ORACLE UNIVERSITY

Integrated Cloud Applications & Platform Services

Oracle Database 12*c*: Program with PL/SQL - Cloud Edition (WDP only)

Activity Guide

D99739GC20

Edition 2.0 | March 2017 | D99761

Learn more from Oracle University at **education.oracle.com**

ORACLE

Table of Contents

Practices for Lesson 1: Introduction

Chapter 1

Practices for Lesson 1: Overview

Lesson Overview

In these practices, you do the following:

- Start SQL Developer
- Create a new database connection
- Browse the schema tables
- Set a SQL Developer preference

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus environment that is available in this course.

Practice 1-1: Getting Started

1. Start SQL Developer.
2. Create a database connection by using the following information (**Hint:** Select the Save Password check box):
   a. Connection Name: MyConnection
   b. Username: `ora41`
   c. Password: `ora41`
   d. Hostname: localhost
   e. Port: 1521
   f. SID: `orcl`
3. Test the new connection. If the Status is Success, connect to the database by using this new connection.
   a. In the Database Connection window, click the Test button.
      Note: The connection status appears in the lower-left corner of the window.
   b. If the status is Success, click the Connect button.
4. Browse the structure of the `EMPLOYEES` table and display its data.
   a. Expand the MyConnection connection by clicking the plus symbol next to it.
   b. Expand the Tables icon by clicking the plus symbol next to it.
   c. Display the structure of the `EMPLOYEES` table.
5. Use the Data tab to view data in the `EMPLOYEES` table.
6. Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than $10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements on the appropriate tabs.

   Note: Take a few minutes to familiarize yourself with the data, or consult Appendix A, which provides the description and data for all the tables in the HR schema that you will use in this course.
7. From the SQL Developer menu, select Tools > Preferences. The Preferences window appears.
8. Select Database > Worksheet Parameters. In the “Select default path to look for scripts” text box, use the Browse button to select the `/home/oracle/labs/plsf` directory. This directory contains the code example scripts, lab scripts, and practice solution scripts that are used in this course. Then, in the Preferences window, click OK to save the Worksheet Parameter setting.

9. Familiarize yourself with the structure of the `/home/oracle/labs/plsf` directory.
   a. Select File > Open. The Open window automatically selects the …`/plsf` directory as your starting location. This directory contains three subdirectories:
      - The `/code_ex` directory contains the code examples found in the course materials. Each .sql script is associated with a particular page in the lesson.
      - The `/labs` directory contains the code that is used in certain lesson practices. You are instructed to run the required script in the appropriate practice.
      - The `/soln` directory contains the solutions for each practice. Each `.sql` script is numbered with the associated practice_exercise reference.
   b. You can also use the Files tab to navigate through directories to open the script files.
   c. Using the Open window, and the Files tab, navigate through the directories and open a script file without executing the code.
   d. Close the SQL Worksheet.

Solution 1-1: Getting Started

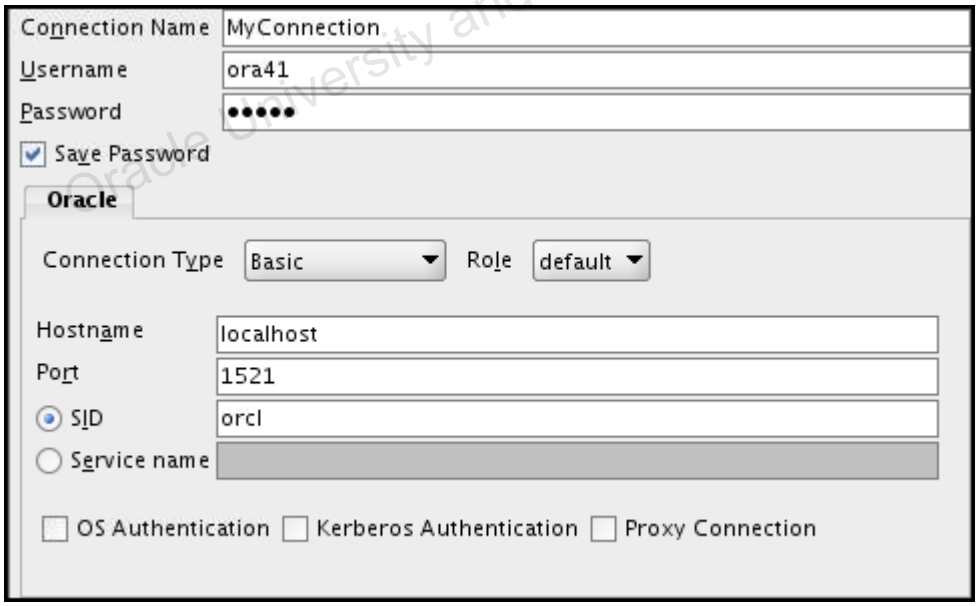
1. Start SQL Developer.
   Click the SQL Developer icon on your desktop.

2. Create a database connection by using the following information (**Hint:** Select the Save Password check box):
   a. Connection Name: MyConnection
   b. Username: `ora41`
   c. Password: `ora41`
   d. Hostname: localhost
   e. Port: 1521
   f. SID: `orcl`

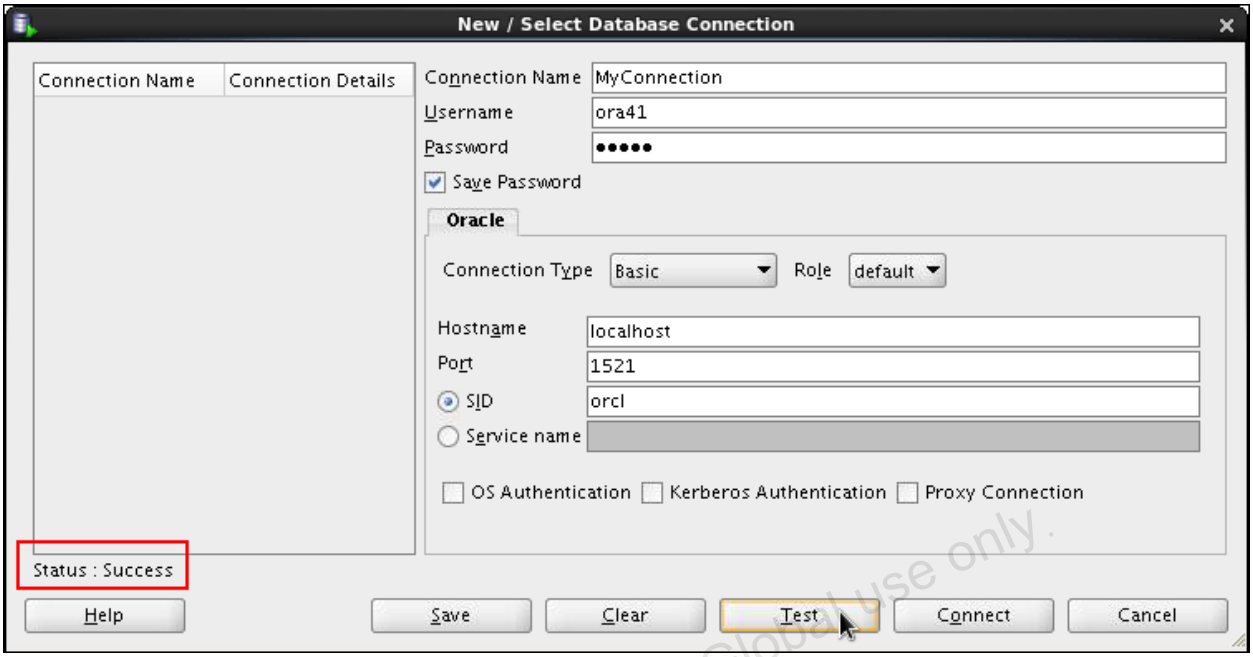
Right-click the Connections node on the Connections tabbed page and select **New Connection…**

Result: The New/Select Database Connection window appears.

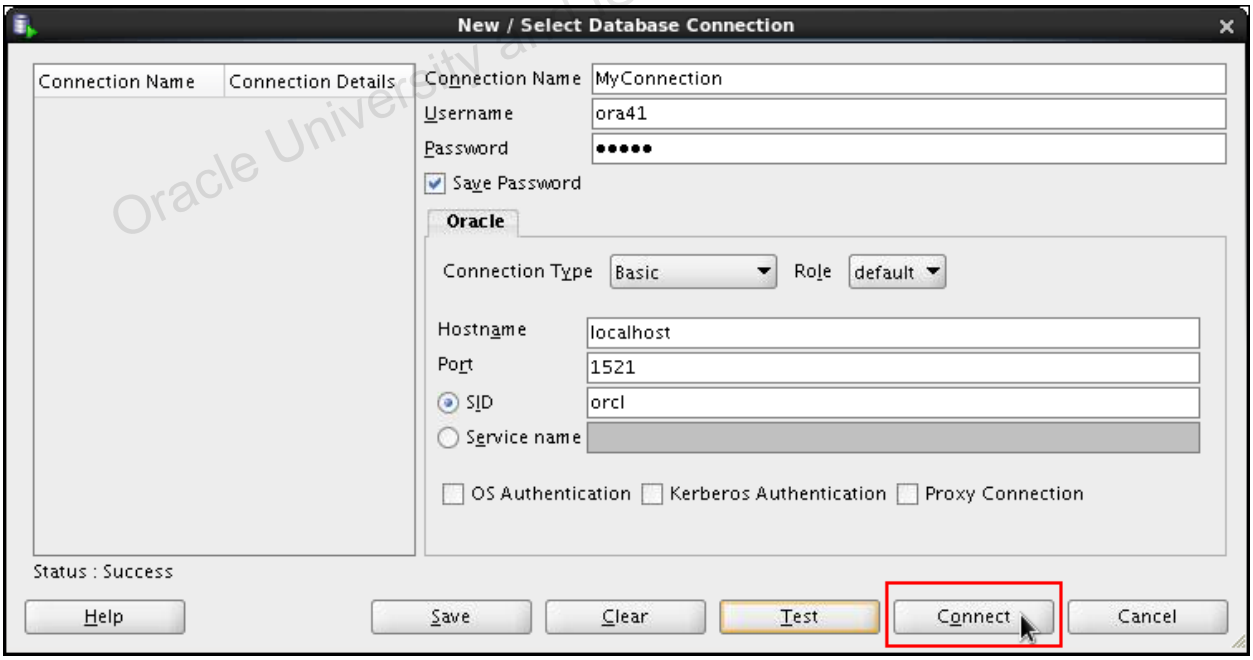
Use the preceding information to create the new database connection. In addition, select the Save Password check box. Example:



3. Test the new connection. If the Status is Success, connect to the database by using this new connection.
   a. In the Database Connection window, click the Test button.
      Note: The connection status appears in the lower-left corner of the window.



   b. If the status is Success, click the Connect button.



Note: To display the properties of an existing connection, right-click the connection name on the Connections tab and select Properties from the shortcut menu.

4. Browse the structure of the `EMPLOYEES` table and display its data.
   a. Expand the MyConnection connection by clicking the plus symbol next to it.

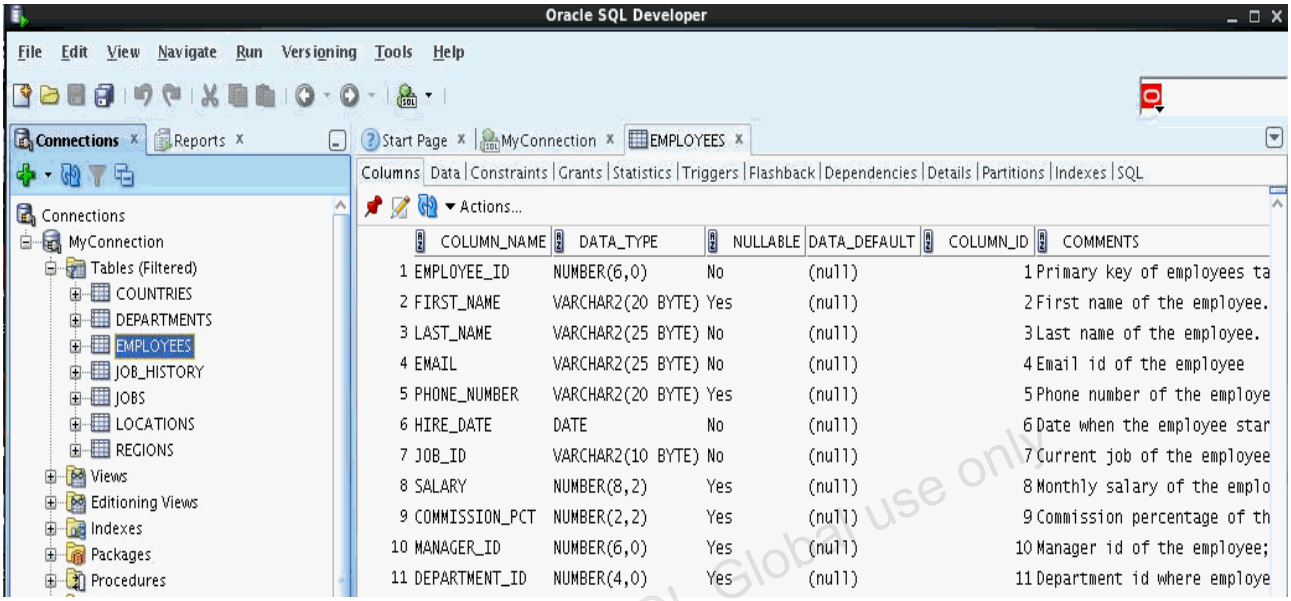
b. Expand Tables by clicking the plus symbol next to it.

c. Display the structure of the `EMPLOYEES` table.

Drill down on the `EMPLOYEES` table by clicking the plus symbol next to it.

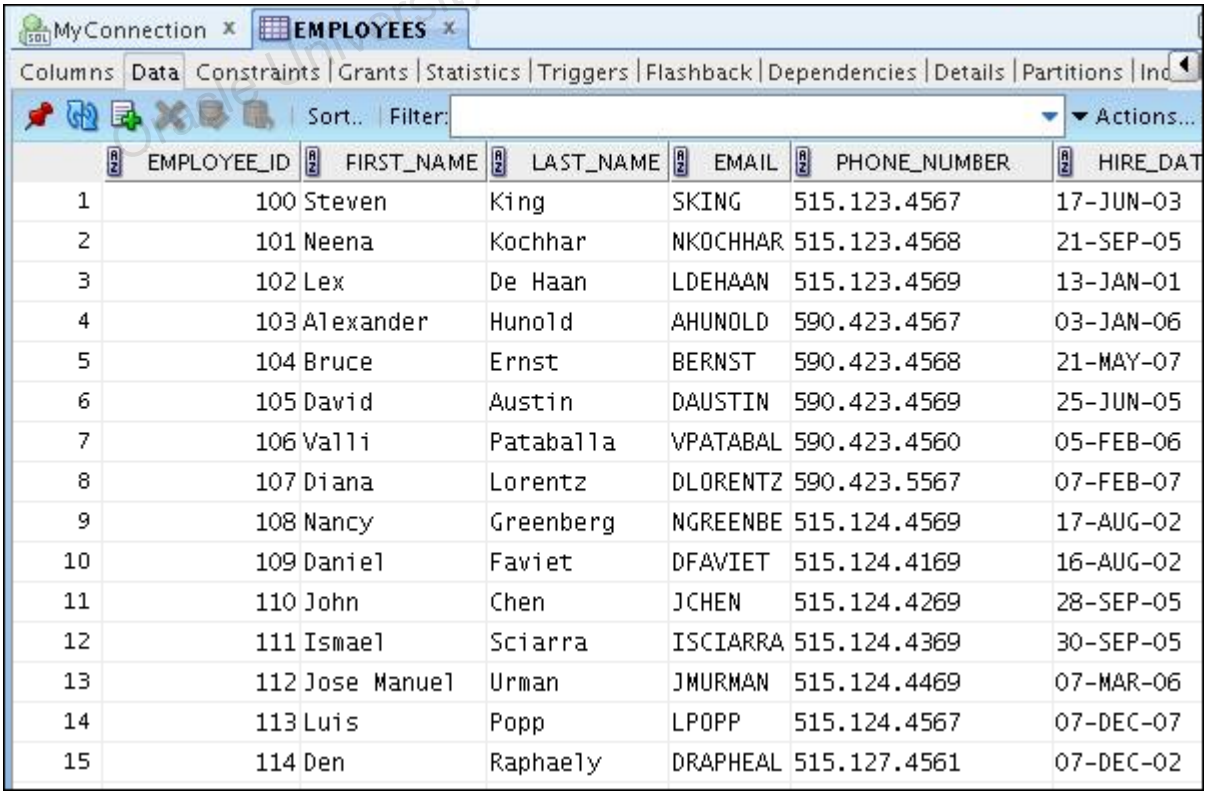
Click the `EMPLOYEES` table.

Result: The Columns tab displays the columns in the `EMPLOYEES` table as follows:



5. Use the Data tab to view the data in the `EMPLOYEES` table.

Result: The `EMPLOYEES` table data is displayed as follows:



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

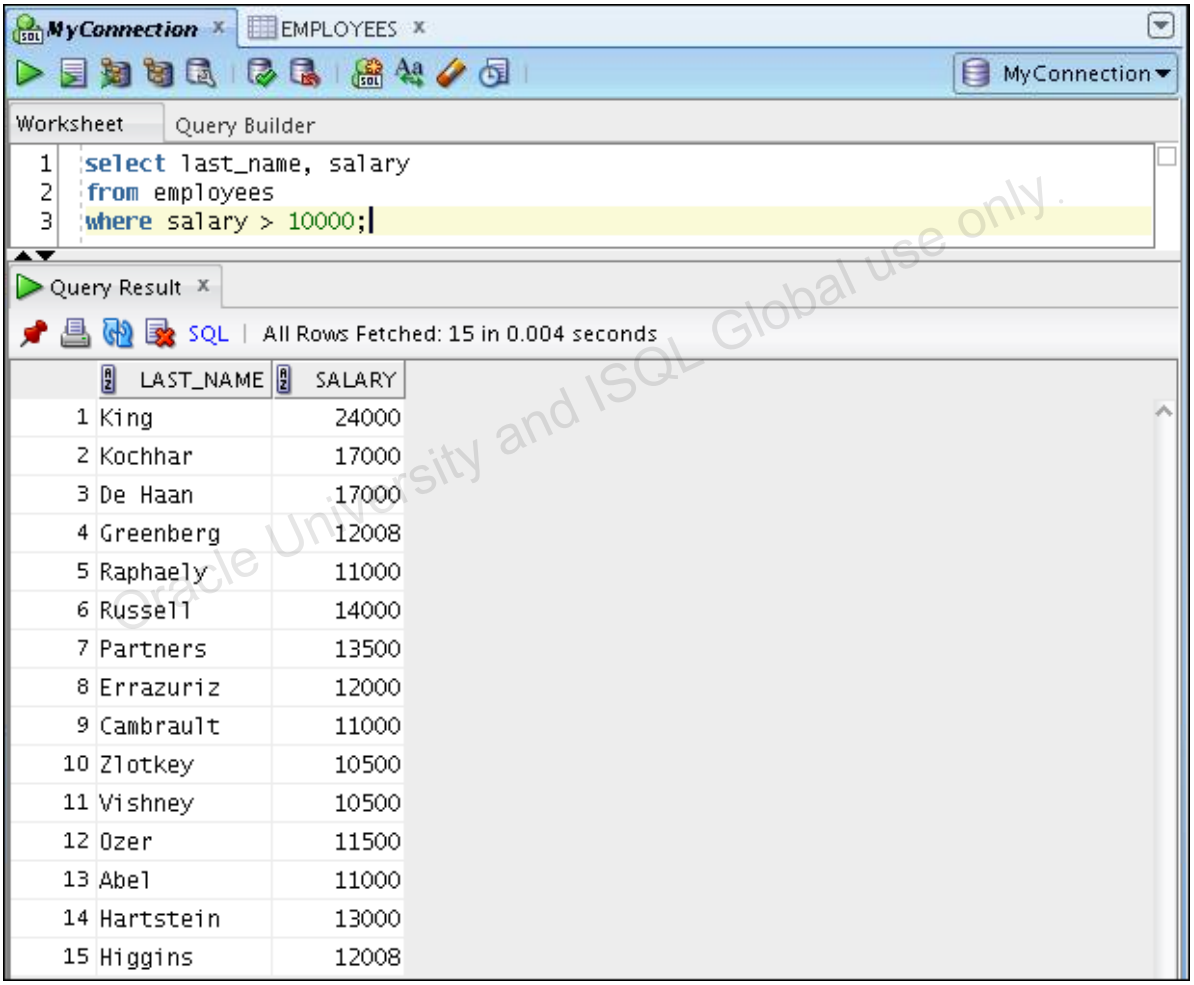
   Note: Take a few minutes to familiarize yourself with the data, or consult Appendix A, which provides the description and data for all the tables in the HR schema that you will use in this course.

To display the SQL Worksheet, click the MyConnection tab.

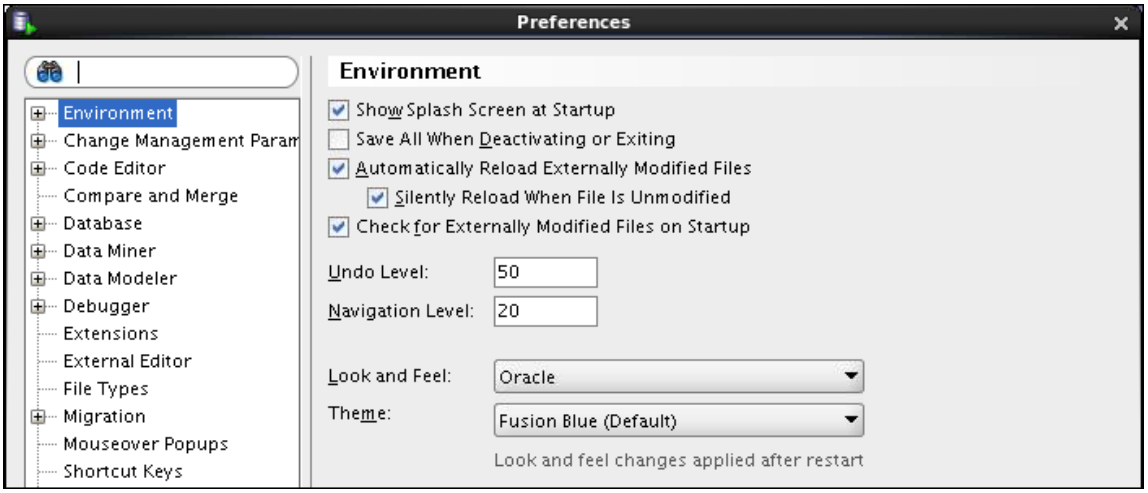
Note: This tab was opened previously when you drilled down on your database connection.

Enter the appropriate `SELECT` statement. Press F9 to execute the query and F5 to execute the query by using the Run Script method.

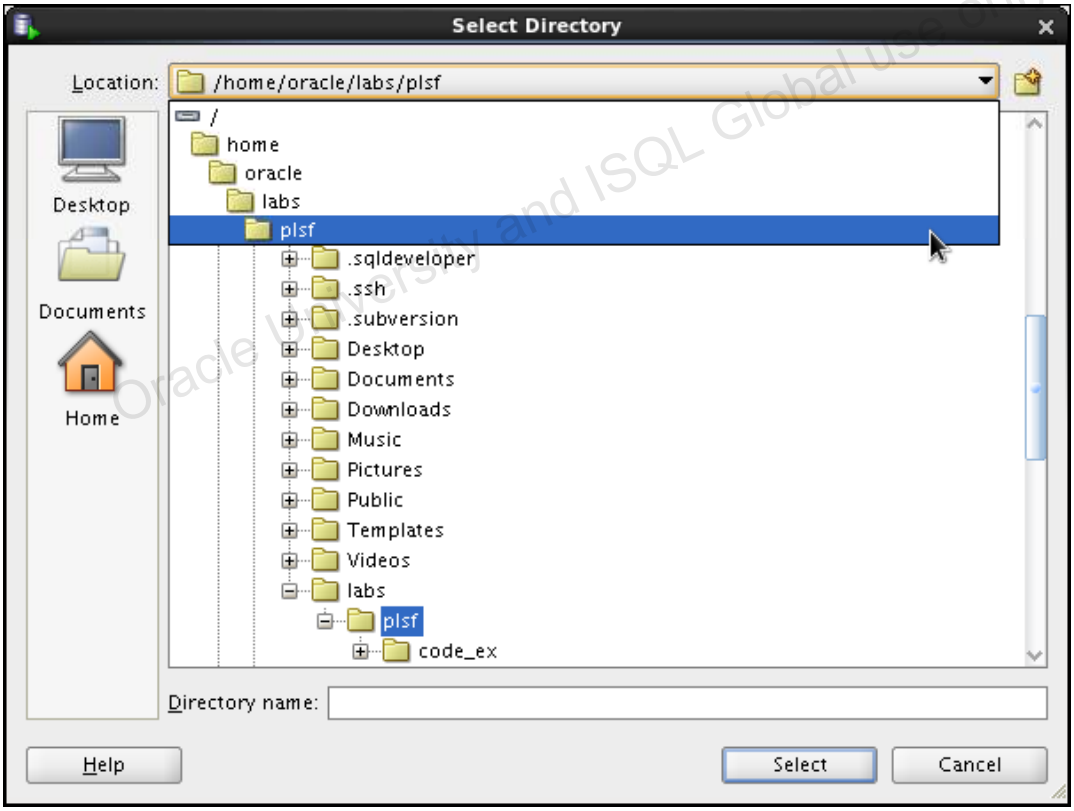
For example, when you press F9, the results appear similar to the following:



7. From the SQL Developer menu, select Tools > Preferences. The Preferences window appears.



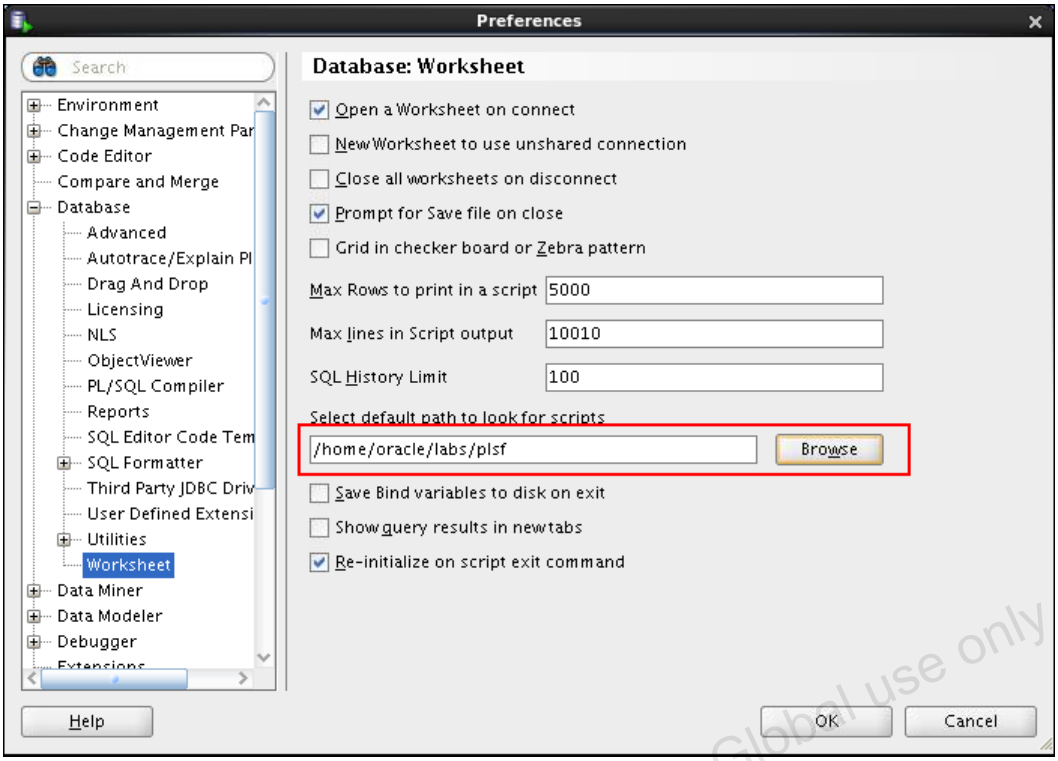
8. Select Database > Worksheet Parameters. In the "Select default path to look for scripts" text box, use the Browse button to select the `/home/oracle/labs/plsf` directory.



This directory contains the code example scripts, lab scripts, and practice solution scripts that are used in this course.

Click Select to choose the directory.

Then, in the Preferences window, click OK to save the Worksheet Parameter setting.

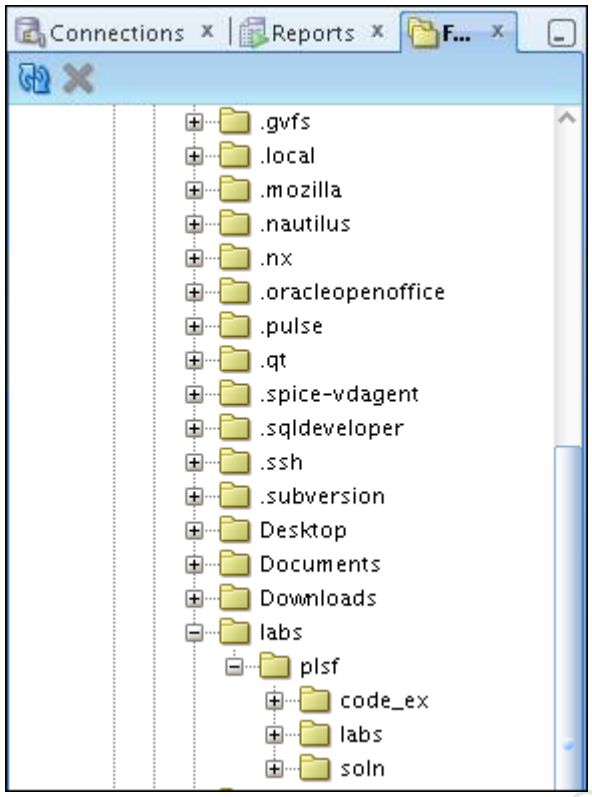


For Practice 11- Practice 22 change the directory to `/home/oracle/labs/plpu/`

9. Familiarize yourself with the structure of the `/home/oracle/labs/plsf` directory.
   a. Select File > Open. Navigate to the `/home/oracle/labs/plsf` directory. This directory contains three subdirectories:

- The `/code_ex` directory contains the code examples found in the course materials. Each `.sql` script is associated with a particular page in the lesson.
- The `/labs` directory contains the code that is used in certain lesson practices. You are instructed to run the required script in the appropriate practice.
- The `/soln` directory contains the solutions for each practice. Each `.sql` script is numbered with the associated practice_exercise reference.

b. You can also use the Files tab to navigate through directories to open script files.



c. Using the Open window, and the Files tab, navigate through the directories and open a script file without executing the code.

d. Close the SQL Worksheet.

To close any SQL Worksheet tab, click X on the tab, as shown here:

Practices for Lesson 2: Introduction to PL/SQL

Chapter 2

Practices for Lesson 2: Overview

Lesson Overview

The `/home/oracle/labs/plsf/labs` folder is the working directory where you save the scripts that you create.

The solutions for all the practices are in the `/home/oracle/labs/plsf/soln` folder.

Practice 2: Introduction to PL/SQL

1. Which of the following PL/SQL blocks execute successfully?

   a.
   ```
   BEGIN
   END;
   ```

   b.
   ```
   DECLARE
   v_amount INTEGER(10);
   END;
   ```

   c.
   ```
   DECLARE
   BEGIN
   END;
   ```

   d.
   ```
   DECLARE
   v_amount INTEGER(10);
   BEGIN
   DBMS_OUTPUT.PUT_LINE(v_amount);
   END;
   ```

2. Create and execute a simple anonymous block that outputs “Hello World.” Execute and save this script as `lab_02_02_soln.sql`.

Solution 2: Introduction to PL/SQL

1. Which of the following PL/SQL blocks execute successfully?

```
a. BEGIN
   END;
b. DECLARE
   v_amount INTEGER(10);
   END;
c. DECLARE
   BEGIN
   END;
d. DECLARE
   v_amount INTEGER(10);
   BEGIN
   DBMS_OUTPUT.PUT_LINE(v_amount);
   END;
```

The block in a does not execute. It has no executable statements.
The block in b does not have the mandatory executable section that starts with the `BEGIN` keyword.

The block in c has all the necessary parts, but no executable statements.

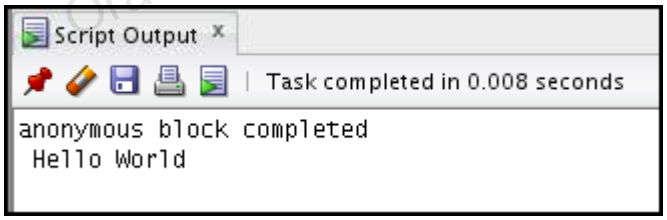
The block in d executes successfully.

2. Create and execute a simple anonymous block that outputs "Hello World." Execute and save this script as `lab_02_02_soln.sql`.

Enter the following code in the workspace, and then press F5.

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

You should see the following output on the Script Output tab:



Click the Save button. Select the folder in which you want to save the file. Enter `lab_02_02_soln.sql` as the file name and click Save.

Practices for Lesson 3: Declaring PL/SQL Variables

Chapter 3

Practice 3: Declaring PL/SQL Variables

In this practice, you declare PL/SQL variables.

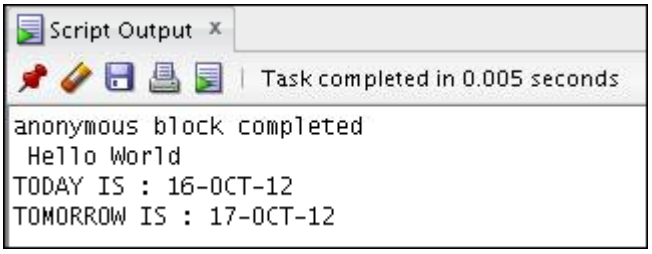
1. Identify valid and invalid identifiers:
   a. `today`
   b. `last_name`
   c. `today's_date`
   d. `Number_of_days_in_February_this_year`
   e. `Isleap$year`
   f. `#number`
   g. `NUMBER#`
   h. `number1to7`
2. Identify valid and invalid variable declaration and initialization:
   a. `number_of_copies PLS_INTEGER;`
   b. `PRINTER_NAME constant VARCHAR2(10);`
   c. `deliver_to VARCHAR2(10)`:=Johnson;
   d. `by_when DATE:= CURRENT_DATE`+1;
3. Examine the following anonymous block, and then select a statement from the following that is true.

```
DECLARE
  v_fname VARCHAR2(20);
  v_lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_fname ||' ' ||v_lname);
END;
```

   a. The block executes successfully and prints "fernandez."
   b. The block produces an error because the `fname` variable is used without initializing.
   c. The block executes successfully and prints "null fernandez."
   d. The block produces an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
   e. The block produces an error because the `v_fname` variable is not declared.

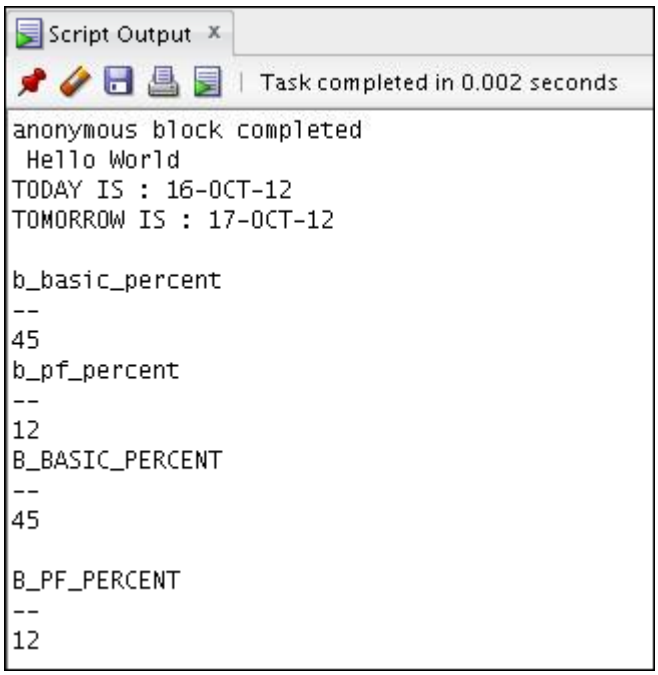
4. Modify an existing anonymous block and save it as a new script.
   a. Open the `lab_02_02_soln.sql` script, which you created in Practice 2.
   b. In this PL/SQL block, declare the following variables:
      1) `v_today` of type `DATE`. Initialize `today` with `SYSDATE`.
      2) `v_tomorrow` of type `today`. Use the `%TYPE` attribute to declare this variable.
   c. In the executable section:
      1) Initialize the `v_tomorrow` variable with an expression, which calculates tomorrow's date (add one to the value in `today`).
      2) Print the value of `v_today` and `tomorrow` after printing "Hello World."
   d. Save your script as `lab_03_04_soln.sql`, and then execute.

      The sample output is as follows (the values of `v_today` and `v_tomorrow` will be different to reflect your current today's and tomorrow's date):



5. Edit the `lab_03_04_soln.sql` script.
   a. Add code to create two bind variables, named `b_basic_percent` and `b_pf_percent`. Both bind variables are of type `NUMBER`.
   b. In the executable section of the PL/SQL block, assign the values 45 and 12 to `b_basic_percent` and `b_pf_percent`, respectively.
   c. Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.

d. Execute and save your script as `lab_03_05_soln.sql`. The sample output is as follows:



Solution 3: Declaring PL/SQL Variables

1. Identify valid and invalid identifiers:
   a. `today` **Valid**
   b. `last_name` **Valid**
   c. `today's_date` **Invalid** – character "'" not allowed
   d. `Number_of_days_in_February_this_year` **Invalid** – Too long
   e. `Isleap$year` **Valid**
   f. `#number` **Invalid –** Cannot start with "#"
   g. `NUMBER#` **Valid**
   h. `number1to7` **Valid**
2. Identify valid and invalid variable declaration and initialization:

a.	`number_of_copies`	`PLS_INTEGER;`	**Valid**
b.	`PRINTER_NAME`	`constant VARCHAR2(10);`	**Invalid**
c.	`deliver_to`	`VARCHAR2(10):=`Johnson;	**Invalid**
d.	`by_when`	`DATE:= CURRENT_DATE`+1;	**Valid**

*The declaration in **b** is invalid because constant variables must be initialized during declaration.*

*The declaration in **c** is invalid because string literals should be enclosed within single quotation marks.*

3. Examine the following anonymous block, and then select a statement from the following that is true.

```
DECLARE
  v_fname VARCHAR2(20);
  v_lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_fname ||' ' ||v_lname);
END;
```

   a. The block executes successfully and prints "fernandez."
   b. The block produces an error because the `fname` variable is used without initializing.
   c. The block executes successfully and prints "null fernandez."
   d. The block produces an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
   e. The block produces an error because the `v_fname` variable is not declared.

   f. The block will execute successfully and print "fernandez."

