `EMPLOYEES` table and add the column `TOTSAL` with data type `NUMBER(9, 2)`.
 - c. Create and save a query to display the name, type, and status of all invalid objects.
 - d. In the `compile_pkg` (created in Practice 7 of the lesson titled “Using Dynamic SQL”), add a procedure called `recompile` that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to alter the invalid object type and compile it.
 - e. Execute the `compile_pkg.recompile` procedure.
 - f. Run the script file that you created in step 3 c. to check the value of the `STATUS` column. Do you still have objects with an `INVALID` status?

Solution 12-1: Managing Dependencies in Your Schema

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

1. Create a tree structure showing all dependencies involving your add_employee procedure and your valid_deptid function.

Note: add_employee and valid_deptid were created in the Practice 3 of lesson titled “Creating Functions and Debugging Subprograms.” Execute the following code to create the add_employee procedure and valid_deptid function.

```

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

BEGIN
    SELECT 1
    INTO   v_dummy

```

```

    FROM      departments
    WHERE     department_id = p_deptid;
    RETURN   TRUE;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END valid_deptid;
/

```

- a. Load and execute the `utldtree.sql` script, which is located in the `/home/oracle/labs/plpu/labs` directory.

Open the `/home/oracle/labs/plpu/solns/utldtree.sql` script. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```

Rem
Rem $Header: utldtree.sql,v 3.2 2012/11/21 16:24:44 RKOOI Stab $
Rem
Rem Copyright (c) 1991 by Oracle Corporation
Rem NAME
Rem deptree.sql - Show objects recursively dependent on
Rem given object
Rem DESCRIPTION
Rem This procedure, view and temp table will allow you to see
Rem all objects that are (recursively) dependent on the given
Rem object.
Rem Note: you will only see objects for which you have
Rem permission.
Rem Examples:
Rem execute deptree_fill('procedure', 'scott', 'billing');
Rem select * from deptree order by seq#;
Rem
Rem execute deptree_fill('table', 'scott', 'emp');
Rem select * from deptree order by seq#;
Rem

Rem execute deptree_fill('package body', 'scott',
Rem 'accts_payable');
Rem select * from deptree order by seq#;
Rem
Rem A prettier way to display this information than
Rem select * from deptree order by seq#;

```

```

Rem is
Rem   select * from ideptree;
Rem   This shows the dependency relationship via indenting.
Rem   Notice that no order by clause is needed with ideptree.
Rem   RETURNS
Rem
Rem   NOTES
Rem   Run this script once for each schema that needs this
Rem   utility.
Rem   MODIFIED      (MM/DD/YY)
Rem   rkooi        10/26/92 - owner -> schema for SQL2
Rem   glumpkin     10/20/92 - Renamed from DEPTREE.SQL
Rem   rkooi        09/02/92 - change ORU errors
Rem   rkooi        06/10/92 - add rae errors
Rem   rkooi        01/13/92 - update for sys vs. regular user
Rem   rkooi        01/10/92 - fix ideptree
Rem   rkooi        01/10/92 - Better formatting, add ideptree
view
Rem   rkooi        12/02/91 - deal with cursors
Rem   rkooi        10/19/91 - Creation

DROP SEQUENCE deptree_seq
/
CREATE SEQUENCE deptree_seq cache 200
-- cache 200 to make sequence faster

/
DROP TABLE deptree_temptab
/
CREATE TABLE deptree_temptab
(
    object_id          number,
    referenced_object_id number,
    nest_level         number,
    seq#               number
)
/
CREATE OR REPLACE PROCEDURE deptree_fill (type char, schema
char, name char) IS
    obj_id number;
BEGIN
    DELETE FROM deptree_temptab;
    COMMIT;

```

```

SELECT object_id INTO obj_id FROM all_objects
  WHERE owner = upper(deptree_fill.schema)

AND    object_name  = upper(deptree_fill.name)
      AND    object_type   = upper(deptree_fill.type);
INSERT INTO deptree_temptab
  VALUES(obj_id, 0, 0, 0);
INSERT INTO deptree_temptab
  SELECT object_id, referenced_object_id,
         level, deptree_seq.nextval
    FROM public_dependency
   CONNECT BY PRIOR object_id = referenced_object_id
   START WITH referenced_object_id = deptree_fill.obj_id;
EXCEPTION
  WHEN no_data_found then
    raise_application_error(-20000, 'ORU-10013: ' ||
      type || ' ' || schema || '.' || name || ' was not
found.');
END;
/

```

```

DROP VIEW deptree
/

```

```

SET ECHO ON

REM This view will succeed if current user is sys. This view
REM shows which shared cursors depend on the given object. If
REM the current user is not sys, then this view get an error
REM either about lack of privileges or about the non-existence
REM of table x$kglnxs.

SET ECHO OFF
CREATE VIEW sys.deptree
  (nested_level, type, schema, name, seq#)
AS
  SELECT d.nest_level, o.object_type, o.owner, o.object_name,
d.seq#
    FROM deptree_temptab d, dba_objects o
   WHERE d.object_id = o.object_id (+)
UNION ALL
  SELECT d.nest_level+1, 'CURSOR', '<shared>',
'''||c.kglnaobj||'''', d.seq#+.5

```

```

        FROM deptree_temptab d, x$kgldp k, x$kglob g, obj$ o, user$ u,
x$kglob c,
          x$kglxss a
        WHERE d.object_id = o.obj#
        AND  o.name = g.kglnaobj
        AND  o.owner# = u.user#
        AND  u.name = g.kglnaown
        AND  g.kglhdadr = k.kglrfhdl
        AND  k.kglhdadr = a.kglhdadr    -- make sure it is not a
transitive
        AND  k.kgldepno = a.kglxsdep    -- reference, but a direct
one
        AND  k.kglhdadr = c.kglhdadr
        AND  c.kglhdns = 0    -- a cursor
/
SET ECHO ON

REM This view will succeed if current user is not sys. This view
REM does *not* show which shared cursors depend on the given
REM object.
REM If the current user is sys then this view will get an error
REM indicating that the view already exists (since prior view
REM create will have succeeded).

SET ECHO OFF
CREATE VIEW deptree
  (nested_level, type, schema, name, seq#)
AS
  select d.nest_level, o.object_type, o.owner, o.object_name,
d.seq#
  FROM deptree_temptab d, all_objects o
  WHERE d.object_id = o.object_id (+)
/
DROP VIEW ideptree
/
CREATE VIEW ideptree (dependencies)
AS
  SELECT lpad(' ',3*(max(nested_level))) || max(nvl(type, '<no
permission>'))
    || ' ' || schema || decode(type, NULL, '', '.') || name
  FROM deptree
  GROUP BY seq# /* So user can omit sort-by when selecting from
ideptree */
/

```

Script Output

| Task completed in 0.321 seconds

```
Error starting at line 43 in command:
drop sequence deptree_seq
Error report:
SQL Error: ORA-02289: sequence does not exist
02289. 00000 - "sequence does not exist"
*Cause: The specified sequence does not exist, or the user does
not have the required privilege to perform this operation.
*Action: Make sure the sequence name is correct, and that you have
the right to perform the desired operation on this sequence.
sequence DEPTREE_SEQ created.

Error starting at line 47 in command:
drop table deptree temptab
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
table DEPTREE_TEMP TAB created.
PROCEDURE deptree_fill compiled

Error starting at line 81 in command:
drop view deptree
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

> REM This view will succeed if current user is sys. This view shows
> REM which shared cursors depend on the given object. If the current
> REM user is not sys, then this view get an error either about lack
> REM of privileges or about the non-existence of table x$kg1xs.

Error starting at line 92 in command:
create view sys.deptree
  (nested_level, type, schema, name, seq#)
as
  select d.nest_level, o.object_type, o.owner, o.object_name, d.seq#
  from deptree temptab d, dba_objects o
  where d.object_id = o.object_id (+)
union all
  select d.nest_level+1, 'CURSOR', '<shared>', ''||c.kglnaobj||'', d.seq#+.5
  from deptree temptab d, x$kgldp k, x$kglob g, obj$ o, user$ u, x$kglob c,
       x$kg1xs a
  where d.object_id = o.obj#
  and   o.name = g.kglnaobj
  and   o.owner# = u.user#
  and   u.name = g.kglnaown
  and   g.kglhdadr = k.kglrfhdl /* make sure it is not a transitive */
  and   k.kgldepno = a.kglxdep /* reference, but a direct one */
  and   k.kglhdadr = c.kglhdadr
  and   c.kglhdnsp = 0 /* a cursor */
Error at Command Line:96 Column:7
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
```

```
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
> REM This view will succeed if current user is not sys. This view
> REM does *not* show which shared cursors depend on the given object.
> REM If the current user is sys then this view will get an error
> REM indicating that the view already exists (since prior view create
> REM will have succeeded).
view DEPTREE created.
Error starting at line 130 in command:
drop view ideptree
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
view IDEPTREE created.
sequence DEPTREE_SEQ dropped.
sequence DEPTREE_SEQ created.
table DEPTREE_TEMP TAB dropped.
table DEPTREE_TEMP TAB created.
PROCEDURE deptree_fill compiled
view DEPTREE dropped.
```

```
> REM This view will succeed if current user is sys. This view shows
> REM which shared cursors depend on the given object. If the current
> REM user is not sys, then this view get an error either about lack
> REM of privileges or about the non-existence of table x$kg1xs.

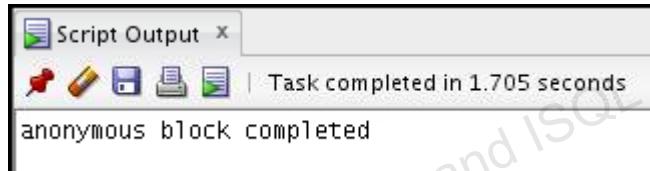
Error starting at line 92 in command:
create view sys.deptree
  (nested_level, type, schema, name, seq#)
as
  select d.nest_level, o.object_type, o.owner, o.object_name, d.seq#
  from deptree_temp tab d, dba_objects o
  where d.object_id = o.object_id (+)
union all
  select d.nest_level+1, 'CURSOR', '<shared>', ''||c.kglnaobj||'', d.seq#+.5
  from deptree_temp tab d, x$kgldp k, x$kglob g, obj$ o, user$ u, x$kglob c,
       x$kg1xs a
  where d.object_id = o.obj#
  and   o.name = g.kglnaobj
  and   o.owner# = u.user#
  and   u.name = g.kglnaown
  and   g.kglhdadr = k.kglrfhd1
  and   k.kglhdadr = a.kglhdadr /* make sure it is not a transitive */
  and   k.kgldepno = a.kglxsdep /* reference, but a direct one */
  and   k.kglhdadr = c.kglhdadr
  and   c.kglhdns = 0 /* a cursor */
```

```
Error at Command Line:96 Column:7
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
> REM This view will succeed if current user is not sys. This view
> REM does *not* show which shared cursors depend on the given object.
> REM If the current user is sys then this view will get an error
> REM indicating that the view already exists (since prior view create
> REM will have succeeded).
view DEPTREE created.
view IDEPTREE dropped.
view IDEPTREE created.
```

- b. Execute the deptree_fill procedure for the add_employee procedure.

Open the /home/oracle/labs/plpu/solns/sol_12.sql script. Uncomment and select the code under task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

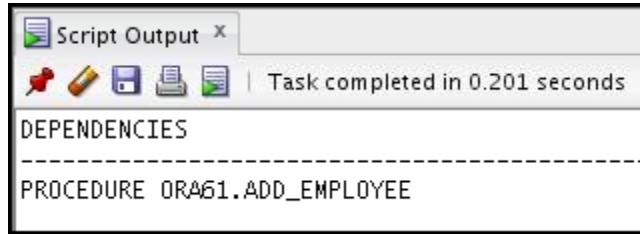
```
EXECUTE deptree_fill('PROCEDURE', USER, 'add_employee')
```



- c. Query the IDEPTREE view to see your results.

Uncomment and select the code under task 1_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

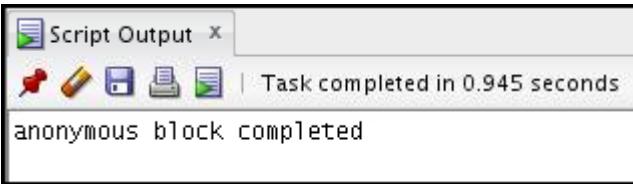
```
SELECT * FROM IDEPTREE;
```



- d. Execute the deptree_fill procedure for the valid_deptid function.

Uncomment and select the code under task 1_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

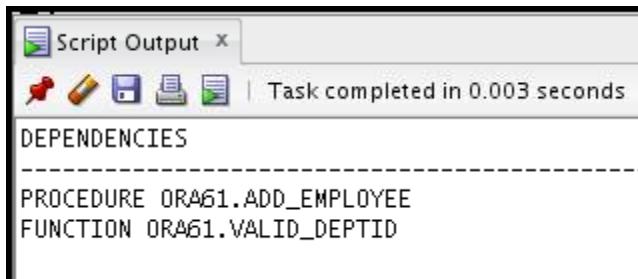
```
EXECUTE deptree_fill('FUNCTION', USER, 'valid_deptid')
```



- e. Query the IDEPTREE view to see your results.

Uncomment and select the code under task 1_e. Click the Execute Statement icon (or press F9) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT * FROM IDEPTREE;
```

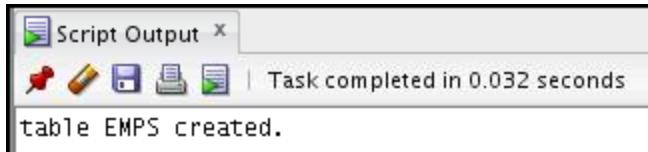


If you have time, complete the following exercise:

2. Dynamically validate invalid objects.
 - a. Make a copy of your EMPLOYEES table, called EMPS.

Uncomment and select the code under task 2_a. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
CREATE TABLE emps AS
SELECT * FROM employees;
```



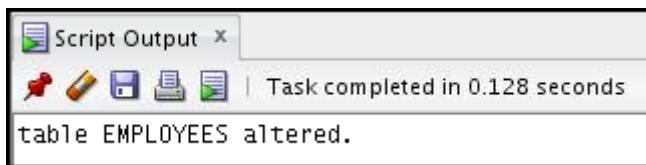
Note: Please ignore the error message, if any while executing the CREATE statement.

- b. Alter your EMPLOYEES table and add the column TOTSAL with data type NUMBER (9, 2).

Uncomment and select the code under task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
ALTER TABLE employees
```

```
ADD (totals NUMBER(9,2)) ;
```



- c. Create and save a query to display the name, type, and status of all invalid objects.

Uncomment and select the code under task 2_c. Click the Execute Statement icon (or press F9) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
1 COMPILE_PKG	PACKAGE BODY	INVALID
2 DELETE_EMP_TRG	TRIGGER	INVALID
3 CHECK_SALARY_TRG	TRIGGER	INVALID
4 EMP_PKG	PACKAGE BODY	INVALID
5 EMP_PKG	PACKAGE	INVALID
6 UPDATE_JOB_HISTORY	TRIGGER	INVALID

Note: Please ignore the difference in the screenshot.

