ORACLE University

Integrated Cloud Applications & Platform Services

Custom CW: Oracle Database 12*c* R2 - Six Payment

Activity Guide

X103804GC10

Edition 1.0 | November 2019

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

ORACLE

Table of Contents

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

Chapter 10

Practices for Lesson 10: 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.

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

Overview

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

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

Task

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

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

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

b. Implement the body of `JOBS_PKG` as follows:
   1) Declare a private PL/SQL index-by table called `jobs_tab_type` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.
   2) Declare a private variable called `jobstab` based on the `jobs_tab_type`.
   3) The `INITIALIZE` procedure reads the rows in the `JOBS` table by using a cursor loop, and uses the `JOB_ID` value for the `jobstab` index that is assigned its corresponding row.
   4) The `GET_MINSALARY` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `min_salary` for that element.
   5) The `GET_MAXSALARY` function uses a `p_jobid` parameter as an index to the `jobstab` and returns the `max_salary` for that element.
   6) The `SET_MINSALARY` procedure uses its `p_jobid` as an index to the `jobstab` to set the `min_salary` field of its element to the value in the `min_salary` parameter.
   7) The `SET_MAXSALARY` procedure uses its `p_jobid` as an index to the `jobstab` to set the `max_salary` field of its element to the value in the `max_salary` parameter.

c. Copy the `CHECK_SALARY` procedure from Practice 9, Exercise 1a, and modify the code by replacing the query on the `JOBS` table with statements to set the local `minsal` and `maxsal` variables with values from the `JOBS_PKG` data by calling the appropriate `GET_*SALARY` functions. This step should eliminate the mutating trigger exception.

d. Implement a `BEFORE INSERT OR UPDATE` statement trigger called `INIT_JOBPKG_TRG` that uses the `CALL` syntax to invoke the `JOBS_PKG.INITIALIZE` procedure to ensure that the package state is current before the DML operations are performed.

e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the `IT_PROG` job type by 1,000 in the `JOBS` table. Follow this up with a query on the employees with the `IT_PROG` job type to check the resulting changes. Which employees' salaries have been set to the minimum for their jobs?

3. Because the `CHECK_SALARY` procedure is fired by `CHECK_SALARY_TRG` before inserting or updating an employee, you must check whether this still works as expected.

   a. Test this by adding a new employee using `EMP_PKG.ADD_EMPLOYEE` with the following parameters: (`'Steve'`, `'Morse'`, `'SMORSE'`, and `sal => 6500`). What happens?

   b. To correct the problem encountered when adding or updating an employee:
      1) Create a `BEFORE INSERT OR UPDATE` statement trigger called `EMPLOYEE_INITJOBS_TRG` on the `EMPLOYEES` table that calls the `JOBS_PKG.INITIALIZE` procedure.
      2) Use the `CALL` syntax in the trigger body.

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

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

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

1. Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salaries. Implement this requirement through a package procedure called by a trigger on the JOBS table. When you attempt to update the minimum salary in the JOBS table and try to update the employees' salaries, the CHECK_SALARY trigger attempts to read the JOBS table, which is subject to change, and you get a mutating table exception that is resolved by creating a new package and additional triggers.
   a. Update your EMP_PKG package (that you last updated in Practice 9) as follows:
      1) Add a procedure called SET_SALARY that updates the employees' salaries.
      2) The SET_SALARY procedure accepts the following two parameters: The job ID for those salaries that may have to be updated, and the new minimum salary for the job ID

      Open sol_10.sql script from /home/oracle/labs/plpu/soln directory. Uncomment and select the code under Task 1_a. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown as follows. The newly added code is highlighted in bold letters in the following code box.

```
-- Package SPECIFICATION

CREATE OR REPLACE PACKAGE emp_pkg IS


  TYPE emp_tab_type IS TABLE OF employees%ROWTYPE;

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

  PROCEDURE add_employee(
```

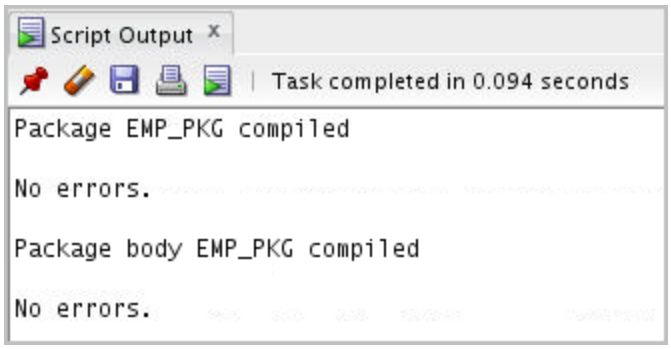
```
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  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


  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;

  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
```