4. Modify an existing anonymous block and save it as a new script.
   a. Open the `lab_02_02_soln.sql` script, which you created in Practice 2.
   b. In the PL/SQL block, declare the following variables:
      1) Variable `v_today` of type `DATE`. Initialize `today` with `SYSDATE`.

```
DECLARE
   v_today DATE:=SYSDATE;
```

      2) Variable `v_tomorrow` of type `today`. Use the `%TYPE` attribute to declare this variable.

```
   v_tomorrow v_today%TYPE;
```

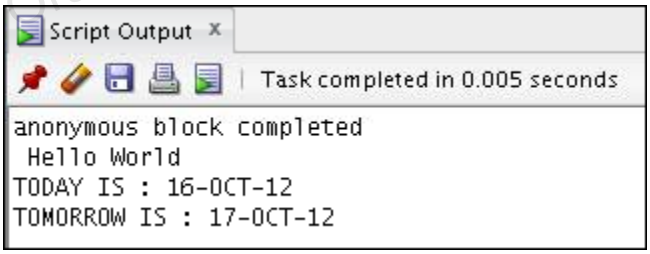
In the executable section:

   1) Initialize the `v_tomorrow` variable with an expression, which calculates tomorrow's date (add one to the value in `v_today`).
   2) Print the value of `v_today` and `v_tomorrow` after printing "Hello World."

```
BEGIN
   v_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello World ');
   DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| v_today);
   DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);
END;
```

   c. Save your script as `lab_03_04_soln.sql`, and then execute.

      The sample output is as follows (the values of `v_today` and `v_tomorrow` will be different to reflect your current today's and tomorrow's date):



5. Edit the `lab_03_04_soln.sql` script.
   a. Add the code to create two bind variables, named `b_basic_percent` and `b_pf_percent`. Both bind variables are of type `NUMBER`.

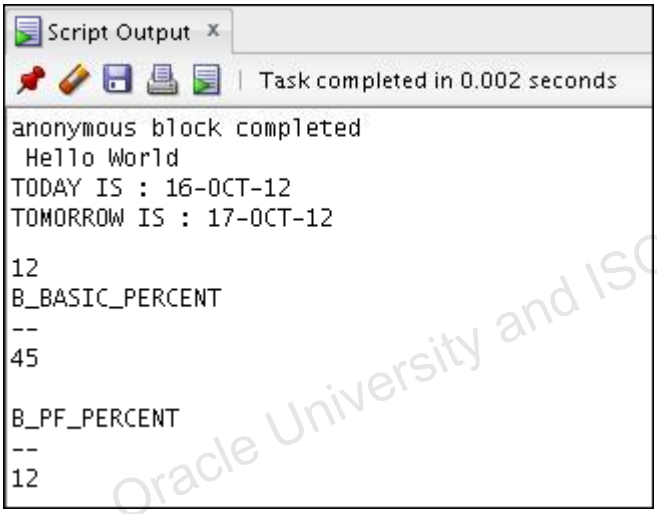
```
VARIABLE b_basic_percent NUMBER
VARIABLE b_pf_percent NUMBER
```

   b. In the executable section of the PL/SQL block, assign the values `45` and `12` to `b_basic_percent` and `b_pf_percent`, respectively.

```
   :b_basic_percent:=45;
   :b_pf_percent:=12;
```

   c. Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.

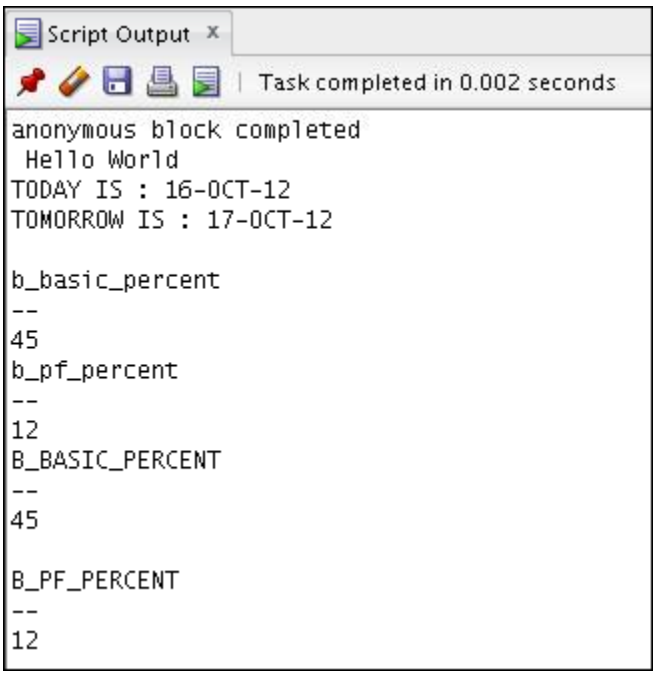
```
/
PRINT b_basic_percent
PRINT b_pf_percent
```



OR

```
PRINT
```

d. Execute and save your script as `lab_03_05_soln.sql`. The sample output is as follows:



Practices for Lesson 4: Writing Executable Statements

Chapter 4

Practice 4: Writing Executable Statements

Note: If you have executed the code examples for this lesson, make sure you execute the following code before starting this practice:

```
DROP sequence my_seq;
```

In this practice, you examine and write executable statements.

```
DECLARE
 v_weight    NUMBER(3) := 600;
 v_message   VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   v_weight    NUMBER(3) := 1;
   v_message   VARCHAR2(255) := 'Product 11001';
   v_new_locn  VARCHAR2(50) := 'Europe';
  BEGIN
   v_weight := v_weight + 1;
   v_new_locn := 'Western ' || v_new_locn;
1 →
  END;
 v_weight := v_weight + 1;
 v_message := v_message || ' is in stock';
 v_new_locn := 'Western ' || v_new_locn;
2 →
END;
/
```

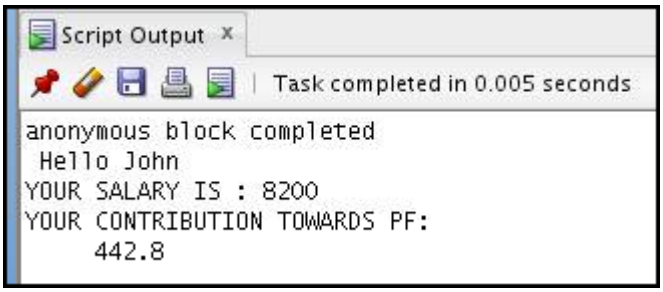
1. Evaluate the preceding PL/SQL block and determine the data type and value of each of the following variables, according to the rules of scoping.
   a. The value of `v_weight` at position 1 is:
   b. The value of `v_new_locn` at position 1 is:
   c. The value of `v_weight` at position 2 is:
   d. The value of `v_message` at position 2 is:
   e. The value of `v_new_locn` at position 2 is:

```
DECLARE
   v_customer     VARCHAR2(50) := 'Womansport';
   v_credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       v_customer  NUMBER(7) := 201;
       v_name VARCHAR2(25) := 'Unisports';
    BEGIN
       v_credit_rating :='GOOD';
       …
    END;
  …
END;
```

2. In the preceding PL/SQL block, determine the values and data types for each of the following cases:
   a. The value of `v_customer` in the nested block is:
   b. The value of `v_name` in the nested block is:
   c. The value of `v_credit_rating` in the nested block is:
   d. The value of `v_customer` in the main block is:
   e. The value of `v_name` in the main block is:
   f. The value of `v_credit_rating` in the main block is:
3. Use the same session that you used to execute the practices in the lesson titled "Declaring PL/SQL Variables." If you have opened a new session, execute `lab_03_05_soln.sql`. Then, edit `lab_03_05_soln.sql` as follows:
   a. Use single-line comment syntax to comment the lines that create the bind variables, and turn on `SERVEROUTPUT`.
   b. Use multiple-line comments in the executable section to comment the lines that assign values to the bind variables.
   c. In the declaration section:
      1) Declare and initialize two temporary variables to replace the commented out bind variables.
      2) Declare two additional variables: `v_fname` of type `VARCHAR2` and size 15, and `v_emp_sal` of type `NUMBER` and size 10.
   d. Include the following SQL statement in the executable section:

```
SELECT first_name, salary INTO v_fname, v_emp_sal
  FROM employees WHERE employee_id=110;
```

   e. Change the line that prints "Hello World" to print "Hello" and the first name. Then, comment the lines that display the dates and print the bind variables.
   f. Calculate the contribution of the employee toward provident fund (PF).
   PF is 12% of the basic salary, and the basic salary is 45% of the salary. Use local variables for the calculation. Try to use only one expression to calculate the PF. Print the employee's salary and his or her contribution toward PF.
   g. Execute and save your script as `lab_04_03_soln.sql`. The sample output is as follows:



Solution 4: Writing Executable Statements

In this practice, you examine and write executable statements.

```
DECLARE
 v_weight    NUMBER(3) := 600;
 v_message   VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   v_weight    NUMBER(3) := 1;
   v_message   VARCHAR2(255) := 'Product 11001';
   v_new_locn  VARCHAR2(50) := 'Europe';
  BEGIN
   v_weight := v_weight + 1;
   v_new_locn := 'Western ' || v_new_locn;
1 ------------------------------------------>
  END;
 v_weight := v_weight + 1;
 v_message := v_message || ' is in stock';
 v_new_locn := 'Western ' || v_new_locn;
2 ------------------------------------------>
END;
/
```

1. Evaluate the preceding PL/SQL block and determine the data type and value of each of the following variables, according to the rules of scoping.
   a. The value of `v_weight` at position 1 is:
      2
      The data type is `NUMBER`.
   b. The value of `v_new_locn` at position 1 is:
      Western Europe
      The data type is `VARCHAR2`.
   c. The value of `v_weight` at position 2 is:
      601
      The data type is `NUMBER`.
   d. The value of `v_message` at position 2 is:
      Product 10012 is in stock
      The data type is `VARCHAR2`.
   e. The value of `v_new_locn` at position 2 is:
      Illegal because `v_new_locn` is not visible outside the subblock

```
DECLARE
   v_customer     VARCHAR2(50) := 'Womansport';
   v_credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       v_customer  NUMBER(7) := 201;
       v_name VARCHAR2(25) := 'Unisports';
    BEGIN
       v_credit_rating :='GOOD';
       …
    END;
  …
END;
```

2. In the preceding PL/SQL block, determine the values and data types for each of the following cases:
   a. The value of `v_customer` in the nested block is:
      201
      The data type is `NUMBER`.
   b. The value of `v_name` in the nested block is:
      Unisports
      The data type is `VARCHAR2`.
   c. The value of `v_credit_rating` in the nested block is:
      GOOD
      The data type is `VARCHAR2`.
   d. The value of `v_customer` in the main block is:
      Womansport
      The data type is `VARCHAR2`.
   e. The value of `v_name` in the main block is:
      Null. **`name` is not visible in the main block and you would see an error.**
   f. The value of `v_credit_rating` in the main block is:
      EXCELLENT
      The data type is `VARCHAR2`.
3. Use the same session that you used to execute the practices in the lesson titled "Declaring PL/SQL Variables." If you have opened a new session, execute `lab_03_05_soln.sql`. Then, edit `lab_03_05_soln.sql` as follows:
   a. Use single-line comment syntax to comment the lines that create the bind variables, and turn on `SERVEROUTPUT`.

```
-- VARIABLE b_basic_percent NUMBER
-- VARIABLE b_pf_percent NUMBER
SET SERVEROUTPUT ON
```

b. Use multiple-line comments in the executable section to comment the lines that assign values to the bind variables.

```
   /*:b_basic_percent:=45;
   :b_pf_percent:=12;*/
```

c. In the declaration section:
   1) Declare and initialize two temporary variables to replace the commented out bind variables.
   2) Declare two additional variables: `v_fname` of type `VARCHAR2` and size 15, and `v_emp_sal` of type `NUMBER` and size 10.

```
DECLARE
   v_basic_percent NUMBER:=45;
   v_pf_percent NUMBER:=12;
   v_fname VARCHAR2(15);
   v_emp_sal NUMBER(10);
```

d. Include the following SQL statement in the executable section:

```
   SELECT first_name, salary INTO v_fname, v_emp_sal
     FROM employees WHERE employee_id=110;
```

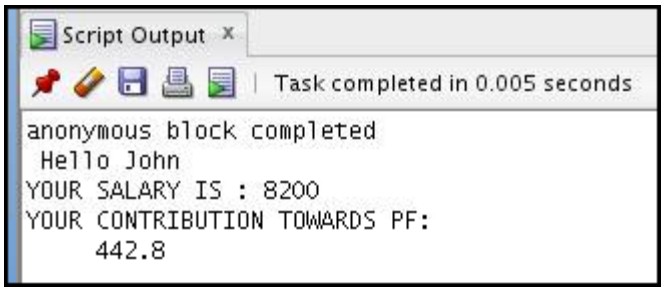
e. Change the line that prints “Hello World” to print “Hello” and the first name. Then, comment the lines that display the dates and print the bind variables.

```
DBMS_OUTPUT.PUT_LINE(' Hello '|| v_fname);
/*   DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| v_today);
DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);*/
...
...

/
--PRINT b_basic_percent
--PRINT b_basic_percent
```

f. Calculate the contribution of the employee toward provident fund (PF).
PF is 12% of the basic salary, and the basic salary is 45% of the salary. Use local variables for the calculation. Try to use only one expression to calculate the PF. Print the employee’s salary and his or her contribution toward PF.

```
   DBMS_OUTPUT.PUT_LINE('YOUR SALARY IS : '||v_emp_sal);
   DBMS_OUTPUT.PUT_LINE('YOUR CONTRIBUTION TOWARDS PF:
     '||v_emp_sal*v_basic_percent/100*v_pf_percent/100);
END;
```

g. Execute and save your script as `lab_04_03_soln.sql`. The sample output is as follows:



Practices for Lesson 5: Using SQL Statements within a PL/SQL Block

Chapter 5

Practice 5: Using SQL Statements Within a PL/SQL

Note: If you have executed the code examples for this lesson, make sure you execute the following code before starting this practice:

```
DROP table employees2;
DROP table copy_emp;
```

In this practice, you use PL/SQL code to interact with the Oracle Server.

1. Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `v_max_deptno` variable. Display the maximum department ID.
   a. Declare a variable `v_max_deptno` of type `NUMBER` in the declarative section.
   b. Start the executable section with the `BEGIN` keyword and include a `SELECT` statement to retrieve the maximum `department_id` from the `departments` table.
   c. Display `v_max_deptno` and end the executable block.
   d. Execute and save your script as `lab_05_01_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
```

2. Modify the PL/SQL block that you created in step 1 to insert a new department into the `departments` table.
   a. Load the `lab_05_01_soln.sql` script. Declare two variables:
      `v_dept_name` of type `departments.department_name` and
      `v_dept_id` of type `NUMBER`
      Assign 'Education' to `v_dept_name` in the declarative section.
   b. You have already retrieved the current maximum department number from the `departments` table. Add 10 to it and assign the result to `v_dept_id`.
   c. Include an `INSERT` statement to insert data into the `department_name`, `department_id`, and `location_id` columns of the `departments` table. Use values in `v_dept_name` and `v_dept_id` for `department_name` and `department_id`, respectively, and use `NULL` for `location_id`.
   d. Use the SQL attribute `SQL%ROWCOUNT` to display the number of rows that are affected.
   e. Execute a `SELECT` statement to check whether the new department is inserted. You can terminate the PL/SQL block with "`/`" and include the `SELECT` statement in your script.
   f. Execute and save your script as `lab_05_02_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
 SQL%ROWCOUNT gives 1

DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          280 Education
```

3. In step 2, you set `location_id` to `NULL`. Create a PL/SQL block that updates `location_id` to `3000` for the new department.
   Note: If you have successfully completed step 2, continue with step 3a. If not, first execute the solution script `/soln/sol_05_02.sql`.

   a. Start the executable block with the `BEGIN` keyword. Include the `UPDATE` statement to set `location_id` to `3000` for the new department (`v_dept_id` =280).

   b. End the executable block with the `END` keyword. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.

   c. Include a `DELETE` statement to delete the department that you added.

   d. Execute and save your script as `lab_05_03_soln.sql`. The sample output is as follows:

```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          280 Education                                        3000

1 rows deleted.
```

Solution 5: Using SQL Statements Within a PL/SQL

In this practice, you use PL/SQL code to interact with the Oracle Server.

1. Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `v_max_deptno` variable. Display the maximum department ID.
   a. Declare a variable `v_max_deptno` of type `NUMBER` in the declarative section.

```
DECLARE
   v_max_deptno  NUMBER;
```

   b. Start the executable section with the `BEGIN` keyword and include a `SELECT` statement to retrieve the maximum `department_id` from the `departments` table.

```
BEGIN
   SELECT MAX(department_id)  INTO v_max_deptno  FROM
     departments;
```

   c. Display `v_max_deptno` and end the executable block.

```
DBMS_OUTPUT.PUT_LINE('The maximum department_id is : ' ||
v_max_deptno);
END;
```

   d. Execute and save your script as `lab_05_01_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
```

2. Modify the PL/SQL block that you created in step 1 to insert a new department into the `departments` table.
   a. Load the `lab_05_01_soln.sql` script. Declare two variables:
   `v_dept_name` of type `departments.department_name` and
   `v_dept_id` of type `NUMBER`
   Assign 'Education' to `v_dept_name` in the declarative section.

```
  v_dept_name departments.department_name%TYPE:= 'Education';
  v_dept_id NUMBER;
```

   b. You have already retrieved the current maximum department number from the `departments` table. Add 10 to it and assign the result to `v_dept_id`.

```
v_dept_id := 10 + v_max_deptno;
```

c. Include an `INSERT` statement to insert data into the `department_name`, `department_id`, and `location_id` columns of the `departments` table. Use values in `v_dept_name` and `v_dept_id` for `department_name` and `department_id`, respectively, and use `NULL` for `location_id`.

```
…
INSERT INTO departments (department_id, department_name,
location_id)
VALUES (v_dept_id, v_dept_name, NULL);
```

d. Use the SQL attribute `SQL%ROWCOUNT` to display the number of rows that are affected.

```
DBMS_OUTPUT.PUT_LINE (' SQL%ROWCOUNT gives ' || SQL%ROWCOUNT);
…
```

e. Execute a `SELECT` statement to check whether the new department is inserted. You can terminate the PL/SQL block with "`/`" and include the `SELECT` statement in your script.

```
…
/
SELECT * FROM  departments  WHERE  department_id= 280;
```

f. Execute and save your script as `lab_05_02_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
 SQL%ROWCOUNT gives 1

DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          280 Education
```

3. In step 2, you set `location_id` to `NULL`. Create a PL/SQL block that updates the `location_id` to `3000` for the new department.
Note: If you successfully completed step 2, continue with step 3a. If not, first execute the solution script `/soln/sol_05_02.sql`.

   a. Start the executable block with the `BEGIN` keyword. Include the `UPDATE` statement to set `location_id` to `3000` for the new department (`v_dept_id` =280).

```
BEGIN
UPDATE departments SET location_id=3000 WHERE
department_id=280;
```

b. End the executable block with the `END` keyword. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.

```
END;
/
SELECT * FROM departments WHERE department_id=280;
```

c. Include a `DELETE` statement to delete the department that you added.

```
DELETE FROM departments WHERE department_id=280;
```

d. Execute and save your script as `lab_05_03_soln.sql`. The sample output is as follows:

```
anonymous block completed
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
          280 Education                                        3000

1 rows deleted.
```

Practices for Lesson 6: Writing Control Structures

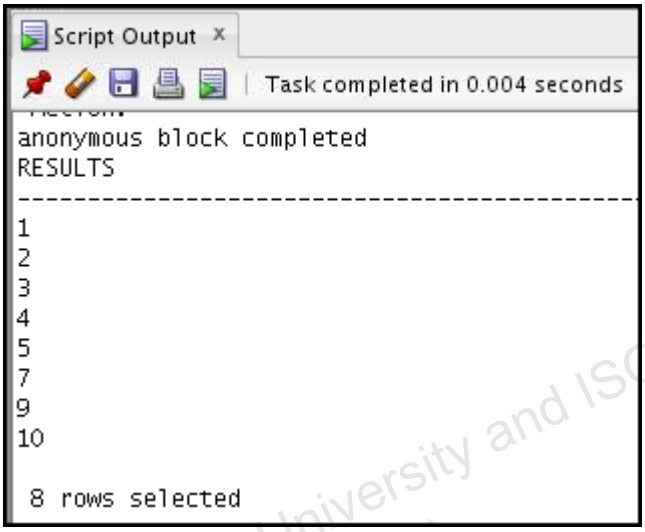
Chapter 6

Practice 6: Writing Control Structures

In this practice, you create PL/SQL blocks that incorporate loops and conditional control structures. This practice tests your understanding of various `IF` statements and `LOOP` constructs.

1. Execute the command in the `lab_06_01.sql` file to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.
   a. Insert the numbers 1 through 10, excluding 6 and 8.
   b. Commit before the end of the block.
   c. Execute a `SELECT` statement to verify that your PL/SQL block worked.

Result: You should see the following output:



2. Execute the `lab_06_02.sql` script. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size 50. Create a PL/SQL block that inserts an asterisk in the `stars` column for every `$1000` of an employee's salary. Save your script as `lab_06_02_soln.sql`.
   a. In the declarative section of the block, declare a variable `v_empno` of type `emp.employee_id` and initialize it to 176. Declare a variable `v_asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `v_sal` of type `emp.salary`.
   b. In the executable section, write logic to append an asterisk (`*`) to the string for every $1,000 of the salary. For example, if the employee earns $8,000, the string of asterisks should contain eight asterisks. If the employee earns $12,500, the string of asterisks should contain 13 asterisks (rounded to the nearest whole number).
   c. Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.
   d. Display the row from the `emp` table to verify whether your PL/SQL block has executed successfully.

e. Execute and save your script as `lab_06_02_soln.sql`. The output is as follows:

Solution 6: Writing Control Structures

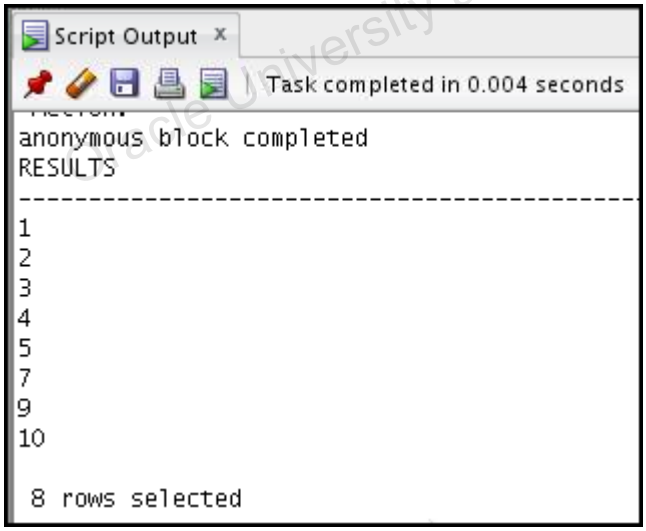
1. Execute the command in the `lab_06_01.sql` file to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.
   a. Insert the numbers 1 through 10, excluding 6 and 8.
   b. Commit before the end of the block.

```
BEGIN
FOR i in 1..10 LOOP
  IF i = 6 or i = 8 THEN
    null;
  ELSE
    INSERT INTO messages(results)
    VALUES (i);
 END IF;
END LOOP;
COMMIT;
END;
/
```

   c. Execute a `SELECT` statement to verify that your PL/SQL block worked.

```
SELECT * FROM messages;
```

Result: You should see the following output:



2. Execute the `lab_06_02.sql` script. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size `50`. Create a PL/SQL block that inserts an asterisk in the `stars` column for every `$1000` of the employee's salary. Save your script as `lab_06_02_soln.sql`.
   a. In the declarative section of the block, declare a variable `v_empno` of type `emp.employee_id` and initialize it to 176. Declare a variable `v_asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `v_sal` of type `emp.salary`.

```
DECLARE
  v_empno       emp.employee_id%TYPE := 176;
  v_asterisk    emp.stars%TYPE := NULL;
  v_sal          emp.salary%TYPE;
```

   b. In the executable section, write logic to append an asterisk (*) to the string for every $1,000 of the salary. For example, if the employee earns $8,000, the string of asterisks should contain eight asterisks. If the employee earns $12,500, the string of asterisks should contain 13 asterisks.

```
BEGIN
    SELECT NVL(ROUND(salary/1000), 0)  INTO v_sal
    FROM emp  WHERE employee_id = v_empno;

    FOR i IN 1..v_sal
       LOOP
        v_asterisk := v_asterisk ||'*';
    END LOOP;
```

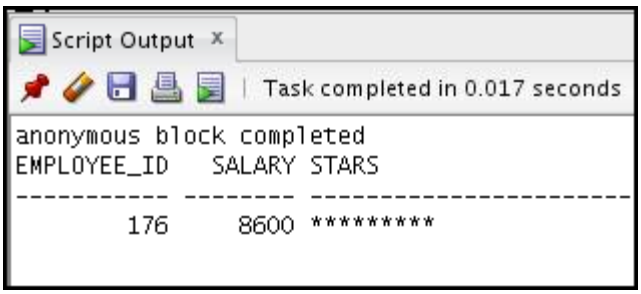
   c. Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

```
   UPDATE emp SET stars = v_asterisk
   WHERE employee_id = v_empno;
   COMMIT;
END;
/
```

   d. Display the row from the `emp` table to verify whether your PL/SQL block has executed successfully.

```
SELECT employee_id,salary, stars
FROM emp WHERE employee_id =176;
```

e. Execute and save your script as `lab_06_02_soln.sql`. The output is as follows:



Practices for Lesson 7: Working with Composite Data Types

Chapter 7

Practice 7: Working with Composite Data Types

Note: If you have executed the code examples for this lesson, make sure you execute the following code before starting this practice:

```
DROP table retired_emps;
DROP table empl;
```

1. Write a PL/SQL block to print information about a given country.
   a. Declare a PL/SQL record based on the structure of the `COUNTRIES` table.
   b. Declare a variable `v_countryid`. Assign `CA` to `v_countryid`.
   c. In the declarative section, use the `%ROWTYPE` attribute and declare the `v_country_record` variable of type `countries`.
   d. In the executable section, get all the information from the `COUNTRIES` table by using `v_countryid`. Display selected information about the country. The sample output is as follows:

```
anonymous block completed
Country Id: CA Country Name: Canada Region: 2
```

   e. You may want to execute and test the PL/SQL block for countries with the IDs DE, UK, and US.
2. Create a PL/SQL block to retrieve the names of some departments from the `DEPARTMENTS` table and print each department name on the screen, incorporating an associative array. Save the script as `lab_07_02_soln.sql`.
   a. Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the names of the departments.
   b. Declare two variables: `f_loop_count` and `v_deptno` of type `NUMBER`. Assign 10 to `f_loop_count` and 0 to `v_deptno`.
   c. Using a loop, retrieve the names of 10 departments and store the names in the associative array. Start with `department_id` 10. Increase `v_deptno` by 10 for every loop iteration. The following table shows the `department_id` for which you should retrieve the `department_name`.

DEPARTMENT_ID	DEPARTMENT_NAME
10	**Administration**
20	**Marketing**
30	**Purchasing**
40	**Human Resources**
50	**Shipping**
60	**IT**
70	**Public Relations**
80	**Sales**

90	Executive
100	Finance

d. Using another loop, retrieve the department names from the associative array and display them.

e. Execute and save your script as `lab_07_02_soln.sql`. The output is as follows:

```
anonymous block completed
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
```

3. Modify the block that you created in Practice 2 to retrieve all information about each department from the `DEPARTMENTS` table and display the information. Use an associative array with the `INDEX BY` table of records method.

   a. Load the `lab_07_02_soln.sql` script.

   b. You have declared the associative array to be of type `departments.department_name`. Modify the declaration of the associative array to temporarily store the number, name, and location of all the departments. Use the `%ROWTYPE` attribute.

   c. Modify the `SELECT` statement to retrieve all department information currently in the `DEPARTMENTS` table and store it in the associative array.

   d. Using another loop, retrieve the department information from the associative array and display the information.

The sample output is as follows:

```
anonymous block completed
Department Number: 10 Department Name: Administration Manager Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id: 201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id: 114 Location Id: 1700
Department Number: 40 Department Name: Human Resources Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id: 121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103 Location Id: 1400
Department Number: 70 Department Name: Public Relations Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145 Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id: 100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id: 108 Location Id: 1700
```

Solution 7: Working with Composite Data Types

1. Write a PL/SQL block to print information about a given country.
   a. Declare a PL/SQL record based on the structure of the COUNTRIES table.
   b. Declare a variable v_countryid. Assign CA to v_countryid.

```
SET SERVEROUTPUT ON

SET VERIFY OFF
DECLARE
  v_countryid varchar2(20):= 'CA';
```

   c. In the declarative section, use the %ROWTYPE attribute and declare the v_country_record variable of type countries.

```
 v_country_record countries%ROWTYPE;
```

   d. In the executable section, get all the information from the COUNTRIES table by using v_countryid. Display selected information about the country.

```
BEGIN
 SELECT *
 INTO   v_country_record
 FROM   countries
 WHERE country_id = UPPER(v_countryid);

 DBMS_OUTPUT.PUT_LINE ('Country Id: ' ||
     v_country_record.country_id ||
     ' Country Name: ' || v_country_record.country_name
     || ' Region: ' || v_country_record.region_id);

END;
```


The sample output after performing all the above steps is as follows:

```
anonymous block completed
Country Id: CA Country Name: Canada Region: 2
```

   e. You may want to execute and test the PL/SQL block for countries with the IDs DE, UK, and US.

2. Create a PL/SQL block to retrieve the names of some departments from the DEPARTMENTS table and print each department name on the screen, incorporating an associative array. Save the script as lab_07_02_soln.sql.
   a. Declare an INDEX BY table dept_table_type of type departments.department_name. Declare a variable my_dept_table of type dept_table_type to temporarily store the names of the departments.

```
SET SERVEROUTPUT ON

DECLARE
   TYPE dept_table_type is table of
     departments.department_name%TYPE
   INDEX BY PLS_INTEGER;
   my_dept_table   dept_table_type;
```

   b. Declare two variables: f_loop_count and v_deptno of type NUMBER. Assign 10 to f_loop_count and 0 to v_deptno.

```
   f_loop_count    NUMBER (2):=10;
   v_deptno        NUMBER (4):=0;
```

   c. Using a loop, retrieve the names of 10 departments and store the names in the associative array. Start with department_id 10. Increase v_deptno by 10 for every iteration of the loop. The following table shows the department_id for which you should retrieve the department_name and store in the associative array.

DEPARTMENT_ID	DEPARTMENT_NAME
10	**Administration**
20	**Marketing**
30	**Purchasing**
40	**Human Resources**
50	**Shipping**
60	**IT**
70	**Public Relations**
80	**Sales**
90	**Executive**
100	**Finance**

```
BEGIN
  FOR i IN 1..f_loop_count
  LOOP
        v_deptno:=v_deptno+10;
   SELECT department_name
   INTO my_dept_table(i)
   FROM departments
   WHERE department_id = v_deptno;
  END LOOP;
```

d. Using another loop, retrieve the department names from the associative array and display them.

```
FOR i IN 1..f_loop_count
  LOOP
    DBMS_OUTPUT.PUT_LINE (my_dept_table(i));
  END LOOP;
END;
```

e. Execute and save your script as `lab_07_02_soln.sql`. The output is as follows:

```
anonymous block completed
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
```

3. Modify the block that you created in Practice 2 to retrieve all information about each department from the `DEPARTMENTS` table and display the information. Use an associative array with the `INDEX BY` table of records method.
   a. Load the `lab_07_02_soln.sql` script.
   b. You have declared the associative array to be of the `departments.department_name` type. Modify the declaration of the associative array to temporarily store the number, name, and location of all the departments. Use the `%ROWTYPE` attribute.

```
SET SERVEROUTPUT ON

DECLARE
   TYPE dept_table_type is table of departments%ROWTYPE
    INDEX BY PLS_INTEGER;
    my_dept_table   dept_table_type;
    f_loop_count       NUMBER (2):=10;
    v_deptno          NUMBER (4):=0;
```

c. Modify the `SELECT` statement to retrieve all department information currently in the `DEPARTMENTS` table and store it in the associative array.

```
BEGIN
  FOR i IN 1..f_loop_count
  LOOP
   v_deptno := v_deptno + 10;
   SELECT *
   INTO my_dept_table(i)
   FROM departments
   WHERE department_id = v_deptno;
  END LOOP;
```

d. Using another loop, retrieve the department information from the associative array and display the information.

```
FOR i IN 1..f_loop_count
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Department Number: ' ||
my_dept_table(i).department_id
     || ' Department Name: ' || my_dept_table(i).department_name
     || ' Manager Id: '||  my_dept_table(i).manager_id
     || ' Location Id: ' || my_dept_table(i).location_id);
  END LOOP;
END;
```

The sample output is as follows:

```
anonymous block completed
Department Number: 10 Department Name: Administration Manager Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id: 201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id: 114 Location Id: 1700
Department Number: 40 Department Name: Human Resources Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id: 121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103 Location Id: 1400
Department Number: 70 Department Name: Public Relations Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145 Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id: 100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id: 108 Location Id: 1700
```

Practices for Lesson 8: Using Explicit Cursors

Chapter 8

Practice 8-1: Using Explicit Cursors

In this practice, you perform two exercises:

- First, you use an explicit cursor to process a number of rows from a table and populate another table with the results by using a cursor `FOR` loop.
- Second, you write a PL/SQL block that processes information with two cursors, including one that uses a parameter.

1. Create a PL/SQL block to perform the following:
   a. In the declarative section, declare and initialize a variable named `v_deptno` of type `NUMBER`. Assign a valid department ID value (see table in step d for values).
   b. Declare a cursor named `c_emp_cursor`, which retrieves the `last_name`, `salary`, and `manager_id` of employees working in the department specified in `v_deptno`.
   c. In the executable section, use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5,000 and if the manager ID is either 101 or 124, display the message "<<*last_name*>> Due for a raise." Otherwise, display the message "<<*last_name*>> Not Due for a raise."
   d. Test the PL/SQL block for the following cases:

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise. . . . . . OConnell Due for a raise Grant Due for a raise
80	Russell Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . Livingston Not Due for a raise Johnson Not Due for a raise

2. Next, write a PL/SQL block that declares and uses two cursors—one without a parameter and one with a parameter. The first cursor retrieves the department number and the department name from the `DEPARTMENTS` table for all departments whose ID number is less than 100. The second cursor receives the department number as a parameter, and retrieves employee details for those who work in that department and whose `employee_id` is less than 120.

   a. Declare a cursor `c_dept_cursor` to retrieve `department_id` and `department_name` for those departments with `department_id` less than 100. Order by `department_id`.
   b. Declare another cursor `c_emp_cursor` that takes the department number as parameter and retrieves the following data from the `EMPLOYEES` table: `last_name`, `job_id`, `hire_date`, and `salary` of those employees who work in that department, with `employee_id` less than 120.
   c. Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.
   d. Open `c_dept_cursor` and use a simple loop to fetch values into the variables declared. Display the department number and department name. Use the appropriate cursor attribute to exit the loop.
   e. Open `c_emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables, and print all the details retrieved from the `EMPLOYEES` table.

   Notes

   - Check whether `c_emp_cursor` is already open before opening the cursor.
   - Use the appropriate cursor attribute for the exit condition.
   - When the loop completes, print a line after you have displayed the details of each department, and close `c_emp_cursor`.

   f. End the first loop and close `c_dept_cursor`. Then end the executable section.

g. Execute the script. The sample output is as follows:

```
anonymous block completed
Department Number : 10  Department Name : Administration
--------------------------------------------------------------------------------
Department Number : 20  Department Name : Marketing
--------------------------------------------------------------------------------
Department Number : 30  Department Name : Purchasing
Raphaely    PU_MAN   07-DEC-02   11000
Khoo    PU_CLERK   18-MAY-03   3100
Baida    PU_CLERK   24-DEC-05   2900
Tobias    PU_CLERK   24-JUL-05   2800
Himuro    PU_CLERK   15-NOV-06   2600
Colmenares    PU_CLERK   10-AUG-07   2500
--------------------------------------------------------------------------------
Department Number : 40  Department Name : Human Resources
--------------------------------------------------------------------------------
Department Number : 50  Department Name : Shipping
--------------------------------------------------------------------------------
Department Number : 60  Department Name : IT
Hunold    IT_PROG   03-JAN-06   9000
Ernst    IT_PROG   21-MAY-07   6000
Austin    IT_PROG   25-JUN-05   4800
Pataballa    IT_PROG   05-FEB-06   4800
Lorentz    IT_PROG   07-FEB-07   4200
--------------------------------------------------------------------------------
Department Number : 70  Department Name : Public Relations
--------------------------------------------------------------------------------
Department Number : 80  Department Name : Sales
--------------------------------------------------------------------------------
Department Number : 90  Department Name : Executive
King    AD_PRES   17-JUN-03   24000
Kochhar    AD_VP   21-SEP-05   17000
De Haan    AD_VP   13-JAN-01   17000
--------------------------------------------------------------------------------
```

Solution 8-1: Using Explicit Cursors

In this practice, you perform two exercises:

- First, you use an explicit cursor to process a number of rows from a table and populate another table with the results by using a cursor `FOR` loop.
- Second, you write a PL/SQL block that processes information with two cursors, including one that uses a parameter.