- d. In the `compile_pkg` (created in Practice 7 of the lesson titled “Using Dynamic SQL”), add a procedure called `recompile` that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to alter the invalid object type and compile it.

Uncomment and select the code under task 2_d. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below. The newly added code is highlighted in bold letters in the following code box.

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

CREATE OR REPLACE PACKAGE BODY compile_pkg_body IS
```

```

PROCEDURE p_execute(stmt VARCHAR2) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE(stmt);
    EXECUTE IMMEDIATE stmt;
END;

FUNCTION get_type(p_name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
BEGIN
    -- The ROWNUM = 1 is added to the condition
    -- to ensure only one row is returned if the
    -- name represents a PACKAGE, which may also
    -- have a PACKAGE BODY. In this case, we can
    -- only compile the complete package, but not
    -- the specification or body as separate
    -- components.
    SELECT object_type INTO proc_type
    FROM user_objects
    WHERE object_name = UPPER(p_name)
    AND ROWNUM = 1;
    RETURN proc_type;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN NULL;
END;

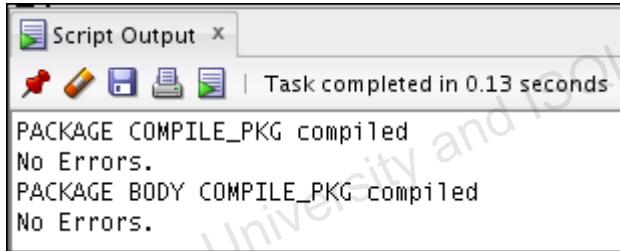
PROCEDURE make(p_name VARCHAR2) IS
    stmt      VARCHAR2(100);
    proc_type VARCHAR2(30) := get_type(p_name);
BEGIN
    IF proc_type IS NOT NULL THEN
        stmt := 'ALTER '|| proc_type ||' ''|| p_name ||' COMPILE';
        p_execute(stmt);
    ELSE
        RAISE_APPLICATION_ERROR(-20001,
            'Subprogram '''|| p_name ||''' does not exist');
    END IF;
END make;

PROCEDURE recompile IS

```

```
stmt VARCHAR2(200);
obj_name user_objects.object_name%type;
obj_type user_objects.object_type%type;
BEGIN
FOR objrec IN (SELECT object_name, object_type
                FROM user_objects
               WHERE status = 'INVALID'
                 AND object_type <> 'PACKAGE BODY')
LOOP
stmt := 'ALTER '|| objrec.object_type ||' ' ||
        objrec.object_name ||' COMPILE';
p_execute(stmt);
END LOOP;
END recompile;

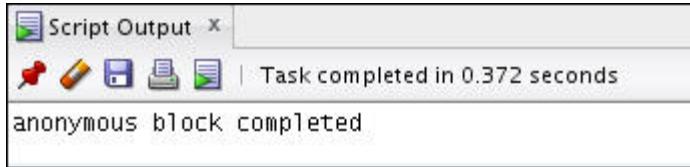
END compile_pkg;
/
SHOW ERRORS
```



- e. Execute the `compile_pkg.recompile` procedure.

Uncomment and select the code under task 2_e. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
EXECUTE compile_pkg.recompile
```



Note: If you come across an error message in the screenshot, please ignore. The procedure would have been compiled.

- f. Run the script file that you created in step 2_c to check the value of the STATUS column. Do you still have objects with an INVALID status?

Uncomment and select the code under task 2_f. Click the Execute Statement icon (or press F9) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
SELECT object_name, object_type, status  
FROM USER_OBJECTS  
WHERE status = 'INVALID';
```

Query Result		
SQL All Rows Fetched: 0 in 0.068 seconds		
OBJECT_NAME	OBJECT_TYPE	STATUS

Note: Compare this output to the output in step 2(c). You see that all the objects from the previous screenshot are now valid.

Additional Practices 1

Chapter 13

Additional Practices 1

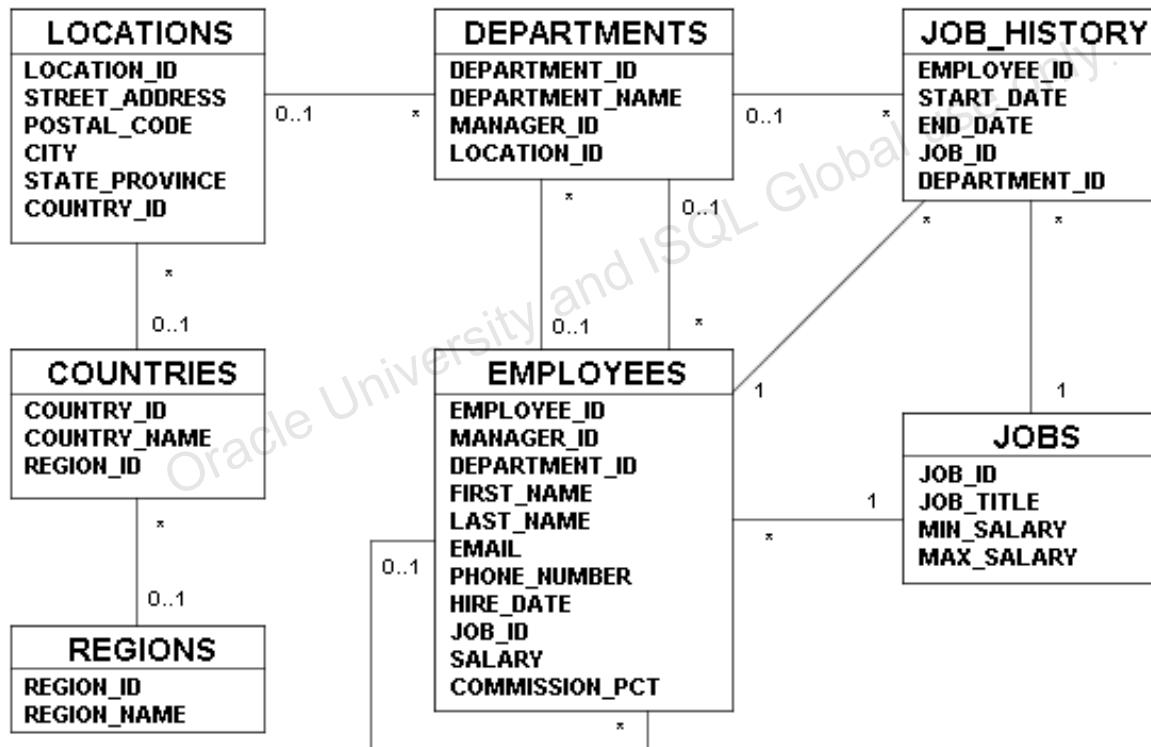
Overview

The additional practices are provided as a supplement to the course *Oracle Database: Develop PL/SQL Program Units*. In these practices, you apply the concepts that you learned in the course. The additional practices comprise two lessons.

Lesson 1 provides supplemental exercises to create stored procedures, functions, packages, and triggers, and to use the Oracle-supplied packages with SQL Developer or SQL*Plus as the development environment. The tables used in this portion of the additional practice include EMPLOYEES, JOBS, JOB_HISTORY, and DEPARTMENTS.

An entity relationship diagram is provided at the start of each practice. Each entity relationship diagram displays the table entities and their relationships. More detailed definitions of the tables and the data contained in them is provided in the appendix titled “Additional Practices: Table Descriptions and Data.”

The Human Resources (HR) Schema Entity Relationship Diagram



Practice 1-1: Creating a New SQL Developer Database Connection

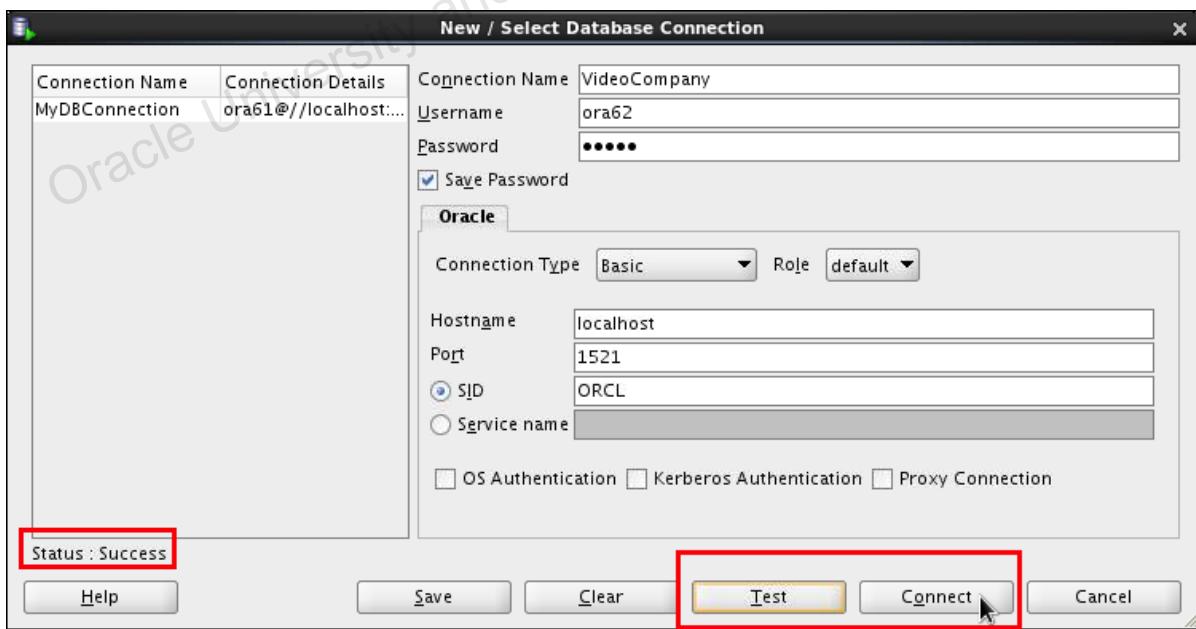
Overview

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

Start up SQL Developer using the user ID and password that are provided to you by the instructor such as ora62.

Task

1. Start up SQL Developer using the user ID and password that are provided to you by the instructor such as ora62.
2. Create a database connection using the following information:
 - a. Connection Name: VideoCompany
 - b. Username: ora62
 - c. Password: ora62
 - d. Select the Save Password check box.
 - e. Hostname: Enter the host name for your PC or alternatively mention localhost
 - f. Port: 1521
 - g. SID: ORCL
3. Test the new connection. If the Status shows as Success, connect to the database using this new connection:
 - a. Click the Test button in the New/Select Database Connection window. If the status shows as Success, click the Connect button.



Solution 1-1: Creating a New SQL Developer Database Connection

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

1. Start up SQL Developer using the user ID and password that are provided to you by the instructor, such as ora62.

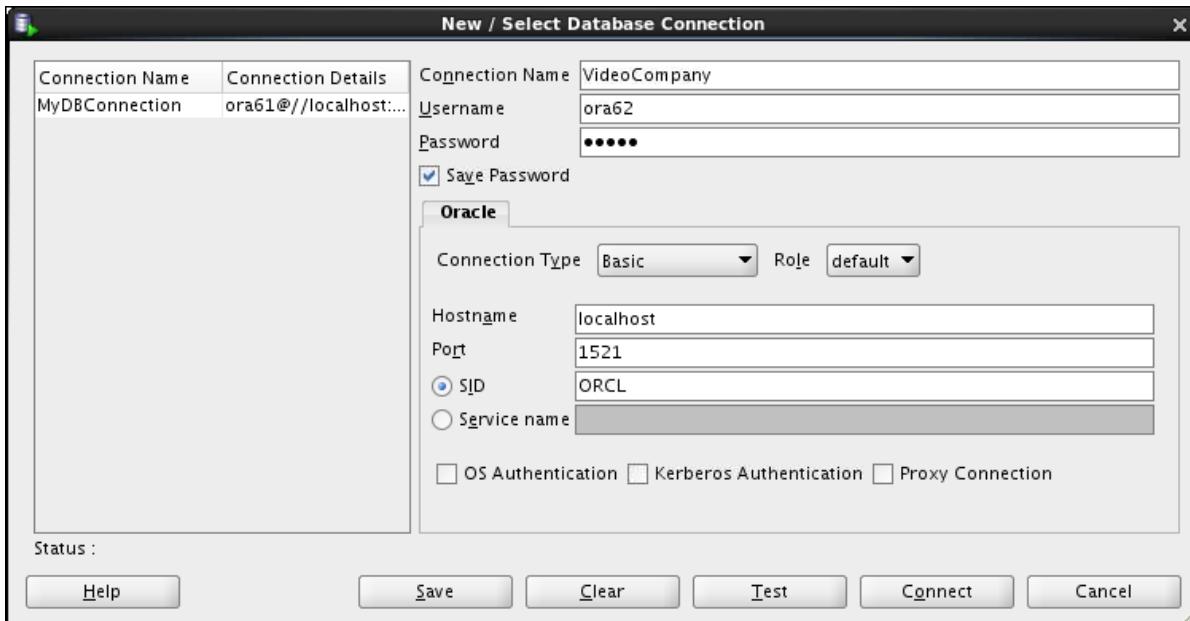
Click the SQL Developer icon on your desktop.



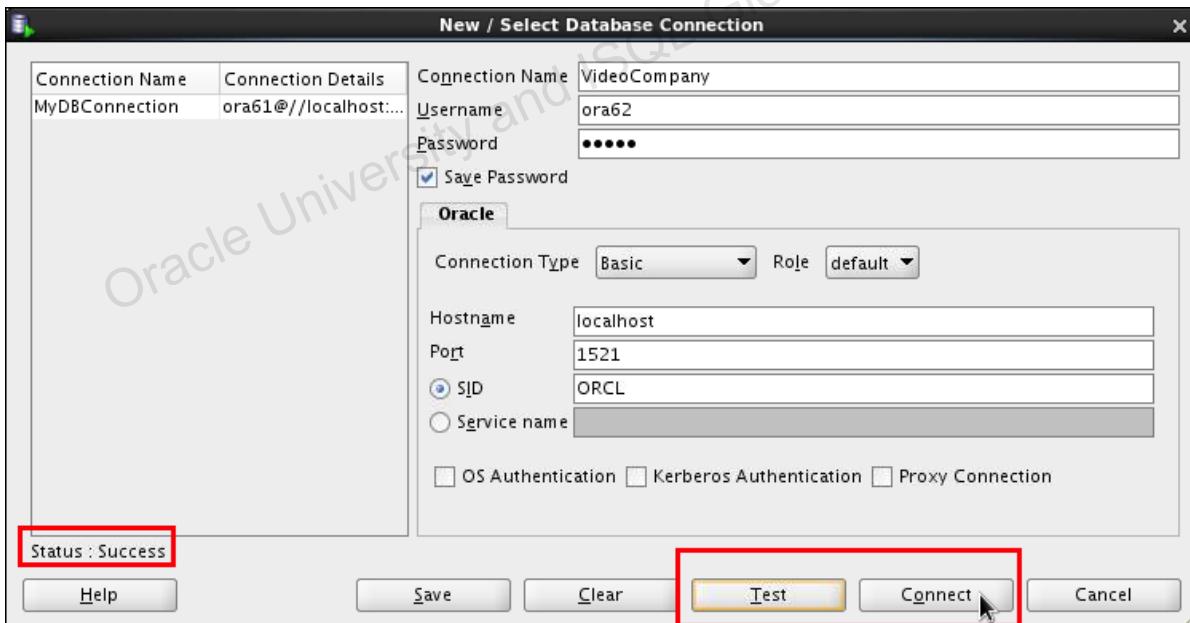
2. Create a database connection using the following information:
 - a. Connection Name: VideoCompany
 - b. Username: ora62
 - c. Password: ora62
 - d. Hostname: Enter the host name for your PC or let the default localhost remain.
 - e. Port: 1521
 - f. SID: ORCL

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

Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Substitute the username, password, host name, and service name with the appropriate information as provided by your instructor. The following is a sample of the newly created database connection for student ora62:



3. Test the new connection. If the status shows as Success, connect to the database using this new connection:
 - a. Click the Test button in the New/Select Database Connection window. If the status shows as Success, click the Connect button.



Practice 1-2: Adding a New Job to the JOBS Table

Overview

In this practice, you create a subprogram to add a new job into the JOBS table.

Tasks

1. Create a stored procedure called NEW_JOB to enter a new order into the JOBS table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.
2. Enable SERVEROUTPUT, and then invoke the procedure to add a new job with job ID 'SY_ANAL', job title 'System Analyst', and minimum salary of 6000.
3. Check whether a row was added and note the new job ID for use in the next exercise. Commit the changes.

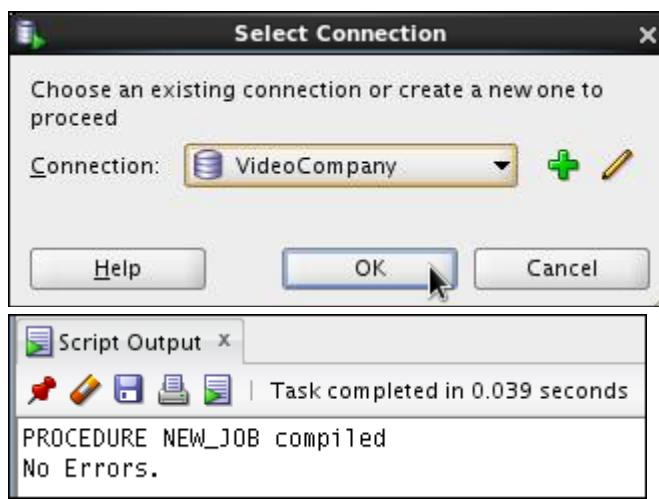
Solution 1-2: Adding a New Job to the JOBS Table

In this practice, you create a subprogram to add a new job into the JOBS table.

1. Create a stored procedure called NEW_JOB to enter a new order into the JOBS table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.

Open the /home/oracle/labs/plpu/solns/sol_ap1.sql script. Uncomment and select the code under Task 1 of Additional Practice 1-2. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. Make sure that you have selected the new VideoCompany connection. The code, connection prompt, and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE new_job(
    p_jobid  IN jobs.job_id%TYPE,
    p_title   IN jobs.job_title%TYPE,
    v_minsal  IN jobs.min_salary%TYPE) IS
    v_maxsal  jobs.max_salary%TYPE := 2 * v_minsal;
BEGIN
    INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
    VALUES (p_jobid, p_title, v_minsal, v_maxsal);
    DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
    DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_title || ' ' ||
                           v_minsal || ' ' || v_maxsal);
END new_job;
/
SHOW ERRORS
```