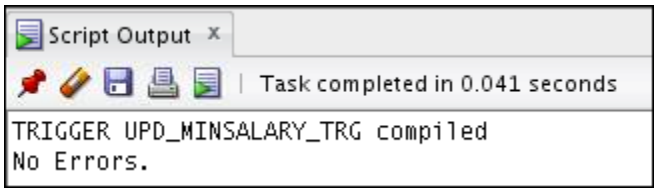
```
      WHERE employee_id = rec_emp.employee_id;
    END LOOP;
  END set_salary;

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



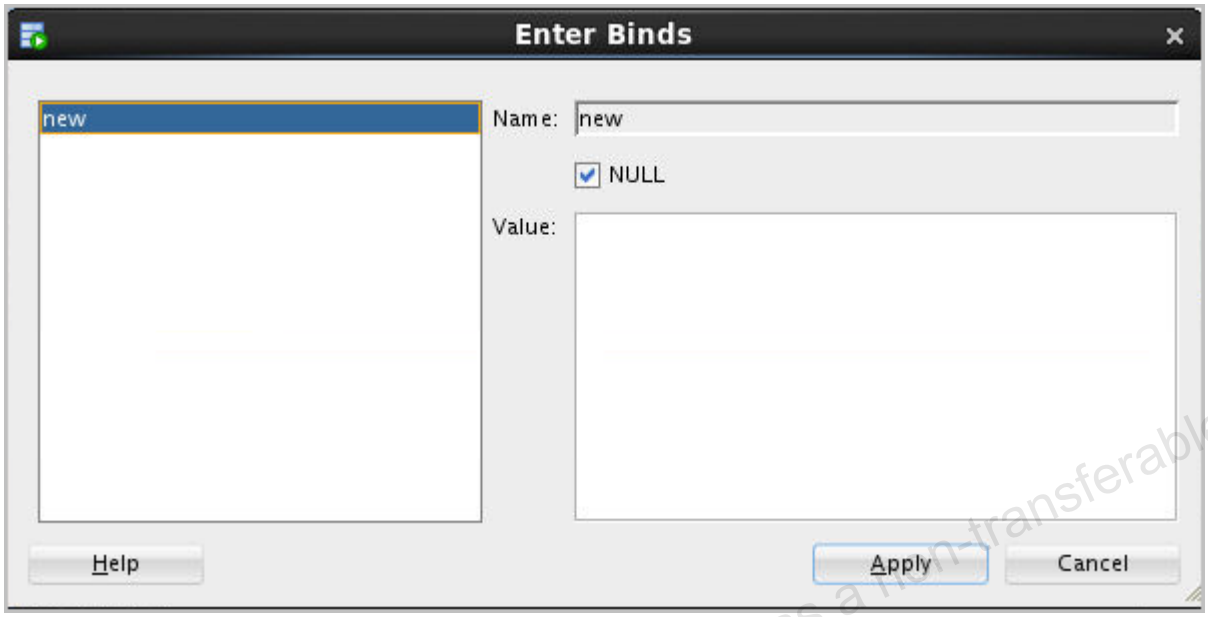
b. Create a row trigger named `UPD_MINSALARY_TRG` on the `JOBS` table that invokes the `EMP_PKG.SET_SALARY` procedure, when the minimum salary in the `JOBS` table is updated for a specified job ID.

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
```



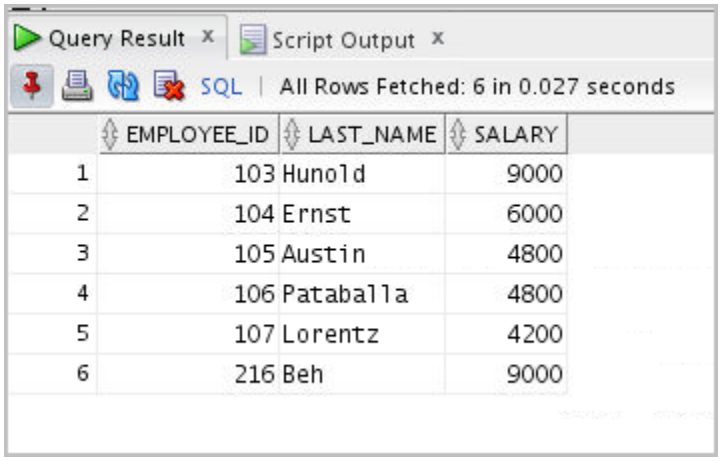
Note: The trigger compilation might ask for values of bind variables while compiling. You may encounter a wizard as the one below. Click Apply.