1. Create a PL/SQL block to perform the following:
   a. In the declarative section, declare and initialize a variable named `v_deptno` of the `NUMBER` type. Assign a valid department ID value (see table in step d for values).

```
DECLARE
v_deptno NUMBER := 10;
```

   b. Declare a cursor named `c_emp_cursor`, which retrieves `last_name`, `salary`, and `manager_id` of employees working in the department specified in `v_deptno`.

```
CURSOR  c_emp_cursor IS
   SELECT     last_name, salary,manager_id
   FROM       employees
   WHERE      department_id = v_deptno;
```

   c. In the executable section, use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5,000 and if the manager ID is either 101 or 124, display the message "<<*last_name*>> Due for a raise." Otherwise, display the message "<<*last_name*>> Not Due for a raise."

```
BEGIN
 FOR emp_record IN c_emp_cursor
  LOOP
   IF emp_record.salary < 5000 AND (emp_record.manager_id=101 OR
emp_record.manager_id=124) THEN
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Due for a
raise');
   ELSE
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Not Due for a
raise');
   END IF;
  END LOOP;
END;
```

d. Test the PL/SQL block for the following cases:

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise. . . . . . OConnell Due for a raise Grant Due for a raise
80	Russell Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . Livingston Not Due for a raise Johnson Not Due for a raise

2. Next, write a PL/SQL block that declares and uses two cursors—one without a parameter and one with a parameter. The first cursor retrieves the department number and the department name from the `DEPARTMENTS` table for all departments whose ID number is less than 100. The second cursor receives the department number as a parameter, and retrieves employee details for those who work in that department and whose `employee_id` is less than 120.

   a. Declare a cursor `c_dept_cursor` to retrieve `department_id` and `department_name` for those departments with `department_id` less than 100. Order by `department_id`.

```
DECLARE
  CURSOR c_dept_cursor IS
        SELECT department_id,department_name
        FROM  departments
        WHERE department_id < 100
        ORDER BY    department_id;
```

b. Declare another cursor `c_emp_cursor` that takes the department number as parameter and retrieves the following data from the `EMPLOYEES` table: `last_name`, `job_id`, `hire_date`, and `salary` of those employees who work in that department, with `employee_id` less than 120.

```
CURSOR c_emp_cursor(v_deptno NUMBER) IS
        SELECT last_name,job_id,hire_date,salary
        FROM  employees
        WHERE department_id = v_deptno
        AND employee_id < 120;
```

c. Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.

```
  v_current_deptno departments.department_id%TYPE;
  v_current_dname  departments.department_name%TYPE;
  v_ename employees.last_name%TYPE;
  v_job employees.job_id%TYPE;
  v_hiredate employees.hire_date%TYPE;
  v_sal employees.salary%TYPE;
```

d. Open `c_dept_cursor` and use a simple loop to fetch values into the variables declared. Display the department number and department name. Use the appropriate cursor attribute to exit the loop.

```
BEGIN
   OPEN c_dept_cursor;
      LOOP
         FETCH c_dept_cursor INTO v_current_deptno,
           v_current_dname;
         EXIT WHEN c_dept_cursor%NOTFOUND;
         DBMS_OUTPUT.PUT_LINE ('Department Number : ' ||
           v_current_deptno || '  Department Name : ' ||
           v_current_dname);
```

e. Open `c_emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables, and print all the details retrieved from the `EMPLOYEES` table.

Notes

- Check whether `c_emp_cursor` is already open before opening the cursor.
- Use the appropriate cursor attribute for the exit condition.
- When the loop completes, print a line after you have displayed the details of each department, and close `c_emp_cursor`.

```
IF c_emp_cursor%ISOPEN THEN
  CLOSE c_emp_cursor;
END IF;
OPEN c_emp_cursor (v_current_deptno);
  LOOP
    FETCH c_emp_cursor INTO  v_ename,v_job,v_hiredate,v_sal;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_ename || '    ' ||  v_job
                    || '   '  ||  v_hiredate || '   ' ||  v_sal);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('-----------------------------------------
-----------------------------------------');
         CLOSE c_emp_cursor;
```

f. End the first loop and close `c_dept_cursor`. Then end the executable section.

```
    END LOOP;
  CLOSE c_dept_cursor;
END;
```

g. Execute the script. The sample output is as follows:

```
anonymous block completed
Department Number : 10  Department Name : Administration
--------------------------------------------------------------------------------
Department Number : 20  Department Name : Marketing
--------------------------------------------------------------------------------
Department Number : 30  Department Name : Purchasing
Raphaely    PU_MAN   07-DEC-02   11000
Khoo    PU_CLERK   18-MAY-03   3100
Baida    PU_CLERK   24-DEC-05   2900
Tobias    PU_CLERK   24-JUL-05   2800
Himuro    PU_CLERK   15-NOV-06   2600
Colmenares    PU_CLERK   10-AUG-07   2500
--------------------------------------------------------------------------------
Department Number : 40  Department Name : Human Resources
--------------------------------------------------------------------------------
Department Number : 50  Department Name : Shipping
--------------------------------------------------------------------------------
Department Number : 60  Department Name : IT
Hunold    IT_PROG   03-JAN-06   9000
Ernst    IT_PROG   21-MAY-07   6000
Austin    IT_PROG   25-JUN-05   4800
Pataballa    IT_PROG   05-FEB-06   4800
Lorentz    IT_PROG   07-FEB-07   4200
--------------------------------------------------------------------------------
Department Number : 70  Department Name : Public Relations
--------------------------------------------------------------------------------
Department Number : 80  Department Name : Sales
--------------------------------------------------------------------------------
Department Number : 90  Department Name : Executive
King    AD_PRES   17-JUN-03   24000
Kochhar    AD_VP   21-SEP-05   17000
De Haan    AD_VP   13-JAN-01   17000
--------------------------------------------------------------------------------
```

Practice 8-2: Using Explicit Cursors: Optional

If you have time, complete the following optional practice. Here, create a PL/SQL block that uses an explicit cursor to determine the top *n* salaries of employees.

1. Run the `lab_08-02.sql` script to create the `TOP_SALARIES` table for storing the salaries of the employees.
2. In the declarative section, declare the `v_num` variable of the `NUMBER` type that holds a number *n*, representing the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter `5`. Declare another variable `sal` of type `employees.salary`. Declare a cursor, `c_emp_cursor`, which retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.
3. In the executable section, open the loop and fetch the top *n* salaries, and then insert them into the `TOP_SALARIES` table. You can use a simple loop to operate on the data. Also, try and use the `%ROWCOUNT` and `%FOUND` attributes for the exit condition.

   Note: Make sure that you add an exit condition to avoid having an infinite loop.
4. After inserting data into the `TOP_SALARIES` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `EMPLOYEES` table.

```
SALARY
----------
24000
17000
17000
14000
13500
```

5. Test a variety of special cases such as `v_num = 0` or where `v_num` is greater than the number of employees in the `EMPLOYEES` table. Empty the `TOP_SALARIES` table after each test.

Solution 8-2: Using Explicit Cursors: Optional

If you have time, complete the following optional exercise. Here, create a PL/SQL block that uses an explicit cursor to determine the top *n* salaries of employees.

1. Execute the `lab_08-02.sql` script to create a new table, `TOP_SALARIES`, for storing the salaries of the employees.
2. In the declarative section, declare a variable `v_num` of type `NUMBER` that holds a number *n*, representing the number of top *n* earners from the `EMPLOYEES` table. For example, to view the top five salaries, enter `5`. Declare another variable `sal` of type `employees.salary`. Declare a cursor, `c_emp_cursor`, which retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.

```
DECLARE
  v_num          NUMBER(3) := 5;
  v_sal          employees.salary%TYPE;
  CURSOR         c_emp_cursor IS
    SELECT       salary
    FROM         employees
    ORDER BY     salary DESC;
```

3. In the executable section, open the loop and fetch the top *n* salaries, and then insert them into the `TOP_SALARIES` table. You can use a simple loop to operate on the data. Also, try and use the `%ROWCOUNT` and `%FOUND` attributes for the exit condition.

   Note: Make sure that you add an exit condition to avoid having an infinite loop.

```
BEGIN
  OPEN c_emp_cursor;
  FETCH c_emp_cursor INTO  v_sal;
  WHILE c_emp_cursor%ROWCOUNT <= v_num AND c_emp_cursor%FOUND LOOP
            INSERT INTO top_salaries (salary)
            VALUES (v_sal);
    FETCH c_emp_cursor INTO v_sal;
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

4. After inserting data into the `TOP_SALARIES` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `EMPLOYEES` table.

```
/
SELECT * FROM top_salaries;
```

The sample output is as follows:

```
SALARY
----------------------
24000
17000
17000
14000
13500
```

5. Test a variety of special cases such as `v_num = 0` or where `v_num` is greater than the number of employees in the `EMPLOYEES` table. Empty the `TOP_SALARIES` table after each test.

Practices for Lesson 9: Handling Exceptions

Chapter 9

Practice 9-1: Handling Predefined Exceptions

In this practice, you write a PL/SQL block that applies a predefined exception to process only one record at a time. The PL/SQL block selects the name of the employee with a given salary value.

1. Execute the command in the `lab_06_01.sql` file to re-create the `messages` table.
2. In the declarative section, declare two variables: `v_ename` of type `employees.last_name` and `v_emp_sal` of type `employees.salary`. Initialize the latter to 6000.
3. In the executable section, retrieve the last names of employees whose salaries are equal to the value in `v_emp_sal`. If the salary entered returns only one row, insert into the `MESSAGES` table the employee's name and the salary amount.
   Note: Do not use explicit cursors.
4. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `MESSAGES` table the message "No employee with a salary of <*salary*>."
5. If the salary entered returns multiple rows, handle the exception with an appropriate exception handler and insert into the `MESSAGES` table the message "More than one employee with a salary of <*salary*>."
6. Handle any other exception with an appropriate exception handler and insert into the `MESSAGES` table the message "Some other error occurred."
7. Display the rows from the `MESSAGES` table to check whether the PL/SQL block has executed successfully. The output is as follows:

```
RESULTS
--------------------------------------------------
More than one employee with a salary of 6000


1 rows selected
```

8. Change the initialized value of `v_emp_sal` to 2000 and re-execute. The output is as follows:

```
RESULTS
--------------------------------------------------
More than one employee with a salary of 6000
No employee with a salary of 2000
```

Solution 9-1: Handling Predefined Exceptions

In this practice, you write a PL/SQL block that applies a predefined exception to process only one record at a time. The PL/SQL block selects the name of the employee with a given salary value.

1. Execute the command in the `lab_06_01.sql` file to re-create the `MESSAGES` table.
2. In the declarative section, declare two variables: `v_ename` of type `employees.last_name` and `v_emp_sal` of type `employees.salary`. Initialize the latter to 6000.

```
DECLARE
  v_ename     employees.last_name%TYPE;
  v_emp_sal   employees.salary%TYPE := 6000;
```

3. In the executable section, retrieve the last names of employees whose salaries are equal to the value in `v_emp_sal`. If the salary entered returns only one row, insert the employee's name and the salary amount into the `MESSAGES` table.
   Note: Do not use explicit cursors.

```
BEGIN
  SELECT last_name
  INTO        v_ename
  FROM        employees
  WHERE       salary = v_emp_sal;
  INSERT INTO messages (results)
  VALUES (v_ename || ' - ' || v_emp_sal);
```

4. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert the message "No employee with a salary of <*salary*>" into the `MESSAGES` table.

```
EXCEPTION
  WHEN no_data_found THEN
    INSERT INTO messages (results)
    VALUES ('No employee with a salary of '||
            TO_CHAR(v_emp_sal));
```

5. If the salary entered returns multiple rows, handle the exception with an appropriate exception handler and insert the message "More than one employee with a salary of <*salary*>" into the `MESSAGES` table.

```
WHEN too_many_rows THEN
    INSERT INTO messages (results)
    VALUES ('More than one employee with a salary of '||
             TO_CHAR(v_emp_sal));
```

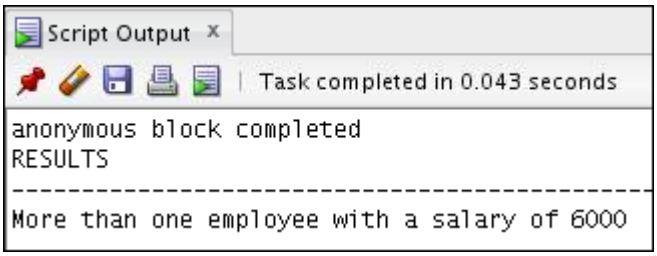
6. Handle any other exception with an appropriate exception handler and insert the message "Some other error occurred" into the MESSAGES table.

```
WHEN others THEN
    INSERT INTO messages (results)
    VALUES ('Some other error occurred.');
END;
```

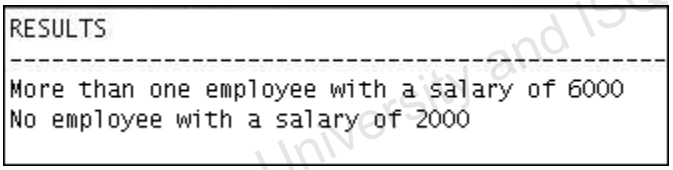
7. Display the rows from the MESSAGES table to check whether the PL/SQL block has executed successfully.

```
/
SELECT * FROM messages;
```

The output is as follows:



8. Change the initialized value of v_emp_sal to 2000 and re-execute. The output is as follows:



Practice 9-2: Handling Standard Oracle Server Exceptions

In this practice, you write a PL/SQL block that declares an exception for the Oracle Server error `ORA-02292 (integrity constraint violated - child record found)`. The block tests for the exception and outputs the error message.

1. In the declarative section, declare an exception `e_childrecord_exists`. Associate the declared exception with the standard Oracle Server error `-02292`.
2. In the executable section, display "Deleting department 40...." Include a `DELETE` statement to delete the department with the `department_id` 40.
3. Include an exception section to handle the `e_childrecord_exists` exception and display the appropriate message.

The sample output is as follows:

```
anonymous block completed
 Deleting department 40........
 Cannot delete this department. There are employees in this department (child records exist.)
```

Solution 9-2: Handling Standard Oracle Server Exceptions

In this practice, you write a PL/SQL block that declares an exception for the Oracle Server error `ORA-02292 (integrity constraint violated - child record found)`. The block tests for the exception and outputs the error message.

1. In the declarative section, declare an exception `e_childrecord_exists`. Associate the declared exception with the standard Oracle Server error `-02292`.

```
SET SERVEROUTPUT ON
DECLARE
   e_childrecord_exists EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_childrecord_exists, -02292);
```

2. In the executable section, display "Deleting department 40...." Include a `DELETE` statement to delete the department with `department_id` 40.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(' Deleting department 40........');
   delete from departments where department_id=40;
```

3. Include an exception section to handle the `e_childrecord_exists` exception and display the appropriate message.

```
EXCEPTION
   WHEN e_childrecord_exists THEN
   DBMS_OUTPUT.PUT_LINE(' Cannot delete this department. There are
employees in this department (child records exist.) ');
END;
```

The sample output is as follows:

```
anonymous block completed
 Deleting department 40........
 Cannot delete this department. There are employees in this department (child records exist.)
```

Practices for Lesson 10: Introducing Stored Procedures and Functions

Chapter 10

Practice 10: Creating and Using Stored Procedures

Note: If you have executed the code examples for this lesson, make sure you execute the following code before starting this practice:

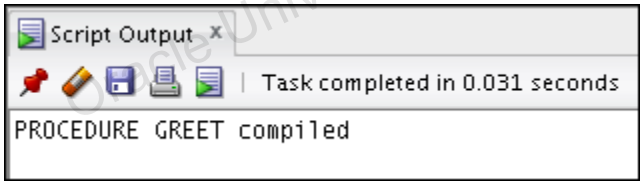
```
DROP table dept;
DROP procedure add_dept;
DROP function check_sal;
```

In this practice, you modify existing scripts to create and use stored procedures.

1. Open `sol_03.sql` script from the `/home/oracle/labs/plsf/soln/` folder. Copy the code under task 4 into a new worksheet.

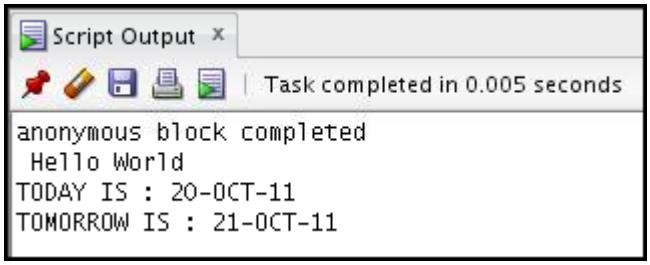
```
SET SERVEROUTPUT ON

DECLARE
   v_today DATE:=SYSDATE;
   v_tomorrow v_today%TYPE;
BEGIN
   v_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello World ');
   DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| v_today);
   DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);
END;
```

   a. Modify the script to convert the anonymous block to a procedure called `greet`. (**Hint:** Also remove the `SET SERVEROUTPUT ON` command.)
   b. Execute the script to create the procedure. The output results should be as follows:



   c. Save this script as `lab_10_01_soln.sql`.
   d. Click the Clear button to clear the workspace.
   e. Create and execute an anonymous block to invoke the `greet` procedure. (**Hint:** Ensure that you enable `SERVEROUTPUT` at the beginning of the block.)

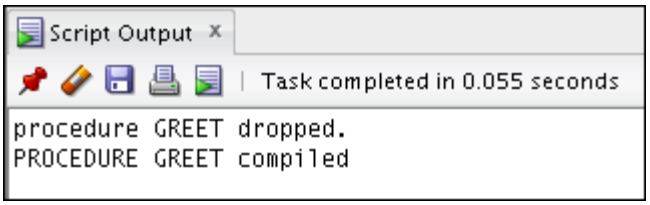
The output should be similar to the following:



2. Modify the `lab_10_01_soln.sql` script as follows:
   a. Drop the `greet` procedure by issuing the following command:

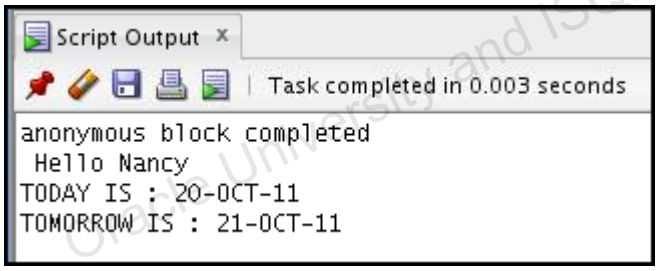
```
DROP PROCEDURE greet;
```

   b. Modify the procedure to accept an argument of type `VARCHAR2`. Call the argument `p_name`.
   c. Print `Hello <name>` (that is, the contents of the argument) instead of printing `Hello World`.
   d. Save your script as `lab_10_02_soln.sql`.
   e. Execute the script to create the procedure. The output results should be as follows:



   f. Create and execute an anonymous block to invoke the `greet` procedure with a parameter value. The block should also produce the output.

The sample output should be similar to the following:



Solution 10: Creating and Using Stored Procedures

In this practice, you modify existing scripts to create and use stored procedures.

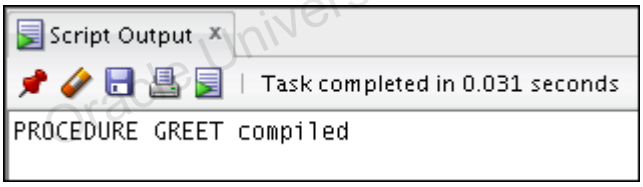
1. Open the `sol_03.sql` script from the `/home/oracle/labs/plsf/soln/` folder. Copy the code under task 4 into a new worksheet.

```
SET SERVEROUTPUT ON

DECLARE
   v_today DATE:=SYSDATE;
   v_tomorrow v_today%TYPE;
BEGIN
   v_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello World ');
   DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| v_today);
   DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);
END;
```

   a. Modify the script to convert the anonymous block to a procedure called `greet`. (**Hint:** Also remove the `SET SERVEROUTPUT ON` command.)

```
CREATE PROCEDURE greet IS
   V_today DATE:=SYSDATE;
   V_tomorrow today%TYPE;
...
```

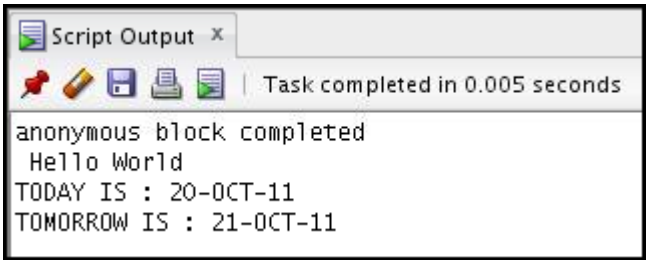
   b. Execute the script to create the procedure. The output results should be as follows:



   c. Save this script as `lab_10_01_soln.sql`.
   d. Click the Clear button to clear the workspace.
   e. Create and execute an anonymous block to invoke the `greet` procedure. (**Hint:** Ensure that you enable `SERVEROUTPUT` at the beginning of the block.)

```
SET SERVEROUTPUT ON

BEGIN
 greet;
END;
```

The output should be similar to the following:



2. Modify the `lab_10_01_soln.sql` script as follows:
   a. Drop the `greet` procedure by issuing the following command:

```
DROP PROCEDURE greet;
```

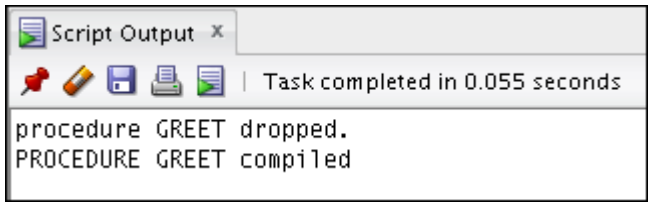
   b. Modify the procedure to accept an argument of type `VARCHAR2`. Call the argument `p_name`.

```
CREATE PROCEDURE greet(p_name VARCHAR2) IS
   V_today DATE:=SYSDATE;
   V_tomorrow today%TYPE;
```

   c. Print `Hello` *<name>* instead of printing `Hello World`.

```
BEGIN
   V_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello '|| p_name);
...
```

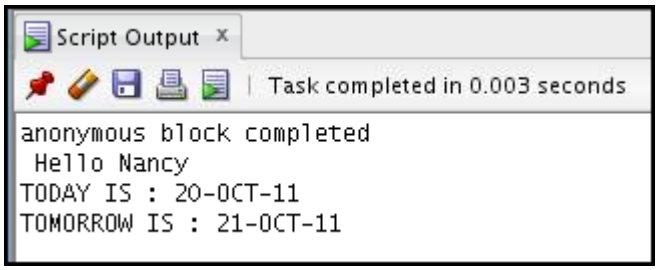
   d. Save your script as `lab_10_02_soln.sql`.
   e. Execute the script to create the procedure. The output results should be as follows:



   f. Create and execute an anonymous block to invoke the `greet` procedure with a parameter value. The block should also produce the output.

```
SET SERVEROUTPUT ON;
BEGIN
 greet('Nancy');
END;
```

The sample output should be similar to the following:



Practices for Lesson 11: Creating Procedures

Chapter 11

Practices for Lesson 11: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 11-1: 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
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
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
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
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.
   a. Create a procedure called `DEL_JOB` to delete a job. Include the necessary exception-handling code if no job is deleted.
   b. Invoke the procedure using the job ID `IT_DBA`. Query the `JOBS` table and view the results.

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

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

```
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.
   a. Create a procedure that returns a value from the `SALARY` and `JOB_ID` columns for a specified employee ID. Remove syntax errors, if any, and then recompile the code.
   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.