2. Enable SERVEROUTPUT, and then invoke the procedure to add a new job with job ID 'SY_ANAL', job title 'System Analyst', and minimum salary of 6000.

Uncomment and select the code under Task 2 of Additional Practice 1-2. When prompted to select a connection, select the new VideoCompany connection. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON
```

```
EXECUTE new_job ('SY_ANAL', 'System Analyst', 6000)
```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the following text:
anonymous block completed
New row added to JOBS table:
SY_ANAL System Analyst 6000 12000

2. Check whether a row was added and note the new job ID for use in the next exercise.
Commit the changes.

Run Uncomment and select the code under Task 3 of Additional Practice 1-2. The code and the results are displayed as follows:

```
SELECT *
FROM   jobs
WHERE  job_id = 'SY_ANAL';
COMMIT;
```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the following text:
JOB_ID JOB_TITLE MIN_SALARY MAX_SALARY

SY_ANAL System Analyst 6000 12000
committed.

Practice 1-3: Adding a New Row to the JOB_HISTORY Table

Overview

In this Additional Practice, you add a new row to the JOB_HISTORY table for an existing employee.

Tasks

1. Create a stored procedure called ADD_JOB_HIST to add a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID ('SY_ANAL') that you created in Task 2 of Practice 1-2.
 - a. The procedure should provide two parameters, one for the employee ID who is changing the job, and the second for the new job ID.
 - b. Read the employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table.
 - c. Make the hire date of this employee as start date and today's date as end date for this row in the JOB_HISTORY table.
 - d. Change the hire date of this employee in the EMPLOYEES table to today's date.
 - e. Update the job ID of this employee to the job ID passed as parameter (use the 'SY_ANAL' job ID) and salary equal to the minimum salary for that job ID + 500.
Note: Include exception handling to handle an attempt to insert a nonexistent employee.
2. Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the ADD_JOB_HIST procedure.
3. Enable SERVEROUTPUT, and then execute the procedure with employee ID 106 and job ID 'SY_ANAL' as parameters.
4. Query the JOB_HISTORY and EMPLOYEES tables to view your changes for employee 106, and then commit the changes.
5. Re-enable the triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables.

Solution 1-3: Adding a New Row to the JOB_HISTORY Table

In this Additional Practice, you add a new row to the JOB_HISTORY table for an existing employee.

1. Create a stored procedure called ADD_JOB_HIST to add a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID ('SY_ANAL') that you created in exercise 2.
 - a. The procedure should provide two parameters, one for the employee ID who is changing the job, and the second for the new job ID.
 - b. Read the employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table.
 - c. Make the hire date of this employee as start date and today's date as end date for this row in the JOB_HISTORY table.
 - d. Change the hire date of this employee in the EMPLOYEES table to today's date.
 - e. Update the job ID of this employee to the job ID passed as parameter (use the 'SY_ANAL' job ID) and salary equal to the minimum salary for that job ID + 500.

Note: Include exception handling to handle an attempt to insert a nonexistent employee.

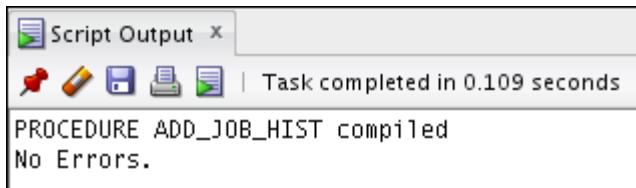
Uncomment and select the code under Task 1 of Additional Practice 1-3. The code and the results are displayed as follows:

```

CREATE OR REPLACE PROCEDURE add_job_hist (
  p_emp_id      IN employees.employee_id%TYPE,
  p_new_jobid   IN jobs.job_id%TYPE) IS
BEGIN
  INSERT INTO job_history
    SELECT employee_id, hire_date, SYSDATE, job_id,
  department_id
    FROM   employees
   WHERE  employee_id = p_emp_id;
  UPDATE employees
    SET   hire_date = SYSDATE,
          job_id = p_new_jobid,
          salary = (SELECT min_salary + 500
                      FROM   jobs
                     WHERE  job_id = p_new_jobid)
    WHERE employee_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
                        ' details to the JOB_HISTORY table');
  DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
                        p_emp_id|| ' to '|| p_new_jobid);

```

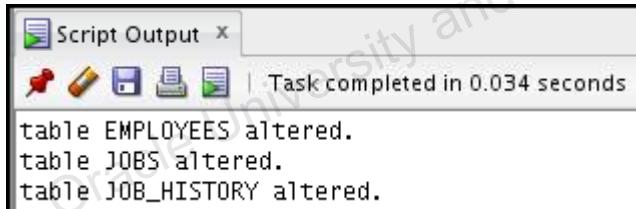
```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20001, 'Employee does not
exist!');
END add_job_hist;
/
SHOW ERRORS
```



2. Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the ADD_JOB_HIST procedure.

Uncomment and select the code under Task 2 of Additional Practice 1-3. The code and the results are displayed as follows:

```
ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;
ALTER TABLE job_history DISABLE ALL TRIGGERS;
```

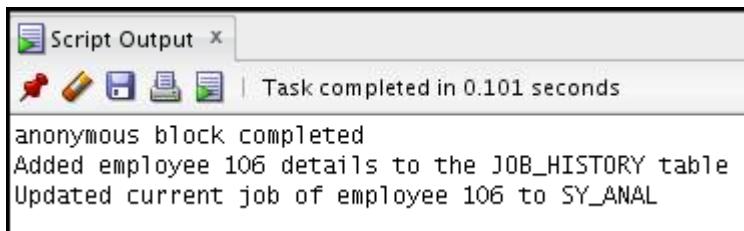


3. Enable SERVEROUTPUT, and then execute the procedure with employee ID 106 and job ID 'SY_ANAL' as parameters.

Uncomment and select the code under Task 3 of Additional Practice 1-3. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE add_job_hist(106, 'SY_ANAL')
```

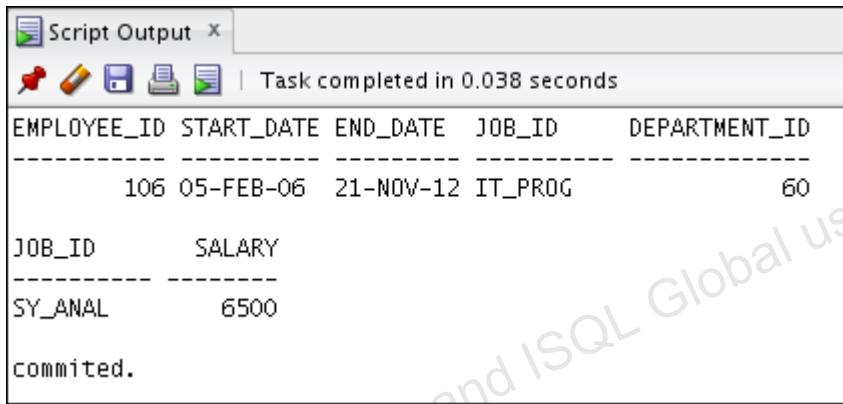


4. Query the `JOB_HISTORY` and `EMPLOYEES` tables to view your changes for employee 106, and then commit the changes.

Uncomment and select the code under Task 4 of Additional Practice 1-3. The code and the results are displayed as follows:

```
SELECT * FROM job_history  
WHERE employee_id = 106;  
  
SELECT job_id, salary FROM employees  
WHERE employee_id = 106;
```

```
COMMIT;
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the results of a COMMIT statement. The output includes the following text:
Task completed in 0.038 seconds
EMPLOYEE_ID START_DATE END_DATE JOB_ID DEPARTMENT_ID

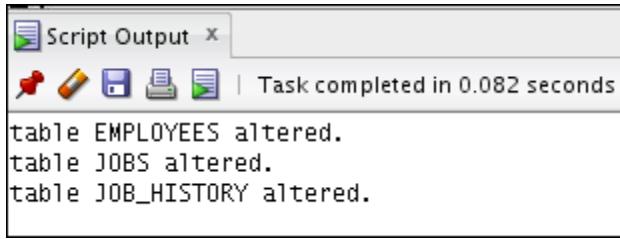
106 05-FEB-06 21-NOV-12 IT_PROG 60
JOB_ID SALARY

SY_ANAL 6500
committed.

5. Re-enable the triggers on the `EMPLOYEES`, `JOB_HISTORY` and `JOB_HISTORY` tables.

Uncomment and select the code under Task 5 of Additional Practice 1-3. The code and the results are displayed as follows:

```
ALTER TABLE employees ENABLE ALL TRIGGERS;  
ALTER TABLE jobs ENABLE ALL TRIGGERS;  
ALTER TABLE job_history ENABLE ALL TRIGGERS;
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the results of enabling triggers on three tables. The output includes the following text:
Task completed in 0.082 seconds
table EMPLOYEES altered.
table JOBS altered.
table JOB_HISTORY altered.

Practice 1-4: Updating the Minimum and Maximum Salaries for a Job

Overview

In this Additional Practice, you create a program to update the minimum and maximum salaries for a job in the JOBS table.

Tasks

1. Create a stored procedure called UPD_JOBSAL to update the minimum and maximum salaries for a specific job ID in the JOBS table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the JOBS table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the JOBS table is locked.

Hint: The resource locked/busy error number is -54.

2. Enable SERVEROUTPUT, and then execute the UPD_JOBSAL procedure by using a job ID of 'SY_ANAL', a minimum salary of 7000 and a maximum salary of 140.

Note: This should generate an exception message.

3. Disable triggers on the EMPLOYEES and JOBS tables.
4. Execute the UPD_JOBSAL procedure using a job ID of 'SY_ANAL', a minimum salary of 7000, and a maximum salary of 14000.
5. Query the JOBS table to view your changes, and then commit the changes.
6. Enable the triggers on the EMPLOYEES and JOBS tables.

Solution 1-4: Updating the Minimum and Maximum Salaries for a Job

In this Additional Practice, you create a program to update the minimum and maximum salaries for a job in the JOBS table.

1. Create a stored procedure called UPD_JOBSAL to update the minimum and maximum salaries for a specific job ID in the JOBS table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the JOBS table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the JOBS table is locked.

Hint: The resource locked/busy error number is -54.