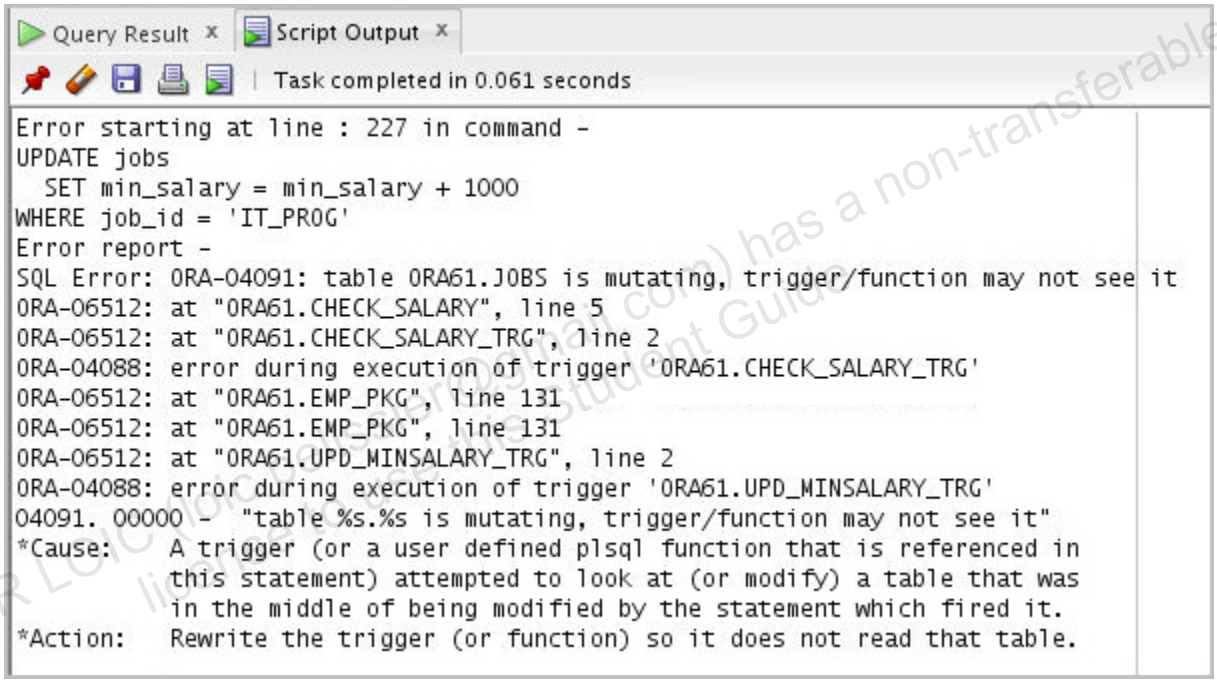


c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their `JOB_ID` is `'IT_PROG'`. Then, update the minimum salary in the `JOBS` table to increase it by $1,000. What happens?

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';
```





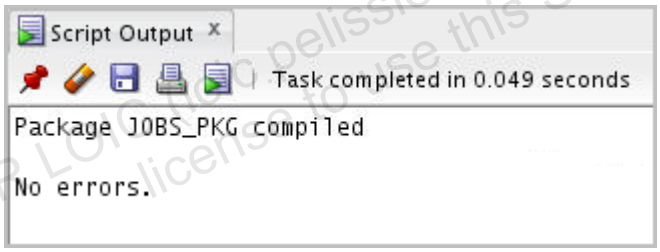
The update of the `min_salary` column for job `'IT_PROG'` fails because the `UPD_MINSALARY_TRG` trigger on the `JOBS` table attempts to update the employees' salaries by calling the `EMP_PKG.SET_SALARY` procedure. The `SET_SALARY` procedure causes the `CHECK_SALARY_TRG` trigger to fire (a cascading effect). 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
```



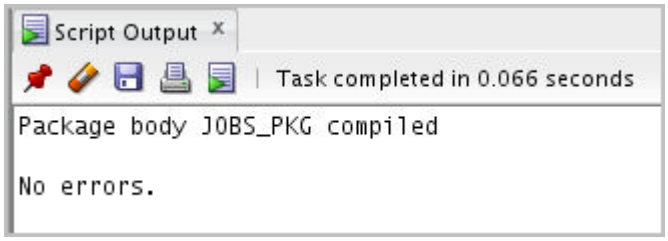
b. Implement the body of `JOBS_PKG` as follows:
   1) Declare a private PL/SQL index-by table called `jobs_tab_type` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.
   2) Declare a private variable called `jobstab` based on the `jobs_tab_type`.
   3) The `INITIALIZE` procedure reads 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;
```

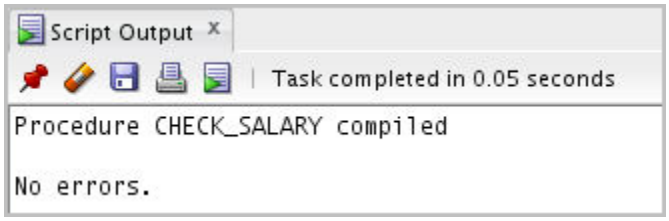
```
END jobs_pkg;
/
SHOW ERRORS
```



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.

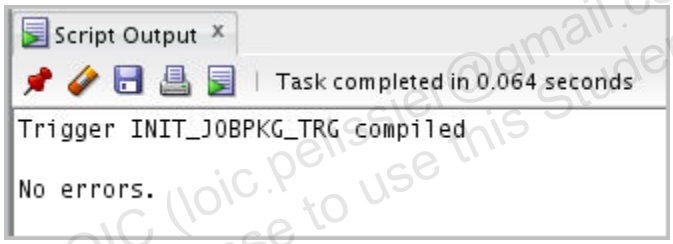
```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN

  -- 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';
```

Query Result | Script Output | Query Result 1

SQL | All Rows Fetched: 6 in 0.003 seconds

	EMPLOYEE_ID	LAST_NAME	SALARY
1	103	Hunold	9000
2	104	Ernst	6000
3	105	Austin	4800
4	106	Pataballa	4800
5	107	Lorentz	4200
6	216	Beh	9000

Query Result | Script Output | Query Result 1

Task completed in 0.265 seconds

```
1 row updated.

>>Query Run In:Query Result 1
```

Query Result | Script Output | Query Result 1

SQL | All Rows Fetched: 6 in 0.002 seconds

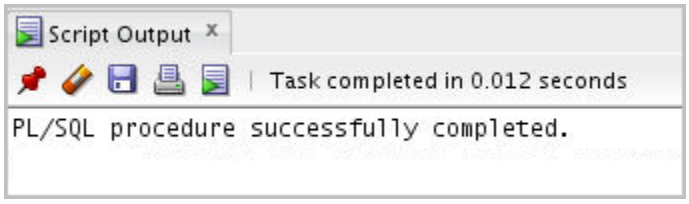
	EMPLOYEE_ID	LAST_NAME	SALARY
1	103	Hunold	9000
2	104	Ernst	6000
3	105	Austin	5000
4	106	Pataballa	5000
5	107	Lorentz	5000
6	216	Beh	9000

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.

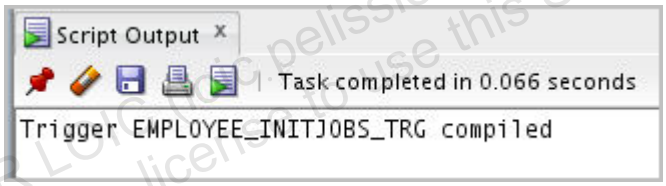
```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal
=> 6500)
```



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.

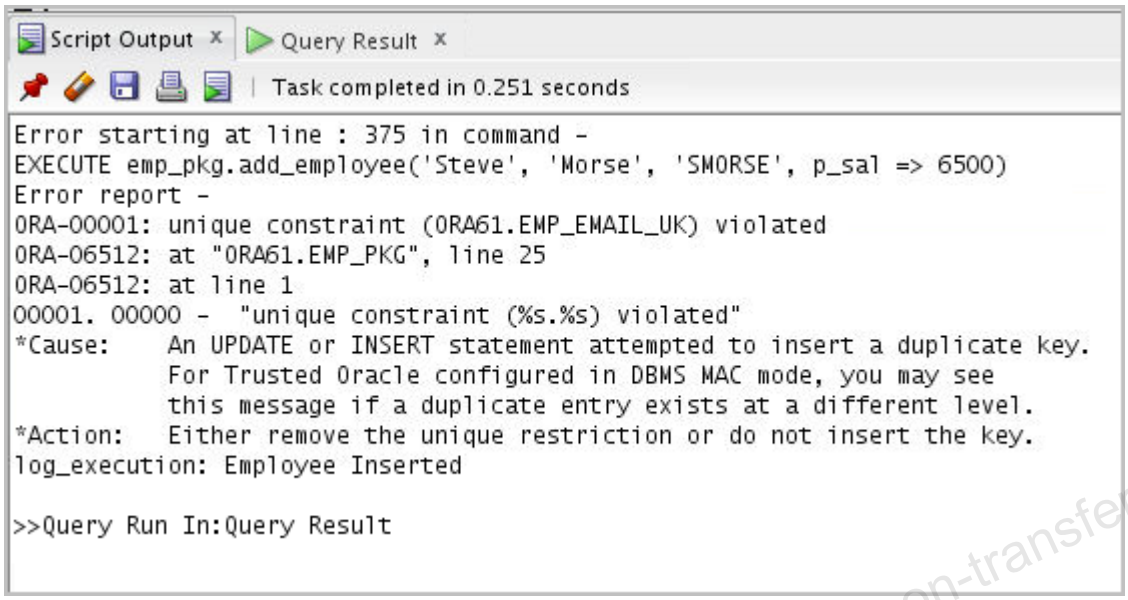
```
CREATE TRIGGER employee_initjobs_trg
BEFORE INSERT OR UPDATE OF job_id, salary ON employees
CALL jobs_pkg.initialize
/
```

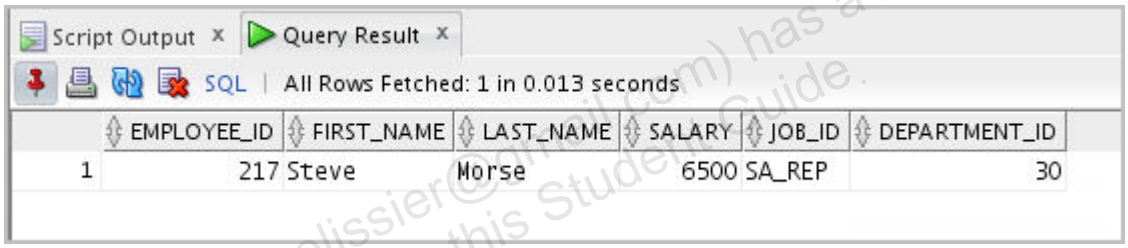


c. Test the trigger by adding employee Steve Morse again. Confirm the inserted record in the `EMPLOYEES` table by displaying the employee ID, first and last names, salary, job ID, and department ID.