```
v_salary
----
8000

v_job
------
ST_MAN
```

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

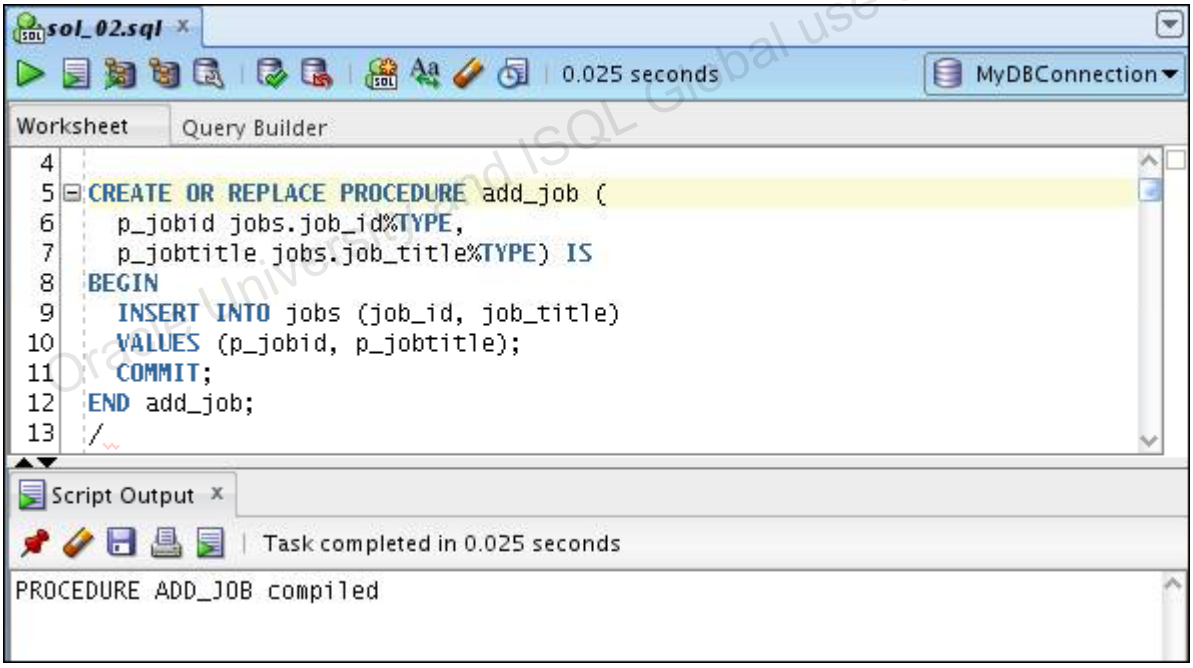
Solution 11-1: Creating, Compiling, and Calling Procedures

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.

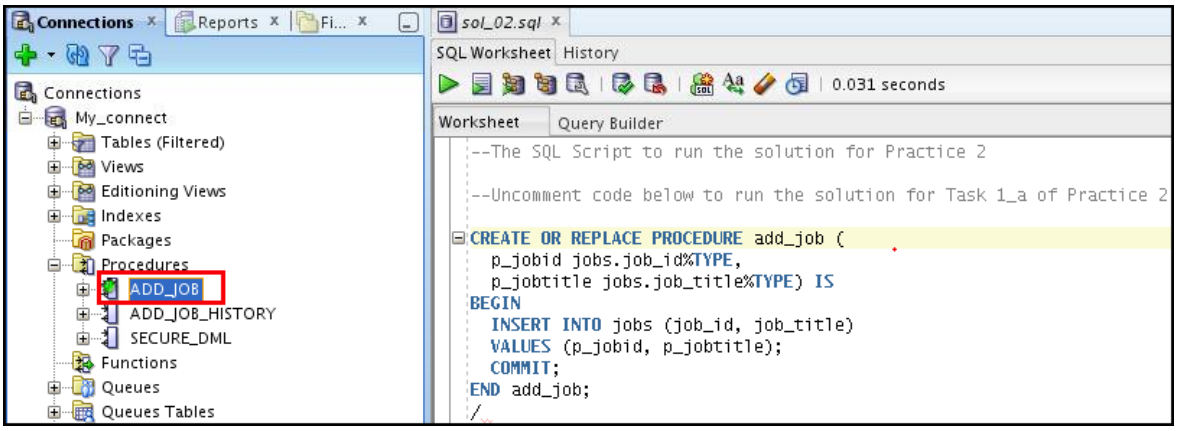
1. Create, compile, and invoke the ADD_JOB procedure and review the results.
   a. Create a procedure called ADD_JOB to insert a new job into the JOBS table. Provide the ID and job title using two parameters.

      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:



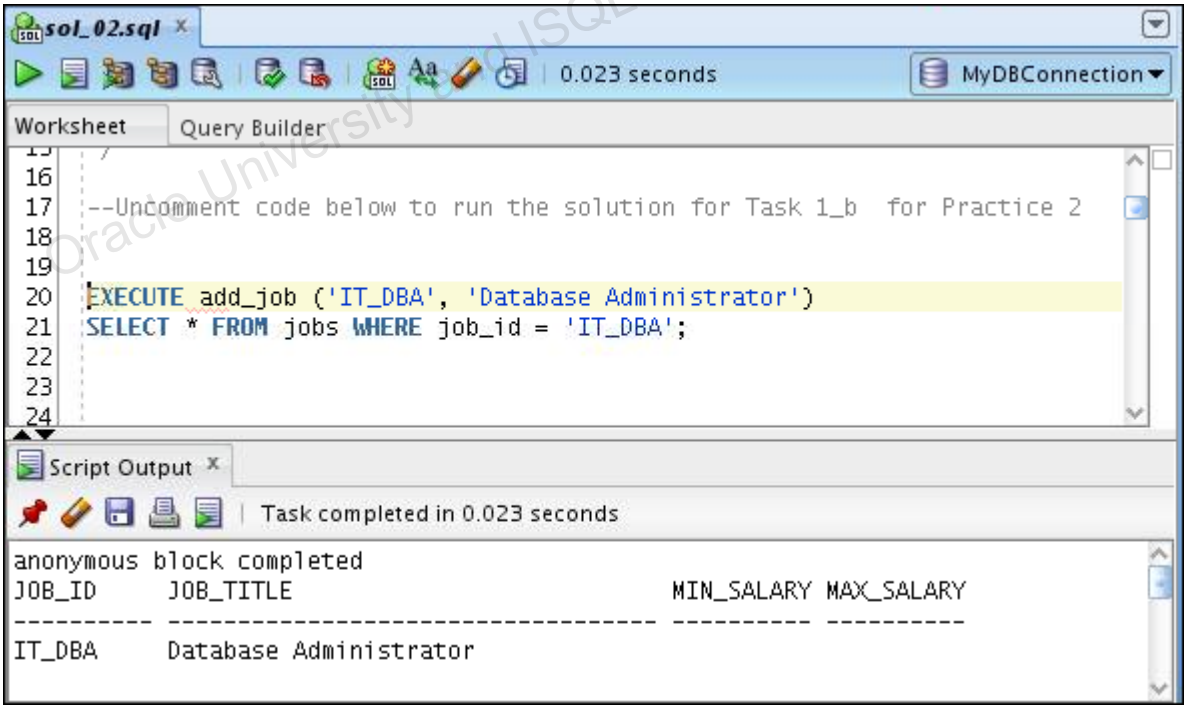
To view the newly created procedure, click the Procedures node in the Object Navigator. If the newly created procedure is not displayed, right-click the Procedures node, and then select Refresh from the shortcut menu. The new procedure is displayed as follows:



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:

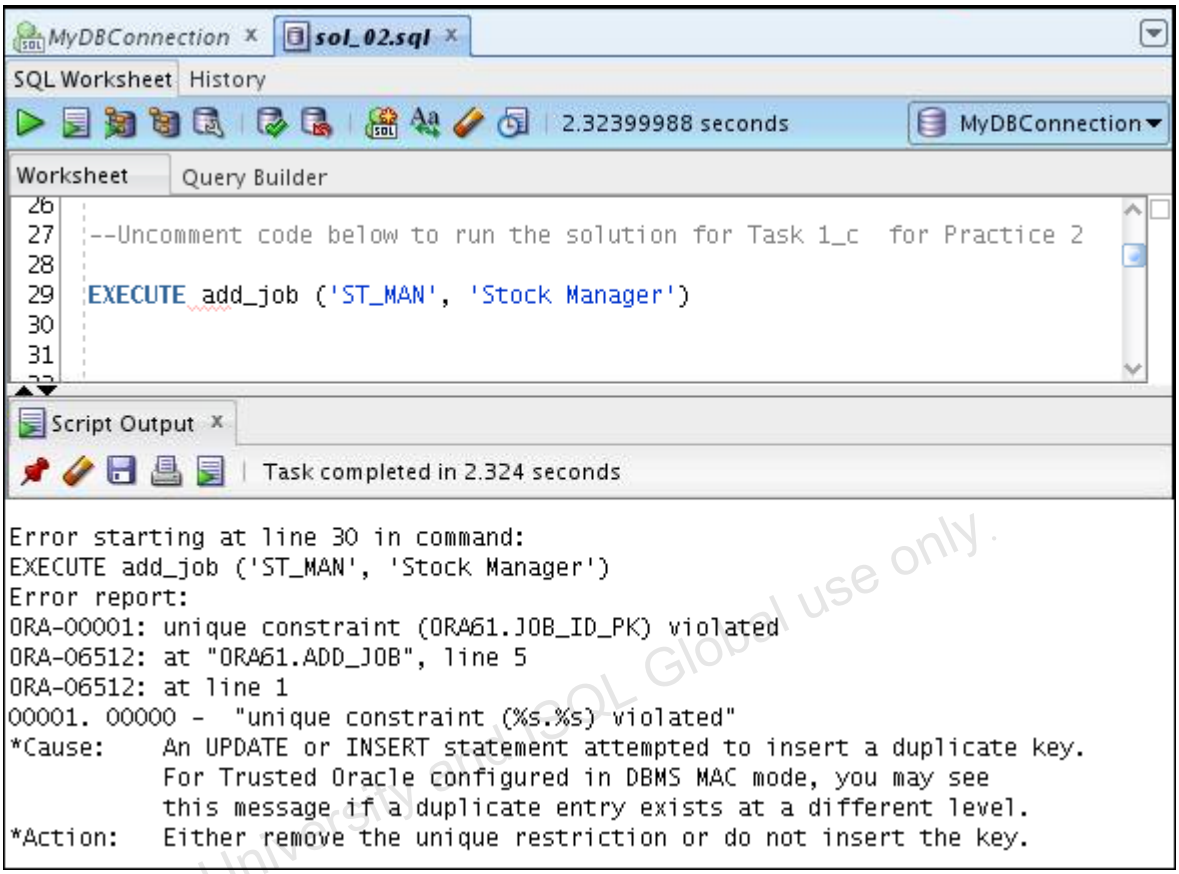
Note: Be sure to comment the previous code before uncommenting the next set of code.



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

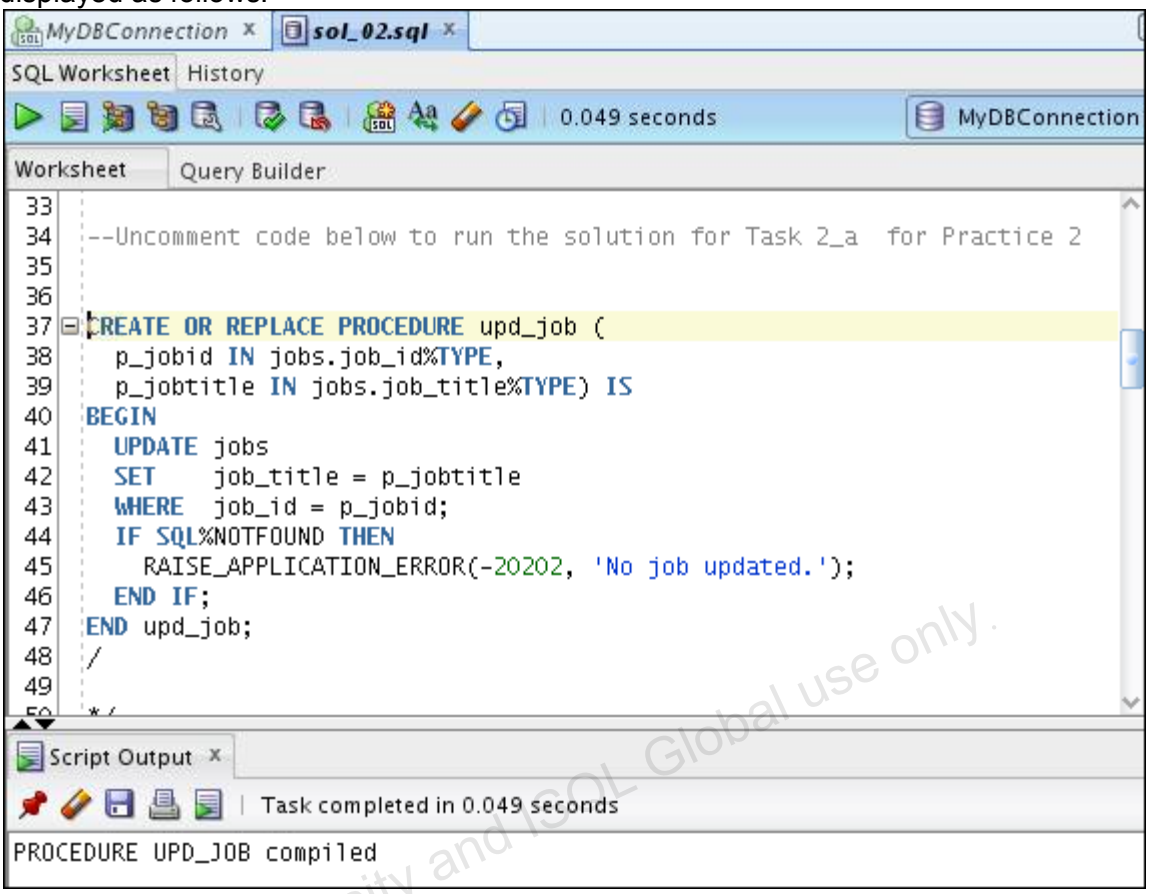
Run the code for Task 1_c from `sol_02.sql` script. The code and the result are displayed as follows:

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



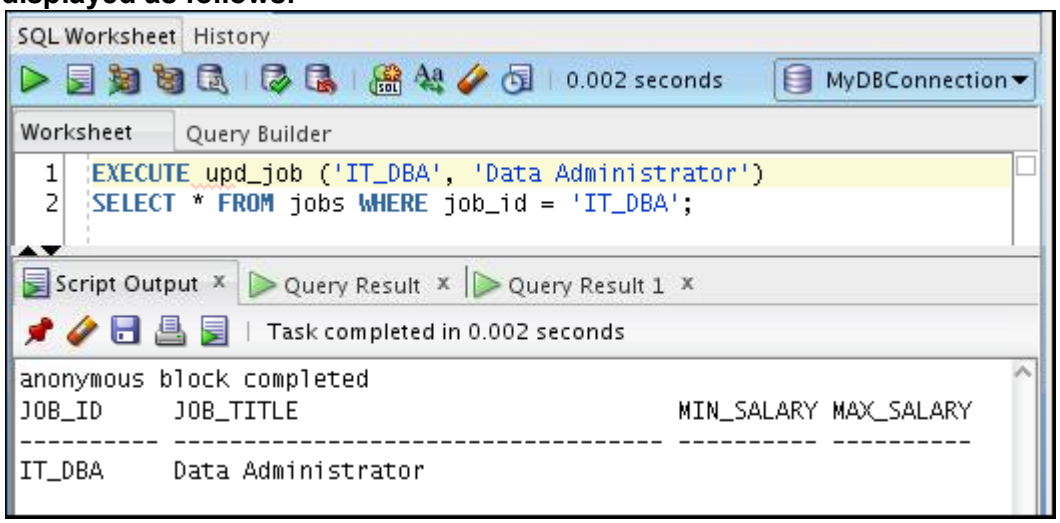
2. Create a procedure called `UPD_JOB` to modify a job in the `JOBS` table.
   a. Create a procedure called `UPD_JOB` to update the job title. Provide the job ID and a new title by using two parameters. Include the necessary exception handling if no update occurs.

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



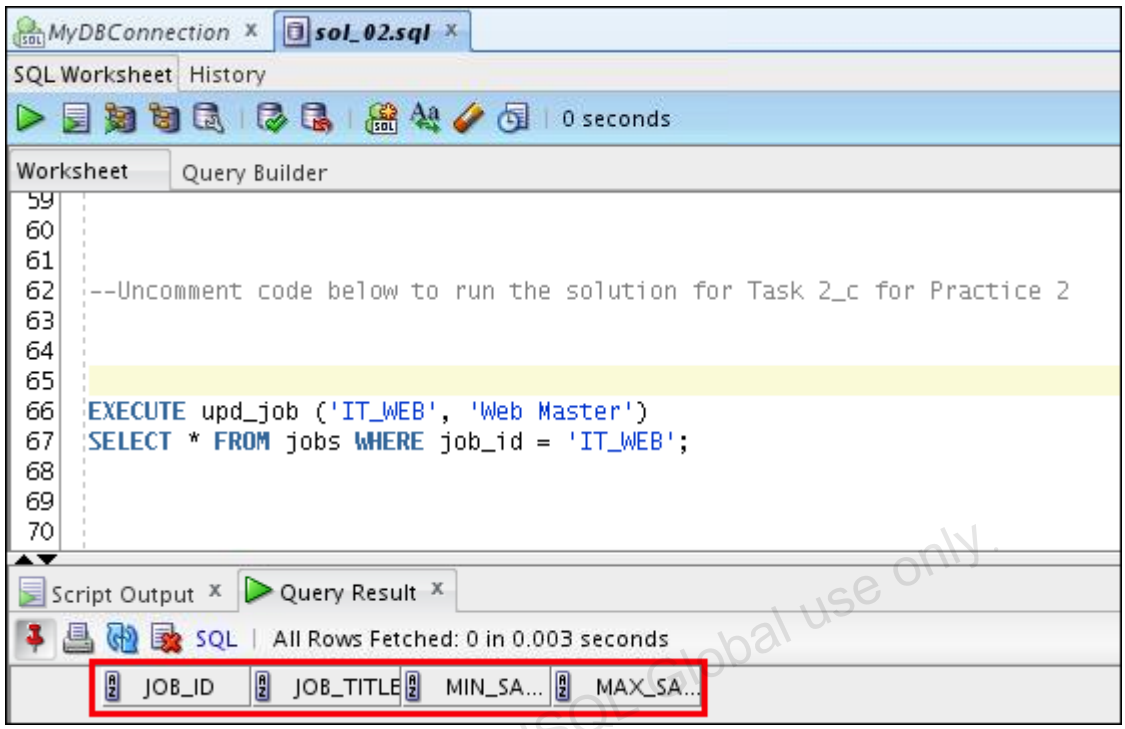
b. Invoke the procedure to change the job title of the job ID `IT_DBA` to `Data Administrator`. Query the `JOBS` table and view the results.

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



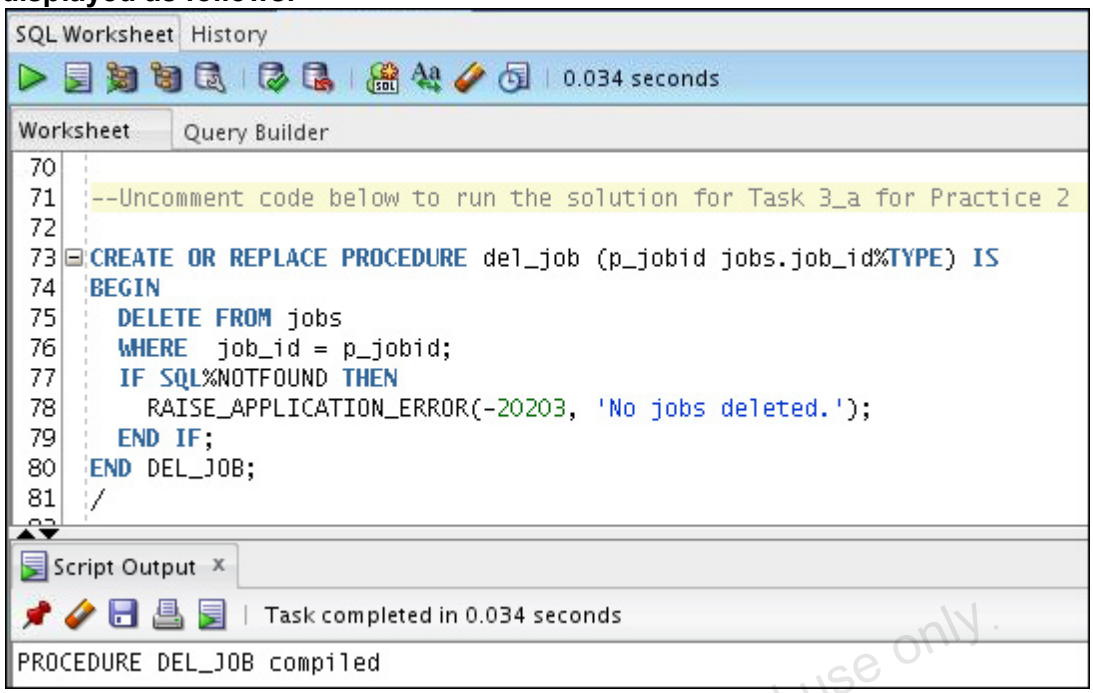
c. Test the exception-handling section of the procedure by trying to update a job that does not exist. You can use the job ID `IT_WEB` and the job title `Web Master`.

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

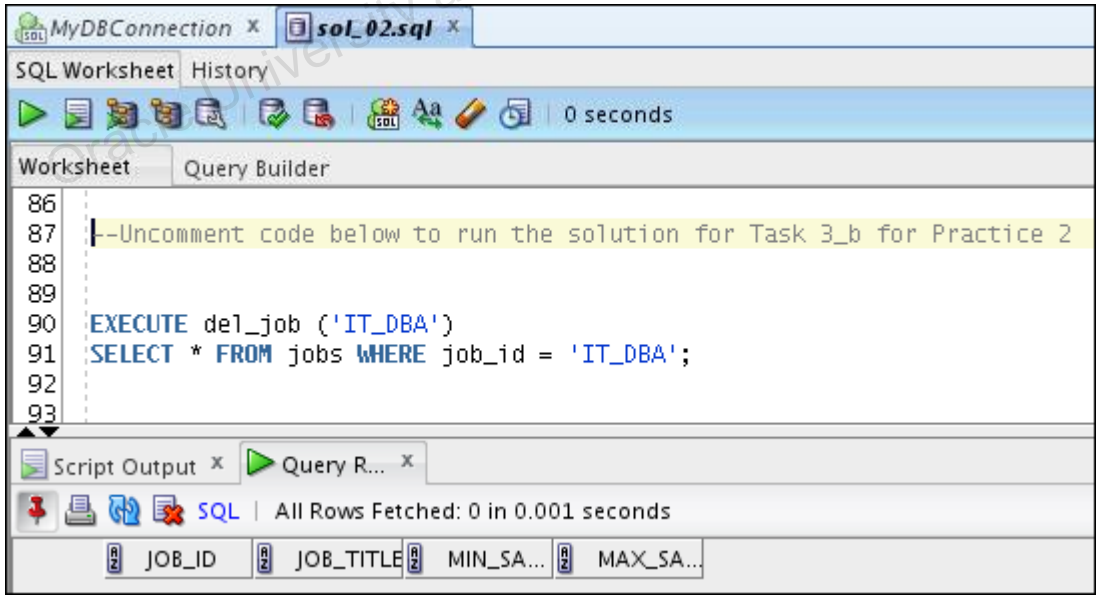


3. Create a procedure called `DEL_JOB` to delete a job from the `JOBS` table.
   a. Create a procedure called `DEL_JOB` to delete a job. Include the necessary exception-handling code if no job is deleted.

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



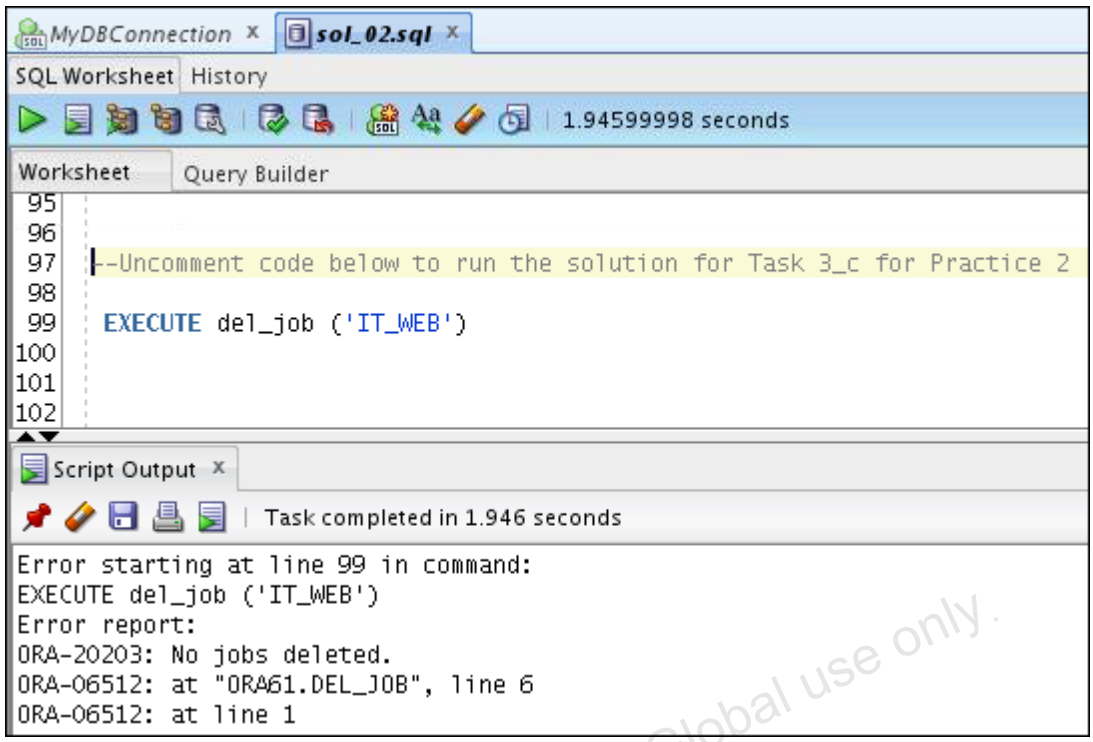
b. **To invoke the procedure and then query the JOBS table, uncomment and select the code under task 3_b in the `/home/oracle/labs/plpu/solns/sol_02.sql` script. Click the Run Script icon (or press F9) icon on the SQL Worksheet toolbar to invoke the procedure. Click the Query Result tab to see the code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



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.

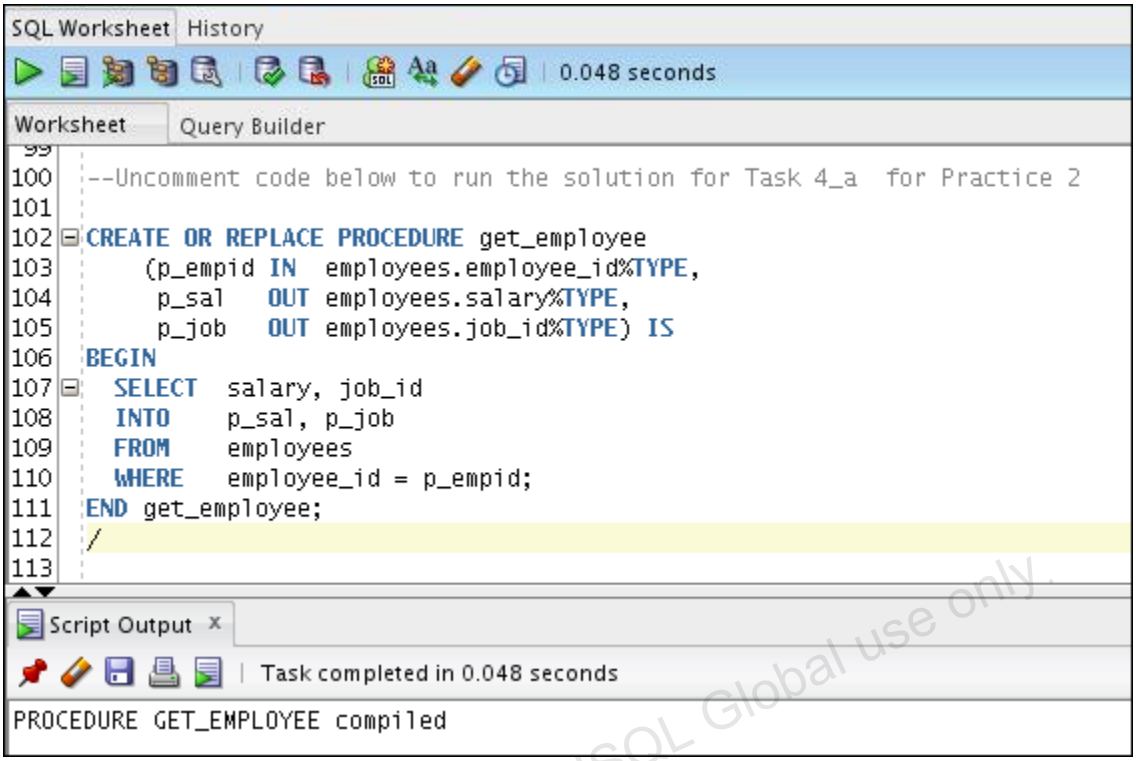
To invoke the procedure and then query the JOBS table, uncomment and select the code under task 3_c in the `/home/oracle/labs/plpu/solns/sol_02.sql`

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:

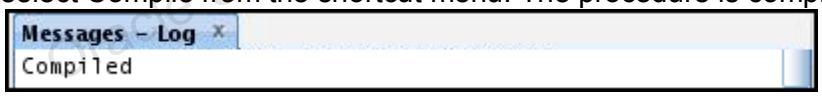


4. Create a procedure called `GET_EMPLOYEE` to query the `EMPLOYEES` table, retrieving the salary and job ID for an employee when provided with the employee ID.
   a. Create a procedure that returns a value from the `SALARY` and `JOB_ID` columns for a specified employee ID. Remove syntax errors, if any,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

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:

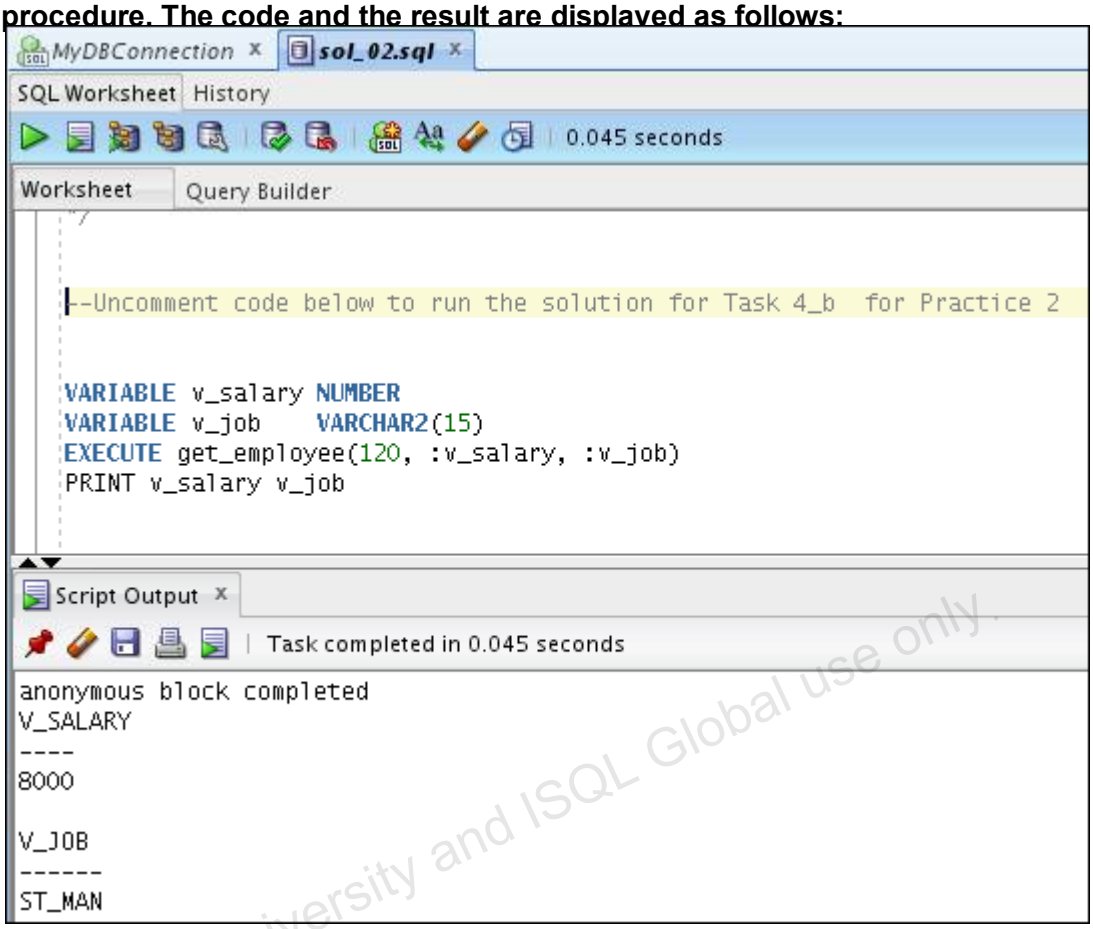


Note: If the newly created procedure is not displayed in the Object Navigator, right-click the Procedures node in the Object Navigator, and then select Refresh from the shortcut menu. Right-click the procedure's name in the Object Navigator, and then select Compile from the shortcut menu. The procedure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



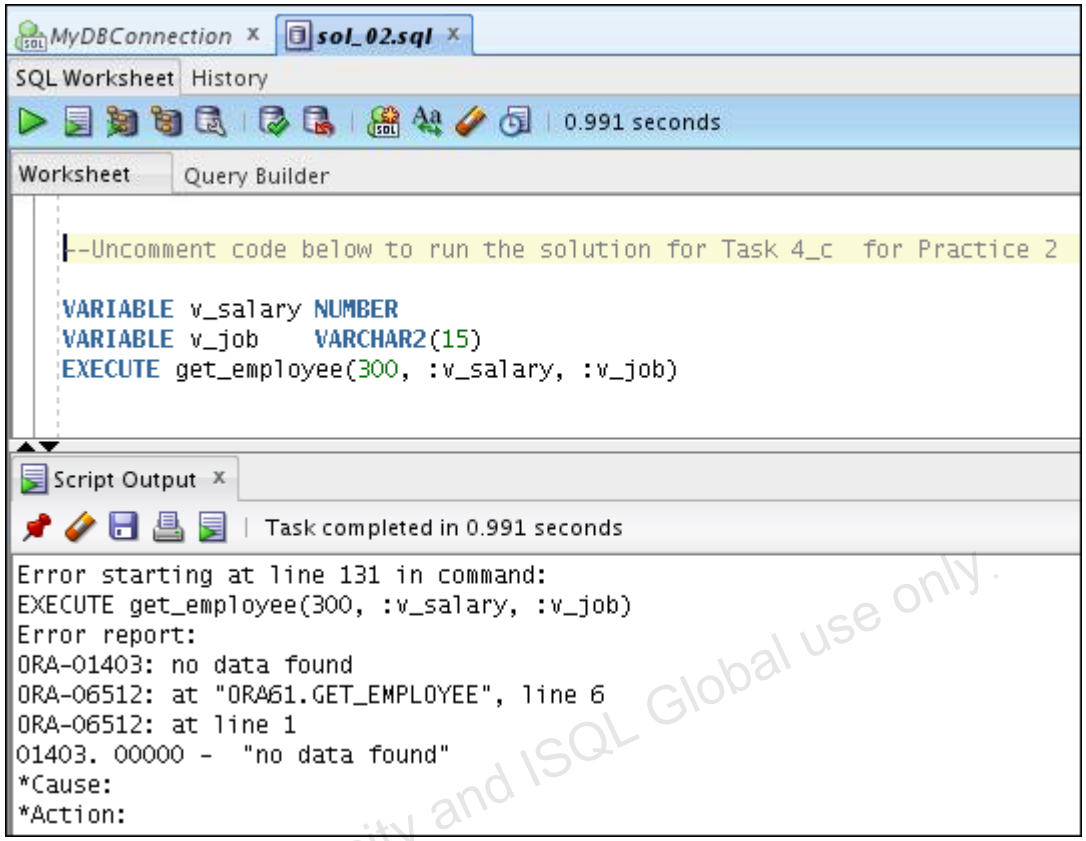
b. Execute the procedure using host variables for the two `OUT` parameters—one for the salary and the other for the job ID. Display the salary and job ID for employee ID 120.

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:



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:



Practices for Lesson 12: Creating Functions

Chapter 12

Practices for Lesson 12: Overview

Overview

In practice 12-1, 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

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 12-1: Creating Functions

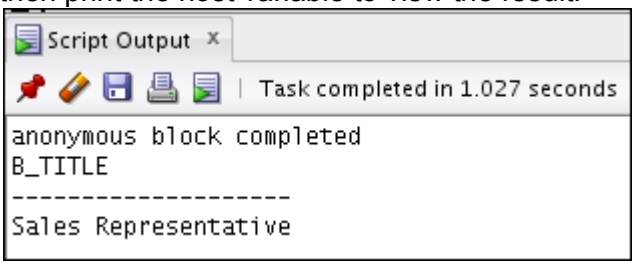
Overview

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

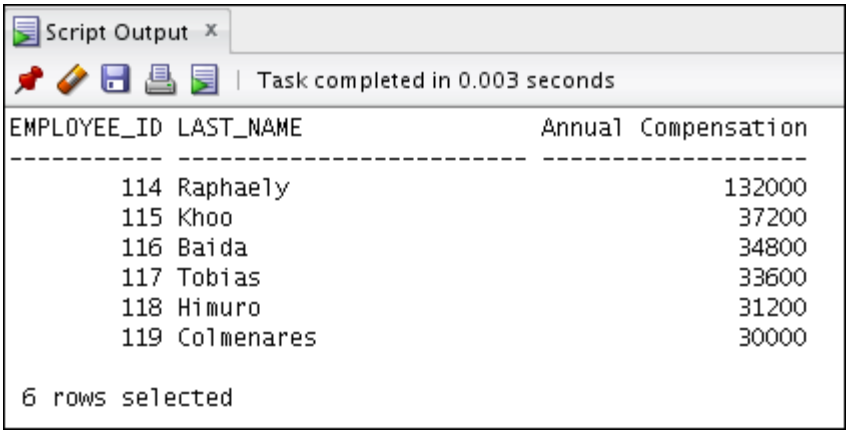
1. Create and invoke the `GET_JOB` function to return a job title.
   a. Create and compile a function called `GET_JOB` to return a job title.
   b. Create a `VARCHAR2` host variable called `b_title`, allowing a length of 35 characters. Invoke the function with job ID `SA_REP` to return the value in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



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)
```

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



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?

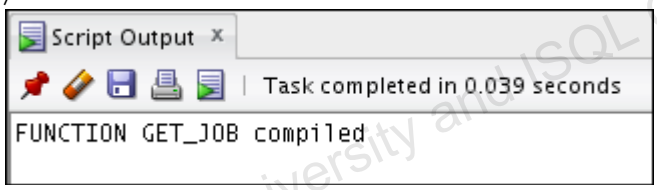
Solution 12-1: Creating Functions

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

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

      Open the `/home/oracle/labs/plpu/solns/sol_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:

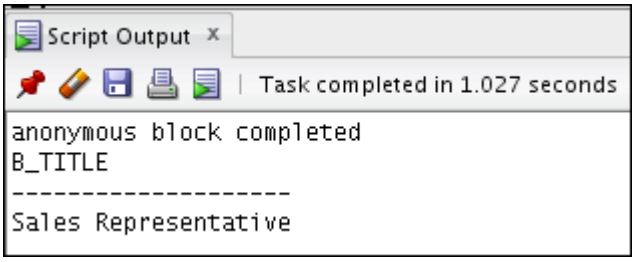
- Open SQL*Plus.
- Connect as SYSDBA.
- Execute the following code:

```
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
```



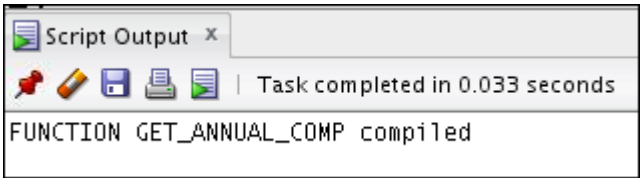
Note: Be sure to add the comments back to the previous code before executing the next set of code. Alternatively, you can select the complete code before using the Run Script icon (or press F5) to execute it.

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:

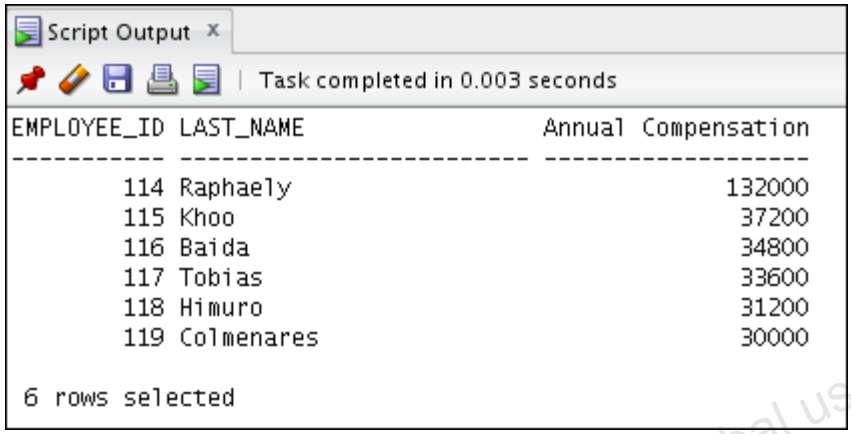
```
CREATE OR REPLACE FUNCTION get_annual_comp(
  p_sal  IN employees.salary%TYPE,
  p_comm IN employees.commission_pct%TYPE)
 RETURN NUMBER IS
BEGIN
  RETURN (NVL(p_sal,0) * 12 + (NVL(p_comm,0) * nvl(p_sal,0) *
12));
END get_annual_comp;
/
```



   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
/
```

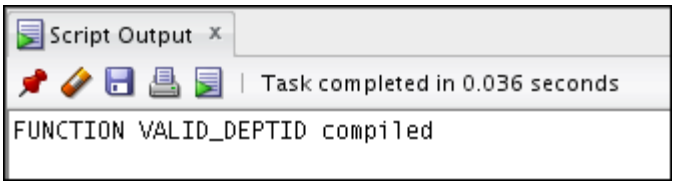


3. Create a procedure, ADD_EMPLOYEE, to insert a new employee into the EMPLOYEES table. The procedure should call a VALID_DEPTID function to check whether the department ID specified for the new employee exists in the DEPARTMENTS table.
   a. Create a function called VALID_DEPTID to validate a specified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

```
/
```

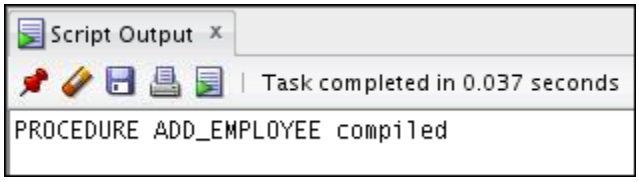


b. Create the `ADD_EMPLOYEE` procedure to add an employee to the `EMPLOYEES` table. The row should be added to the `EMPLOYEES` table if the `VALID_DEPTID` function returns `TRUE`; otherwise, alert the user with an appropriate message. Provide the following parameters:
   - `first_name`
   - `last_name`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

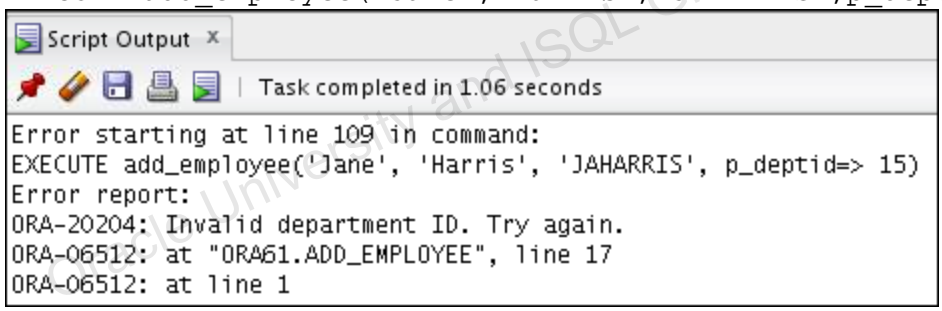
```
   VALUES (employees_seq.NEXTVAL, p_first_name, p_last_name,
p_email,
     p_job, p_mgr, TRUNC(SYSDATE), p_sal, p_comm, p_deptid);
 ELSE
   RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try
again.');
 END IF;
END add_employee;
/
```



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:

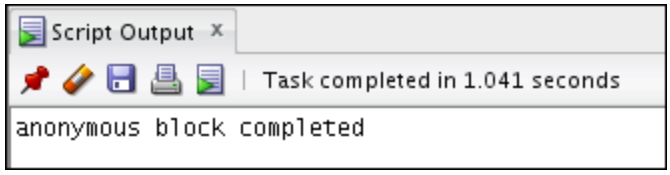
```
EXECUTE add_employee('Jane','Harris','JAHARRIS',p_deptid=> 15)
```



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)
```



Practices for Lesson 13: Debugging Subprograms

Chapter 13

Practices for Lesson 13: Overview

Overview

In practice 13-1 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. 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.
2. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at /`home/oracle/labs/plpu/`

Practice 13-1: 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 13-1: 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.

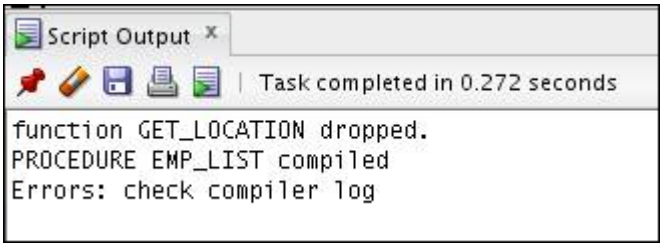
```
SET SERVEROUTPUT ON
```

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:

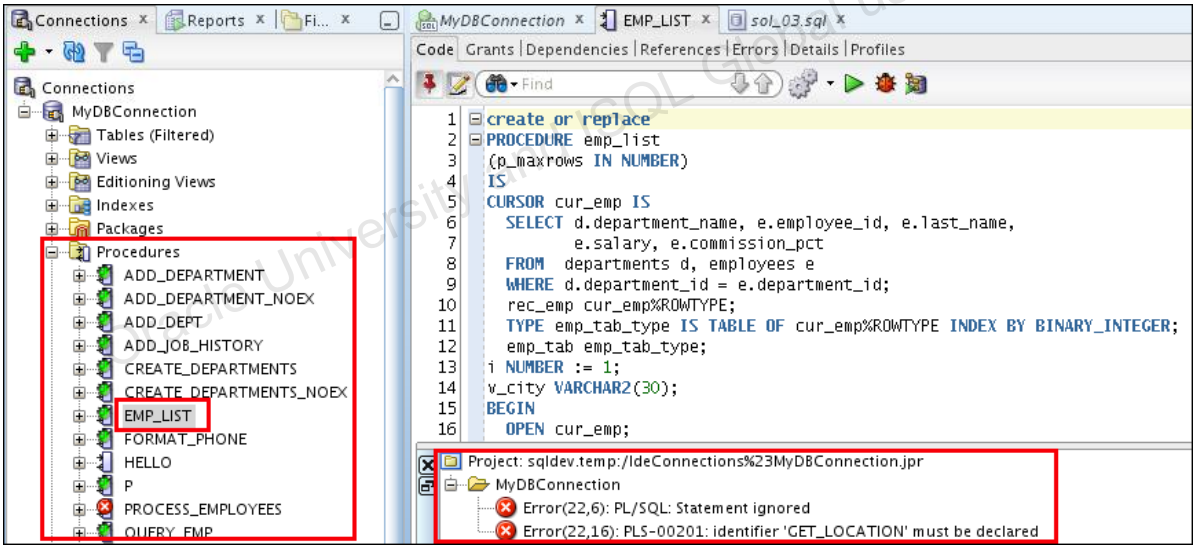
```
DROP FUNCTION get_location;

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:

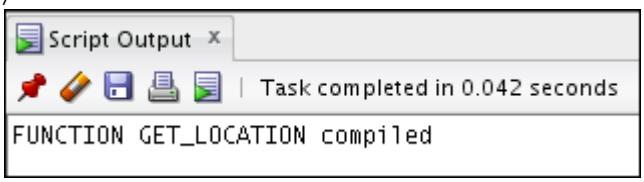


3. Run the solution under Task 3 of practice 3-2 to create the `get_location` function. Examine the code of the function, compile the function, and then correct any errors, if any.

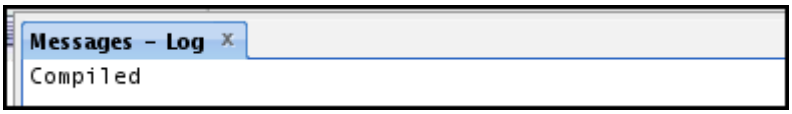
   Uncomment and select the code under Task 3 of Practice 3-2. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. The codex and the result are displayed as follows:

```
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;
/
```



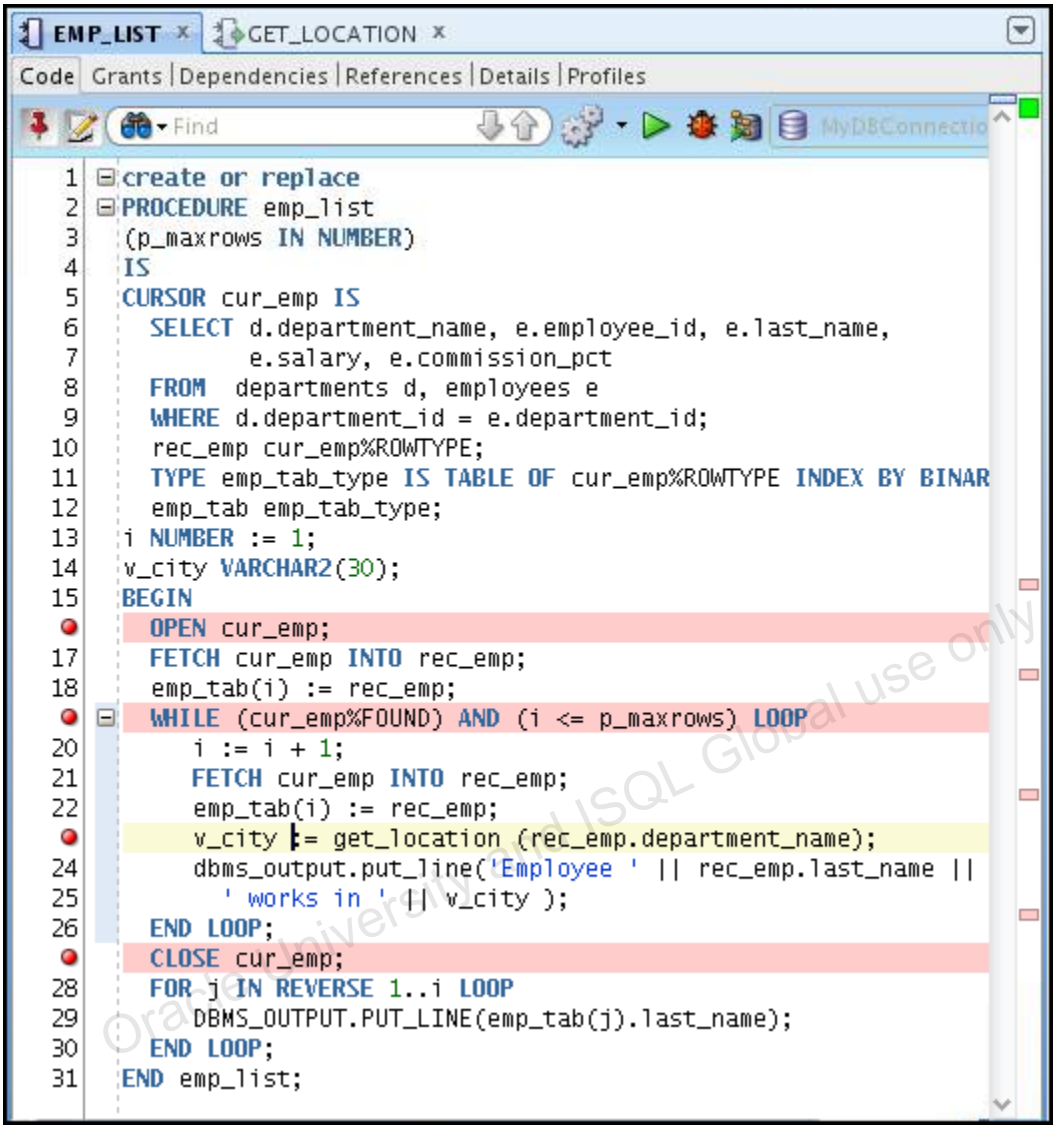
4. Recompile the `emp_list` procedure. The procedure should compile successfully.
   To recompile the procedure, right-click the procedure's name, and then select Compile from the shortcut menu.



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.

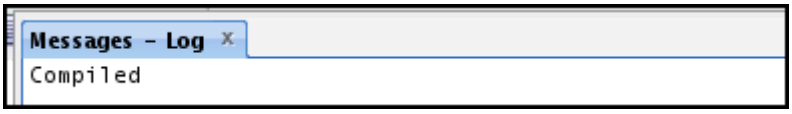
6. Add four breakpoints to the `emp_list` procedure to the following lines of code:
   a. `OPEN cur_emp;`
   b. `WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP`
   c. `v_city := get_location (rec_emp.department_name);`
   d. `CLOSE cur_emp;`

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



7. Compile the `emp_list` procedure for debugging.

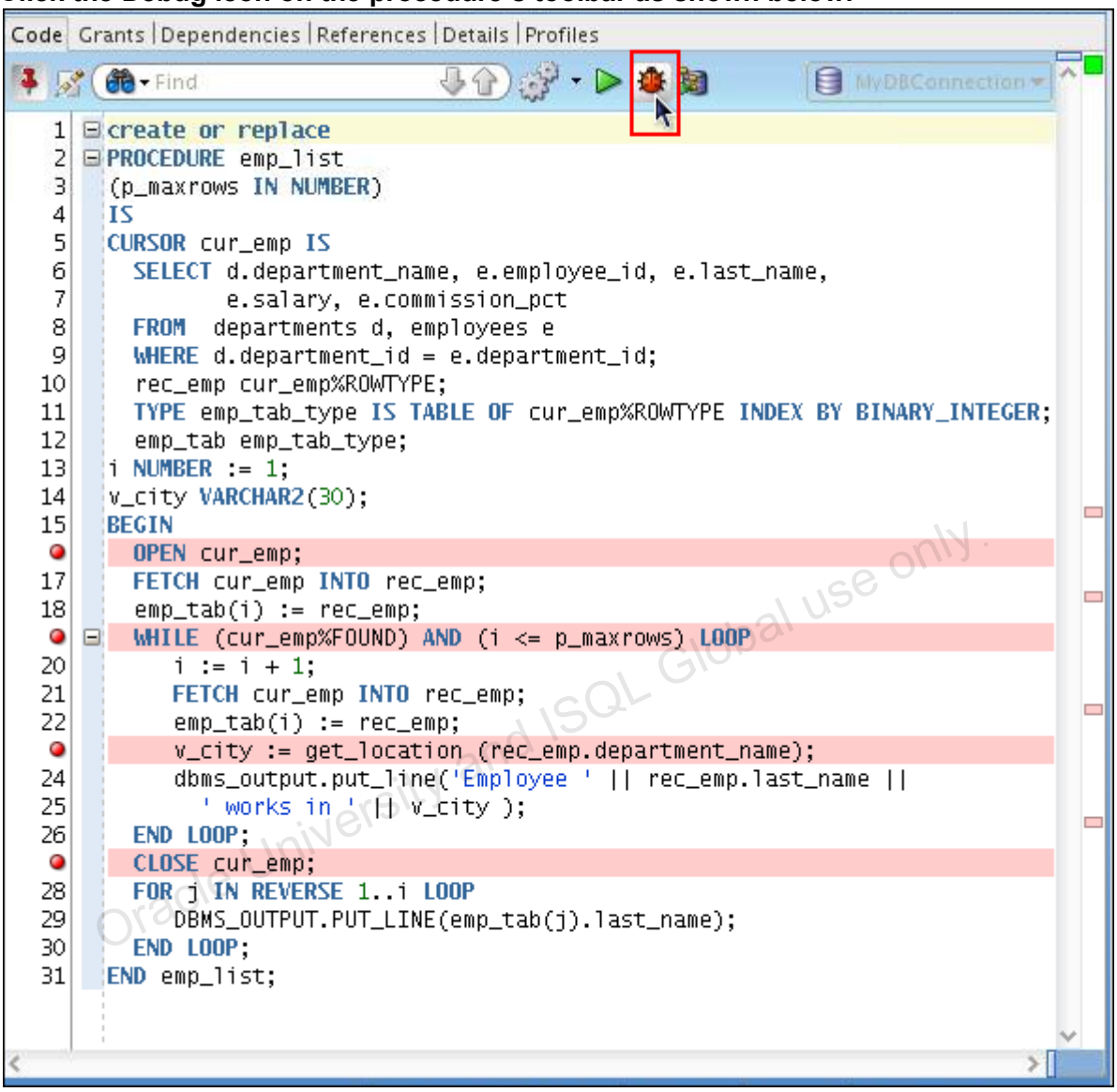
Click the "Compile for Debug" icon on the procedure's toolbar and you get the output as shown below:



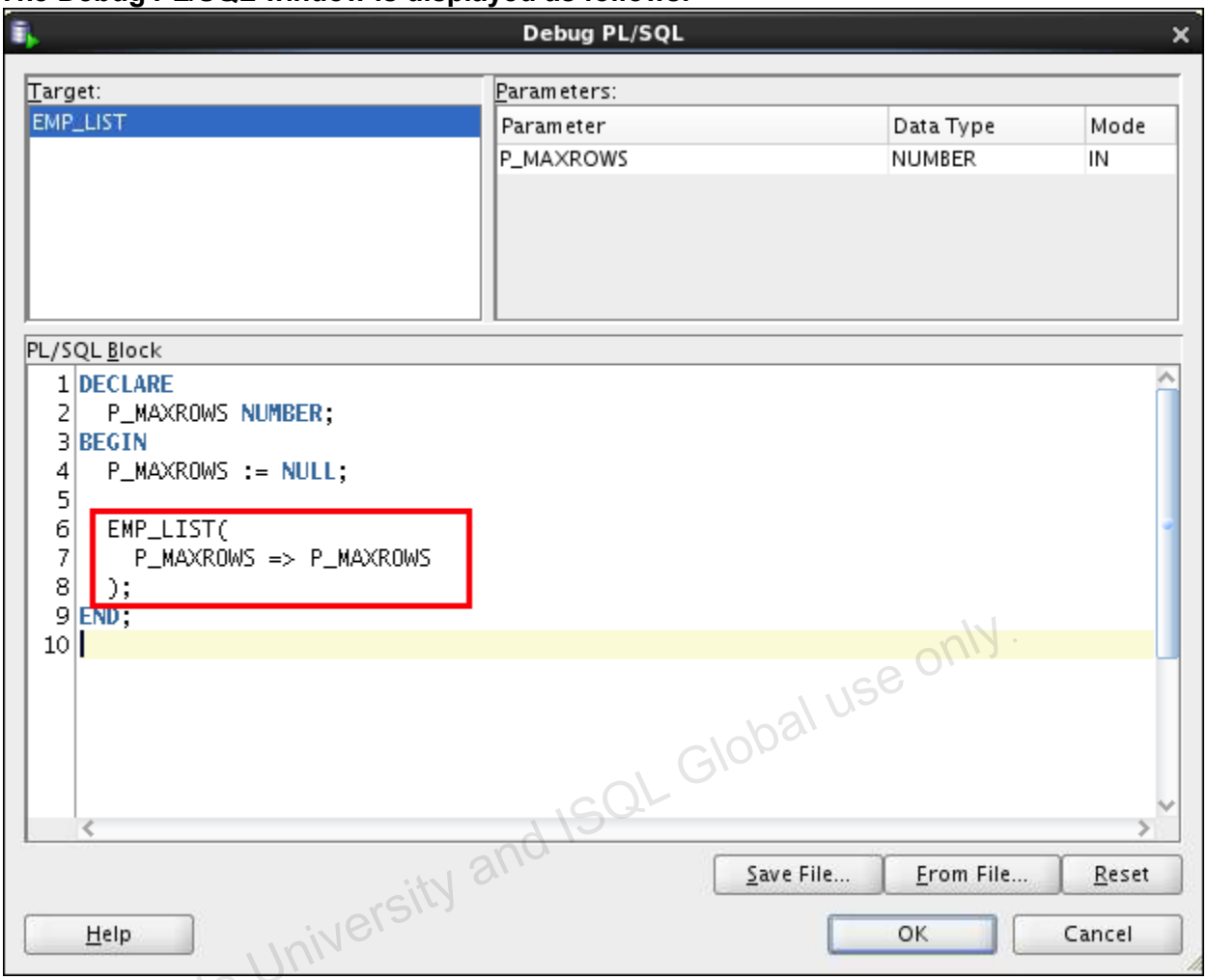
Note: If you get any warnings, it is expected. The two warnings are because the `PLSQL_DEBUG` parameter was deprecated in Oracle Database 11*g*, while SQL Developer is still using that parameter.

8. Debug the procedure.

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

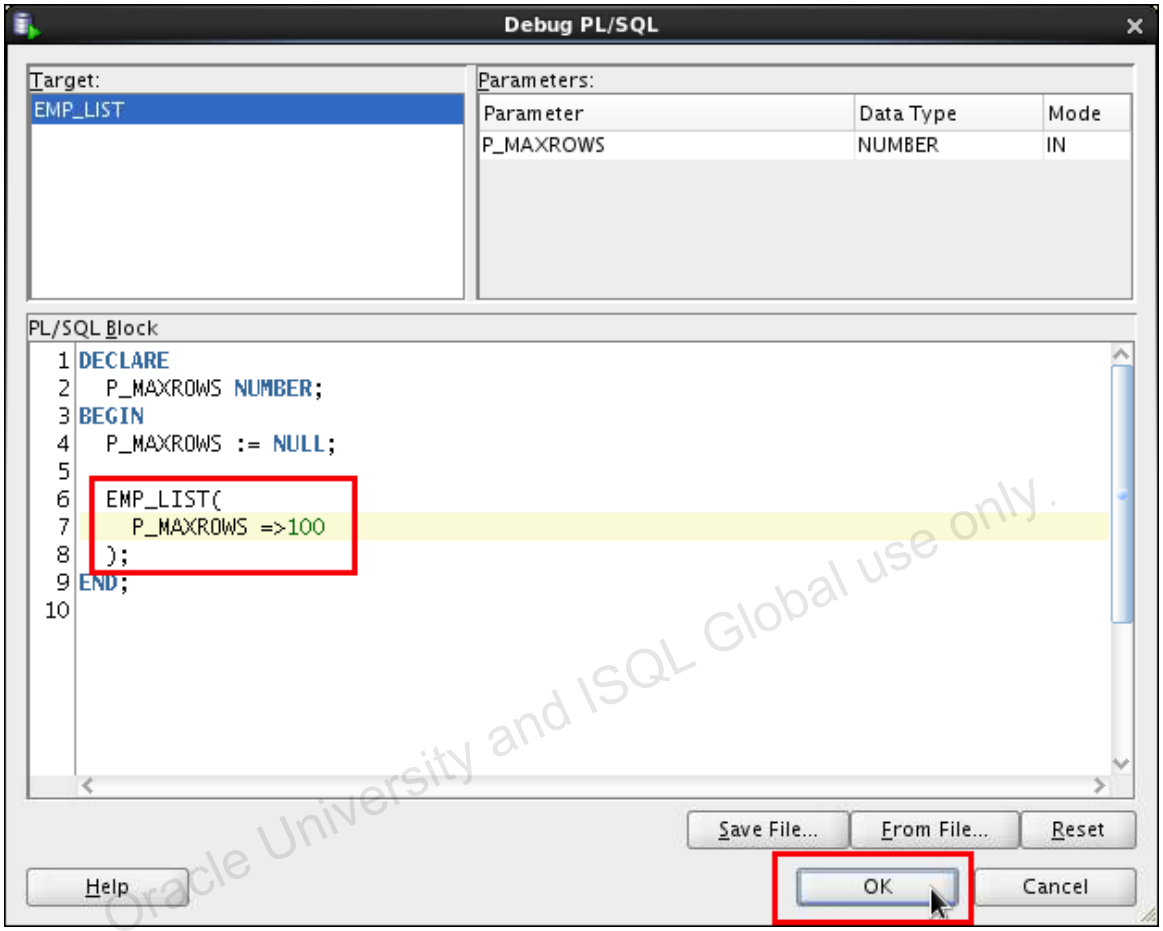


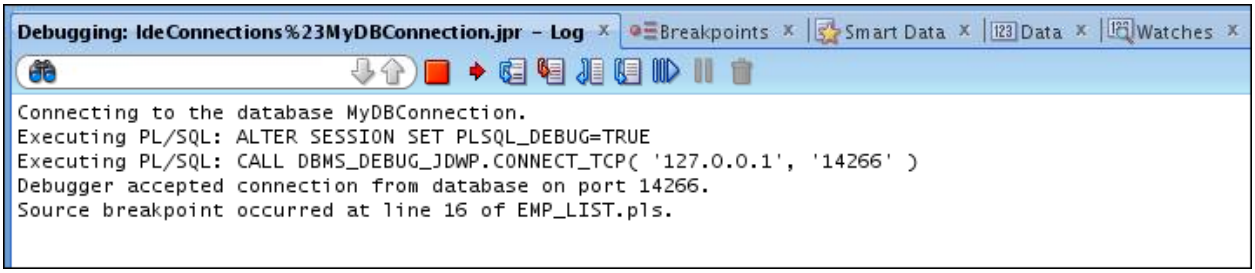
The Debug PL/SQL window is displayed as follows:



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

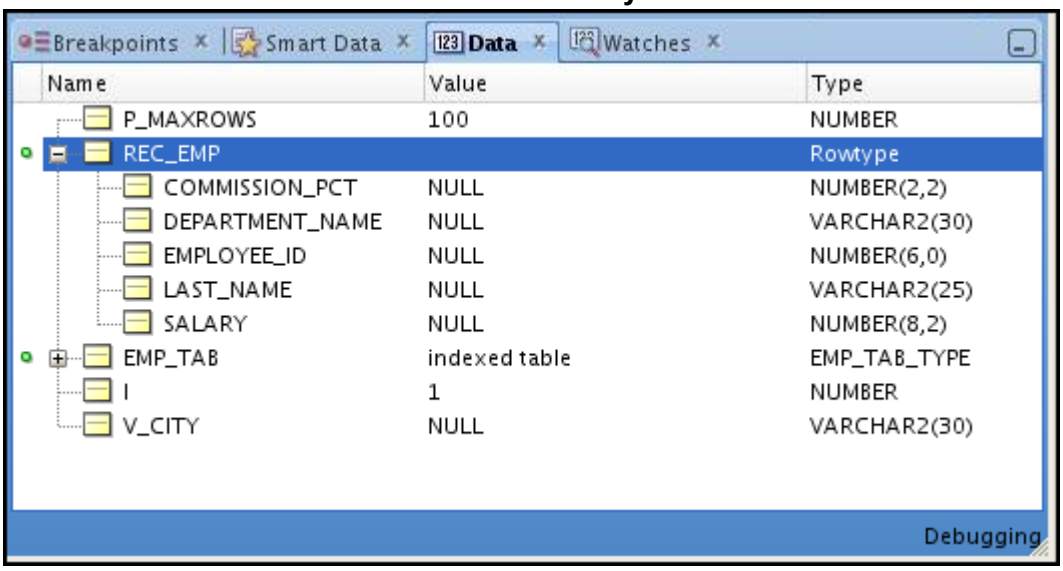
Replace the second P_MAXROWS with 100, and then click OK. Notice how the program control stops at the first breakpoint in the procedure as indicated by the blue highlight color and the red arrow pointing to that line of code. The additional debugging tabs are displayed at the bottom of the page.





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.

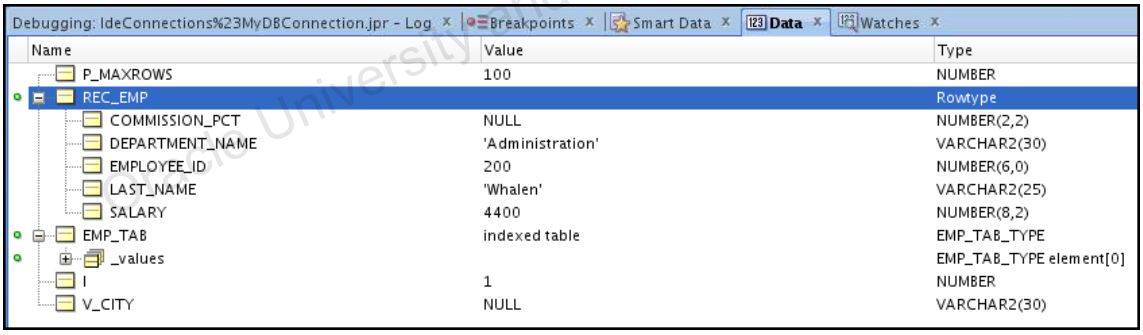


11. Use the Step Into debug option to step in to each line of code in `emp_list` and go through the while loop only once.

    Press F7 to step into the code only once.

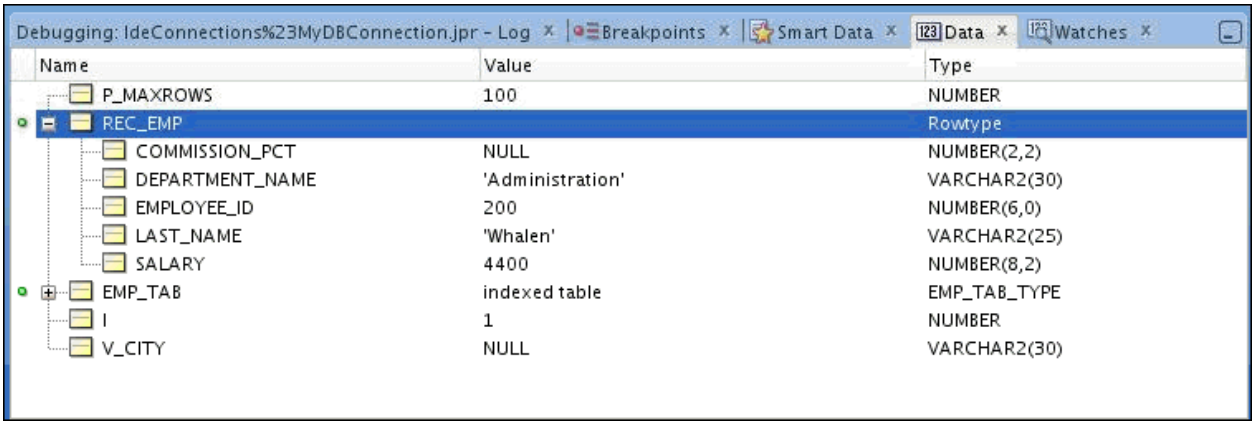
12. Examine the value of the variables on the Data tab. What are the values assigned to `REC_EMP` and `EMP_TAB`?

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



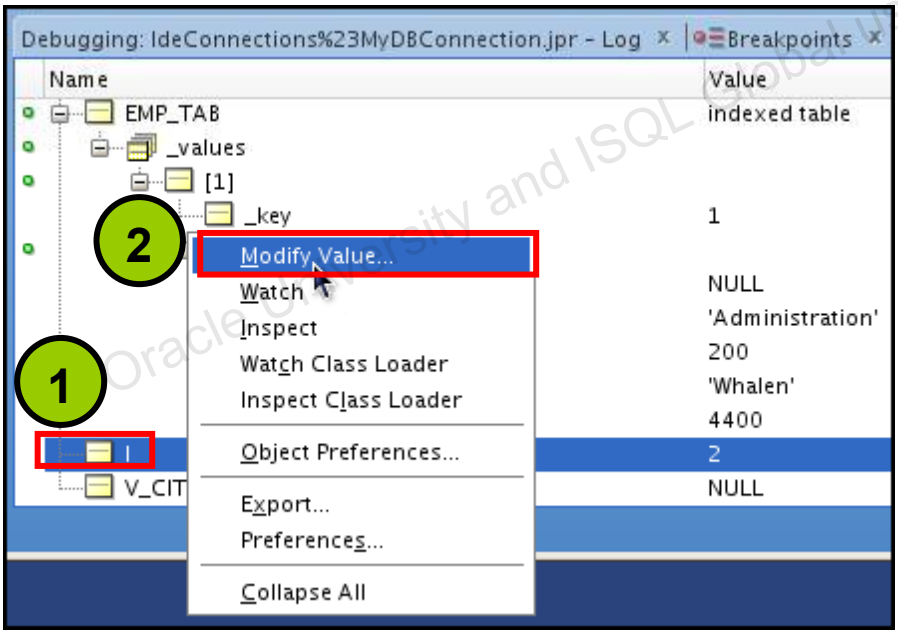
13. Continue pressing F7 until the `emp_tab(i) := rec_emp;` line is executed. Examine the value of the variables on the Data tab. What are the values assigned to `EMP_TAB`?

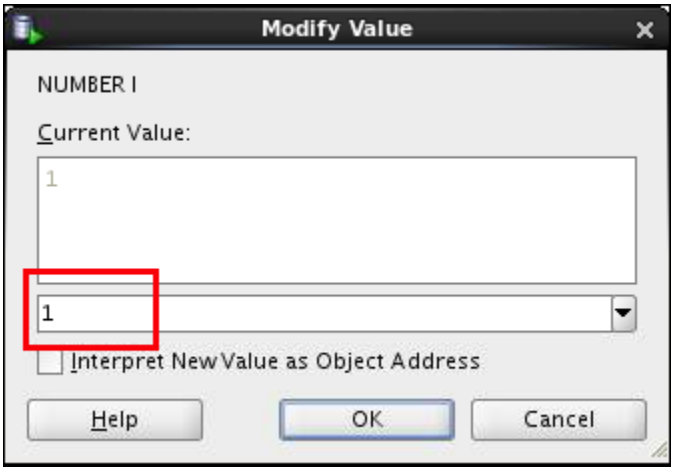
When the line `emp_tab(i) := rec_emp;` is executed, `emp_tab` is initialized to `rec_emp` as shown below:

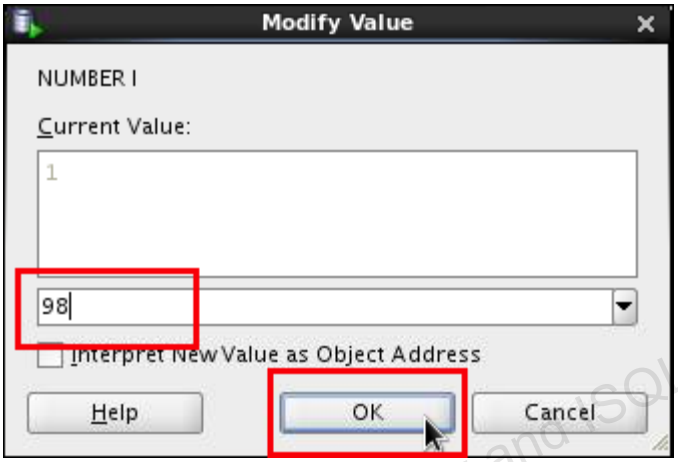


14. Use the Data tab to modify the value of the counter i to 98.

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:

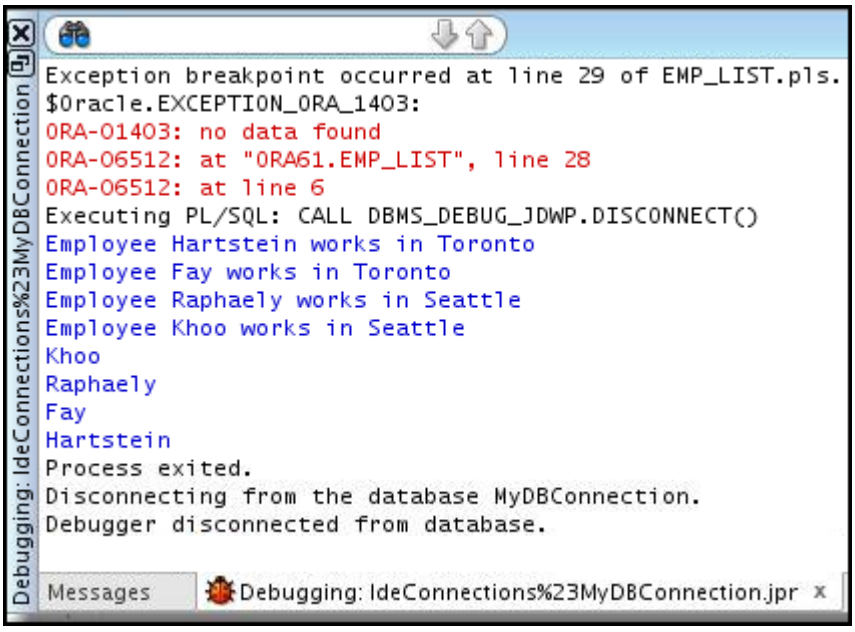






15. Continue pressing F7 until you observe the list of employees displayed on the Debugging – Log tab. How many employees are displayed?

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



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 14: Creating Packages

Chapter 14

Practices for Lesson 14: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 14-1: 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.

```
Script Output x
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

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 14-1: 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.

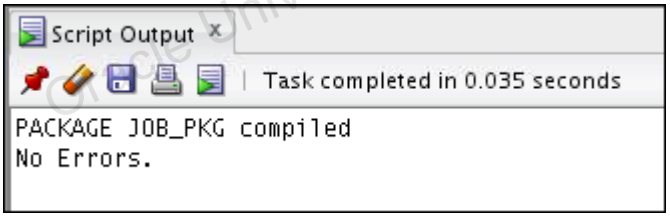
1. Create a package specification and body called `JOB_PKG`, containing a copy of your `ADD_JOB`, `UPD_JOB`, and `DEL_JOB` procedures as well as your `GET_JOB` function.

   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
```

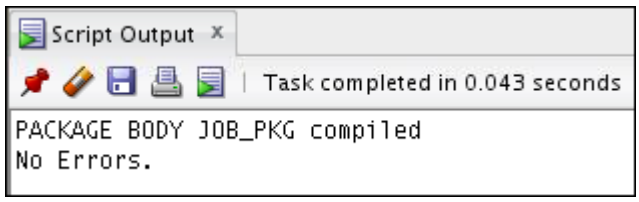


   b. Create the package body with the implementations for each of the subprograms.

      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')
```

```
Script Output x
Task completed in 1.951 seconds
anonymous block completed
```

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';
```

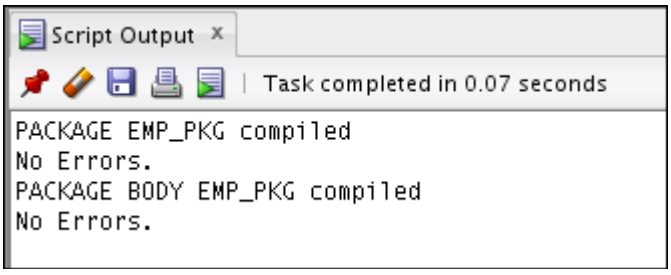
```
Script Output x
Task completed in 0.003 seconds
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
IT_SYSAN   Systems Analyst
```

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_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);
PROCEDURE get_employee(
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

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

END valid_deptid;

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

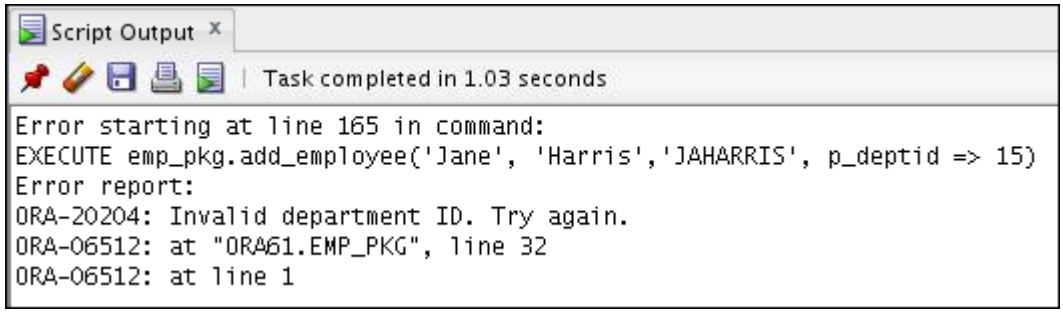


b. Invoke the `EMP_PKG.ADD_EMPLOYEE` procedure, using department ID 15 for employee Jane Harris with the email ID `JAHARRIS`. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.

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.

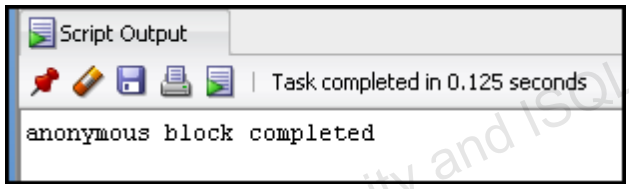
```
EXECUTE emp_pkg.add_employee('Jane', 'Harris','JAHARRIS',
p_deptid => 15)
```



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)
```

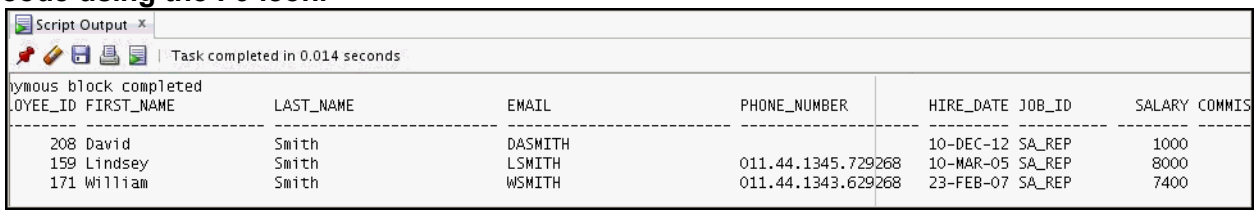


d. Query the `EMPLOYEES` table to verify that the new employee was added.

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.



Practices for Lesson 15: Working with Packages

Chapter 15

Practices for Lesson 15: Overview

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:

a. Before starting this practice, execute `/home/oracle/labs/plpu/code_ex/cleanup_scripts/cleanup_05.sql` script.

b. 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.

c. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 15-1: 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 15-1: 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.

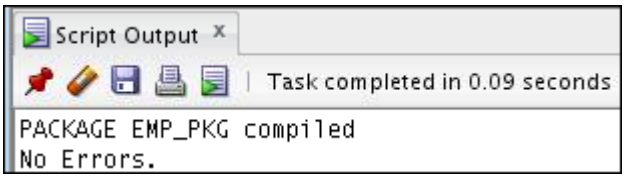
1. Modify the code for the EMP_PKG package that you created in Practice 4 step 2, and overload the ADD_EMPLOYEE procedure.
   a. In the package specification, add a new procedure called ADD_EMPLOYEE that accepts the following three parameters:
      1) First name
      2) Last name
      3) Department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

      Open the /home/oracle/labs/plpu/solns/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_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

-- New overloaded add_employee

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

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

```
SHOW ERRORS
```

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



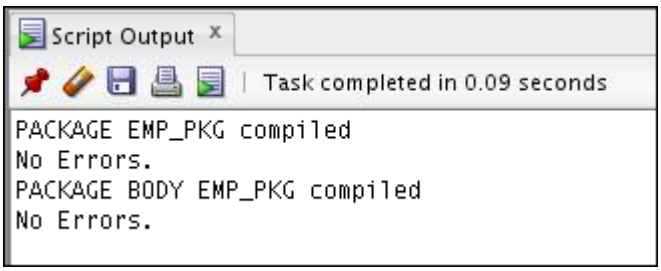
c. Implement the new ADD_EMPLOYEE procedure in the package body as follows:
   1) Format the email address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name.
   2) The procedure should call the existing ADD_EMPLOYEE procedure 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
```

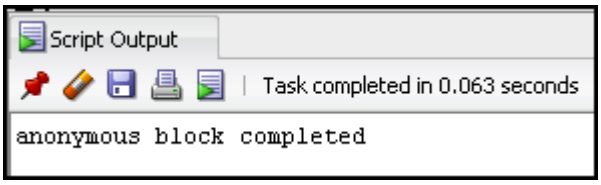
```
      again.');
    END IF;
    END add_employee;

  -- 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:

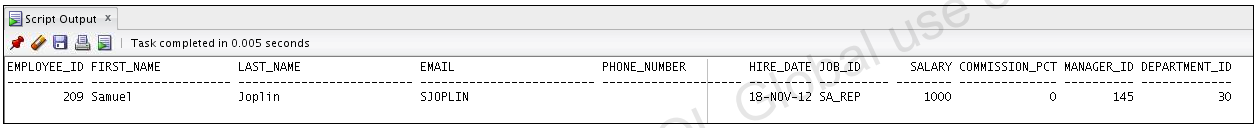
```
EXECUTE emp_pkg.add_employee('Samuel', 'Joplin', 30)
```



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';
```



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.

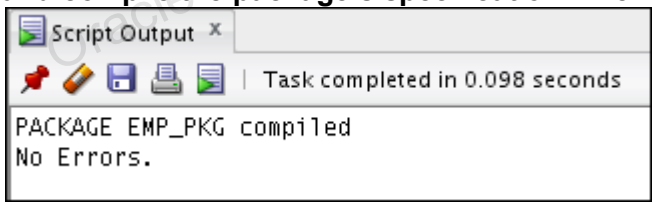
      Uncomment and select the code under Task 2_a.

```
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(
```

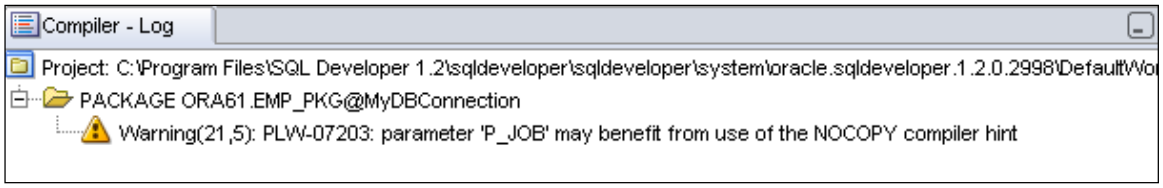
```
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  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.

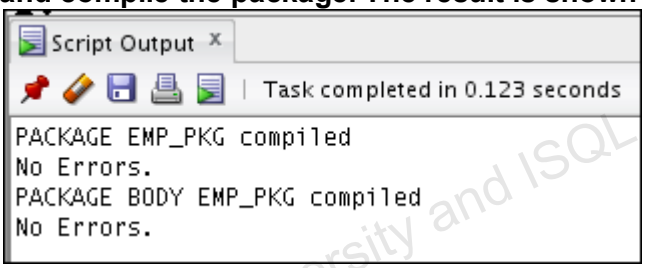
```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE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

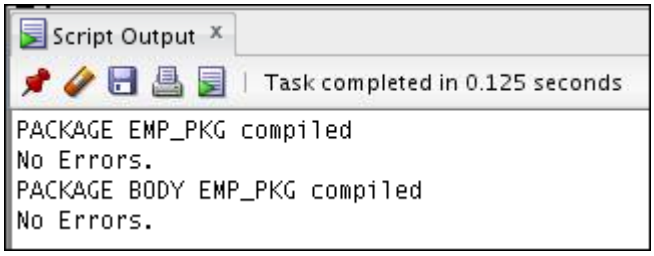
```
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE 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 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
```

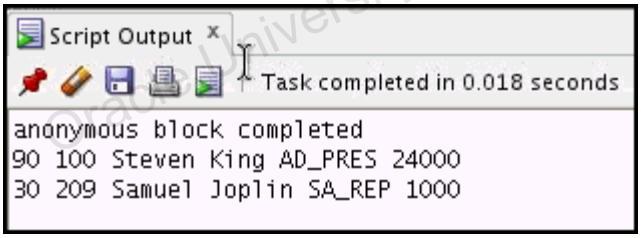
f. Click the Run Script icon (or press F5) to create and compile the package.
Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create and compile the package.



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(
```

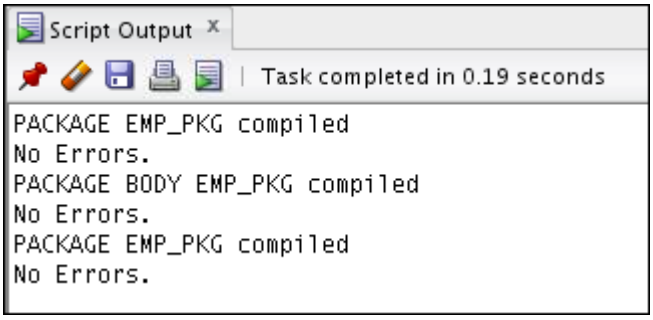
```
    p_empid IN employees.employee_id%TYPE,
    p_sal OUT employees.salary%TYPE,
    p_job OUT employees.job_id%TYPE);

  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_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

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.



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.

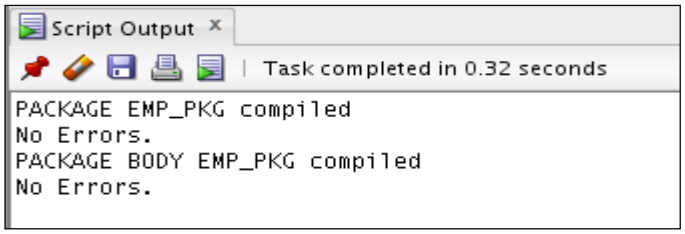
   Uncomment and select the code under Task 4_a. Click the Run Script icon (or press F5) to create and compile the package. The newly added code is highlighted in the following code box.

```
-- 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
```

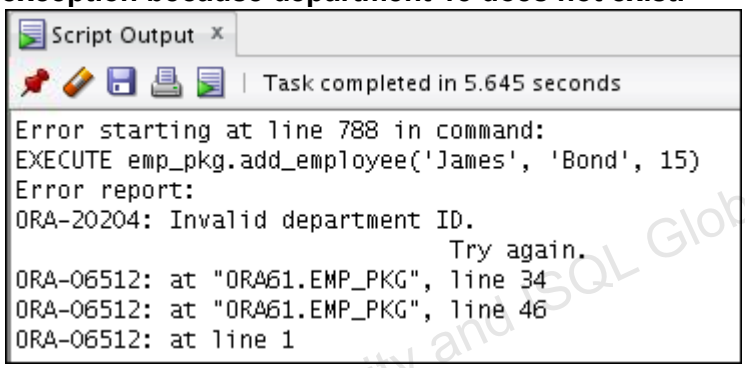


b. Test your code by calling `ADD_EMPLOYEE` using the name `James Bond` in department `15`. What happens?

Uncomment and select the code under Task 4_b.

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

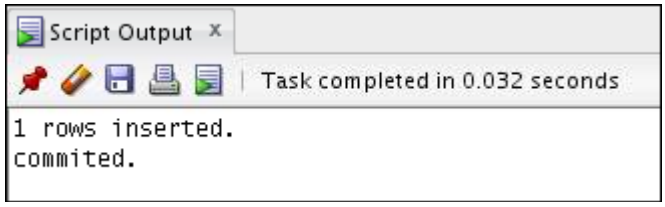
Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to test inserting a new employee. The insert operation to add the employee fails with an exception because department 15 does not exist.



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:

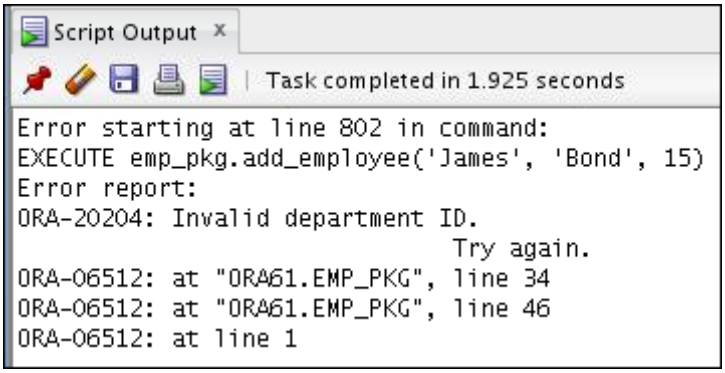
```
INSERT INTO departments (department_id, department_name)
VALUES (15, 'Security');
COMMIT;
```



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)
```

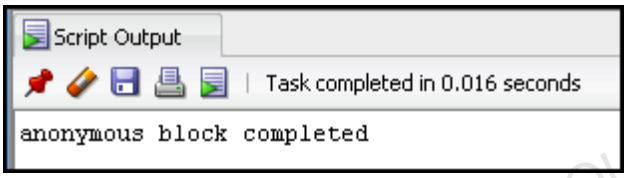


The insert operation to add the employee fails with an exception. Department 15 does not exist as an entry in the PL/SQL associative array (index-by table) package state variable.