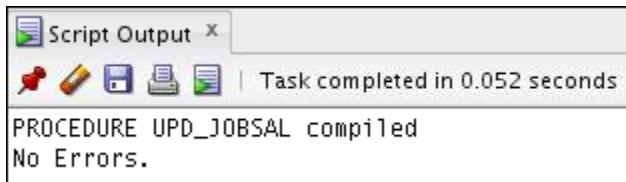
Uncomment and select the code under Task 1 of Additional Practice 1-4. The code and the results are displayed as follows:

```

CREATE OR REPLACE PROCEDURE upd_jobsal(
    p_jobid      IN jobs.job_id%type,
    p_new_minsal IN jobs.min_salary%type,
    p_new_maxsal IN jobs.max_salary%type) IS
    v_dummy          PLS_INTEGER;
    e_resource_busy EXCEPTION;
    e_sal_error      EXCEPTION;
    PRAGMA           EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
    IF (p_new_maxsal < p_new_minsal) THEN
        RAISE e_sal_error;
    END IF;
    SELECT 1 INTO v_dummy
        FROM jobs
        WHERE job_id = p_jobid
        FOR UPDATE OF min_salary NOWAIT;
    UPDATE jobs
        SET min_salary =  p_new_minsal,
            max_salary =  p_new_maxsal
        WHERE job_id  = p_jobid;
EXCEPTION
    WHEN e_resource_busy THEN
        RAISE_APPLICATION_ERROR (-20001,
            'Job information is currently locked, try later.');
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20001, 'This job ID does not
exist');
    WHEN e_sal_error THEN
        RAISE_APPLICATION_ERROR(-20001,

```

'Data error: Max salary should be more than min salary');
END upd_jobsal;
/
SHOW ERRORS

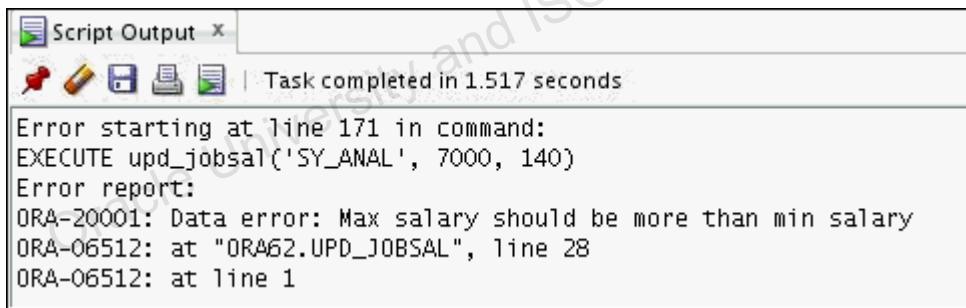


2. Enable SERVEROUTPUT, and then execute the UPD_JOBSAL procedure by using a job ID of 'SY_ANAL', a minimum salary of 7000 and a maximum salary of 140.

Note: This should generate an exception message.

Uncomment and select the code under Task 2 of Additional Practice 1-4. The code and the results are displayed as follows:

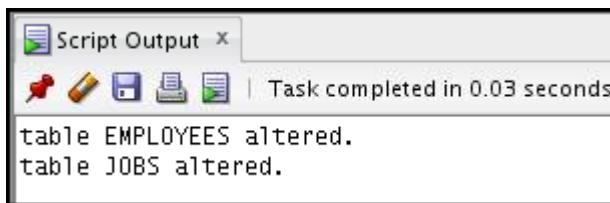
```
SET SERVEROUTPUT ON  
  
EXECUTE upd_jobsal('SY_ANAL', 7000, 140)
```



3. Disable triggers on the EMPLOYEES and JOBS tables.

Uncomment and select the code under Task 3 of Additional Practice 1-4. The code and the results are displayed as follows:

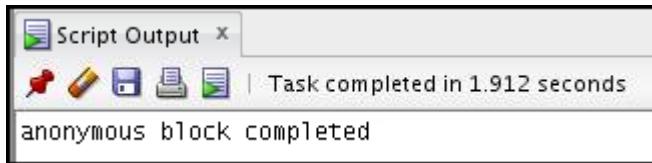
```
ALTER TABLE employees DISABLE ALL TRIGGERS;  
ALTER TABLE jobs DISABLE ALL TRIGGERS;
```



4. Execute the `UPD_JOBSAL` procedure using a job ID of '`SY_ANAL`', a minimum salary of 7000, and a maximum salary of 14000.

Uncomment and select the code under Task 4 of Additional Practice 1-4. The code and the results are displayed as follows:

```
EXECUTE upd_jobsal('SY_ANAL', 7000, 14000)
```



5. Query the `JOB$` table to view your changes, and then commit the changes.

Uncomment and select the code under Task 5 of Additional Practice 1-4. The code and the results are displayed as follows:

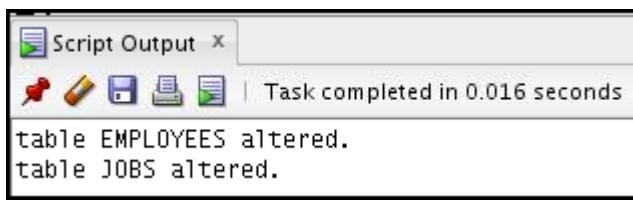
```
SELECT *
FROM   jobs
WHERE  job_id = 'SY_ANAL';
```

Script Output			
Task completed in 0.003 seconds			
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

6. Enable the triggers on the `EMPLOYEES` and `JOB$` tables.

Uncomment and select the code under Task 6 of Additional Practice 1-4. The code and the results are displayed as follows:

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE jobs ENABLE ALL TRIGGERS;
```



Practice 1-5: Monitoring Employees Salaries

Overview

In this Additional Practice, you create a procedure to monitor whether employees have exceeded their average salaries for their job type.

Tasks

1. Disable the `SECURE_EMPLOYEES` trigger.
2. In the `EMPLOYEES` table, add an `EXCEED_AVGSAL` column to store up to three characters and a default value of `NO`. Use a check constraint to allow the values `YES` or `NO`.
3. Create a stored procedure called `CHECK_AVGSAL` that checks whether each employee's salary exceeds the average salary for the `JOB_ID`.
 - a. The average salary for a job is calculated from the information in the `JOBS` table.
 - b. If the employee's salary exceeds the average for his or her job, then update the `EXCEED_AVGSAL` column in the `EMPLOYEES` table to a value of `YES`; otherwise, set the value to `NO`.
 - c. Use a cursor to select the employee's rows using the `FOR UPDATE` option in the query.
 - d. Add exception handling to account for a record being locked.
Hint: The resource locked/busy error number is `-54`.
 - e. Write and use a local function called `GET_JOB_AVGSAL` to determine the average salary for a job ID specified as a parameter.
4. Execute the `CHECK_AVGSAL` procedure. To view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary and the `exceed_avgsal` indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

Note: These exercises can be used for extra practice when discussing how to create functions.

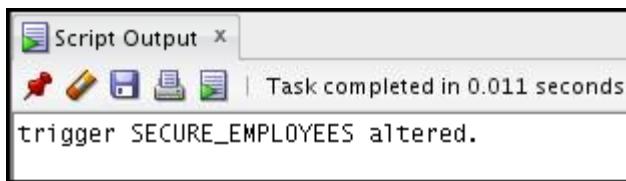
Solution 1-5: Monitoring Employees Salaries

In this practice, you create a procedure to monitor whether employees have exceeded their average salaries for their job type.

1. Disable the SECURE_EMPLOYEES trigger.

Uncomment and select the code under Task 1 of Additional Practice 1-5. The code and the results are displayed as follows:

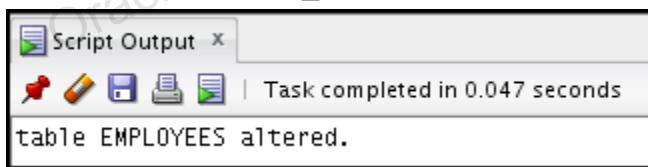
```
ALTER TRIGGER secure_employees DISABLE;
```



2. In the EMPLOYEES table, add an EXCEED_AVGSAL column to store up to three characters and a default value of NO. Use a check constraint to allow the values YES or NO.

Uncomment and select the code under Task 2 of Additional Practice 1-5. The code and the results are displayed as follows:

```
ALTER TABLE employees ADD (exceed_avgsal VARCHAR2(3) DEFAULT
  'NO'
  CONSTRAINT employees_exceed_avgsal_ck
  CHECK (exceed_avgsal IN ('YES', 'NO')));
```



3. Create a stored procedure called CHECK_AVGSAL that checks whether each employee's salary exceeds the average salary for the JOB_ID.
 - a. The average salary for a job is calculated from the information in the JOBS table.
 - b. If the employee's salary exceeds the average for his or her job, then update the EXCEED_AVGSAL column in the EMPLOYEES table to a value of YES; otherwise, set the value to NO.
 - c. Use a cursor to select the employee's rows using the FOR UPDATE option in the query.
 - d. Add exception handling to account for a record being locked.
Hint: The resource locked/busy error number is -54.
 - e. Write and use a local function called GET_JOB_AVGSAL to determine the average salary for a job ID specified as a parameter.

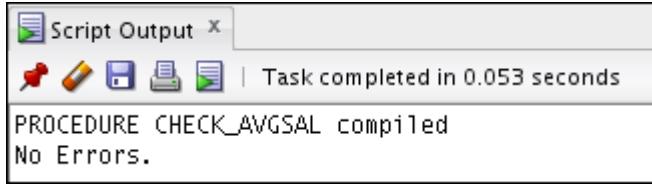
Uncomment and select the code under Task 3 of Additional Practice 1-5. The code and the results are displayed as follows::

```

CREATE OR REPLACE PROCEDURE check_avgsal IS
    emp_exceed_avgsal_type employees.exceed_avgsal%type;
    CURSOR c_emp_csr IS
        SELECT employee_id, job_id, salary
        FROM employees
        FOR UPDATE;
    e_resource_busy EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_resource_busy, -54);
    FUNCTION get_job_avgsal (jobid VARCHAR2) RETURN NUMBER IS
        avg_sal employees.salary%type;
    BEGIN
        SELECT (max_salary + min_salary)/2 INTO avg_sal
        FROM jobs
        WHERE job_id = jobid;
        RETURN avg_sal;
    END;

    BEGIN
        FOR emprec IN c_emp_csr
        LOOP
            emp_exceed_avgsal_type := 'NO';
            IF emprec.salary >= get_job_avgsal(emprec.job_id) THEN
                emp_exceed_avgsal_type := 'YES';
            END IF;
            UPDATE employees
            SET exceed_avgsal = emp_exceed_avgsal_type
            WHERE CURRENT OF c_emp_csr;
        END LOOP;
    EXCEPTION
        WHEN e_resource_busy THEN
            ROLLBACK;
            RAISE_APPLICATION_ERROR (-20001, 'Record is busy, try
later.');
    END check_avgsal;
/
SHOW ERRORS

```



4. Execute the `CHECK_AVGSAL` procedure. To view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary and the `exceed_avgsal` indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

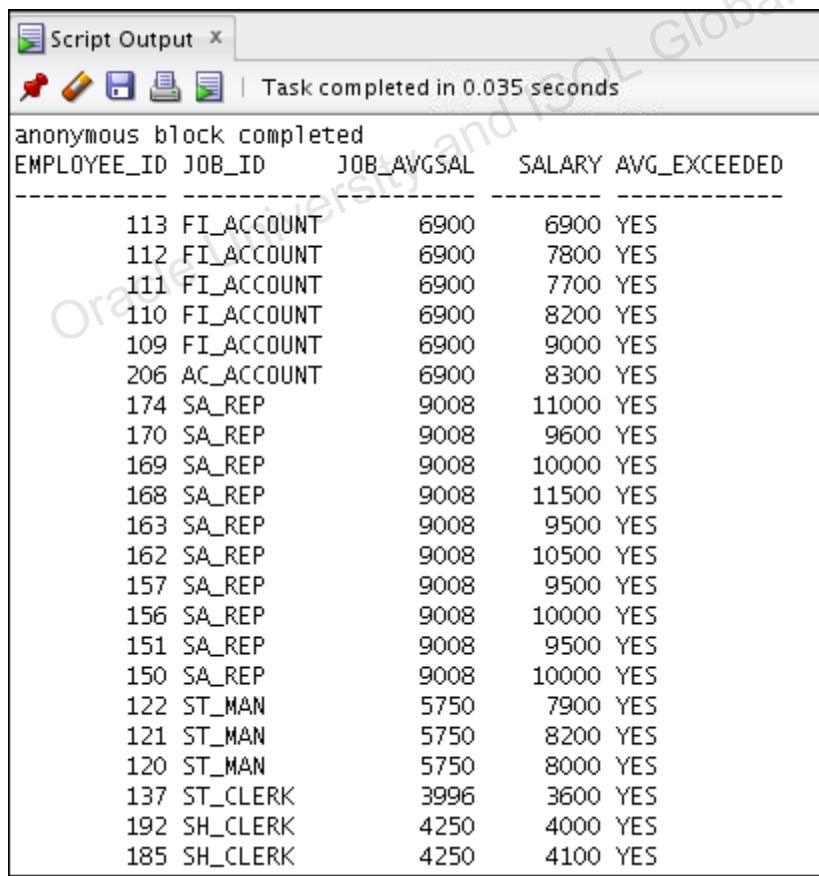
Note: These exercises can be used for extra practice when discussing how to create functions.

Uncomment and select the code under Task 4 of Additional Practice 1-5. The code and the results are displayed as follows:

```
EXECUTE check_avgsal

SELECT e.employee_id, e.job_id, (j.max_salary-j.min_salary/2)
job_avgsal,
      e.salary, e.exceed_avgsal avg_exceeded
FROM   employees e, jobs j
WHERE  e.job_id = j.job_id
and e.exceed_avgsal = 'YES';

COMMIT;
```



The screenshot shows the Oracle SQL Developer interface with a 'Script Output' window. The window title is 'Script Output'. Below the title, there are icons for redo, undo, run, stop, and refresh, followed by the message 'Task completed in 0.035 seconds'. The output area displays the results of an anonymous block. The output starts with 'anonymous block completed' and then shows a table with four columns: EMPLOYEE_ID, JOB_ID, JOB_AVGSAL, and SALARY_AVG_EXCEEDED. The data consists of 25 rows, each representing an employee's information and whether their salary exceeds the average for their job.

EMPLOYEE_ID	JOB_ID	JOB_AVGSAL	SALARY_AVG_EXCEEDED
113	FI_ACCOUNT	6900	6900 YES
112	FI_ACCOUNT	6900	7800 YES
111	FI_ACCOUNT	6900	7700 YES
110	FI_ACCOUNT	6900	8200 YES
109	FI_ACCOUNT	6900	9000 YES
206	AC_ACCOUNT	6900	8300 YES
174	SA_REP	9008	11000 YES
170	SA_REP	9008	9600 YES
169	SA_REP	9008	10000 YES
168	SA_REP	9008	11500 YES
163	SA_REP	9008	9500 YES
162	SA_REP	9008	10500 YES
157	SA_REP	9008	9500 YES
156	SA_REP	9008	10000 YES
151	SA_REP	9008	9500 YES
150	SA_REP	9008	10000 YES
122	ST_MAN	5750	7900 YES
121	ST_MAN	5750	8200 YES
120	ST_MAN	5750	8000 YES
137	ST_CLERK	3996	3600 YES
192	SH_CLERK	4250	4000 YES
185	SH_CLERK	4250	4100 YES

```
184 SH_CLERK      4250    4200 YES
103 IT_PROG       8000    9000 YES
201 MK_MAN        10500   13000 YES
203 HR REP        7000    6500 YES
204 PR REP        8250   10000 YES