```
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 13: Managing Dependencies

Chapter 13

Practices for Lesson 13: 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_13.sql` script.
2. If you missed a step in a practice, please run the appropriate solution script for that practice step before proceeding to the next step or the next practice.

Practice 13-1: 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_13.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.

      Note: The view `sys.deptree` will not be created if you are not a sys user. If you are not a `sys` user `deptree` view is the alternative and that will be created.

   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 8 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 13-1: 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.

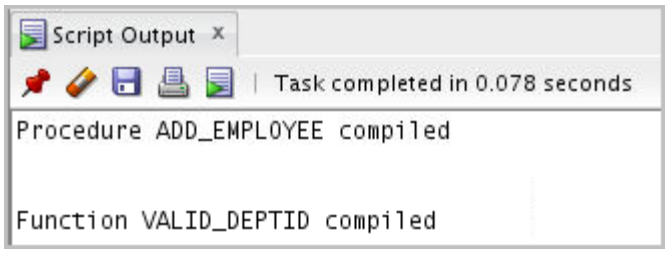
1. a.

   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." 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;
/
```



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.

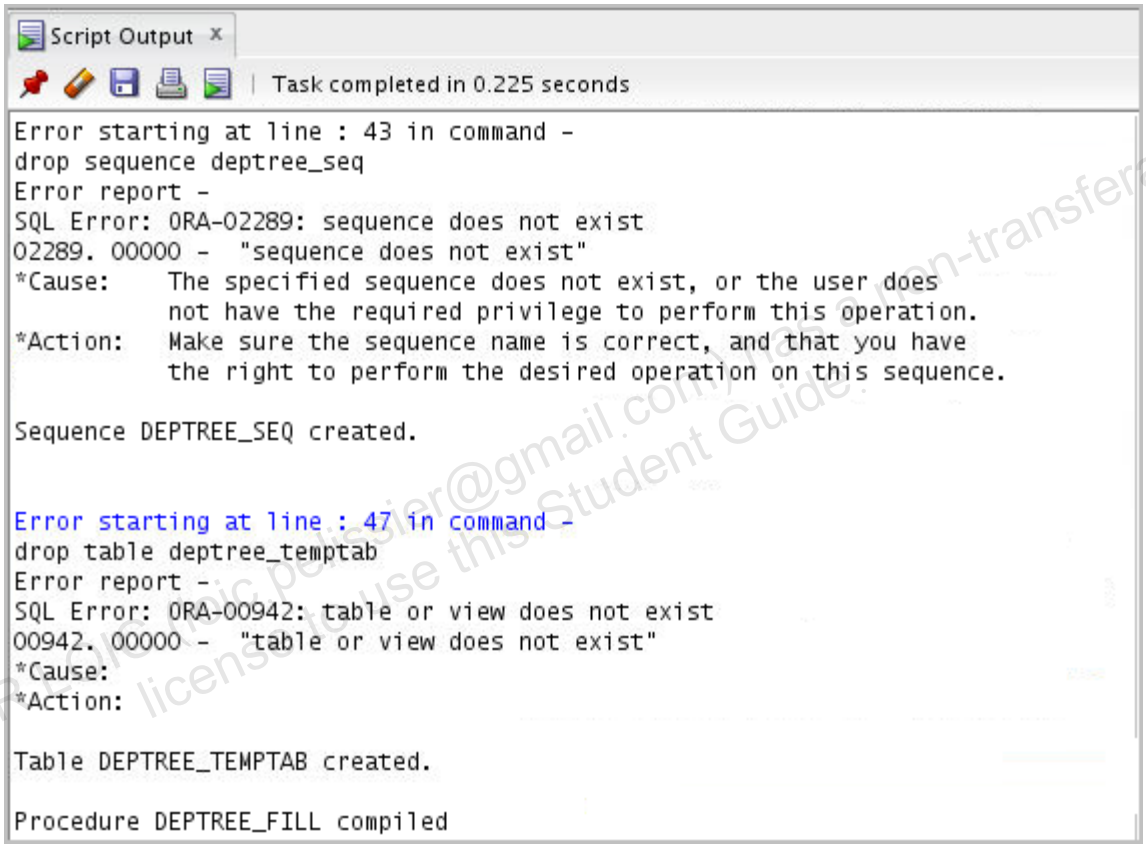
```
Rem
Rem $Header: utldtree.sql,v 3.2 2012/11/21 16:24:44 RKOOI Stab $
Rem
Rem  Copyright (c) 1991 by Oracle Corporation
Rem   NAME
Rem   deptree.sql - Show objects recursively dependent on
Rem   given object
Rem   DESCRIPTION
Rem   This procedure, view and temp table will allow you to see
Rem   all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