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
```

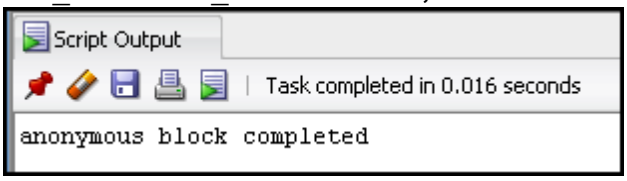


f. Test your code by calling `ADD_EMPLOYEE` using the employee name James Bond, who works in department 15. What happens?

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)
```

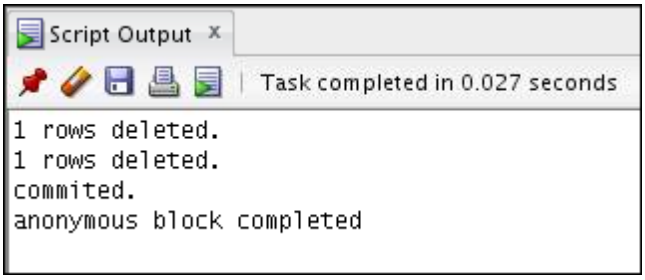
The row is finally inserted because the department 15 record exists in the database and the package's PL/SQL index-by table, due to invoking `EMP_PKG.INIT_DEPARTMENTS`, which refreshes the package state data.



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
```

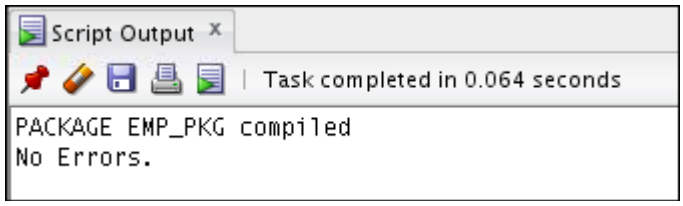


5. Reorganize the subprograms in the package specification and the body so that they are in alphabetical sequence.
   a. Edit the package specification and reorganize subprograms alphabetically. Click Run Script to re-create the package specification. Compile the package specification. What happens?

      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?

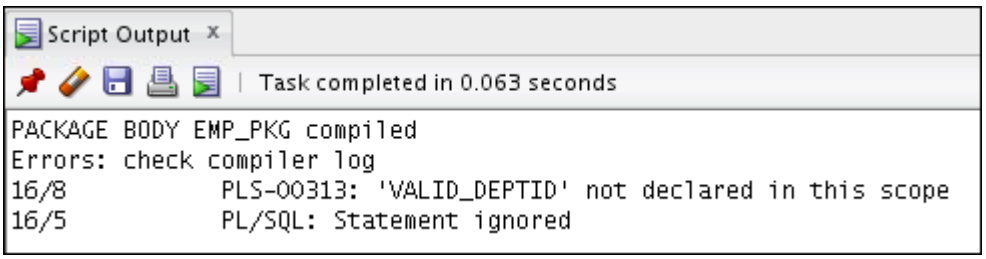
Uncomment and select the code under Task 5_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to re-create the package. The code and the result are displayed as follows.

```
-- 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.



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.

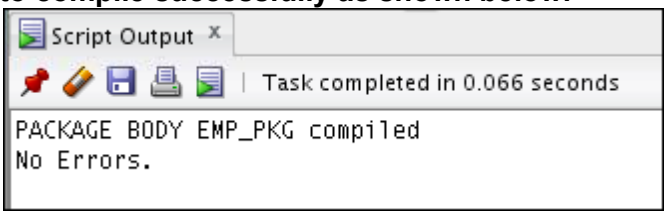
```
-- Package BODY

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 16: Using Oracle-Supplied Packages in Application Development

Chapter 16

Practices for Lesson 16: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 16-1: 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 16-1: Using the UTL_FILE Package

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

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

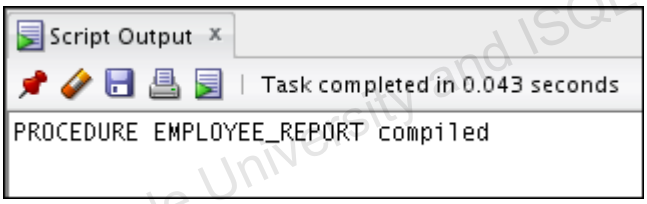
      Note: Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.

      Open the file in the `/home/oracle/labs/plpu/solns/sol_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;
/
```

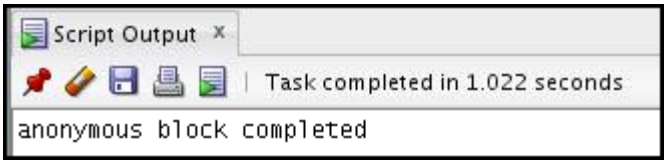
b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure.



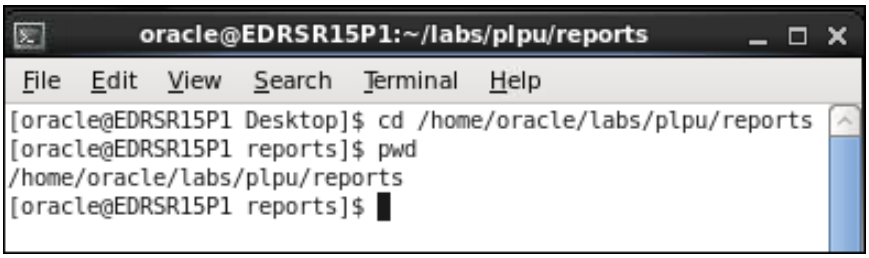
2. Invoke the procedure using the following as arguments:
   a. Use `REPORTS_DIR` as the alias for the directory object as the first parameter.
   b. Use `sal_rpt61.txt` as the second parameter.

   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')
```

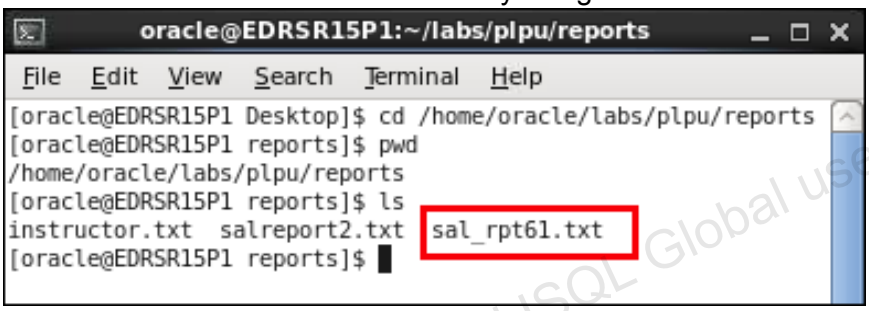


3. View the generated output text file as follows:
   a. Double-click the **Terminal** icon on your desktop. The **Terminal** window is displayed.
   b. At the **$** prompt, change to the **`/home/oracle/labs/plpu/reports`** directory that contains the generated output file, `sal_rpt61.txt` using the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



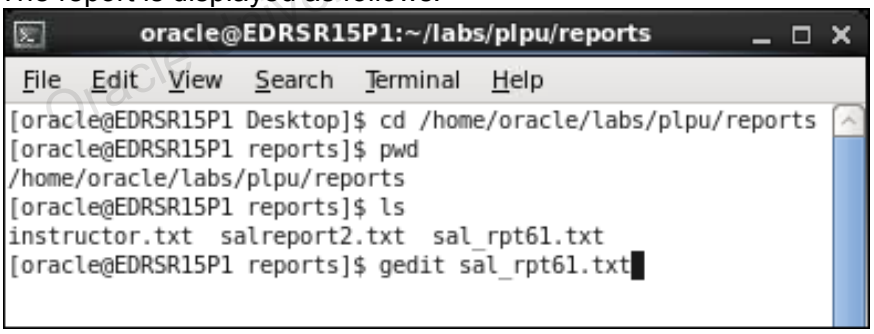
Note: You can use the `pwd` command to list the current working directory as shown in the screenshot above.

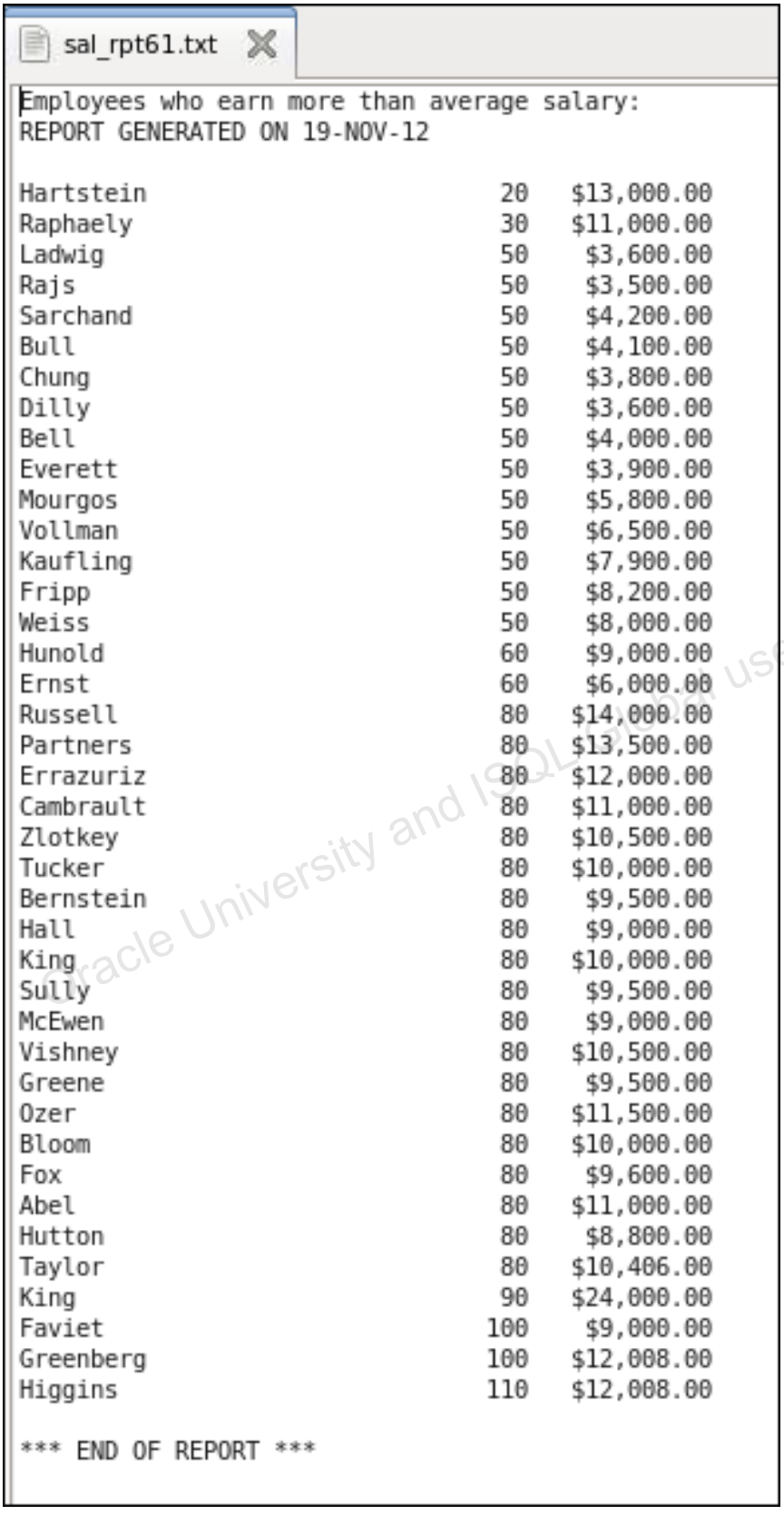
   c. List the contents of the current directory using the `ls` command as follows:



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:





sal_rpt61.txt

```
Employees who earn more than average salary:
REPORT GENERATED ON 19-NOV-12

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 17: Using Dynamic SQL

Chapter 17

Practices for Lesson 17: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 17-1: 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 17-1: Using Native Dynamic SQL

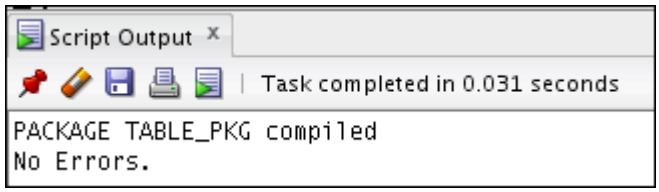
In this practice, you create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table. In addition, you create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an `INVALID` status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

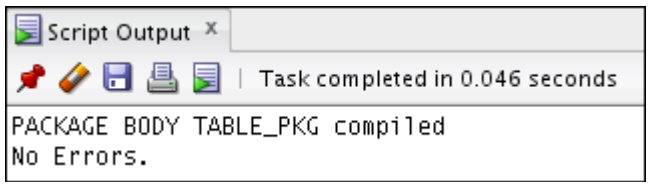


b. Create the package body that accepts the parameters and dynamically constructs the appropriate SQL statements that are executed using Native Dynamic SQL, except for the remove procedure. This procedure should be written using the `DBMS_SQL` package.

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;
```

```
    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_SQL.PARSE(cur_id, v_stmt, DBMS_SQL.NATIVE);
    -- Parse executes DDL statements,no EXECUTE is required.
    DBMS_SQL.CLOSE_CURSOR(cur_id);
  END;

END table_pkg;
/
SHOW ERRORS
```

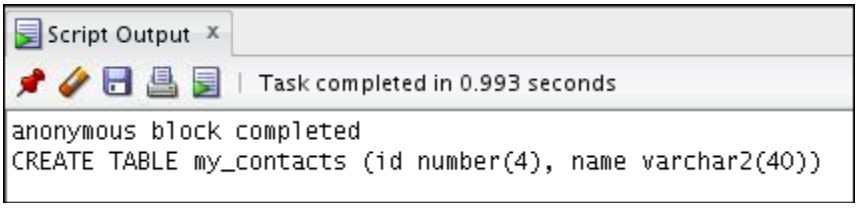


c. Execute the `MAKE` package procedure to create a table as follows:

```
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:

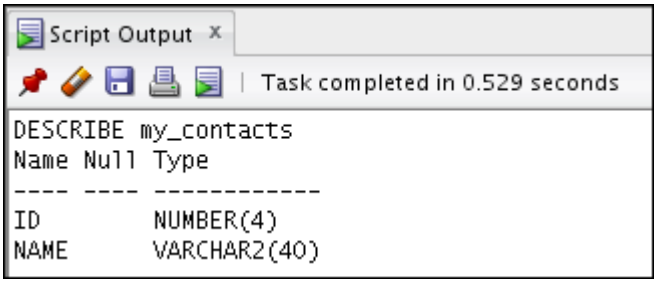
```
EXECUTE table_pkg.make('my_contacts', 'id number(4), name
varchar2(40)')
```



d. Describe the `MY_CONTACTS` table structure.

```
DESCRIBE my_contacts
```

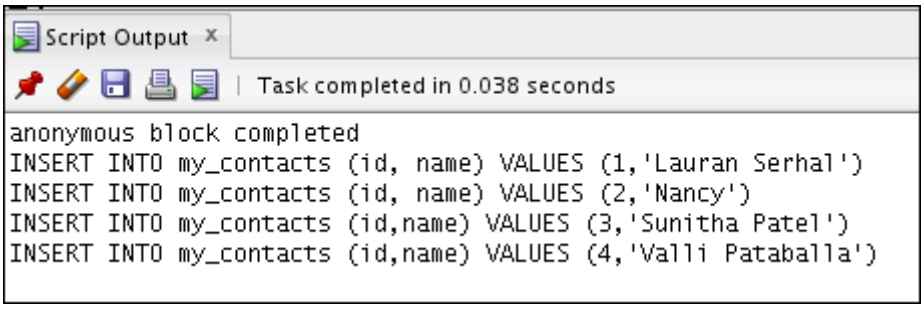
The result is displayed as follows:



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

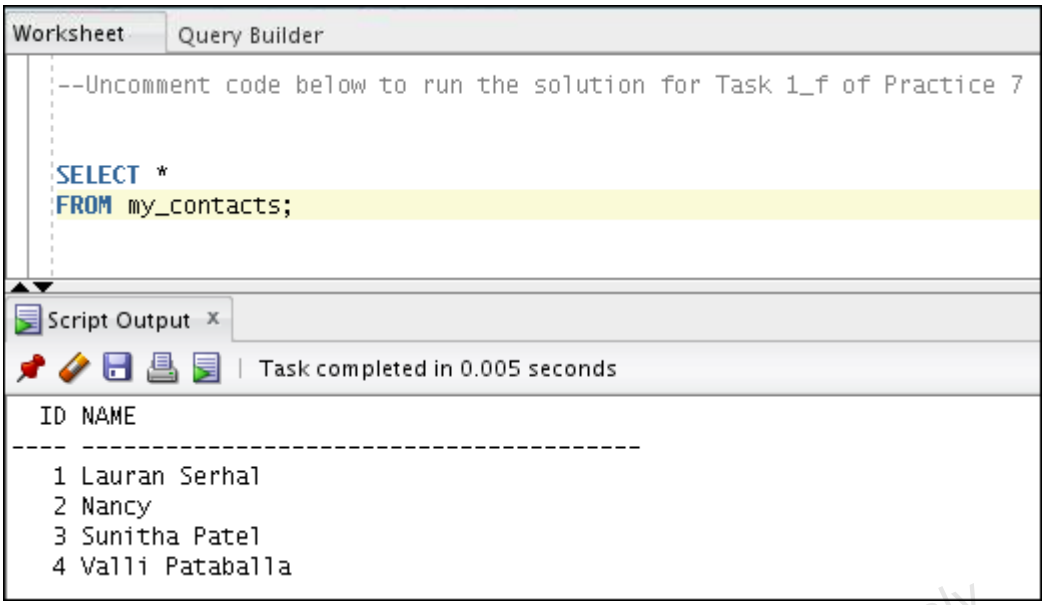
```
SET SERVEROUTPUT ON

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.



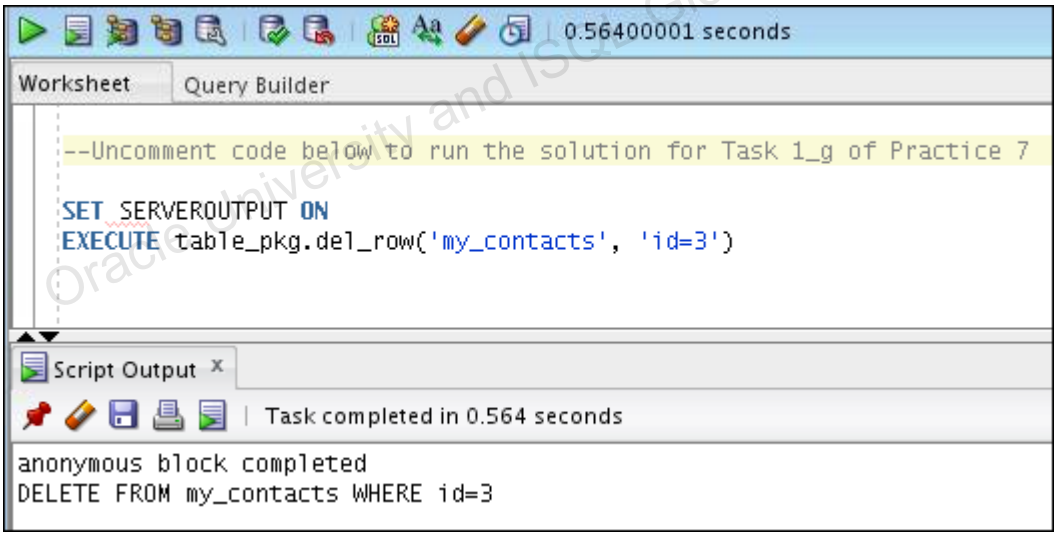
f. Query the `MY_CONTACTS` table contents to verify the additions.

The code and the results are displayed as follows:

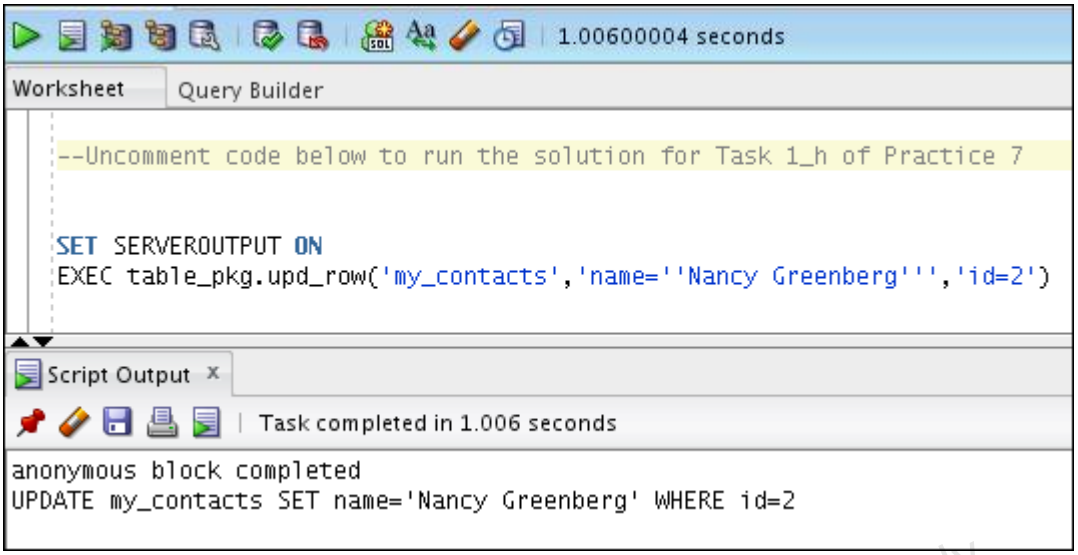


g. Execute the `DEL_ROW` package procedure to delete a contact with an ID value of `3`.

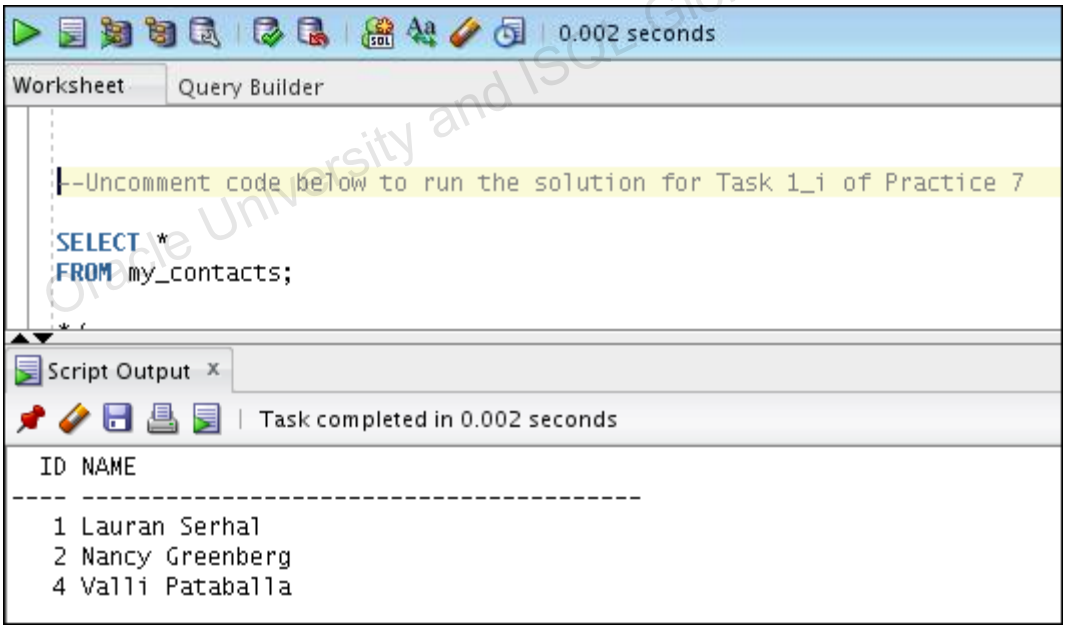
The code and the results are displayed as follows:



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:

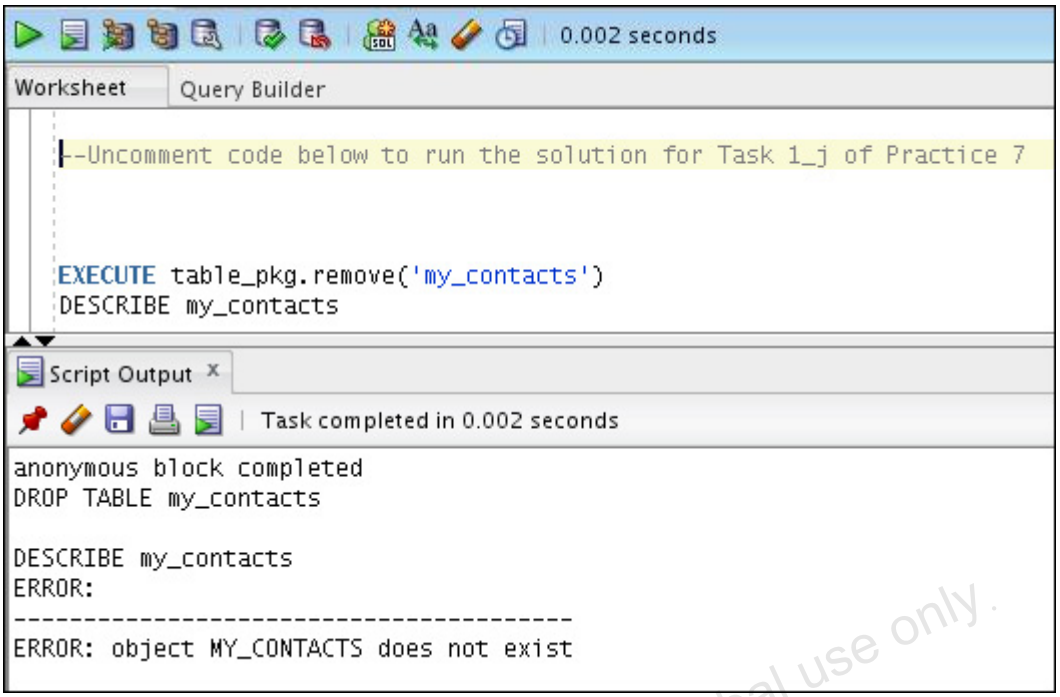


i. Query the `MY_CONTACTS` table contents to verify the changes.
The code and the results are displayed as follows:



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

The code and the results are displayed as follows:

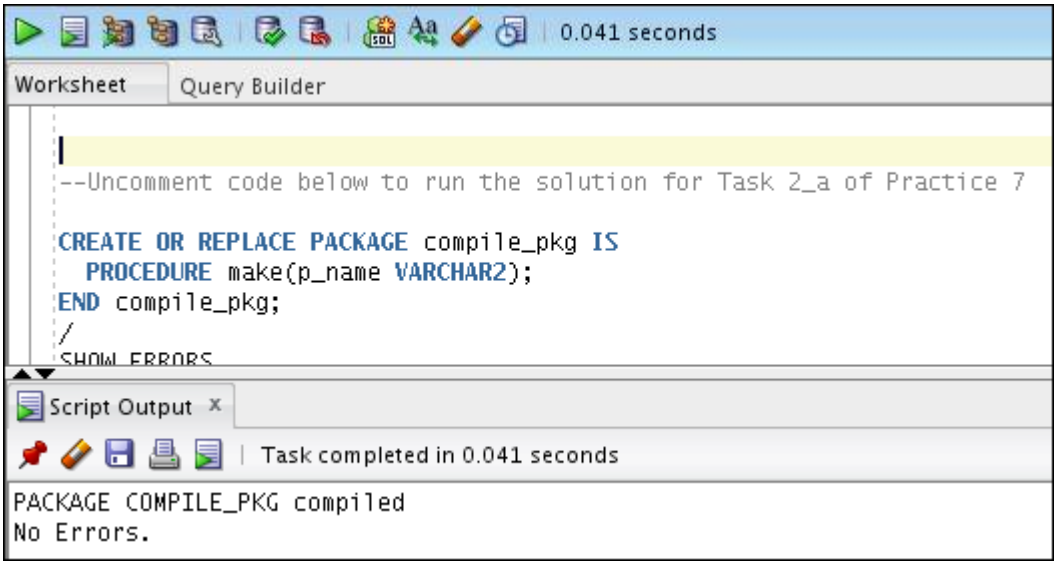


2. Create a `COMPILE_PKG` package that compiles the PL/SQL code in your schema.

   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
```



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:

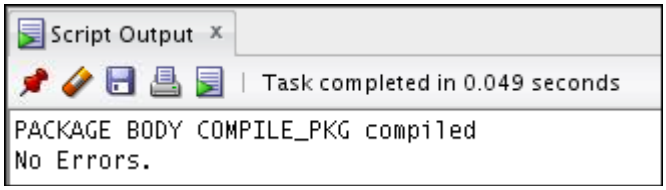
   – The `MAKE` procedure accepts one argument, name, which represents the object name.

   – 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
```

```
    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
```

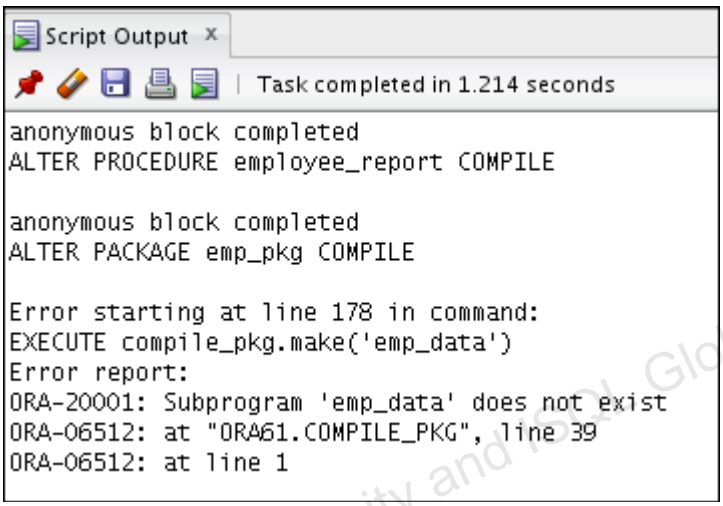


c. Use the `COMPILE_PKG.MAKE` procedure to compile the following:

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')
```



Practices for Lesson 18: Creating Triggers

Chapter 18

Practices for Lesson 18: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 18-1: 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.

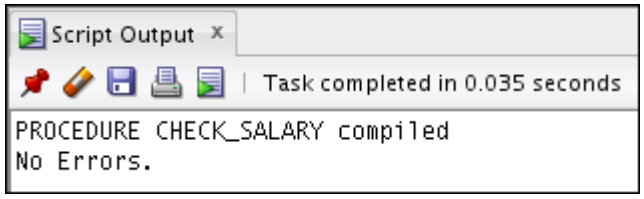
Solution 18-1: Creating Statement and Row Triggers

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
```



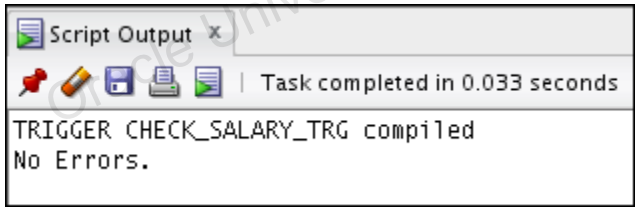
b. Create a trigger called `CHECK_SALARY_TRG` on the `EMPLOYEES` table that fires before an `INSERT` or `UPDATE` operation on each row:

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
```

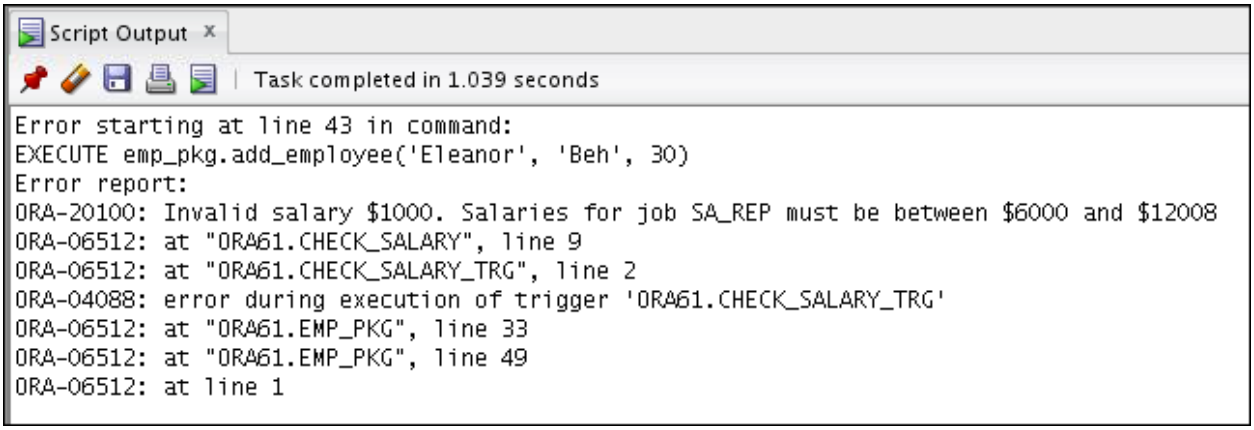


2. Test the `CHECK_SALARY_TRG` trigger using the following cases:

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)
```

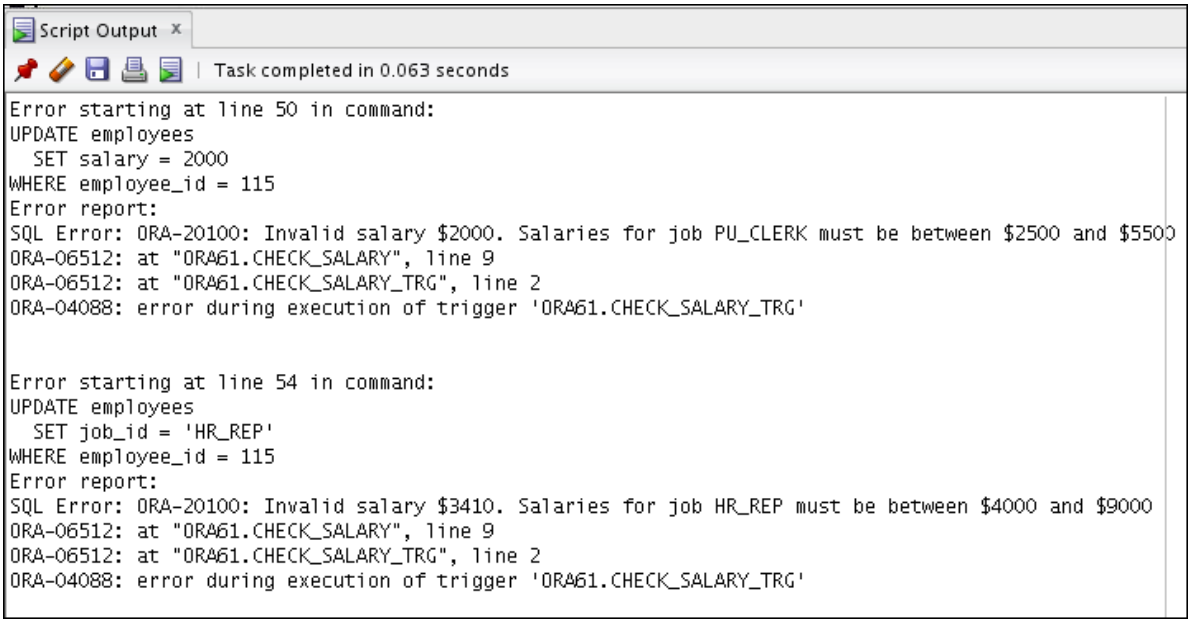


The trigger raises an exception because the `EMP_PKG.ADD_EMPLOYEE` procedure invokes an overloaded version of itself that uses the default salary of $1,000 and a default job ID of `SA_REP`. However, the `JOBS` table stores a minimum salary of $6,000 for the `SA_REP` type.

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;
```



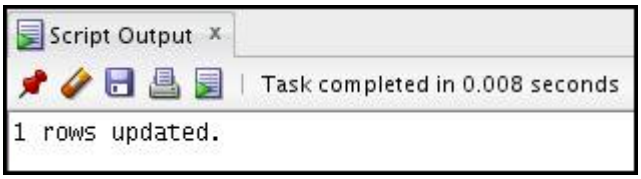
The first update statement fails to set the salary to $2,000. The check salary trigger rule fails the update operation because the new salary for employee 115 is less than the minimum allowed for the `PU_CLERK` job ID.

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;
```



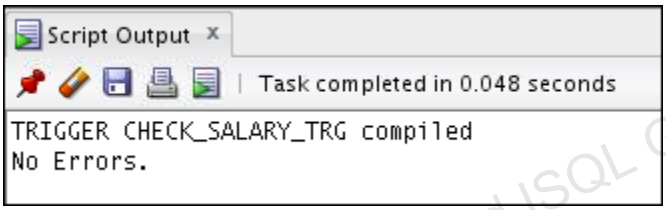
The update operation is successful because the new salary falls within the acceptable range for the current job ID.

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
```

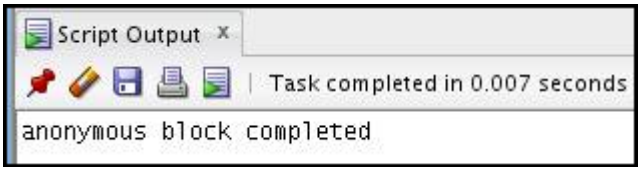


b. Test the trigger by executing the `EMP_PKG.ADD_EMPLOYEE` procedure with the following parameter values:

- `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.