27 rows selected

committed.
```

Practice 1-6: Retrieving the Total Number of Years of Service for an Employee

Overview

In this practice, you create a subprogram to retrieve the number of years of service for a specific employee.

Tasks

1. Create a stored function called `GET_YEARS_SERVICE` to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.
2. Invoke the `GET_YEARS_SERVICE` function in a call to `DBMS_OUTPUT.PUT_LINE` for an employee with ID 999.
3. Display the number of years of service for employee 106 with `DBMS_OUTPUT.PUT_LINE` invoking the `GET_YEARS_SERVICE` function. Make sure that you enable `SERVEROUTPUT`.
4. Query the `JOB_HISTORY` and `EMPLOYEES` tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those that you get when you run these queries.

Solution 1-6: Retrieving the Total Number of Years of Service for an Employee

In this practice, you create a subprogram to retrieve the number of years of service for a specific employee.

1. Create a stored function called GET_YEARS_SERVICE to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.

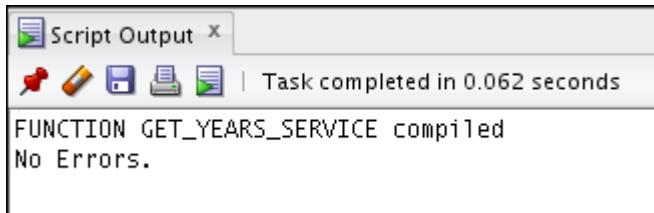
Uncomment and select the code under Task 1 of Additional Practice 1-6. The code and the results are displayed as follows:

```

CREATE OR REPLACE FUNCTION get_years_service(
    p_emp.empid_type IN employees.employee_id%TYPE) RETURN NUMBER
IS
    CURSOR c_jobh_csr IS
        SELECT MONTHS_BETWEEN(end_date, start_date)/12
    v_years_in_job
        FROM job_history
        WHERE employee_id = p_emp.empid_type;
    v_years_service NUMBER(2) := 0;
    v_years_in_job NUMBER(2) := 0;
BEGIN
    FOR jobh_rec IN c_jobh_csr
    LOOP
        EXIT WHEN c_jobh_csr%NOTFOUND;
        v_years_service := v_years_service +
    jobh_rec.v_years_in_job;
    END LOOP;
    SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO
    v_years_in_job
    FROM employees
    WHERE employee_id = p_emp.empid_type;
    v_years_service := v_years_service + v_years_in_job;
    RETURN ROUND(v_years_service);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20348,
            'Employee with ID '|| p_emp.empid_type || ' does not
exist.');
        RETURN NULL;
END get_years_service;

```

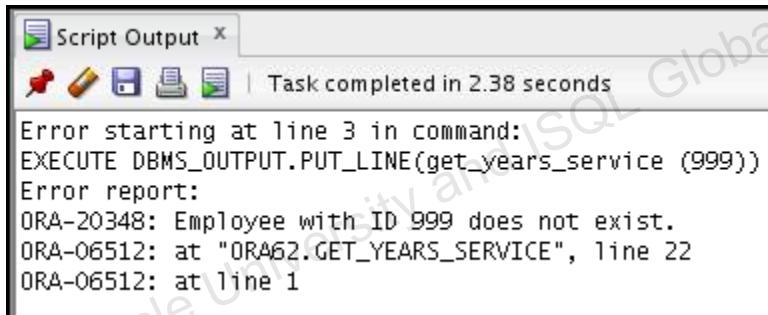
```
/  
SHOW ERRORS
```



2. Invoke the GET_YEARS_SERVICE function in a call to DBMS_OUTPUT.PUT_LINE for an employee with ID 999.

Uncomment and select the code under Task 2 of Additional Practice 1-6. The code and the results are displayed as follows:

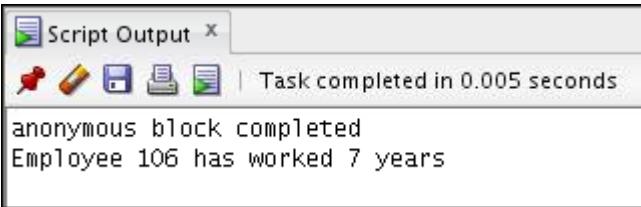
```
SET SERVEROUTPUT ON  
EXECUTE DBMS_OUTPUT.PUT_LINE(get_years_service (999))
```



3. Display the number of years of service for employee 106 with DBMS_OUTPUT.PUT_LINE invoking the GET_YEARS_SERVICE function. Make sure that you enable SERVEROUTPUT.

Uncomment and select the code under Task 1 of Additional Practice 1-6. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON  
  
BEGIN  
    DBMS_OUTPUT.PUT_LINE (  
        'Employee 106 has worked ' || get_years_service(106) || '  
years');  
END;  
/
```

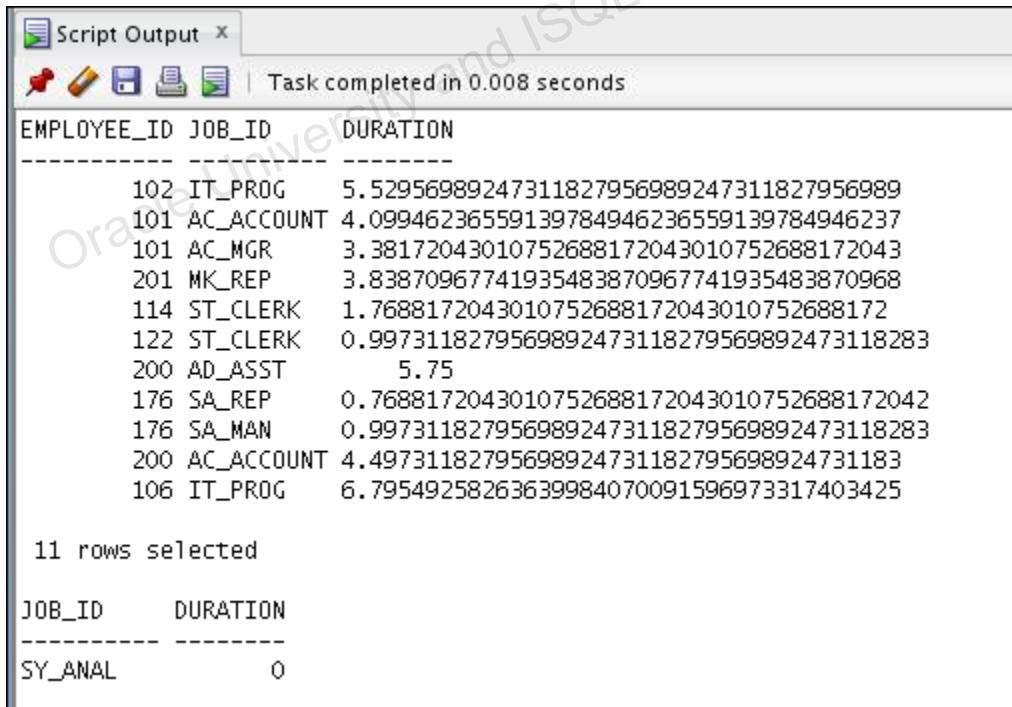


4. Query the JOB_HISTORY and EMPLOYEES tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those you get when you run these queries.

Uncomment and select the code under Task 4 of Additional Practice 1-6. The code and the results are displayed as follows:

```
SELECT employee_id, job_id,
       MONTHS_BETWEEN(end_date, start_date)/12 duration
  FROM job_history;

SELECT job_id, MONTHS_BETWEEN(SYSDATE, hire_date)/12 duration
  FROM employees
 WHERE employee_id = 106;
```



Practice 1-7: Retrieving the Total Number of Different Jobs for an Employee

Overview

In this practice, you create a program to retrieve the number of different jobs that an employee worked on during his or her service.

Tasks

1. Create a stored function called `GET_JOB_COUNT` to retrieve the total number of different jobs on which an employee worked.
 - a. The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job.
 - b. Add exception handling to account for an invalid employee ID.
Hint: Use the distinct job IDs from the `JOB_HISTORY` table, and exclude the current job ID, if it is one of the job IDs on which the employee has already worked.
 - c. Write a `UNION` of two queries and count the rows retrieved into a PL/SQL table.
 - d. Use a `FETCH` with `BULK COLLECT INTO` to obtain the unique jobs for the employee.
2. Invoke the function for the employee with the ID of 176. Make sure that you enable `SERVERROUTPUT`.
Note: These exercises can be used for extra practice when discussing how to create packages.

Solution 1-7: Retrieving the Total Number of Different Jobs for an Employee

In this practice, you create a program to retrieve the number of different jobs that an employee worked on during his or her service.

1. Create a stored function called GET_JOB_COUNT to retrieve the total number of different jobs on which an employee worked.
 - a. The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job.
 - b. Add exception handling to account for an invalid employee ID.

Hint: Use the distinct job IDs from the JOB_HISTORY table, and exclude the current job ID if it is one of the job IDs on which the employee has already worked.

 - c. Write a UNION of two queries and count the rows retrieved into a PL/SQL table.
 - d. Use a FETCH with BULK COLLECT INTO to obtain the unique jobs for the employee.

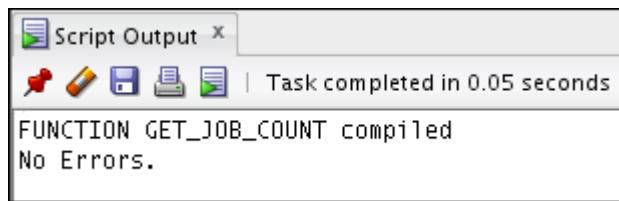
Uncomment and select the code under Task 1 of Additional Practice 1-7. The code and the results are displayed as follows:

```

CREATE OR REPLACE FUNCTION get_job_count(
  p_emp.empid_type IN employees.employee_id%TYPE) RETURN NUMBER
IS
  TYPE jobs_table_type IS TABLE OF jobs.job_id%type;
  v_jobtab jobs_table_type;
  CURSOR c_empjob_csr IS
    SELECT job_id
    FROM job_history
    WHERE employee_id = p_emp.empid_type
    UNION
    SELECT job_id
    FROM employees
    WHERE employee_id = p_emp.empid_type;
BEGIN
  OPEN c_empjob_csr;
  FETCH c_empjob_csr BULK COLLECT INTO v_jobtab;
  CLOSE c_empjob_csr;
  RETURN v_jobtab.count;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp.empid_type || ' does not
      exist!');
  RETURN NULL;

```

END get_job_count;
/
SHOW ERRORS



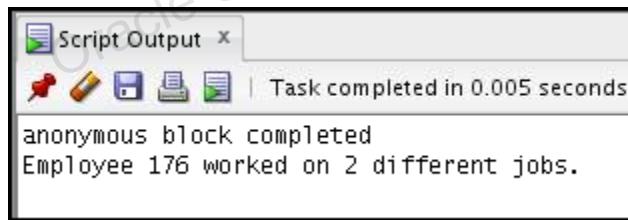
2. Invoke the function for the employee with the ID of 176. Make sure that you enable SERVEROUTPUT.

Note: These exercises can be used for extra practice when discussing how to create packages.

Uncomment and select the code under Task 2 of Additional Practice 1-7. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

BEGIN
    DBMS_OUTPUT.PUT_LINE('Employee 176 worked on ' ||
        get_job_count(176) || ' different jobs.');
END;
/
```



Practice 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions

Overview

In this practice, you create a package called `EMPJOB_PKG` that contains your `NEW_JOB`, `ADD_JOB_HIST`, `UPD_JOBSAL` procedures, as well as your `GET_YEARS_SERVICE` and `GET_JOB_COUNT` functions.

Tasks

1. Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.
2. Create the package body with the subprogram implementation; remember to remove, from the subprogram implementations, any types that you moved into the package specification.
3. Invoke your `EMPJOB_PKG.NEW_JOB` procedure to create a new job with the ID `PR_MAN`, the job title Public Relations Manager, and the salary 6250. Make sure that you enable `SERVERTOUTPUT`.
4. Invoke your `EMPJOB_PKG.ADD_JOB_HIST` procedure to modify the job of employee ID 110 to job ID `PR_MAN`.
Note: You need to disable the `UPDATE_JOB_HISTORY` trigger before you execute the `ADD_JOB_HIST` procedure, and re-enable the trigger after you have executed the procedure.
5. Query the `JOBES`, `JOB_HISTORY`, and `EMPLOYEES` tables to verify the results.
Note: These exercises can be used for extra practice when discussing how to create database triggers.

Solution 1-8: Creating a New Package that Contains the Newly Created Procedures and Functions

In this practice, you create a package called `EMPJOB_PKG` that contains your `NEW_JOB`, `ADD_JOB_HIST`, `UPD_JOBSAL` procedures, as well as your `GET_YEARS_SERVICE` and `GET_JOB_COUNT` functions.

1. Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.

Uncomment and select the code under Task 1 of Additional Practice 1-8. The code and the results are displayed as follows:

```
CREATE OR REPLACE PACKAGE empjob_pkg IS
    TYPE jobs_table_type IS TABLE OF jobs.job_id%TYPE;

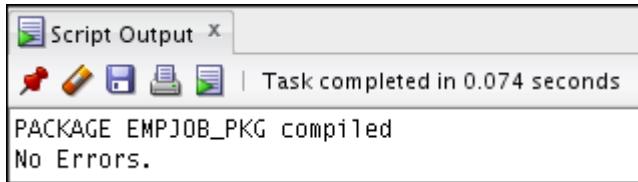
    PROCEDURE add_job_hist(
        p_emp_id IN employees.employee_id%TYPE,
        p_new_jobid IN jobs.job_id%TYPE);

    FUNCTION get_job_count(
        p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

    FUNCTION get_years_service(
        p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

    PROCEDURE new_job(
        p_jobid IN jobs.job_id%TYPE,
        p_title IN jobs.job_title%TYPE,
        p_minsal IN jobs.min_salary%TYPE);

    PROCEDURE upd_jobsal(
        p_jobid IN jobs.job_id%type,
        p_new_minsal IN jobs.min_salary%type,
        p_new_maxsal IN jobs.max_salary%type);
END empjob_pkg;
/
SHOW ERRORS
```



2. Create the package body with the subprogram implementation; remember to remove from the subprogram implementations any types that you moved into the package specification.

Uncomment and select the code under Task 2 of Additional Practice 1-8. The code and the results are displayed as follows:

```

CREATE OR REPLACE PACKAGE BODY empjob_pkg IS
  PROCEDURE add_job_hist(
    p_emp_id IN employees.employee_id%TYPE,
    p_new_jobid IN jobs.job_id%TYPE) IS
  BEGIN
    INSERT INTO job_history
      SELECT employee_id, hire_date, SYSDATE, job_id,
        department_id
      FROM employees
      WHERE employee_id = p_emp_id;
    UPDATE employees
      SET hire_date = SYSDATE,
          job_id = p_new_jobid,
          salary = (SELECT min_salary + 500
                     FROM jobs
                    WHERE job_id = p_new_jobid)
      WHERE employee_id = p_emp_id;
    DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
      ' details to the JOB_HISTORY table');
    DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
      p_emp_id|| ' to '|| p_new_jobid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR (-20001, 'Employee does not
exist!');
    END add_job_hist;

  FUNCTION get_job_count (
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
    v_jobtab jobs_table_type;
    CURSOR c_empjob_csr IS
      SELECT job_id

```

```

        FROM job_history
        WHERE employee_id = p_emp_id
      UNION
        SELECT job_id
        FROM employees
        WHERE employee_id = p_emp_id;
BEGIN
  OPEN c.empjob_csr;
  FETCH c.empjob_csr BULK COLLECT INTO v_jobtab;
  CLOSE c.empjob_csr;
  RETURN v_jobtab.count;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_id ||' does not exist!');
    RETURN 0;
END get_job_count;