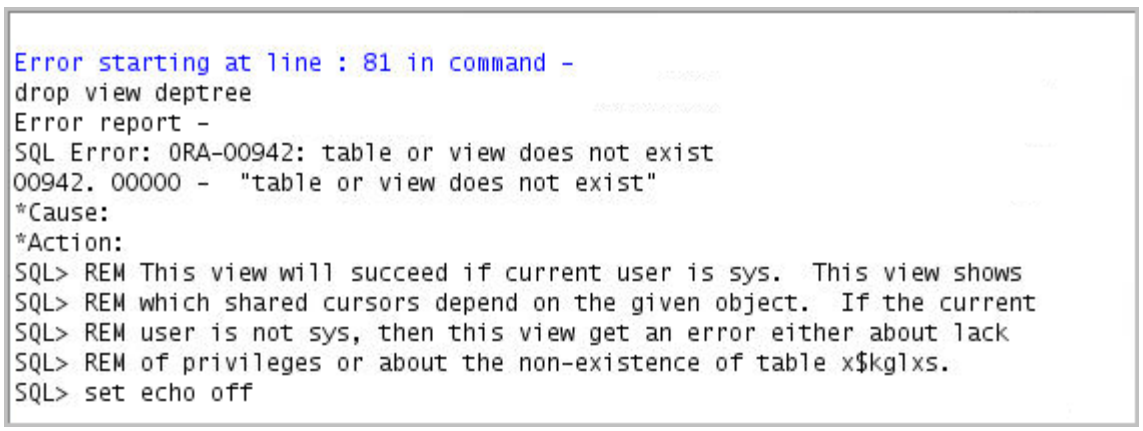
```
Rem   execute deptree_fill('package body', 'scott',
Rem   'accts_payable');
Rem   select * from deptree order by seq#;
Rem
Rem   A prettier way to display this information than
Rem   select * from deptree order by seq#;
Rem is
Rem   select * from ideptree;
Rem   This shows the dependency relationship 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 */
/
```





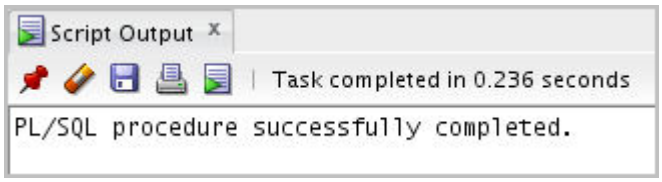
```
Error starting at line : 92 in command -
create view sys.deptree
  (nested_level, type, schema, name, seq#)
as
  select d.nest_level, o.object_type, o.owner, o.object_name, d.seq#
  from deptree_temptab d, dba_objects o
  where d.object_id = o.object_id (+)
union all
  select d.nest_level+1, 'CURSOR', '<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 report -
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:
```

```
SQL> REM This view will succeed if current user is not sys.  This view
SQL> REM does *not* show which shared cursors depend on the given object.
SQL> REM If the current user is sys then this view will get an error
SQL> REM indicating that the view already exists (since prior view create
SQL> REM will have succeeded).
SQL> set echo off

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.
```

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.