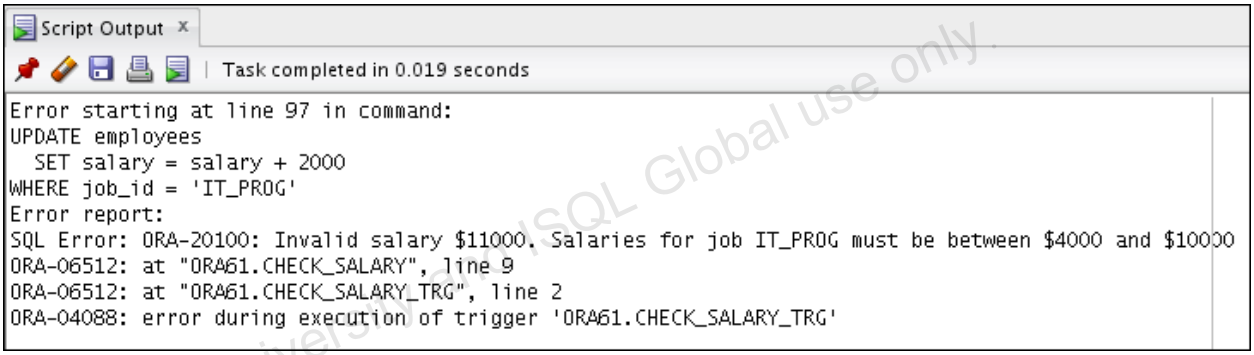
```
BEGIN
  emp_pkg.add_employee('Eleanor', 'Beh', 'EBEH',
                       p_job => 'IT_PROG', p_sal => 5000);
END;
/
```



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';
```



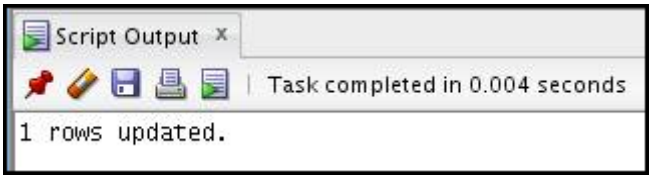
An employee's salary in the specified job type exceeds the maximum salary for that job type. No employee salaries in the `IT_PROG` job type are updated.

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.

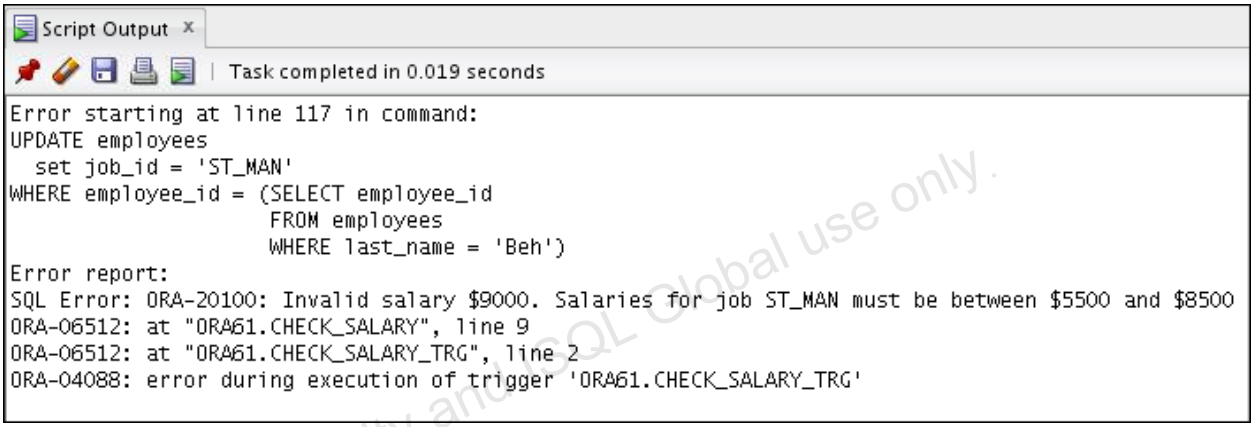
```
UPDATE employees
  SET salary = 9000
WHERE employee_id = (SELECT employee_id
                     FROM employees
                     WHERE last_name = 'Beh');
```



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');
```



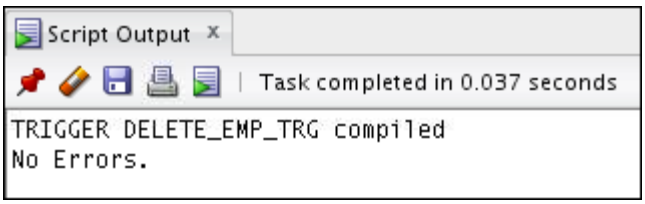
The maximum salary of the new job type is less than the employee's current salary; therefore, the update operation fails.

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

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
```

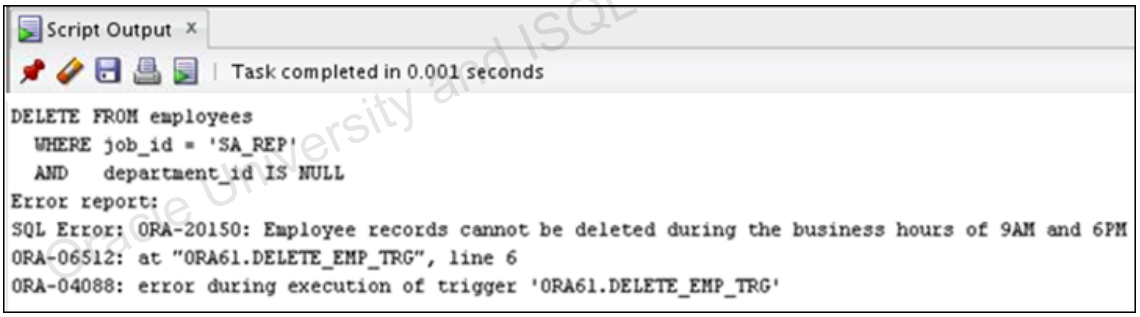


b. Attempt to delete employees with `JOB_ID` of `SA_REP` who are not assigned to a department.

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;
```



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 19: Creating Compound, DDL, and Event Database Triggers

Chapter 19

Practices for Lesson 19: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 19-1: 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 19-1: 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_comm employees.commission_pct%TYPE DEFAULT 0,
    p_deptid employees.department_id%TYPE DEFAULT 30);

  PROCEDURE add_employee(
```

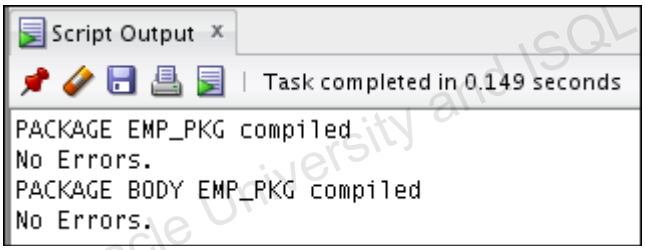
```
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  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;

  -- 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;
```

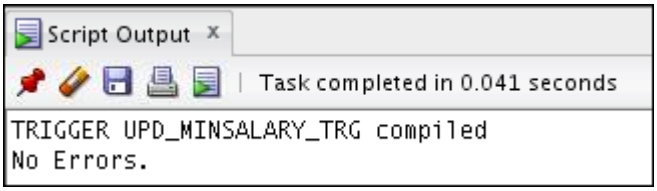
```
-- New set_salary procedure
PROCEDURE set_salary(p_jobid VARCHAR2, p_min_salary NUMBER) IS
    CURSOR cur_emp IS
      SELECT employee_id
      FROM employees
      WHERE job_id = p_jobid AND salary < p_min_salary;
  BEGIN
    FOR rec_emp IN cur_emp
    LOOP
      UPDATE employees
        SET salary = p_min_salary
      WHERE employee_id = rec_emp.employee_id;
    END LOOP;
  END set_salary;

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



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.

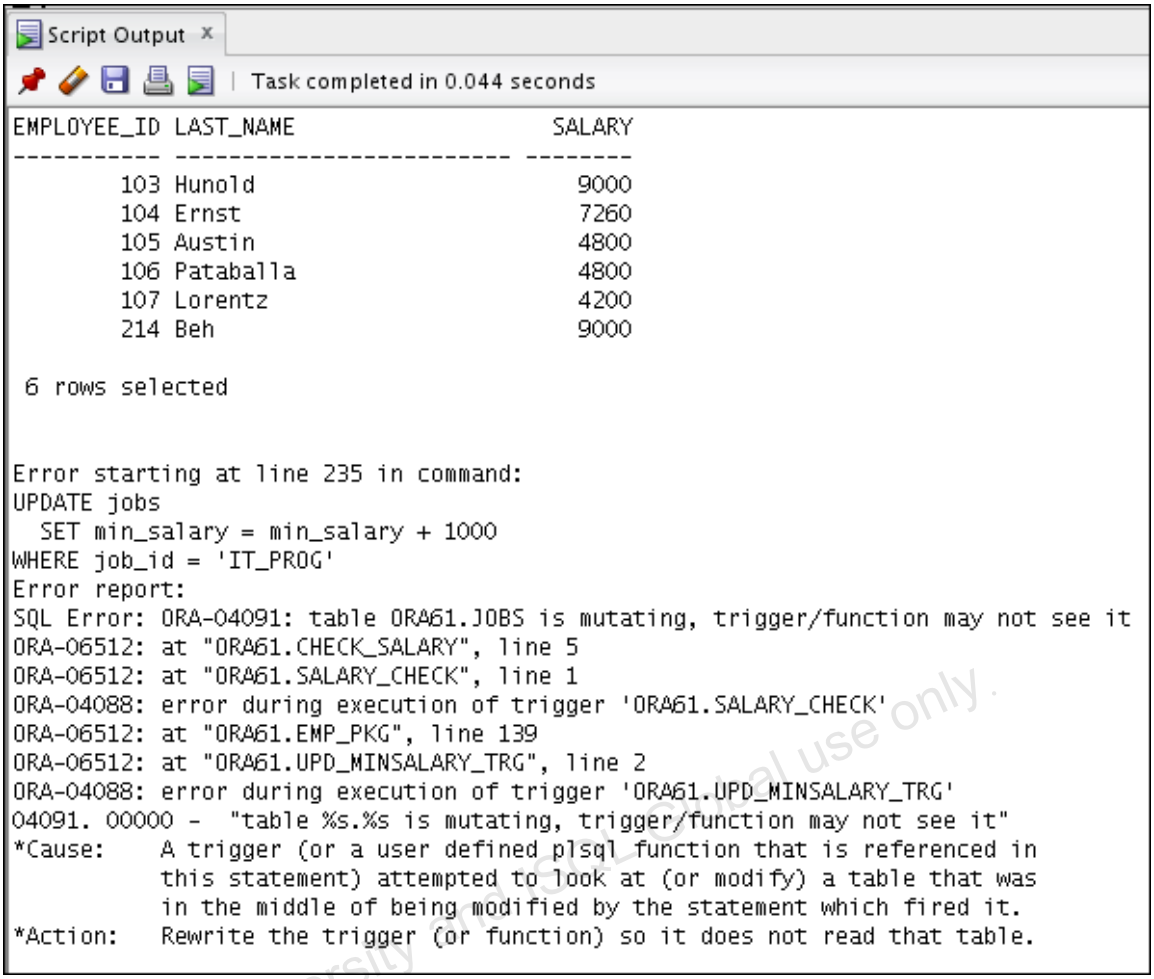
```
CREATE OR REPLACE TRIGGER upd_minsalary_trg
AFTER UPDATE OF min_salary ON JOBS
FOR EACH ROW
BEGIN
  emp_pkg.set_salary(:new.job_id, :new.min_salary);
END;
/
SHOW ERRORS
```



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';
```



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.

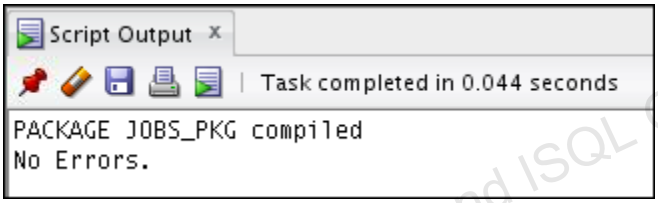
2. To resolve the mutating table issue, create a `JOBS_PKG` package to maintain in memory a copy of the rows in the `JOBS` table. Next, modify the `CHECK_SALARY` procedure to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, you must create a `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
```



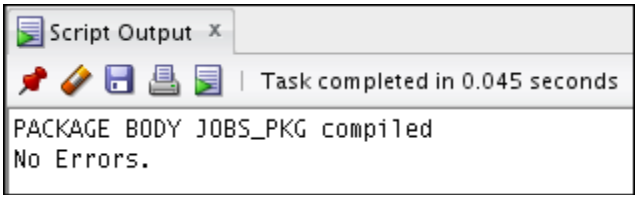
b. Implement the body of `JOBS_PKG` as follows:

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.

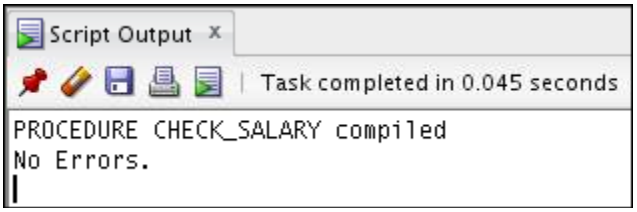
```
CREATE OR REPLACE PACKAGE BODY jobs_pkg IS
  TYPE jobs_tab_type IS TABLE OF jobs%rowtype
    INDEX BY jobs.job_id%type;
  jobstab jobs_tab_type;

  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.

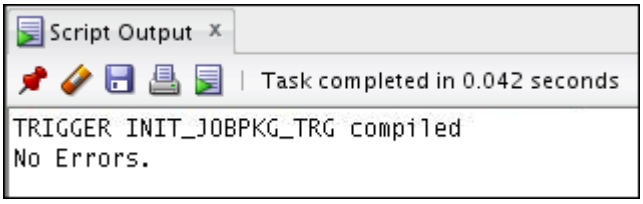
```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN

  -- 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.

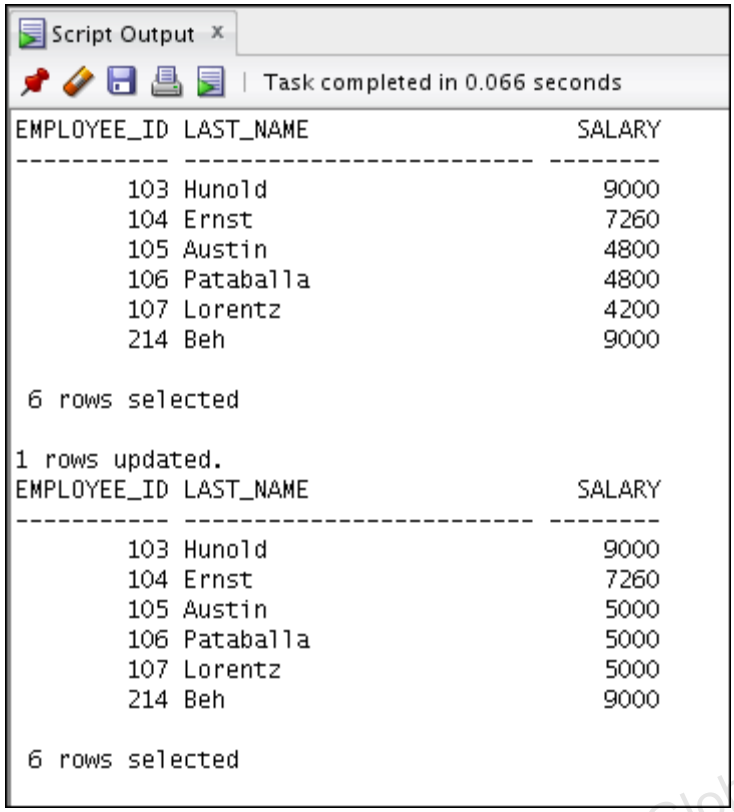
```
CREATE OR REPLACE TRIGGER init_jobpkg_trg
BEFORE INSERT OR UPDATE ON jobs
CALL jobs_pkg.initialize
/
SHOW ERRORS
```



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';
```

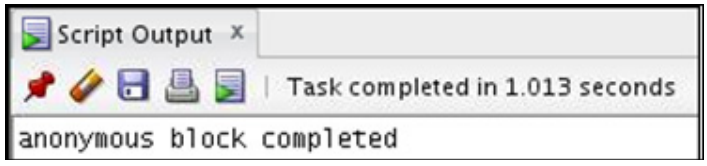


The employees with last names `Austin`, `Pataballa`, and `Lorentz` have all had their salaries updated. No exception occurred during this process, and you implemented a solution for the mutating table trigger exception.

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.

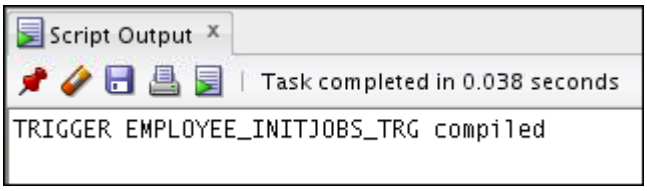
```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal
=> 6500)
```



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.

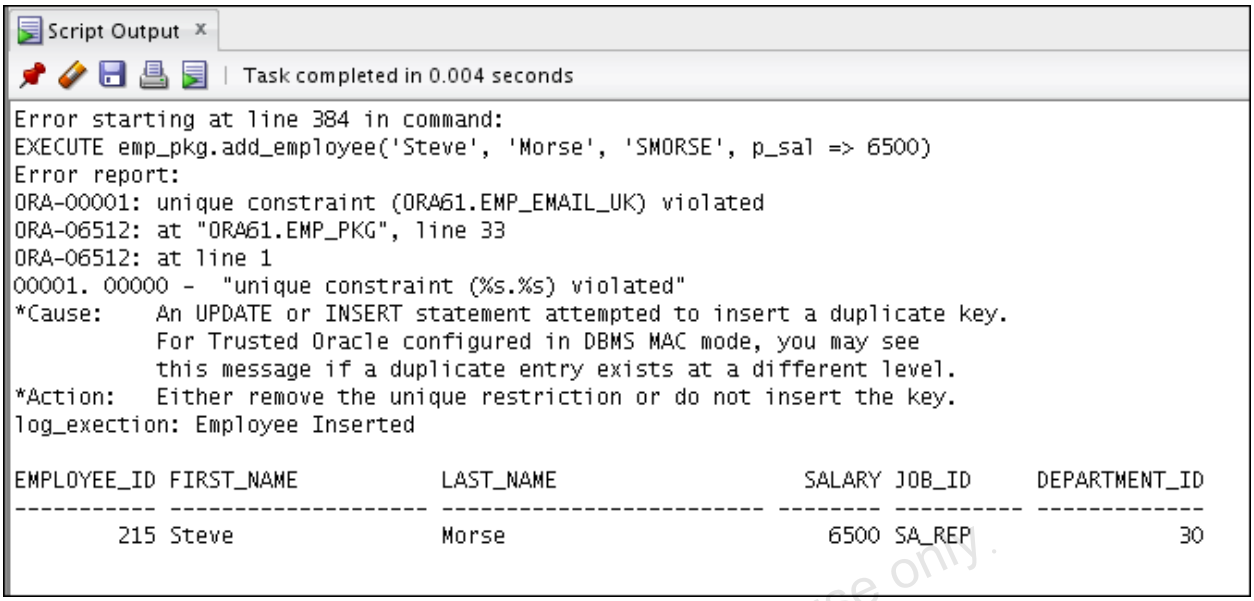
```
CREATE TRIGGER employee_initjobs_trg
BEFORE INSERT OR UPDATE OF job_id, salary ON employees
CALL jobs_pkg.initialize
/
```



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.



Practices for Lesson 20: Design Considerations for PL/SQL Code

Chapter 20

Practices for Lesson 20: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 20-1: 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 `SERVEROUTPUT`. Invoke the `emp_pkg.get_employees` procedure for department `30`, and then invoke `emp_pkg.show_employees`. Repeat this for department `60`.
2. Your manager wants to keep a log whenever the `add_employee` procedure in the package is invoked to insert a new employee into the `EMPLOYEES` table.
   a. First, load and execute the 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 20-1: Using Bulk Binding and Autonomous Transactions

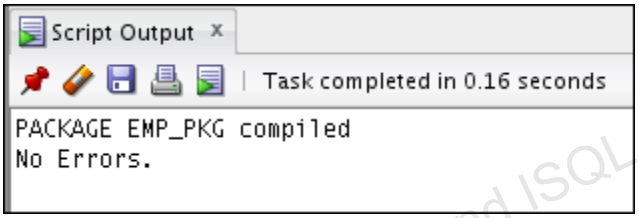
In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table. In addition, you create the `add_employee` procedure that inserts new employees. 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

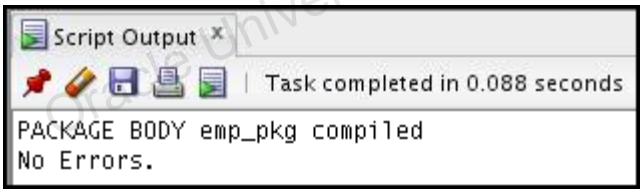
Uncomment and select the code under Task 1_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the package body. The code and the results are shown below. The newly added code is highlighted in bold letters in the code box below.

```
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.

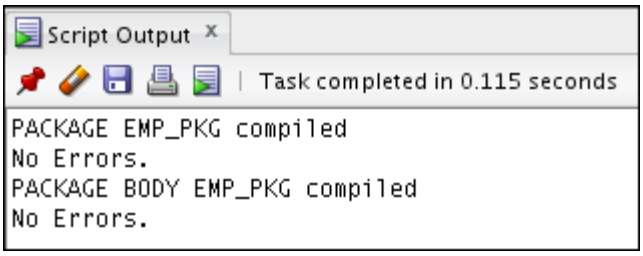
```
-- Package SPECIFICATION

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
```

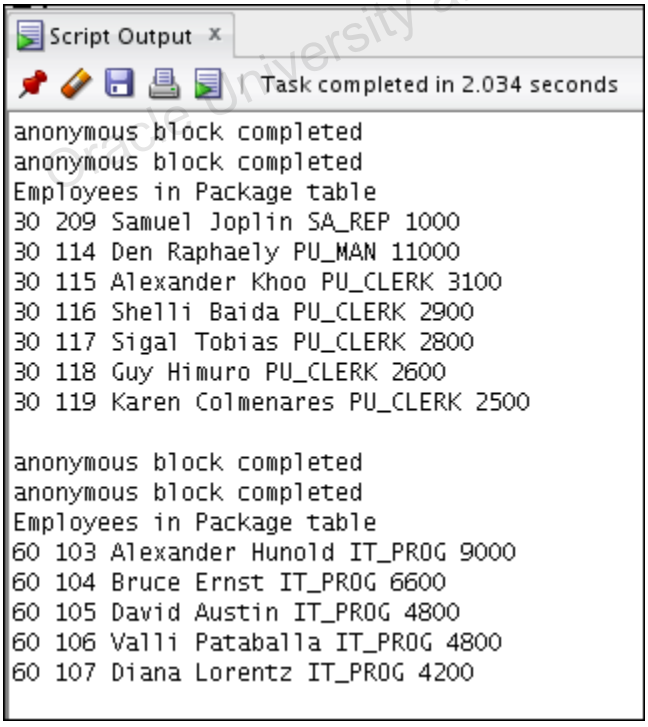
```
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;

  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
```



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:

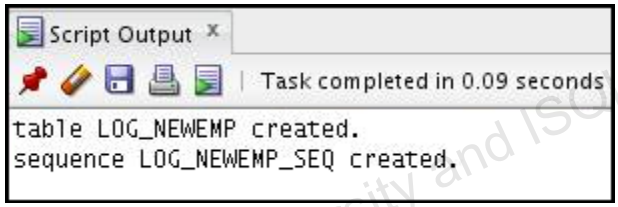
```
SET SERVEROUTPUT ON

EXECUTE emp_pkg.get_employees(30)
EXECUTE emp_pkg.show_employees

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



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
```

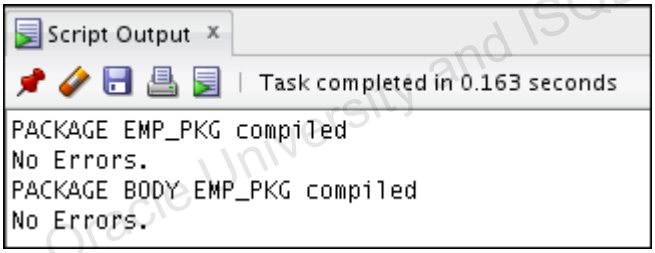
```
  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

  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(
```

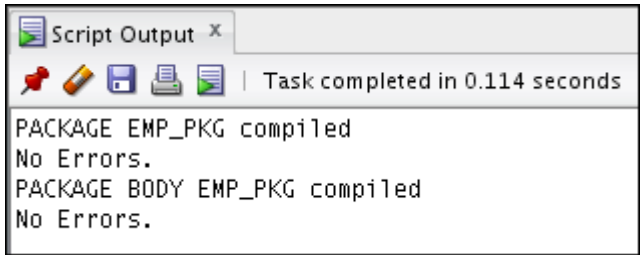
```
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_email employees.email%TYPE,
    p_job employees.job_id%TYPE DEFAULT 'SA_REP',
    p_mgr employees.manager_id%TYPE DEFAULT 145,
    p_sal employees.salary%TYPE DEFAULT 1000,
    p_comm employees.commission_pct%TYPE 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

    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
```

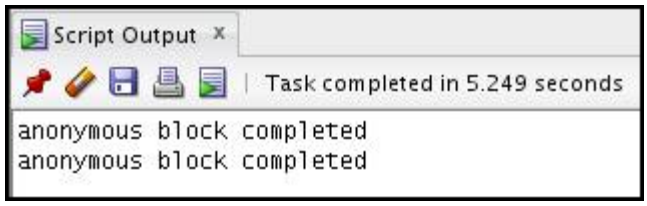
```
    RETURN valid_departments.exists(p_deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN

    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)
```



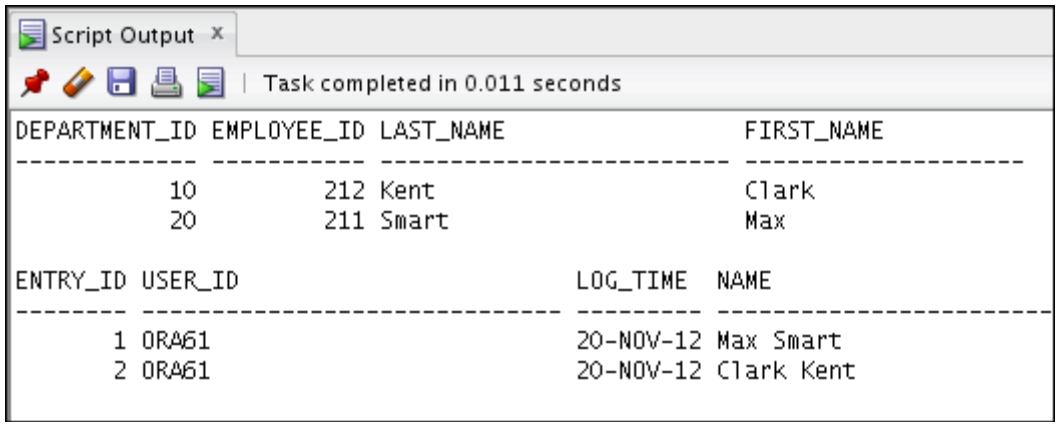
Both insert statements complete successfully. The log table has two log records as shown in the next step.

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
```

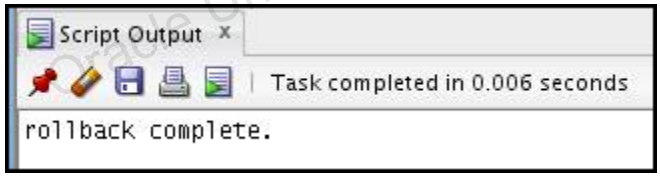
```
where last_name in ('Kent', 'Smart');

select * from log_newemp;
```

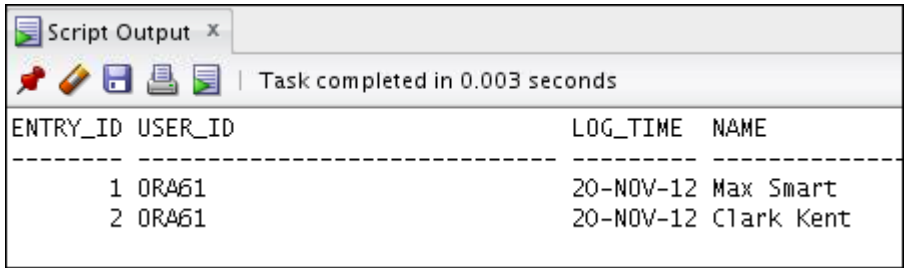


There are two log records, one for `Smart` and another for `Kent`.

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 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 21: Tuning the PL/SQL Compiler

Chapter 21

Practices for Lesson 21: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 21-1: 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 `SERVEROUTPUT` before running the script.

Solution 21-1: Using the PL/SQL Compiler Parameters and Warnings

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

1. 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.
   a. Object name
   b. Object type
   c. The object's compilation mode
   d. 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;
```

Query Result

SQL | Fetched 50 rows in 0.009 seconds

	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.

2. Alter the `PLSQL_CODE_TYPE` parameter to enable native compilation for your session, and compile `ADD_JOB_HISTORY`.

   a. Execute the `ALTER SESSION` command to enable native compilation for the session.
   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

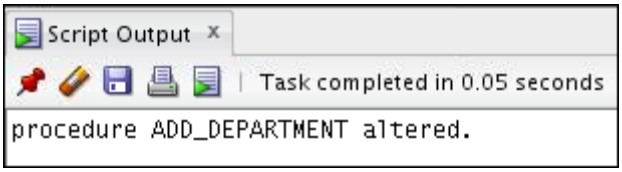
Task completed in 0.002 seconds

```
session SET altered.
```

b. Compile the `ADD_DEPARTMENT` procedure.

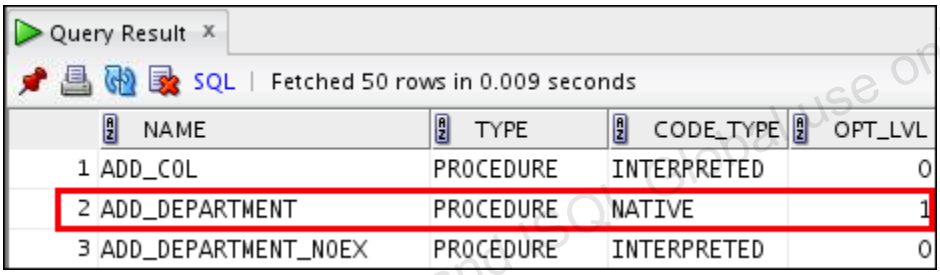
Uncomment and select the code under Task 2_b. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the query. The code and the results are shown below.

```
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;
```

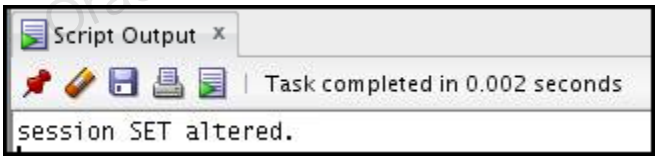


	NAME	TYPE	CODE_TYPE	OPT_LVL
1	ADD_COL	PROCEDURE	INTERPRETED	0
2	ADD_DEPARTMENT	PROCEDURE	NATIVE	1
3	ADD_DEPARTMENT_NOEX	PROCEDURE	INTERPRETED	0

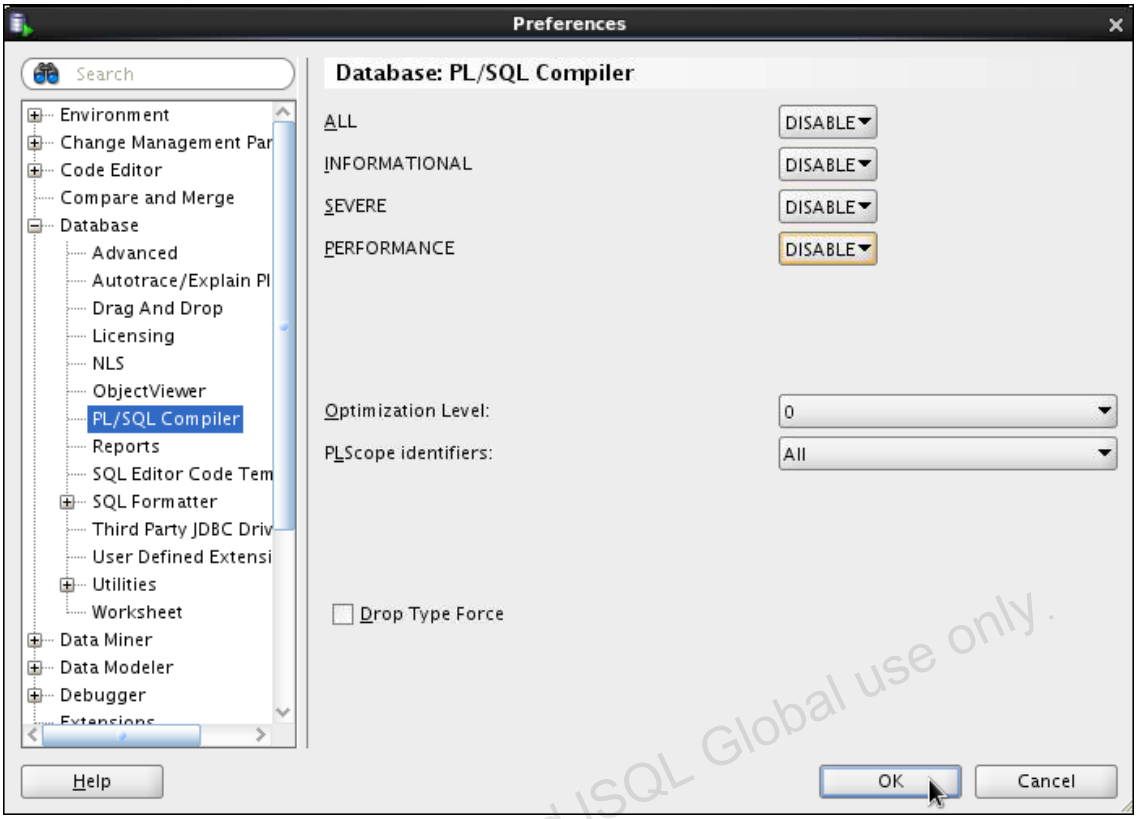
...

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.

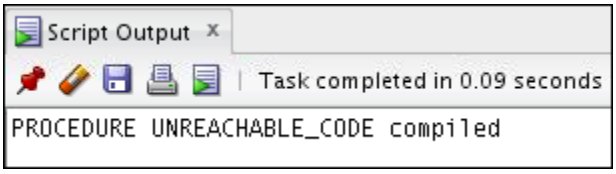


Select DISABLE for all four PL/SQL compiler warnings categories, and then click OK.

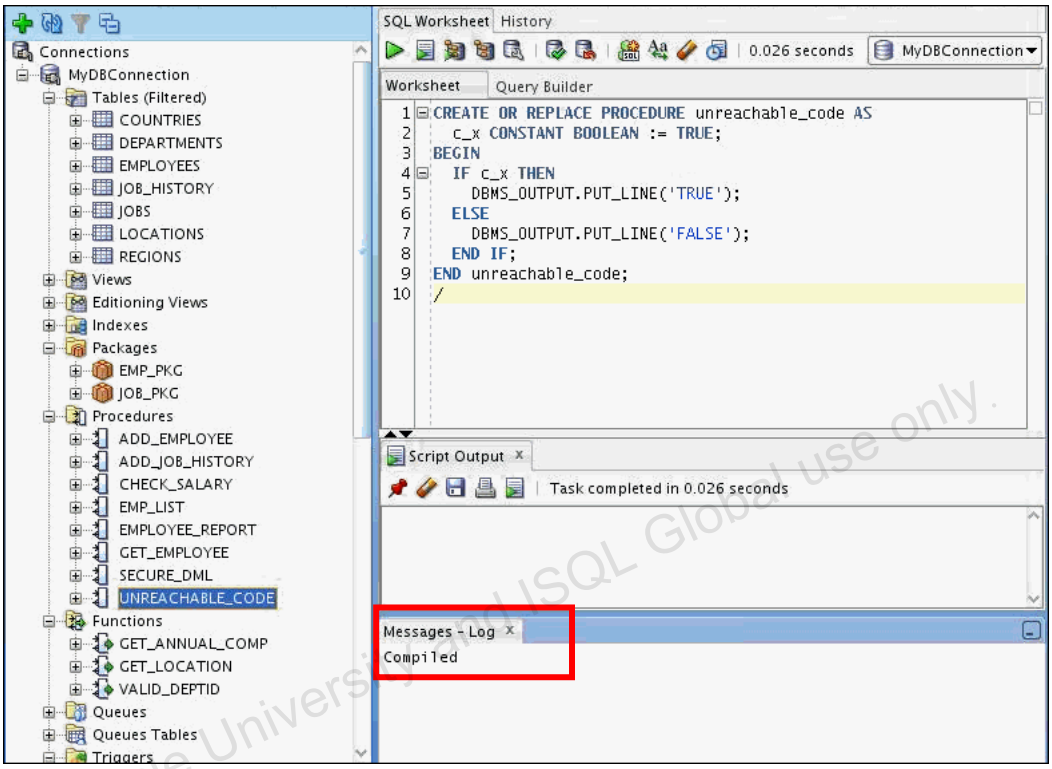
4. Edit, examine, and execute the `lab_11_04.sql` script to create the `UNREACHABLE_CODE` procedure. Click the Run Script icon (or press F5) to create and compile the procedure.

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.

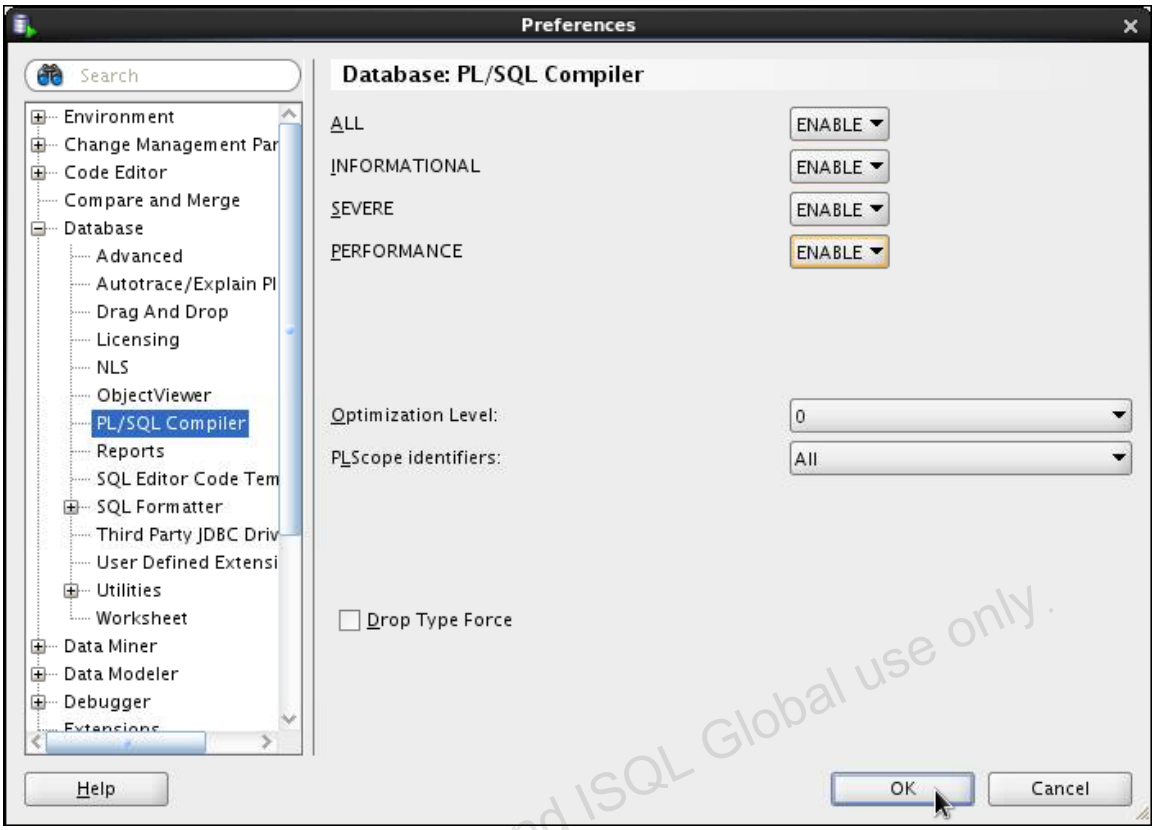


Note

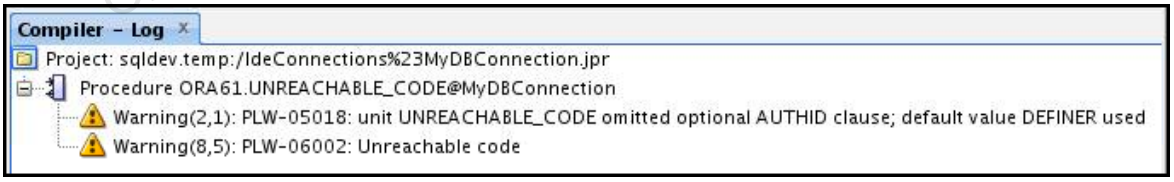
- If the procedure is not displayed in the Navigation tree, click the Refresh icon on the Connections tab.
- Make sure your Messages – Log tab is displayed (select View > Log from the menu bar).