FUNCTION get_years_service(
  p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
  CURSOR c_jobh_csr IS
    SELECT MONTHS_BETWEEN(end_date, start_date)/12
  v_years_in_job
    FROM job_history
    WHERE employee_id = p_emp_id;
  v_years_service NUMBER(2) := 0;
  v_years_in_job NUMBER(2) := 0;
BEGIN
  FOR jobh_rec IN c_jobh_csr
  LOOP
    EXIT WHEN c_jobh_csr%NOTFOUND;
    v_years_service := v_years_service +
  jobh_rec.v_years_in_job;
  END LOOP;
  SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO
  v_years_in_job
  FROM employees
  WHERE employee_id = p_emp_id;
  v_years_service := v_years_service + v_years_in_job;
  RETURN ROUND(v_years_service);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,

```

```

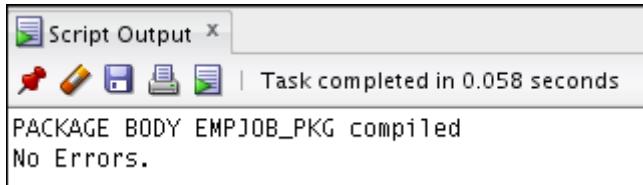
        'Employee with ID '|| p_emp_id ||' does not exist.');
      RETURN 0;
END get_years_service;

PROCEDURE new_job(
  p_jobid IN jobs.job_id%TYPE,
  p_title IN jobs.job_title%TYPE,
  p_minsal IN jobs.min_salary%TYPE) IS
  v_maxsal jobs.max_salary%TYPE := 2 * p_minsal;
BEGIN
  INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
  VALUES (p_jobid, p_title, p_minsal, v_maxsal);
  DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
  DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_title || ' ' ||
                        p_minsal || ' ' || v_maxsal);
END new_job;

PROCEDURE upd_jobsal(
  p_jobid IN jobs.job_id%type,
  p_new_minsal IN jobs.min_salary%type,
  p_new_maxsal IN jobs.max_salary%type) IS
  v_dummy PLS_INTEGER;
  e_resource_busy EXCEPTION;
  e_sal_error EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
  IF (p_new_maxsal < p_new_minsal) THEN
    RAISE e_sal_error;
  END IF;
  SELECT 1 INTO v_dummy
  FROM jobs
  WHERE job_id = p_jobid
  FOR UPDATE OF min_salary NOWAIT;
  UPDATE jobs
    SET min_salary = p_new_minsal,
        max_salary = p_new_maxsal
    WHERE job_id = p_jobid;
EXCEPTION
  WHEN e_resource_busy THEN
    RAISE_APPLICATION_ERROR (-20001,
                            'Job information is currently locked, try later.');
  WHEN NO_DATA_FOUND THEN

```

RAISE_APPLICATION_ERROR(-20001, 'This job ID does not exist');
 WHEN e_sal_error THEN
 RAISE_APPLICATION_ERROR(-20001,
 'Data error: Max salary should be more than min salary');
 END upd_jobsal;
 END empjob_pkg;
 /
 SHOW ERRORS



3. Invoke your `EMPJOB_PKG.NEW_JOB` procedure to create a new job with the ID `PR_MAN`, the job title Public Relations Manager, and the salary 6250. Make sure that you enable SERVEROUTPUT.

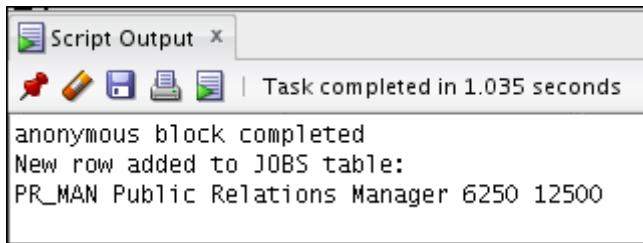
Uncomment and select the code under Task 3 of Additional Practice 1-8. The code and the results are displayed as follows:

```

SET SERVEROUTPUT ON

EXECUTE empjob_pkg.new_job('PR_MAN', 'Public Relations Manager',
6250)

```



4. Invoke your `EMPJOB_PKG.ADD_JOB_HIST` procedure to modify the job of employee ID 110 to job ID `PR_MAN`.

Note: You need to disable the `UPDATE_JOB_HISTORY` trigger before you execute the `ADD_JOB_HIST` procedure, and re-enable the trigger after you have executed the procedure.

Uncomment and select the code under Task 4 of Additional Practice 1-8. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON
ALTER TRIGGER update_job_history DISABLE;
EXECUTE empjob_pkg.add_job_hist(110, 'PR_MAN')
ALTER TRIGGER update_job_history ENABLE;
```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The output pane contains the following text:
trigger UPDATE_JOB_HISTORY altered.
anonymous block completed
Added employee 110 details to the JOB_HISTORY table
Updated current job of employee 110 to PR_MAN

trigger UPDATE_JOB_HISTORY altered.

5. Query the JOBS, JOB_HISTORY, and EMPLOYEES tables to verify the results.

Note: These exercises can be used for extra practice when discussing how to create database triggers.

Uncomment and select the code under Task 5 of Additional Practice 1-8. The code and the results are displayed as follows:

```
SELECT * FROM jobs WHERE job_id = 'PR_MAN';
SELECT * FROM job_history WHERE employee_id = 110;
SELECT job_id, salary FROM employees WHERE employee_id = 110;
```

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The output pane contains the results of three SELECT statements:

JOBS table:

JOB_ID JOB_TITLE MIN_SALARY MAX_SALARY

PR_MAN Public Relations Manager 6250 12500

JOB_HISTORY table:

EMPLOYEE_ID START_DATE END_DATE JOB_ID DEPARTMENT_ID

 110 28-SEP-05 22-NOV-12 FI_ACCOUNT 100

EMPLOYEES table:

JOB_ID SALARY

PR_MAN 6750

Practice 1-9: Creating a Trigger to Ensure that the Employees' Salaries Are Within the Acceptable Range

Overview

In this practice, you create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.

Tasks

1. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row that is updated in the `MIN_SALARY` and `MAX_SALARY` columns in the `JOB`s table.
 - a. For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the `EMPLOYEES` table falls within the new range of salaries specified for this job ID.
 - b. Include exception handling to cover a salary range change that affects the record of any existing employee.
2. Test the trigger using the `SY_ANAL` job, setting the new minimum salary to 5000, and the new maximum salary to 7000. Before you make the change, write a query to display the current salary range for the `SY_ANAL` job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the `JOB`s table for the specified job ID.
3. Using the `SY_ANAL` job, set the new minimum salary to 7,000, and the new maximum salary to 18000. Explain the results.

Solution 1-9: Creating a Trigger to Ensure that the Employees Salaries are Within the Acceptable Range

In this practice, you create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.

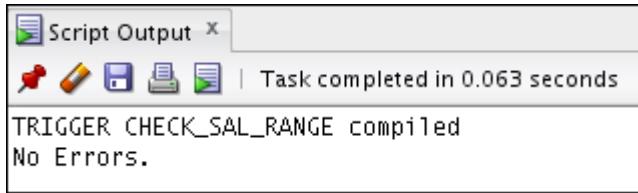
1. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row that is updated in the `MIN_SALARY` and `MAX_SALARY` columns in the `JOBS` table.
 - a. For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the `EMPLOYEES` table falls within the new range of salaries specified for this job ID.
 - b. Include exception handling to cover a salary range change that affects the record of any existing employee.

Uncomment and select the code under Task 1 of Additional Practice 1-9. The code and the results are displayed as follows:

```

CREATE OR REPLACE TRIGGER check_sal_range
BEFORE UPDATE OF min_salary, max_salary ON jobs
FOR EACH ROW
DECLARE
  v_minsal employees.salary%TYPE;
  v_maxsal employees.salary%TYPE;
  e_invalid_salrange EXCEPTION;
BEGIN
  SELECT MIN(salary), MAX(salary) INTO v_minsal, v_maxsal
  FROM employees
  WHERE job_id = :NEW.job_id;
  IF (v_minsal < :NEW.min_salary) OR (v_maxsal >
:NEW.max_salary) THEN
    RAISE e_invalid_salrange;
  END IF;
EXCEPTION
  WHEN e_invalid_salrange THEN
    RAISE_APPLICATION_ERROR(-20550,
      'Employees exist whose salary is out of the specified
      range. ' ||
      'Therefore the specified salary range cannot be updated.');
END check_sal_range;
/
SHOW ERRORS

```



- Test the trigger using the SY_ANAL job, setting the new minimum salary to 5000, and the new maximum salary to 7000. Before you make the change, write a query to display the current salary range for the SY_ANAL job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the JOBS table for the specified job ID.

Uncomment and select the code under Task 2 of Additional Practice 1-9. The code and the results are displayed as follows:

```

SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';

SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'SY_ANAL';

UPDATE jobs
SET min_salary = 5000, max_salary = 7000
WHERE job_id = 'SY_ANAL';

SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';

```

The screenshot shows a 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output'. Below it are several icons: a red script, a yellow pencil, a blue square, a green document, and a blue folder. To the right of these icons is the text 'Task completed in 0.018 seconds'. The main content area displays the results of four SQL queries:

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

EMPLOYEE_ID	LAST_NAME	SALARY
106	Pataballa	6500

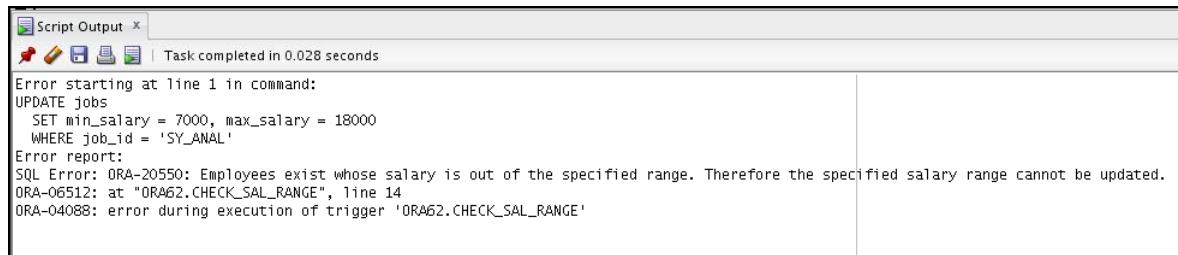
1 rows updated.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	5000	7000

3. Using the SY_ANAL job, set the new minimum salary to 7,000, and the new maximum salary to 18000. Explain the results.

Uncomment and select the code under Task 3 of Additional Practice 1-9. The code and the results are displayed as follows:

```
UPDATE jobs
   SET min_salary = 7000, max_salary = 18000
 WHERE job_id = 'SY_ANAL';
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the following text:

```
Script Output X
Task completed in 0.028 seconds
Error starting at line 1 in command:
UPDATE jobs
  SET min_salary = 7000, max_salary = 18000
 WHERE job_id = 'SY_ANAL'
Error report:
SQL Error: ORA-20550: Employees exist whose salary is out of the specified range. Therefore the specified salary range cannot be updated.
ORA-06512: at "ORA62.CHECK_SAL_RANGE", line 14
ORA-04088: error during execution of trigger 'ORA62.CHECK_SAL_RANGE'
```

The update fails to change the salary range due to the functionality provided by the CHECK_SAL_RANGE trigger because employee 106 who has the SY_ANAL job ID has a salary of 6500, which is less than the minimum salary for the new salary range specified in the UPDATE statement.

Oracle University and ISQL Global use only.

Additional Practices 2

Chapter 14

Additional Practices 2

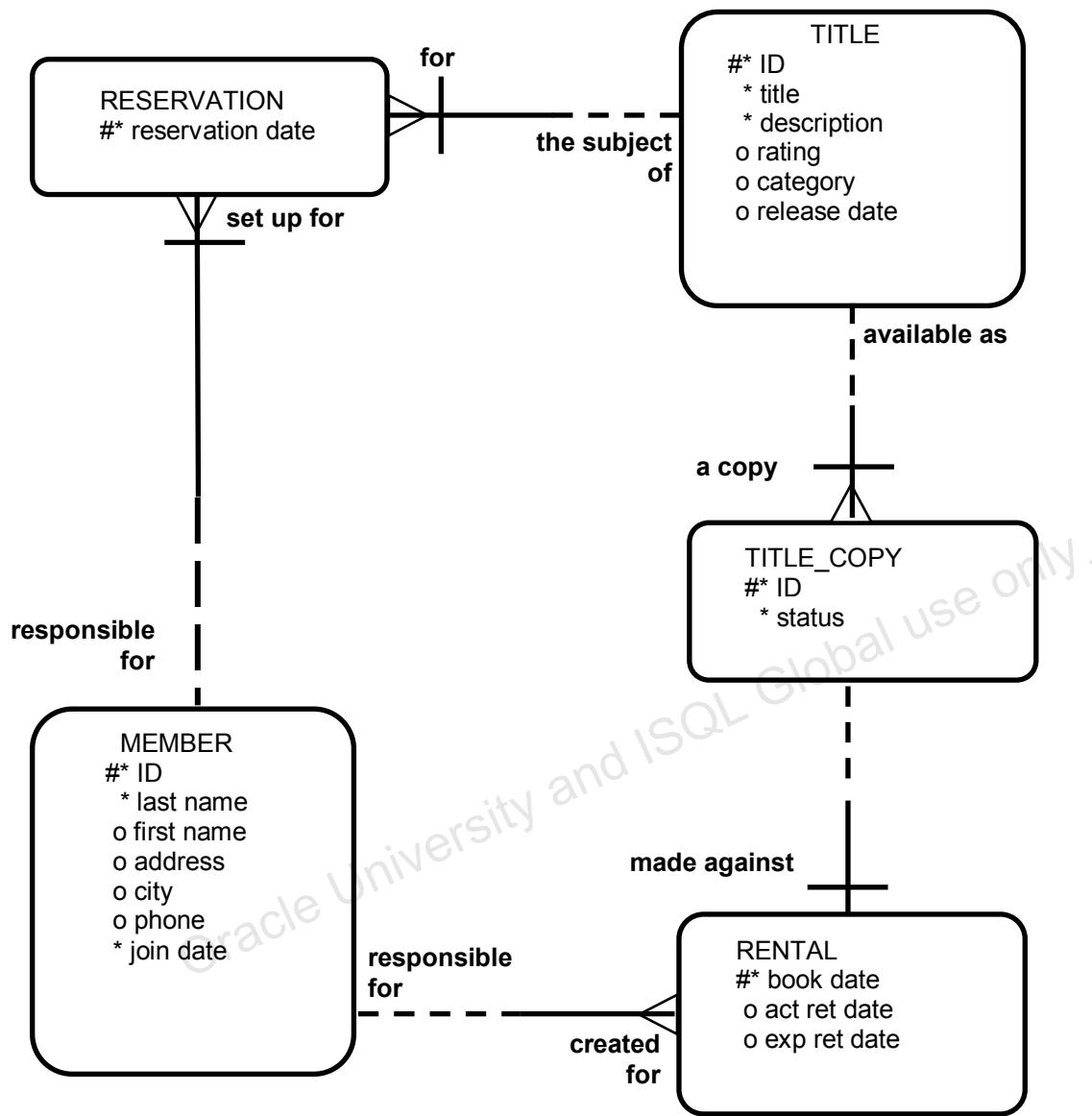
Overview

In this case study, you create a package named VIDEO_PKG that contains procedures and functions for a video store application. This application enables customers to become a member of the video store. Any member can rent movies, return rented movies, and reserve movies. Additionally, you create a trigger to ensure that any data in the video tables is modified only during business hours.

Create the package by using SQL*Plus and use the DBMS_OUTPUT Oracle-supplied package to display messages.

The video store database contains the following tables: TITLE, TITLE_COPY, RENTAL, RESERVATION, and MEMBER.

The video store database entity relationship diagram



Practice 2-1: Creating the VIDEO_PKG Package

Overview

In this practice, you create a package named VIDEO_PKG that contains procedures and functions for a video store application.

Task

1. Load and execute the /home/oracle/labs/plpu/labs/buildvid1.sql script to create all the required tables and sequences that are needed for this exercise.
2. Load and execute the /home/oracle/labs/plpu/labs/buildvid2.sql script to populate all the tables created through the buildvid1.sql script.
3. Create a package named VIDEO_PKG with the following procedures and functions:
 - a. **NEW_MEMBER:** A public procedure that adds a new member to the MEMBER table. For the member ID number, use the sequence MEMBER_ID_SEQ; for the join date, use SYSDATE. Pass all other values to be inserted into a new row as parameters.
 - b. **NEW_RENTAL:** An overloaded public function to record a new rental. Pass the title ID number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as AVAILABLE in the TITLE_COPY table for one copy of this title, then update this TITLE_COPY table and set the status to RENTED. If there is no copy available, the function must return NULL. Then, insert a new record into the RENTAL table identifying the booked date as today's date, the copy ID number, the member ID number, the title ID number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return NULL, and display a list of the customers' names that match and their ID numbers.
 - c. **RETURN_MOVIE:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title ID, the copy ID, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the RENTAL table and set the actual return date to today's date. Update the status in the TITLE_COPY table based on the status parameter passed into the procedure.
 - d. **RESERVE_MOVIE:** A private procedure that executes only if all the video copies requested in the NEW_RENTAL procedure have a status of RENTED. Pass the member ID number and the title ID number to this procedure. Insert a new record into the RESERVATION table and record the reservation date, member ID number, and title ID number. Print a message indicating that a movie is reserved and its expected date of return.
 - e. **EXCEPTION_HANDLER:** A private procedure that is called from the exception handler of the public programs. Pass the SQLCODE number to this procedure, and the name of the program (as a text string) where the error occurred. Use RAISE_APPLICATION_ERROR to raise a customized error. Start with a unique key violation (-1) and foreign key violation (-2292). Allow the exception handler to raise a generic error for any other errors.
4. Use the following scripts located in the /home/oracle/labs/plpu/soln directory to test your routines:
 - a. Add two members using the code under Task 4_a from sol_ap2.sql script.

- b. Add new video rentals using the code under Task 4_b from `sol_ap2.sql` script.
 - c. Return movies using the code under Task 4_c from `sol_ap2.sql` script.
5. The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 12:00 PM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.
 - a. Create a stored procedure called `TIME_CHECK` that checks the current time against business hours. If the current time is not within business hours, use the `RAISE_APPLICATION_ERROR` procedure to give an appropriate message.
 - b. Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your `TIME_CHECK` procedure from each of these triggers.
 - c. Test your triggers.

Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

Solution 2-1: Creating the VIDEO_PKG Package

In this practice, you create a package named VIDEO_PKG that contains procedures and functions for a video store application.

1. Load and execute the /home/oracle/labs/plpu/labs/buildvid1.sql script to create all the required tables and sequences that are needed for this exercise.

Run the /home/oracle/labs/plpu/labs/buildvid1.sql script. The code, the connection prompt, and the results are displayed as follows:

```

SET ECHO OFF
/* Script to build the Video Application (Part 1 - buildvid1.sql)
   for the Oracle Introduction to Oracle with Procedure Builder
course.
   Created by: Debby Kramer Creation date: 12/10/95
   Last updated: 11/21/12
   Modified by Supriya Ananth on 21-NOV-2012
   For the course Oracle Database: PL/SQL Program Units
   This part of the script creates tables and sequences that are
used
   by Task 4 of the Additional Practices of the course.
*/
DROP TABLE rental CASCADE CONSTRAINTS;
DROP TABLE reservation CASCADE CONSTRAINTS;
DROP TABLE title_copy CASCADE CONSTRAINTS;
DROP TABLE title CASCADE CONSTRAINTS;
DROP TABLE member CASCADE CONSTRAINTS;

PROMPT Please wait while tables are created.....

CREATE TABLE MEMBER
  (member_id  NUMBER (10)          CONSTRAINT member_id_pk PRIMARY
KEY
  , last_name  VARCHAR2(25)
    CONSTRAINT member_last_nn NOT NULL
  , first_name VARCHAR2(25)
  , address    VARCHAR2(100)
  , city       VARCHAR2(30)
  , phone      VARCHAR2(25)
  , join_date  DATE DEFAULT SYSDATE
    CONSTRAINT join_date_nn NOT NULL)
/
CREATE TABLE TITLE
  (title_id   NUMBER(10)
    CONSTRAINT title_id_pk PRIMARY KEY
  , title      VARCHAR2(60)
    CONSTRAINT title_nn NOT NULL
  , description VARCHAR2(400)

```

```

        CONSTRAINT title_desc_nn NOT NULL
, rating      VARCHAR2(4)
        CONSTRAINT title_rating_ck CHECK (rating IN
('G','PG','R','NC17','NR'))
, category    VARCHAR2(20) DEFAULT 'DRAMA'
        CONSTRAINT title_categ_ck CHECK (category IN
('DRAMA','COMEDY','ACTION',
'CHILD','SCIFI','DOCUMENTARY'))
, release_date DATE
/
CREATE TABLE TITLE_COPY
(copy_id      NUMBER(10)
, title_id    NUMBER(10)
        CONSTRAINT copy_title_id_fk
            REFERENCES title(title_id)
, status      VARCHAR2(15)
        CONSTRAINT copy_status_nn NOT NULL
        CONSTRAINT copy_status_ck CHECK (status IN ('AVAILABLE',
'DESTROYED',
'RENTED', 'RESERVED'))
, CONSTRAINT copy_title_id_pk PRIMARY KEY(copy_id, title_id))
/
CREATE TABLE RENTAL
(book_date DATE DEFAULT SYSDATE
, copy_id   NUMBER(10)
, member_id NUMBER(10)
        CONSTRAINT rental_mbr_id_fk REFERENCES member(member_id)
, title_id  NUMBER(10)
, act_ret_date DATE
, exp_ret_date DATE DEFAULT SYSDATE+2
, CONSTRAINT rental_copy_title_id_fk FOREIGN KEY (copy_id,
title_id)
            REFERENCES title_copy(copy_id,title_id)
, CONSTRAINT rental_id_pk PRIMARY KEY(book_date, copy_id,
title_id, member_id))
/
CREATE TABLE RESERVATION
(res_date    DATE
, member_id NUMBER(10)
, title_id  NUMBER(10)
, CONSTRAINT res_id_pk PRIMARY KEY(res_date, member_id, title_id))
/
PROMPT Tables created.

DROP SEQUENCE title_id_seq;
DROP SEQUENCE member_id_seq;

PROMPT Creating Sequences...

```

```
CREATE SEQUENCE member_id_seq
    START WITH 100
    NOCACHE
/
CREATE SEQUENCE title_id_seq
    START WITH 91
    NOCACHE
/
PROMPT Sequences created.

PROMPT Run buildvid2.sql now to populate the above tables.
```



```
Script Output X
| Task completed in 1.53 seconds

Error starting at line 12 in command:
DROP TABLE rental CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 13 in command:
DROP TABLE reservation CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 14 in command:
DROP TABLE title_copy CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 15 in command:
DROP TABLE title CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:

Error starting at line 16 in command:
DROP TABLE member CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
```

```
*Cause:  
*Action:  
Please wait while tables are created....  
table MEMBER created.  
table TITLE created.  
table TITLE_COPY created.  
table RENTAL created.  
table RESERVATION created.  
Tables created.  
  
Error starting at line 80 in command:  
DROP SEQUENCE title_id_seq  
Error report:  
SQL Error: ORA-02289: sequence does not exist  
02289. 00000 - "sequence does not exist"  
*Cause: The specified sequence does not exist, or the user does  
not have the required privilege to perform this operation.  
*Action: Make sure the sequence name is correct, and that you have  
the right to perform the desired operation on this sequence.  
  
Error starting at line 81 in command:  
DROP SEQUENCE member_id_seq  
Error report:  
SQL Error: ORA-02289: sequence does not exist  
02289. 00000 - "sequence does not exist"  
*Cause: The specified sequence does not exist, or the user does  
not have the required privilege to perform this operation.  
*Action: Make sure the sequence name is correct, and that you have  
the right to perform the desired operation on this sequence.  
Creating Sequences...  
sequence MEMBER_ID_SEQ created.  
sequence TITLE_ID_SEQ created.  
Sequences created.  
Run buildvid2.sql now to populate the above tables.
```

2. Load and execute the /home/oracle/labs/plpu/labs/buildvid2.sql script to populate all the tables created through the buildvid1.sql script.

Run the /home/oracle/labs/plpu/labs/buildvid2.sql script. The code, the connection prompt, and the results are displayed as follows:

```
/* Script to build the Video Application (Part 2 -  
buildvid2.sql)  
This part of the script populates the tables that are created  
using buildvid1.sql  
These are used by Part B of the Additional Practices of the  
course.  
You should run the script buildvid1.sql before running this  
script to create the above tables.  
*/
```

```

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Velasquez',
'Carmen', '283 King Street', 'Seattle', '587-99-6666', '03-MAR-
90');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Ngao',
'LaDoris', '5 Modrany', 'Bratislava', '586-355-8882', '08-MAR-
90');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Nagayama',
'Midori', '68 Via Centrale', 'Sao Paolo', '254-852-5764', '17-
JUN-91');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Quick-To-
See', 'Mark', '6921 King Way', 'Lagos', '63-559-777', '07-APR-
90');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Ropeburn',
'Audry', '86 Chu Street', 'Hong Kong', '41-559-87', '04-MAR-
90');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Urguhart',
'Molly', '3035 Laurier Blvd.', 'Quebec', '418-542-9988', '18-
JAN-91');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Menchu',
'Roberta', 'Boulevard de Waterloo 41', 'Brussels', '322-504-
2228', '14-MAY-90');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Biri', 'Ben',
'398 High St.', 'Columbus', '614-455-9863', '07-APR-90');

INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Catchpole',
'Antoinette', '88 Alfred St.', 'Brisbane', '616-399-1411', '09-
FEB-92');

COMMIT;

```

```

INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Willie and Christmas Too', 'All
of Willie''s friends made a Christmas list for Santa, but Willie
has yet to create his own wish list.', 'G', 'CHILD', '05-OCT-
95');

INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Alien Again', 'Another
installment of science fiction history. Can the heroine save the
planet from the alien life form?', 'R', 'SCIFI', '19-
MAY-95');

INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'The Glob', 'A meteor crashes near
a small American town and unleashes carnivorous goo in this
classic.', 'NR', 'SCIFI', '12-AUG-95');

```

```

INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'My Day Off', 'With a little luck
and a lot of ingenuity, a teenager skips school for a day in New
York.', 'PG', 'COMEDY', '12-JUL-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Miracles on Ice', 'A six-year-old
has doubts about Santa Claus. But she discovers that miracles
really do exist.', 'PG', 'DRAMA', '12-SEP-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Soda Gang', 'After discovering a
cache of drugs, a young couple find themselves pitted against a
vicious gang.', 'NR', 'ACTION', '01-JUN-95');
INSERT INTO title (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Interstellar Wars', 'Futuristic
interstellar action movie. Can the rebels save the humans from
the evil Empire?', 'PG', 'SCIFI', '07-JUL-77');

COMMIT;

INSERT INTO title_copy VALUES (1,92, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,93, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,93, 'RENTED');
INSERT INTO title_copy VALUES (1,94, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (3,95, 'RENTED');
INSERT INTO title_copy VALUES (1,96, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,97, 'AVAILABLE');
COMMIT;

INSERT INTO reservation VALUES (sysdate-1, 101, 93);
INSERT INTO reservation VALUES (sysdate-2, 106, 102);

COMMIT;

INSERT INTO rental VALUES (sysdate-1, 2, 101, 93, null,
sysdate+1);
INSERT INTO rental VALUES (sysdate-2, 3, 102, 95, null,
sysdate);

```

```
INSERT INTO rental VALUES (sysdate-4, 1, 106, 97, sysdate-2,  
sysdate-2);  
INSERT INTO rental VALUES (sysdate-3, 1, 101, 92, sysdate-2,  
sysdate-1);  
  
COMMIT;
```

PROMPT ** Tables built and data loaded **



```
1 rows inserted.  
1 rows inserted.  
committed.  
1 rows inserted.  
committed.  
** Tables built and data loaded **
```

3. Create a package named `VIDEO_PKG` with the following procedures and functions:
 - a. **NEW_MEMBER:** A public procedure that adds a new member to the `MEMBER` table. For the member ID number, use the sequence `MEMBER_ID_SEQ`; for the join date, use `SYSDATE`. Pass all other values to be inserted into a new row as parameters.
 - b. **NEW_RENTAL:** An overloaded public function to record a new rental. Pass the title ID number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as `AVAILABLE` in the `TITLE_COPY` table for one copy of this title, then update this `TITLE_COPY` table and set the status to `RENTED`. If there is no copy available, the function must return `NULL`. Then, insert a new record into the `RENTAL` table identifying the booked date as today's date, the copy ID number, the member ID number, the title ID number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return `NULL`, and display a list of the customers' names that match and their ID numbers.
 - c. **RETURN_MOVIE:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title ID, the copy ID, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the `RENTAL` table and set the actual return date to today's date. Update the status in the `TITLE_COPY` table based on the status parameter passed into the procedure.
 - d. **RESERVE_MOVIE:** A private procedure that executes only if all the video copies requested in the `NEW_RENTAL` procedure have a status of `RENTED`. Pass the member ID number and the title ID number to this procedure. Insert a new record into the `RESERVATION` table and record the reservation date, member ID number, and title ID number. Print a message indicating that a movie is reserved and its expected date of return.
 - e. **EXCEPTION_HANDLER:** A private procedure that is called from the exception handler of the public programs. Pass the `SQLCODE` number to this procedure, and the name of the program (as a text string) where the error occurred. Use `RAISE_APPLICATION_ERROR` to raise a customized error. Start with a unique key violation (-1) and foreign key violation (-2292). Allow the exception handler to raise a generic error for any other errors.

**Uncomment and run the code under Task 3 from
`/home/oracle/labs/plpu/solns/sol_ap2.sql` script. The code, the
connection prompt, and the results are displayed as follows:**

VIDEO_PKG Package Specification