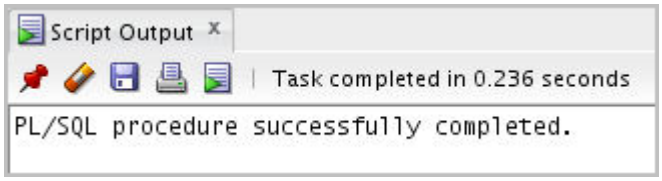
Uncomment and select the code under task 1_c. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

```
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.

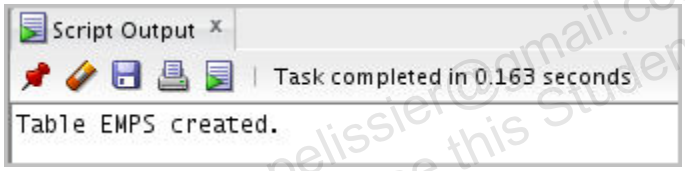
```
SELECT * FROM IDEPTREE;
```

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;
      ```

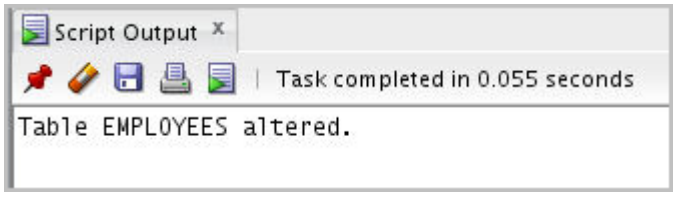
      

      Note: Please ignore the error message, if any while executing the `CREATE` statement.

   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	SECURE_EMPLOYEES	TRIGGER	INVALID
2	FORWARD_PKG	PACKAGE BODY	INVALID
3	LOG_EMPLOYEE	TRIGGER	INVALID
4	GET_EMPLOYEE	PROCEDURE	INVALID
5	DISPLAY	PROCEDURE	INVALID
6	RAISE_SAL	PROCEDURE	INVALID
7	DISPLAY_NEW_SAL	PROCEDURE	INVALID
8	QUERY_EMP	PROCEDURE	INVALID
9	EMP_MAILS	VIEW	INVALID

Note: Please ignore the difference in the screenshot.

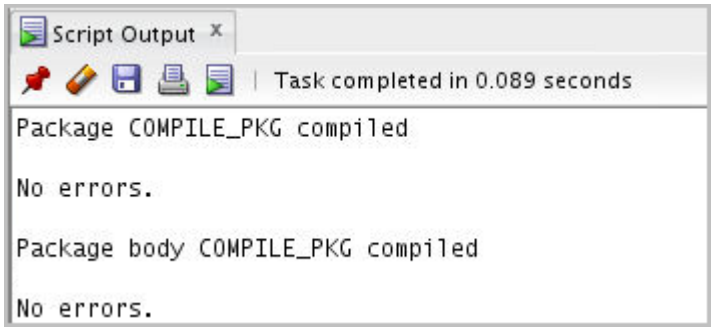
d. In the `compile_pkg` (created in Practice 8 of the lesson titled “Using Dynamic SQL”), add a procedure called `recompile` that recompiles all invalid procedures, functions, and packages in your schema. Use Native Dynamic SQL to alter the invalid object type and compile it.

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

CREATE OR REPLACE PACKAGE BODY compile_pkg IS

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

  FUNCTION get_type(name VARCHAR2) RETURN VARCHAR2 IS
    proc_type VARCHAR2(30) := NULL;
  BEGIN
     -- The ROWNUM = 1 is added to the condition
     -- to ensure only one row is returned if the
     -- name represents a PACKAGE, which may also
     -- have a PACKAGE BODY. In this case, we can
```

```
   -- only compile the complete package, but not
   -- the specification or body as separate
   -- components.
SELECT object_type INTO proc_type
FROM user_objects
WHERE object_name = UPPER(name)
 AND ROWNUM = 1;
  RETURN proc_type;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN NULL;
END;

PROCEDURE make(name VARCHAR2) IS
  stmt       VARCHAR2(100);
  proc_type  VARCHAR2(30) := get_type(name);
BEGIN
  IF proc_type IS NOT NULL THEN
    stmt := 'ALTER '|| proc_type ||' '|| name ||' COMPILE';

execute(stmt);
  ELSE
    RAISE_APPLICATION_ERROR(-20001,
       'Subprogram '''|| name ||''' does not exist');
  END IF;
END make;

PROCEDURE recompile IS
  stmt VARCHAR2(200);
  obj_name user_objects.object_name%type;
  obj_type user_objects.object_type%type;
BEGIN
  FOR objrec IN (SELECT object_name, object_type
                 FROM user_objects
                 WHERE status = 'INVALID'
                 AND object_type <> 'PACKAGE BODY')
  LOOP
    stmt := 'ALTER '|| objrec.object_type ||' '||
                 objrec.object_name ||' COMPILE';
    execute(stmt);
  END LOOP;
END recompile;
```

```
END compile_pkg;
/
```