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.

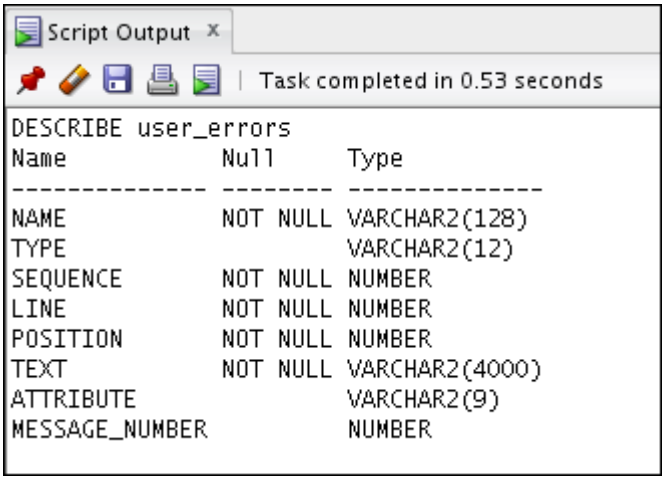


Note: If you get the following two warnings in SQL Developer, it is expected in some versions of SQL Developer. If you do get the following warnings, it is because your version of SQL Developer still uses the Oracle 11*g* database deprecated `PLSQL_DEBUG` parameter.

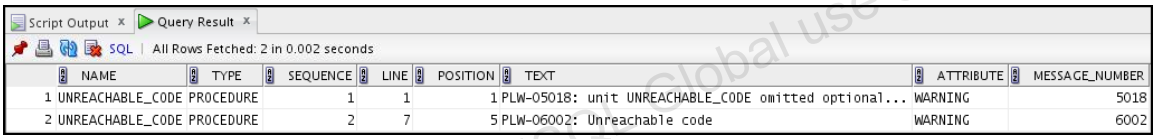
```
Warning(1):PLW-06015:parameter PLSQL_DEBUG is deprecated ; use
PLSQL_OPTIMIZE_LEVEL=1

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
```




```
SELECT *
FROM user_errors;
```



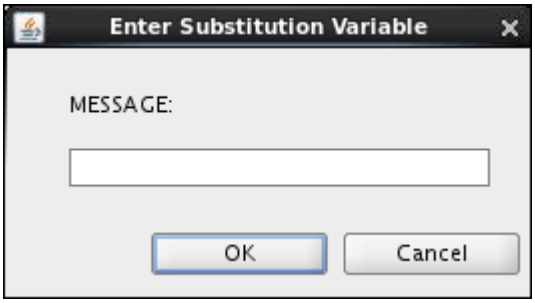
...

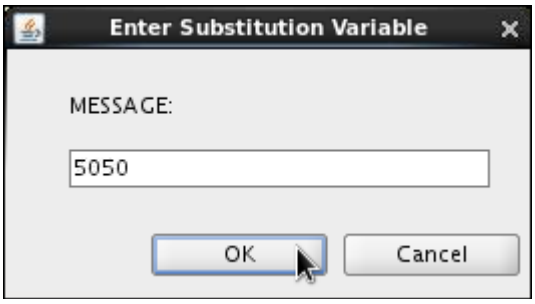
Note: The output was displayed on the Results tab because we used the F9 key to execute the `SELECT` statement. The results of the `SELECT` statement might be different depending on the amount of errors you had in your session.

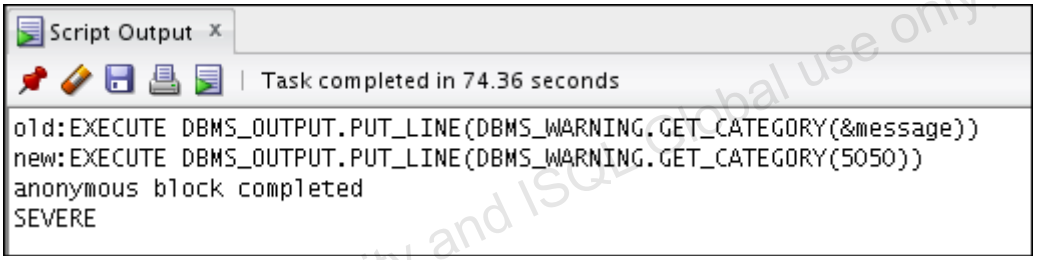
9. Create a script named `warning_msgs` that uses the `EXECUTE DBMS_OUTPUT` and the `DBMS_WARNING` packages to identify the categories for the following compiler-warning message numbers: 5050, 6075, and 7100. Enable `SERVEROUTPUT` before running the script.

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.

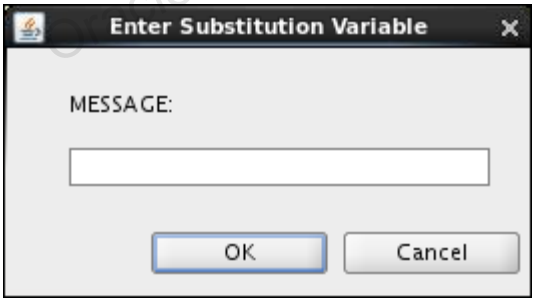
```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```

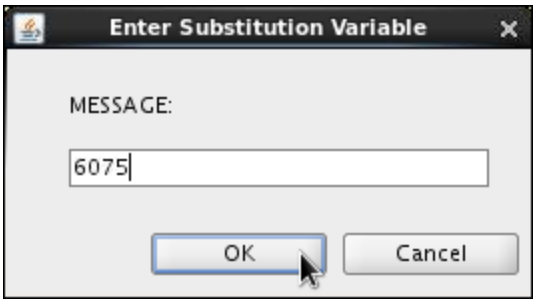


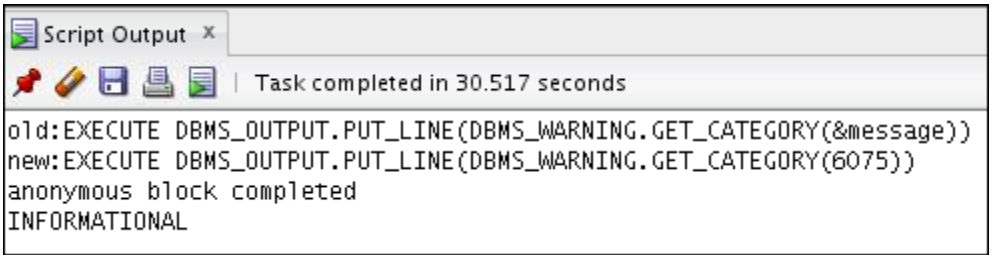




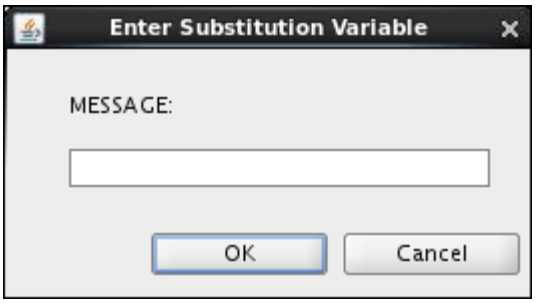

```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```

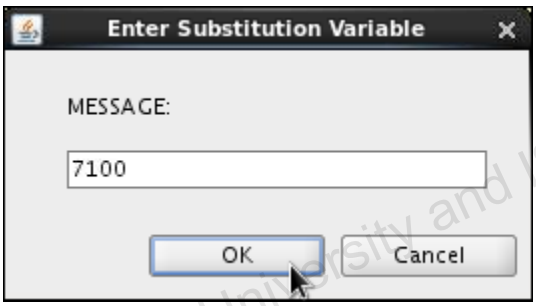


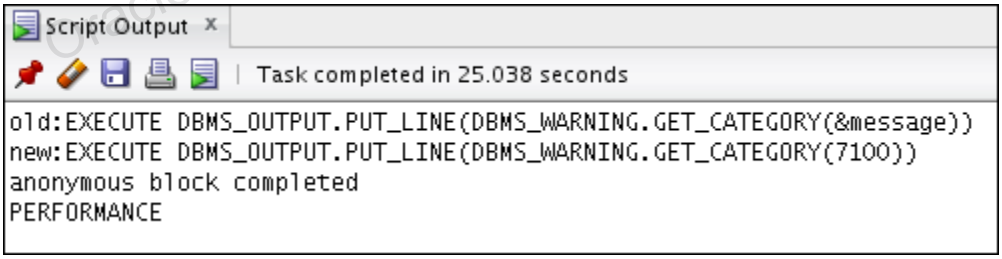





```
EXECUTE
DBMS_OUTPUT.PUT_LINE(DBMS_WARNING.GET_CATEGORY(&message));
```







Practices for Lesson 22: Managing Dependencies

Chapter 22

Practices for Lesson 22: Overview

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.
3. All the required lab and solution files are inside their respective directories under the folder `plpu`, located at `/home/oracle/labs/plpu/`

Practice 22-1: 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 22-1: 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.

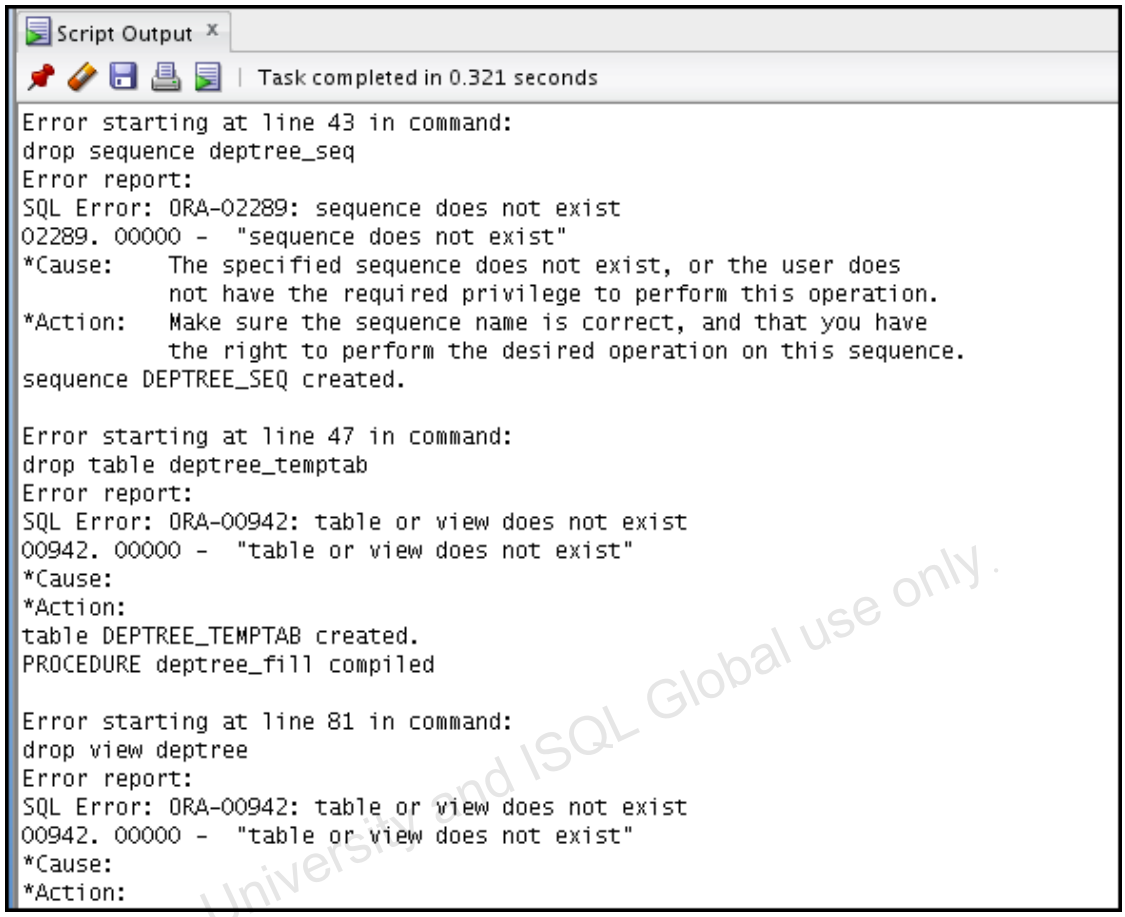
```
Rem
Rem $Header: utldtree.sql,v 3.2 2012/11/21 16:24:44 RKOOI Stab $
Rem
Rem  Copyright (c) 1991 by Oracle Corporation
Rem   NAME
Rem   deptree.sql - Show objects recursively dependent on
Rem   given object
Rem   DESCRIPTION
Rem   This procedure, view and temp table will allow you to see
Rem   all 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;
  COMMITT;
```

```
  SELECT object_id INTO obj_id FROM all_objects
    WHERE owner = upper(deptree_fill.schema)

AND   object_name  = upper(deptree_fill.name)
    AND   object_type  = upper(deptree_fill.type);
  INSERT INTO deptree_temptab
    VALUES(obj_id, 0, 0, 0);
  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
of REM table x$kglxs.

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$kglob c,
      x$kglxs a
    WHERE d.object_id = o.obj#
    AND   o.name = g.kglnaobj
    AND   o.owner# = u.user#
    AND   u.name = g.kglnaown
    AND   g.kglhdadr = k.kglrfhdl
    AND   k.kglhdadr = a.kglhdadr   -- make sure it is not a
transitive
    AND   k.kgldepno = a.kglxsdep   -- reference, but a direct
one
    AND   k.kglhdadr = c.kglhdadr
    AND   c.kglhdnsp = 0 -- a cursor
/

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 */
/
```



```
> 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$kglxs.

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$kglob c,
      x$kglxs a
    where d.object_id = o.obj#
    and   o.name = g.kglnaobj
    and   o.owner# = u.user#
    and   u.name = g.kglnaown
    and   g.kglhdadr = k.kglrfhdl
    and   k.kglhdadr = a.kglhdadr   /* make sure it is not a transitive */
    and   k.kgldepno = a.kglxsdep   /* reference, but a direct one */
    and   k.kglhdadr = c.kglhdadr
    and   c.kglhdnsp = 0 /* a cursor */
Error at Command Line: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_TEMPTAB dropped.
table DEPTREE_TEMPTAB created.
PROCEDURE deptree_fill compiled
view DEPTREE dropped.
```

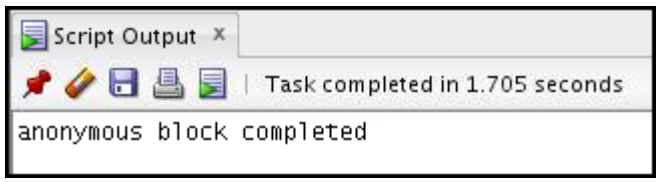
```
> REM This view will succeed if current user is sys.  This view shows
> REM which shared cursors depend on the given object.  If the current
> REM user is not sys, then this view get an error either about lack
> REM of privileges or about the non-existence of table x$kglxs.

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$kglob c,
      x$kglxs a
    where d.object_id = o.obj#
    and   o.name = g.kglnaobj
    and   o.owner# = u.user#
    and   u.name = g.kglnaown
    and   g.kglhdadr = k.kglrfhdl
    and   k.kglhdadr = a.kglhdadr   /* make sure it is not a transitive */
    and   k.kgldepno = a.kglxsdep   /* reference, but a direct one */
    and   k.kglhdadr = c.kglhdadr
    and   c.kglhdnsp = 0 /* a cursor */
```

```
Error at Command Line: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.

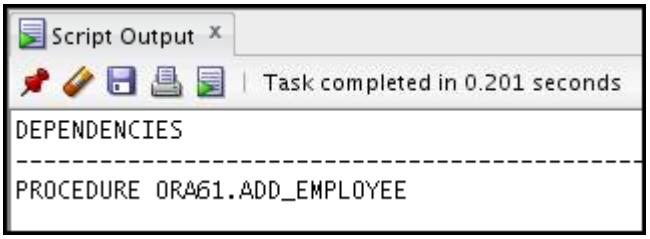
```
EXECUTE deptree_fill('PROCEDURE', USER, 'add_employee')
```



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.

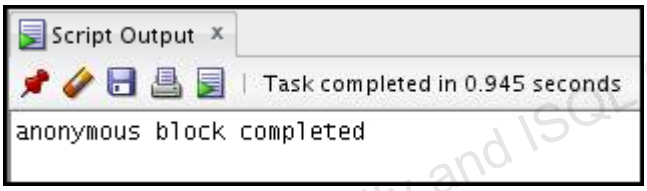
```
SELECT * FROM IDEPTREE;
```



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.

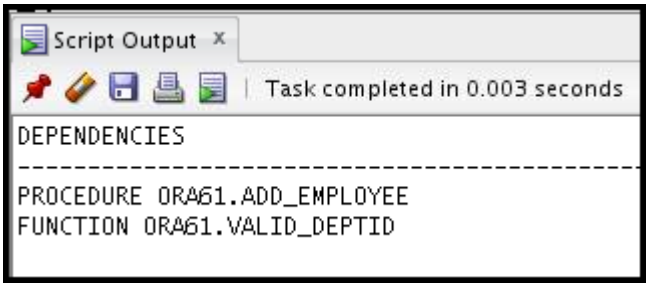
```
EXECUTE deptree_fill('FUNCTION', USER, 'valid_deptid')
```



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;
```

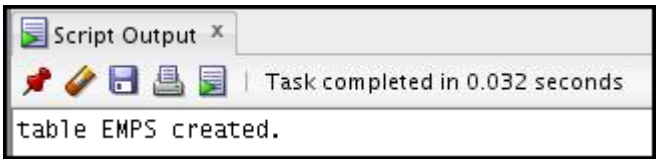


If you have time, complete the following exercise:

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;
```

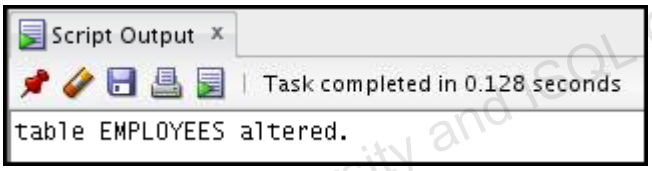


Note: Please ignore the error message, if any while executing the `CREATE` statement.

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 (totsal 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

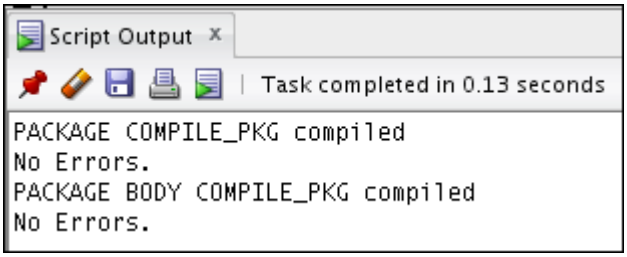
Note: Please ignore the difference in the screenshot.

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
```

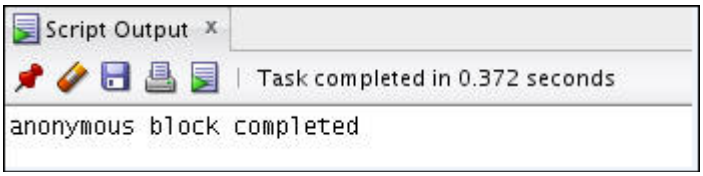
```
    RETURN NULL;
  END;

  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
```

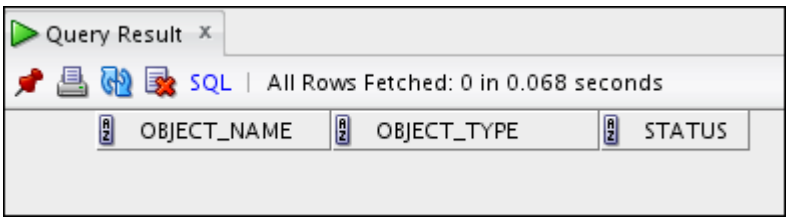


Note: If you come across an error message in the screenshot, please ignore. The procedure would have been compiled.

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';
```



Note: Compare this output to the output in step 2(c). You see that all the objects from the previous screenshot are now valid.

Practices for Lesson 23: Oracle Cloud Overview

Chapter 23

Practices for Lesson 23: Overview

Practice Overview

There is no hands-on practice for this lesson. However you have:

- Steps to request Oracle Database Cloud Service Trial Account
- Getting Started with Oracle Public Cloud DBaaS demonstration

Practice 23-1: Requesting an Oracle Cloud Trial Account

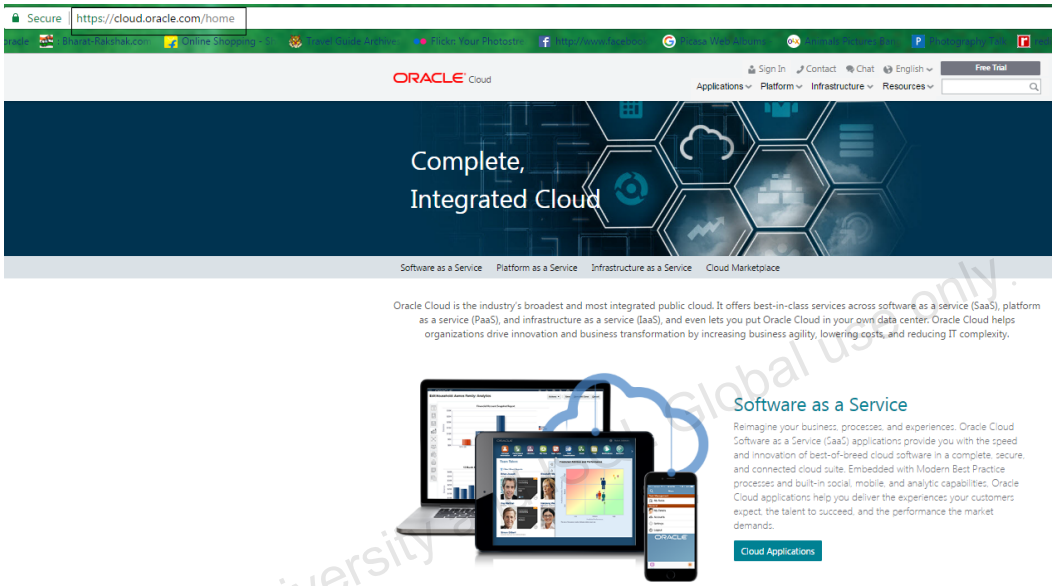
Overview

In this practice, you get an overview on how to request and activate an Oracle Cloud Trial account.

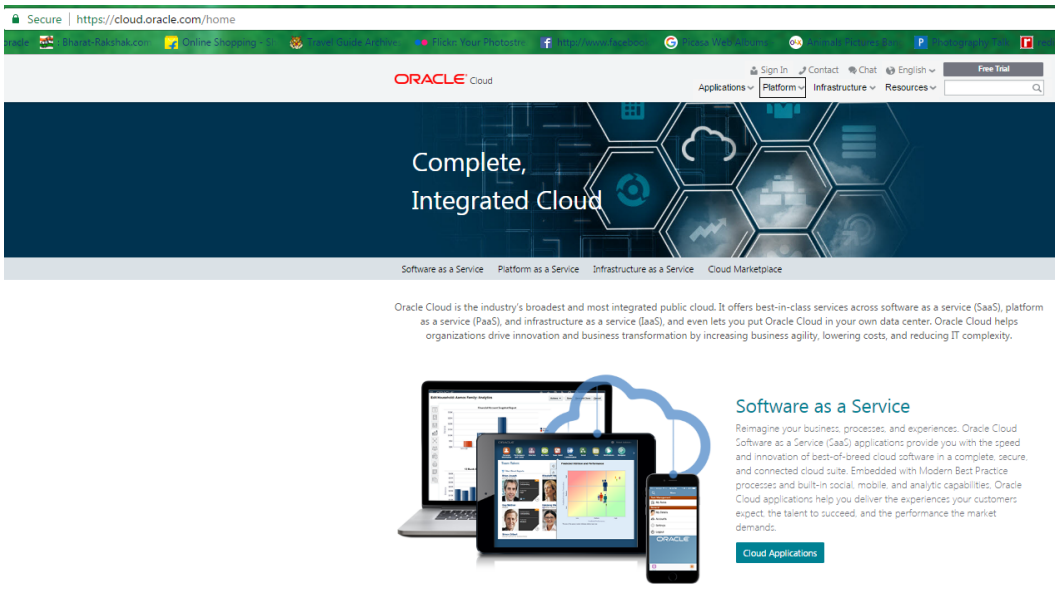
Tasks

A. Request a trial subscription:

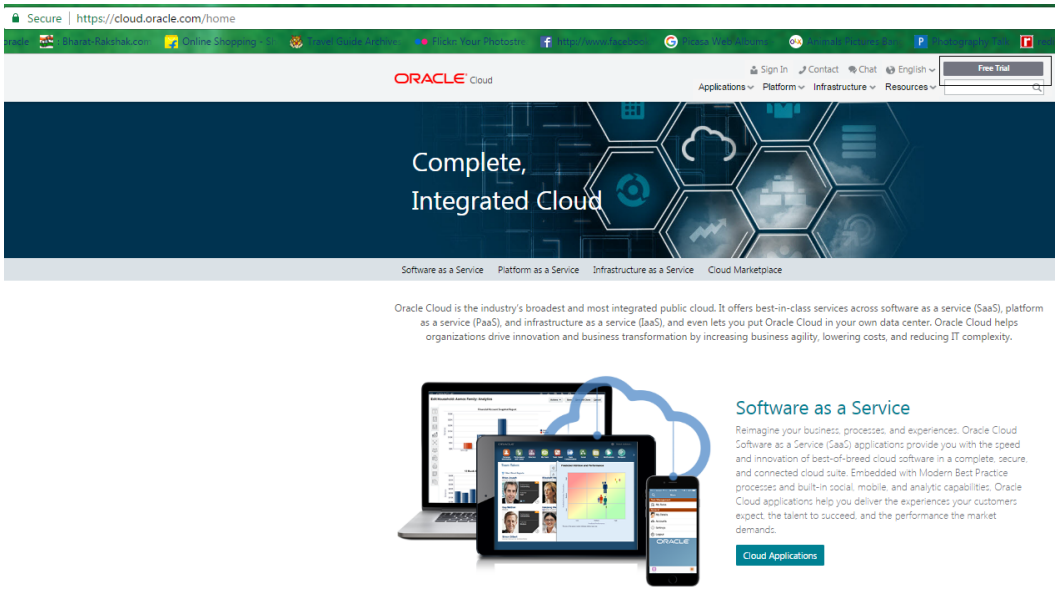
1. Open your web browser and go to the Oracle Cloud website: http://cloud.oracle.com



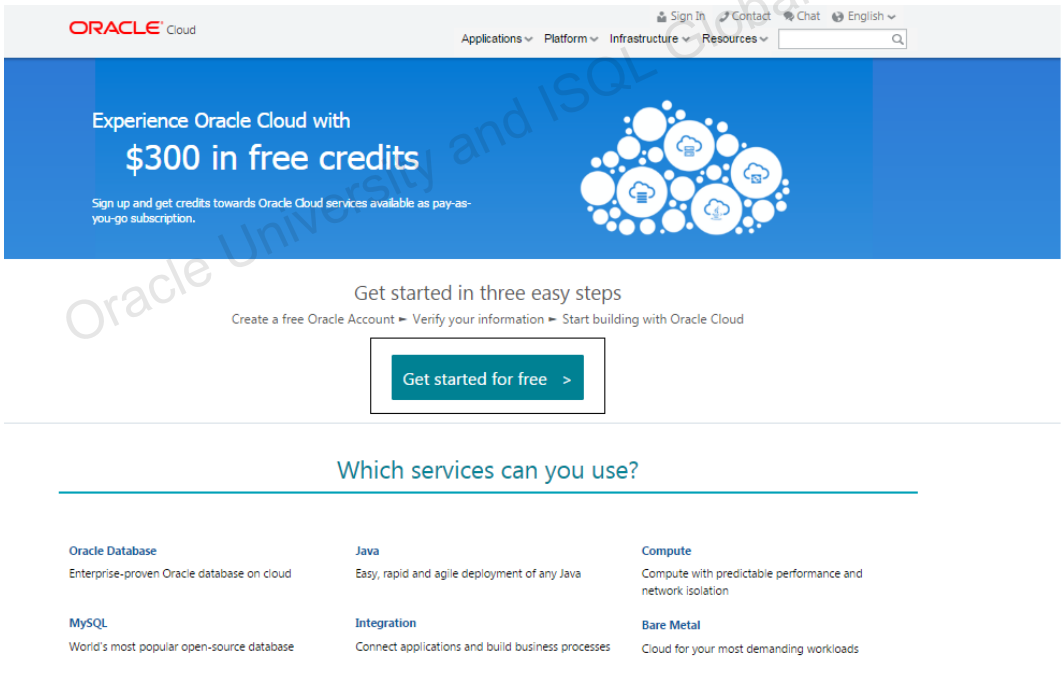
2. Select one of the service category tabs. For example, Applications or Platform.



3. Click **Free Trial**. The page lists the services that have free subscriptions.



4. Click **Get started for free**.



5. Click **Try It** to setup a free trial account.

6. Select one of the following options to continue:
   - If you already have an Oracle.com account, enter your single sign-on (SSO) user name and password, and click **Sign In**. The Sign Up for a Trial Subscription wizard opens.

     Note that if you are already signed in to your Oracle.com account, the system does not prompt for your credentials again. The Sign Up for a Trial Subscription wizard opens immediately.
   - If you don't have an Oracle.com account, then click **Sign Up** to register for a free account. Follow the on-screen instructions. Your account gets created and you will receive a confirmation email. Follow the instructions in the email to verify the status of your email address. You can then use your Oracle.com account to register for Oracle Cloud services.

7. Enter the information required to set up your Oracle Cloud account as follows:

Field	Description
First Name, Last Name	Enter your name.
Company, Country	Enter a new company name or select an existing one. Parentheses are allowed in the company name. If you are requesting your first trial, enter the company

Field	Description
	name, and select the country. If you have already requested a trial, the Company field shows a company name by default. You can select an existing company name from the list or you can enter a new company name.
Country Calling Code	The country calling code is automatically selected based on the country you choose from this list. If you select **Other** in this field, then you're prompted to enter the country code.
Mobile Number	Enter a valid mobile number.
Request Code/ Verification Code	You must request a verification code, which will be sent to the mobile number you specified, to verify your identity and complete the trial flow. Click **Request Code**. Enter the code (that you received on the mobile phone) in this field. Verification codes are valid for one-time use.
Service Name /Identity Domain/Account Name	Enter the service name that you want to use for which you are requesting a trial subscription. Oracle Cloud determines your options for the **Identity Domain** field based on the company name and country you entered on the Account Information page. For metered services, this is the account name. You can either create a new identity domain when requesting for metered service trials or use an existing domain. Note that you can't activate metered trials in a domain containing applications such as Oracle Human Capital Management Cloud Service (Oracle HCM) or Oracle Customer Relationship Management Cloud Service (Oracle CRM). • If an identity domain for trial subscriptions does not exist for the company and country you entered, then Oracle Cloud automatically generates and displays a unique name for the identity domain. You cannot change the value. • If an identity domain for trial subscriptions already exists for the company and country you entered, then the Identity Domain field displays the name of an existing domain by default. You can select any existing domain from the list or create a new identity

Field	Description
	domain. • If you create a new identity domain, then Oracle Cloud automatically generates a unique name for the identity domain. You can see the assigned name in the **Identity Domain** field. If you change the domain name, then ensure that you enter a unique domain name in the **Identity Domain** field. If you don't, then you'll get an error message when you try to go to the next step of the workflow. When generating names for identity domains, Oracle Cloud uses either of the following formats: • For metered trial subscriptions: *country**company**nnnnn* • For nonmetered trial subscriptions: *country**company**trialnnnnn* where: • *country*: Standard two-letter abbreviation for the country. • ***company***: • For metered trial subscriptions: Up to the first eight characters of the company name that you specified previously. • For nonmetered trial subscriptions: Up to the first 13 characters of the company name that you specified previously. • *trial*: The word “trial” • *nnnnn*: A 5-digit number, randomly generated. Examples: • For metered trial subscriptions: *usopenbree94621, caopenbree37518* • For nonmetered trial subscriptions: *usopenbreezetrial94621, caopenbreezetrial37518*
Data Jurisdiction	If you are prompted, then select the jurisdiction where you want us to set up your trial service. Data jurisdictions are filtered based on the requested service and subscription

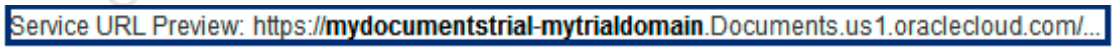
Field	Description
	type. A data jurisdiction is automatically selected based on the Company's country. However, you can select another data jurisdiction to set up your trial, if supported. You can't select a data jurisdiction when you request metered trials. The company's country is mapped to configured data jurisdiction in the system such as: • APAC • EMEA • South America • North America North America is selected by default if the country doesn't belong to the other 3 jurisdictions. If North America isn't available, then the first available data jurisdiction is selected. You can customize the data jurisdiction settings as required.

Note:

The generated service URL preview is displayed at the bottom:

https://<*service_name*>-<*identity_domain_name*>.<*cloud_service*>.<*data_center_name*>.oraclecloud.com

For example:



The generated service URL preview changes as and when you change the service name, the identity domain name, or both. Note that the actual format of service URL preview varies based on the service type.

Read and accept the terms and conditions of the trial agreement before continuing.

Click **Sign up**. The system confirms that Oracle has received your request for a trial.

The Review Summary page displays the following details:

- **Service Information:** Displays the type of service you requested, the name of the service, and the identity domain to which the service belongs.

  In addition, the Review Summary page lists the name and types of other services included with your trial subscription request, if any. For example, the trial subscription for Oracle Java Cloud Service includes Oracle Database Cloud Service, which is created in the same identity domain.

- **Order Information:** Displays the order ID, which is a unique identifier for this order, and the order date. Refer to the order ID whenever you contact us about billing or payment issues.
- **Trial Information:** Displays the trial duration (usually 30 days).

Once your request for a trial subscription gets processed, you will receive an email with the following subject:

`Welcome to Oracle Cloud. Activate your trial.`

The email includes details about your order and your service. It also includes a link to activate your service. Activating the service makes it available for you to use.

Alternatively, you can sign in to My Account at any time to monitor the status of your services, including when a service is ready for activation.

B. Activating a Trial Subscription

When your trial subscription to a service is ready to be activated, you'll get an email from Oracle Cloud. You then use the My Account application to activate your service.

Only trial requests that were processed by Oracle Cloud team can be activated.

You can activate your service from the link in the email or from My Account.

Notes About Activating a Trial Subscription

These are some points to bear in mind when activating a trial subscription.

When you activate a trial subscription, note that:

- If you activate an Oracle Java Cloud Service trial, both Oracle Java Cloud Service and Oracle Database Cloud Service, which is included with the Java trial, will be activated.
- If you view the record for the trial service just after activation, the service is listed but it may not be fully activated yet by Oracle. When fully activated, the status is set to `Active`.
- When the service activation process is complete, the service and its details will be available in My Services, where you can monitor the status and usage of the service.

If you already have trial or paid subscriptions to Oracle Cloud services, you can go to My Services before you activate your service. However, if this request is your first request for a trial or paid subscription, you will not have access to My Services until after the activation process.

Activating Trial Subscriptions from the Email Link

One way to activate a trial is to use the activation link provided in the email from Oracle Cloud. You will receive the email when your service is ready to be activated.

To activate your trial subscription using the email link:

1. Open the Welcome email you received from Oracle Cloud.
2. Click **Activate My Trial**.
   - If you are not signed in, then the Oracle Sign In page opens. Enter your Oracle.com account user name and password, and click **Sign In**. The My Account application opens and displays the details page for the service.
   - If you are already signed in to your Oracle.com account, then the My Account application opens and displays the details page for the service. You don't need to sign in again.

On the details page for the service, note that the system:

- Displays a message that indicates the service was submitted for activation. You'll get another email when the service is active and ready to use.
- Updates the cloud icon to indicate the current status.
- Updates the **Status** field in the Additional Information section to indicate the current status.

Activating Trial Subscriptions From Oracle Cloud

If you activated your trial subscription by using the email link, then you can skip this section.

To open My Account to get your trial subscription activated:

1. Sign in to My Account.
   a. Open your web browser and go to the Oracle Cloud website: `http://cloud.oracle.com`
   b. Click **Sign In** and then click **Sign In to My Account**.
   c. Enter your Oracle.com account user name and password, and click **Sign In**.

The Dashboard page in My Account opens.

2. Navigate to the listing of the trial service that you want to activate.

javatrial6314 (JCS - SaaS Extension)
Subscription: Trial (Activate by 16-Nov-2014 3:13 PM IST)
Data Center: US Commercial 1
Identity Domain: inmycompanytrial80830

Note

- The cloud icon and its hover text indicate that the service hasn't been activated.
- The **Subscription** field specifies the date by which you must activate the trial subscription for this service. If you don't activate the service by the deadline, then Oracle Cloud cancels the subscription.
- The **Activate** button, which appears only if the service needs to be activated, is now available.

Click **Activate**.

Note that the system:

- Displays a message at the top of the page that indicates the service was submitted for activation. You'll get another email when the service is active and ready to use.
- Places the service listing in alphabetic order on the page.
- Updates the cloud icon and the **Subscription** field to indicate that the activation is in progress.

Practice 23-2: Getting Started with Oracle Public Cloud DBaaS demonstration

Overview

In this demonstration, you get an overview on how to create and access your first Database Cloud Service Instance.

Tasks

Review the Getting Started with Oracle Public Cloud DBaaS (http://oukc.oracle.com/public/redir.html?type=player&offid=1957025749) demonstration. This demonstration covers everything that is needed to create and access your first Database Cloud Service Instance.

Tasks include

- Creating a backup container
- Creating SSH Keys
- Creating the database itself