```

CREATE OR REPLACE PACKAGE video_pkg IS
  PROCEDURE new_member
    (p_lname      IN member.last_name%TYPE,
     p_fname      IN member.first_name%TYPE          DEFAULT NULL,
     p_address    IN member.address%TYPE            DEFAULT NULL,
     p_city       IN member.city%TYPE              DEFAULT NULL,
     p_phone      IN member.phone%TYPE             DEFAULT NULL);

  FUNCTION new_rental
    (p_memberid   IN rental.member_id%TYPE,
     p_titleid    IN rental.title_id%TYPE)
    RETURN DATE;

  FUNCTION new_rental
    (p_membername IN member.last_name%TYPE,
     p_titleid    IN rental.title_id%TYPE)
    RETURN DATE;

  PROCEDURE return_movie
    (p_titleid    IN rental.title_id%TYPE,
     p_copyid     IN rental.copy_id%TYPE,
     p_sts        IN title_copy.status%TYPE);
END video_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY video_pkg IS
  PROCEDURE exception_handler(errcode IN NUMBER, p_context IN VARCHAR2) IS
    BEGIN
      IF errcode = -1 THEN
        RAISE_APPLICATION_ERROR(-20001,
          'The number is assigned to this member is already in use,
          |||'
          'try again.');
      ELSIF errcode = -2291 THEN
        RAISE_APPLICATION_ERROR(-20002, p_context ||
          ' has attempted to use a foreign key value that is
          invalid');
      ELSE
        RAISE_APPLICATION_ERROR(-20999, 'Unhandled error in ' ||
          p_context || '. Please contact your application ' ||
          'administrator with the following information: ' ||
          CHR(13) || SQLERRM);
      END IF;
    END exception_handler;

  PROCEDURE reserve_movie
    (p_memberid   IN reservation.member_id%TYPE,
     p_titleid    IN reservation.title_id%TYPE) IS

```

```

CURSOR c_rented_csr IS
    SELECT exp_ret_date
    FROM rental
    WHERE title_id = p_titleid
    AND act_ret_date IS NULL;
BEGIN
    INSERT INTO reservation (res_date, member_id, title_id)
VALUES (SYSDATE, p_memberid, p_titleid);
    COMMIT;
    FOR rented_rec IN c_rented_csr LOOP
        DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on: '
        || rented_rec.exp_ret_date);
        EXIT WHEN c_rented_csr%found;
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'RESERVE_MOVIE');
END reserve_movie;

PROCEDURE return_movie(
    p_titleid IN rental.title_id%TYPE,
    p_copyid IN rental.copy_id%TYPE,
    p_sts IN title_copy.status%TYPE) IS
    v_dummy VARCHAR2(1);
    CURSOR c_res_csr IS
        SELECT *
        FROM reservation
        WHERE title_id = p_titleid;
BEGIN
    SELECT '' INTO v_dummy
    FROM title
    WHERE title_id = p_titleid;
    UPDATE rental
    SET act_ret_date = SYSDATE
    WHERE title_id = p_titleid
    AND copy_id = p_copyid AND act_ret_date IS NULL;
    UPDATE title_copy
    SET status = UPPER(p_sts)
    WHERE title_id = p_titleid AND copy_id = p_copyid;
    FOR res_rec IN c_res_csr LOOP
        IF c_res_csr%FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Put this movie on hold -- ' ||
            'reserved by member #' || res_rec.member_id);
        END IF;
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'RETURN_MOVIE');
END return_movie;

FUNCTION new_rental(
    p_memberid IN rental.member_id%TYPE,
    p_titleid IN rental.title_id%TYPE) RETURN DATE IS

```

```

CURSOR c_copy_csr IS
    SELECT * FROM title_copy
    WHERE title_id = p_titleid
    FOR UPDATE;
v_flag    BOOLEAN  := FALSE;
BEGIN
    FOR copy_rec IN c_copy_csr LOOP
        IF copy_rec.status = 'AVAILABLE' THEN
            UPDATE title_copy
            SET status = 'RENTED'
            WHERE CURRENT OF c_copy_csr;
            INSERT INTO rental(book_date, copy_id, member_id,
                               title_id, exp_ret_date)
            VALUES (SYSDATE, copy_rec.copy_id, p_memberid,
                    p_titleid, SYSDATE + 3);
            v_flag := TRUE;
            EXIT;
        END IF;
    END LOOP;
    COMMIT;
    IF v_flag THEN
        RETURN (SYSDATE + 3);
    ELSE
        reserve_movie(p_memberid, p_titleid);
        RETURN NULL;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'NEW_RENTAL');
        RETURN NULL;
END new_rental;

FUNCTION new_rental(
    p_membername IN member.last_name%TYPE,
    p_titleid    IN rental.title_id%TYPE) RETURN DATE IS
CURSOR c_copy_csr IS
    SELECT * FROM title_copy
    WHERE title_id = p_titleid
    FOR UPDATE;
v_flag    BOOLEAN  := FALSE;
v_memberid member.member_id%TYPE;
CURSOR c_member_csr IS
    SELECT member_id, last_name, first_name
    FROM member
    WHERE LOWER(last_name) = LOWER(p_membername)
    ORDER BY last_name, first_name;
BEGIN
    SELECT member_id INTO v_memberid
    FROM member
    WHERE lower(last_name) = lower(p_membername);
    FOR copy_rec IN c_copy_csr LOOP
        IF copy_rec.status = 'AVAILABLE' THEN
            UPDATE title_copy

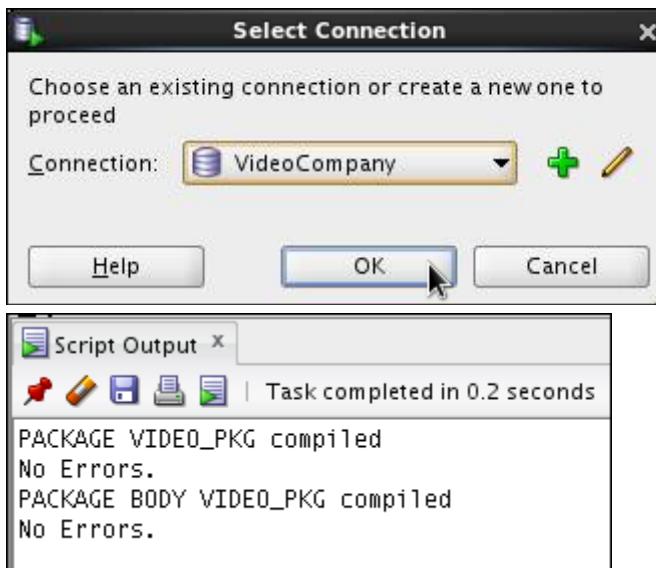
```

```

        SET status = 'RENTED'
        WHERE CURRENT OF c_copy_csr;
        INSERT INTO rental (book_date, copy_id, member_id,
                            title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, v_memberid,
                p_titleid, SYSDATE + 3);
        v_flag := TRUE;
        EXIT;
    END IF;
END LOOP;
COMMIT;
IF v_flag THEN
    RETURN(SYSDATE + 3);
ELSE
    reserve_movie(v_memberid, p_titleid);
    RETURN NULL;
END IF;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE(
            'Warning! More than one member by this name.');
        FOR member_rec IN c_member_csr LOOP
            DBMS_OUTPUT.PUT_LINE(member_rec.member_id || CHR(9) ||
                member_rec.last_name || ', ' || member_rec.first_name);
        END LOOP;
        RETURN NULL;
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'NEW_RENTAL');
        RETURN NULL;
END new_rental;

PROCEDURE new_member(
    p_lname      IN member.last_name%TYPE,
    p_fname       IN member.first_name%TYPE      DEFAULT NULL,
    p_address     IN member.address%TYPE         DEFAULT NULL,
    p_city        IN member.city%TYPE            DEFAULT NULL,
    p_phone       IN member.phone%TYPE           DEFAULT NULL) IS
BEGIN
    INSERT INTO member(member_id, last_name, first_name,
                      address, city, phone, join_date)
    VALUES(member_id_seq.NEXTVAL, p_lname, p_fname,
           p_address, p_city, p_phone, SYSDATE);
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        exception_handler(SQLCODE, 'NEW_MEMBER');
END new_member;
END video_pkg;
/
SHOW ERRORS

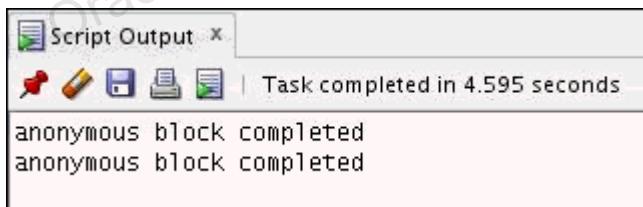
```



4. Use the following scripts located in the /home/oracle/labs/plpu/soln directory to test your routines. Make sure you enable SERVEROUTPUT:
 - a. Add two members using the code under Task 4_a.

Uncomment and run the code under Task 4_a. The code and the results are displayed as follows:

```
EXECUTE video_pkg.new_member('Haas', 'James', 'Chestnut Street',
'Boston', '617-123-4567')
EXECUTE video_pkg.new_member('Biri', 'Allan', 'Hiawatha
Drive', 'New York', '516-123-4567')
```

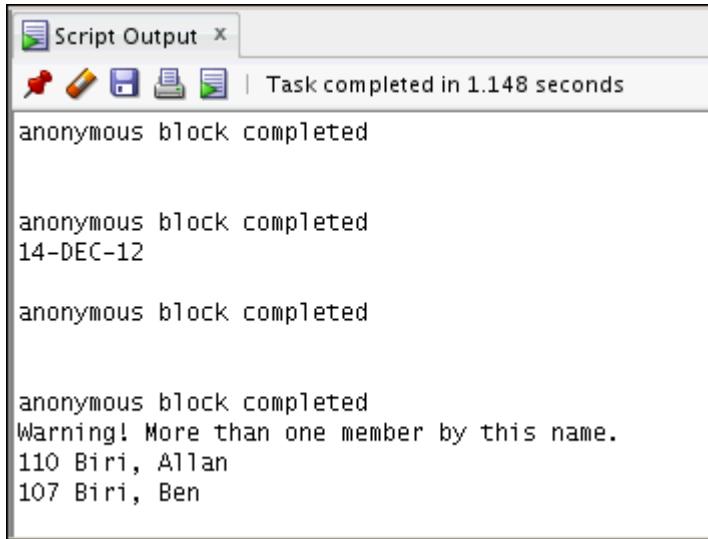


- b. Add new video rentals using the code under Task 4_b.

Uncomment and run the code under Task 4_b. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON
```

```
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(110, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(109, 93))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(107, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental('Biri', 97))
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the following text:
anonymous block completed

anonymous block completed
14-DEC-12

anonymous block completed

anonymous block completed
Warning! More than one member by this name.
110 Biri, Allan
107 Biri, Ben

- c. Return movies using the code under Task 4_c.

Uncomment and run the code under Task 4_c. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE video_pkg.return_movie(92, 3, 'AVAILABLE')
EXECUTE video_pkg.return_movie(95, 3, 'AVAILABLE')
EXECUTE video_pkg.return_movie(93, 1, 'RENTED')
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. It displays the following text:
anonymous block completed
anonymous block completed
anonymous block completed
Put this movie on hold -- reserved by member #101

5. The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 12:00 PM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.
- Create a stored procedure called TIME_CHECK that checks the current time against business hours. If the current time is not within business hours, use the RAISE_APPLICATION_ERROR procedure to give an appropriate message.

Uncomment and run the code under task 5_a. The code and the results are displayed as follows:

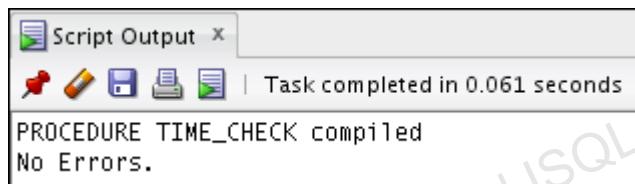
```
CREATE OR REPLACE PROCEDURE time_check IS
```

```

BEGIN
    IF ((TO_CHAR(SYSDATE, 'D') BETWEEN 1 AND 6) AND
        (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT
        BETWEEN
            TO_DATE('08:00', 'hh24:mi') AND TO_DATE('22:00',
            'hh24:mi')))

            OR ((TO_CHAR(SYSDATE, 'D') = 7)
            AND (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT
            BETWEEN
                TO_DATE('08:00', 'hh24:mi') AND TO_DATE('24:00',
                'hh24:mi')))) THEN
                RAISE_APPLICATION_ERROR(-20999,
                'Data changes restricted to office hours.');
            END IF;
    END time_check;
/
SHOW ERRORS

```



- b. Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your TIME_CHECK procedure from each of these triggers.

Uncomment and run the code under Task 5_b. The code and the result are displayed as follows:

```

CREATE OR REPLACE TRIGGER member_trig
    BEFORE INSERT OR UPDATE OR DELETE ON member
CALL time_check
/

CREATE OR REPLACE TRIGGER rental_trig
    BEFORE INSERT OR UPDATE OR DELETE ON rental
CALL time_check
/

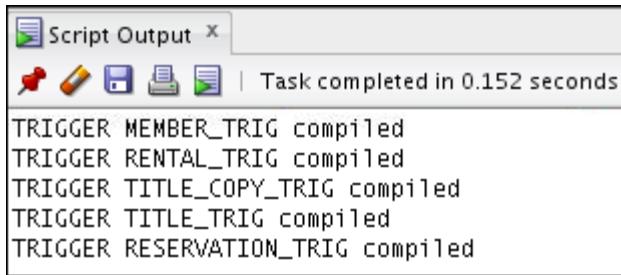
CREATE OR REPLACE TRIGGER title_copy_trig
    BEFORE INSERT OR UPDATE OR DELETE ON title_copy
CALL time_check
/

```

```

CREATE OR REPLACE TRIGGER title_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title
  CALL time_check
/
CREATE OR REPLACE TRIGGER reservation_trig
  BEFORE INSERT OR UPDATE OR DELETE ON reservation
  CALL time_check
/

```



c. Test your triggers.

Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

Uncomment and run the code under Task 5_c. The code and the result are displayed as follows:

```

-- First determine current timezone and time
SELECT SESSIONTIMEZONE,
       TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY HH24:MI') CURR_DATE
  FROM DUAL;

-- Change your time zone using [+|-]HH:MI format such that --
-- the current time returns a time between 6pm and 8am

ALTER SESSION SET TIME_ZONE=' -07:00';

-- Add a new member (for a sample test)

EXECUTE video_pkg.new_member('Elias', 'Elliane', 'Vine
Street', 'California', '789-123-4567')

BEGIN video_pkg.new_member('Elias', 'Elliane', 'Vine Street',
'California', '789-123-4567'); END;

```

-- Restore the original time zone for your session.
ALTER SESSION SET TIME_ZONE='-00:00';

Script Output	
Task completed in 0.006 seconds	
SESSIONTIMEZONE	CURR_DATE
+00:00	22-NOV-2012 08:04
session SET altered. anonymous block completed session SET altered.	

Oracle University and ISQL Global use only .