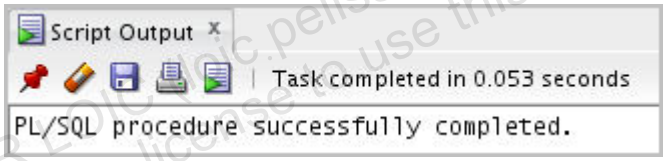
SHOW ERRORS



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
```

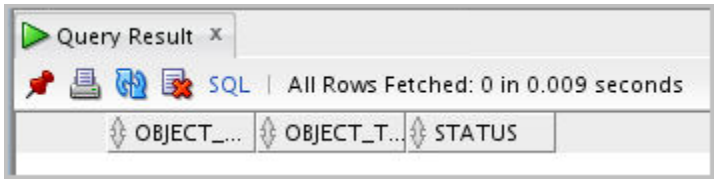


Note: If you come across an error message in the screenshot, please ignore. The procedure would have been compiled.

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 6: Working with JSON Data

Chapter 6

Practices for Lesson 6: Overview

Lesson Overview

This practice covers the following topics:

- Creating JSON object column types in tables and inserting data in it.
- Generating JSON data from data in tables.
- Working with JSON data in PL/SQL blocks.

Practice 6-1: JSON Data in Tables

Overview

In this practice, you create a table with JSON columns and set constraints on the column. Insert data into the JSON data column.

Use the OE connection to complete this practice.

Task

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

Column Name	Data Type	Length
SALES_REP_ID	NUMBER	6
SALES_REP_FNAME	VARCHAR2	20
SALES_REP_LNAME	VARCHAR2	20
ORDER_DETAILS	VARCHAR2	4000

This table contains the information of all the sales representatives who are handling various orders as indicated in table ORDERS.

Note: The SALES_REP_ID column in the ORDERS table references the EMPLOYEES table in the HR schema.

b. Define IS_JSON constraint on the ORDER_DETAILS column.

c. Populate the SALES_REPORT table with first name and last name data of the sales representatives referred to in the ORDERS table.

d. Update the SALES_REPORT table, to populate the ORDER_DETAILS column. Insert a group of JSON objects into the column, where each JSON object has details of the order handled by the sales representative. The JSON object should have information on order_id with the corresponding customer_id and order value.

Practice 6-2: JSON Data in PL/SQL Blocks

Create a procedure `SALES_DATA` that initializes a nested JSON object with properties – `sales_rep_fname`, `sales_rep_lname`, `order_details`. The order_details is further a JSON object with properties – `customer_id`, `order_value`. Access the `order_value` value of the JSON object using `get_number` method and display it to the output.

Solution 6-1: JSON Data in Tables

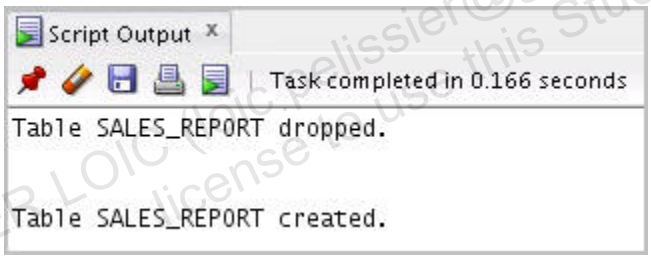
In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS_LOB package to populate the table and manipulate the data.

Use your OE connection.

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

Column Name	Data Type	Length
SALES_REP_ID	NUMBER	6
SALES_REP_FNAME	VARCHAR2	20
SALES_REP_LNAME	VARCHAR2	20
ORDER_DETAILS	VARCHAR2	4000

```
DROP TABLE sales_report
/
CREATE TABLE sales_report(sales_rep_id number(6,0) PRIMARY
KEY,sales_rep_fname VARCHAR2(20) , sales_rep_lname VARCHAR2(20),
order_details VARCHAR2(4000));
```

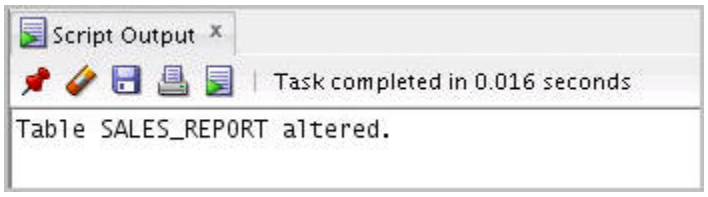


The drop command would return an error if the SALES_REPORT table is not already created.

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

b. Define IS_JSON constraint on the ORDER_DETAILS column.

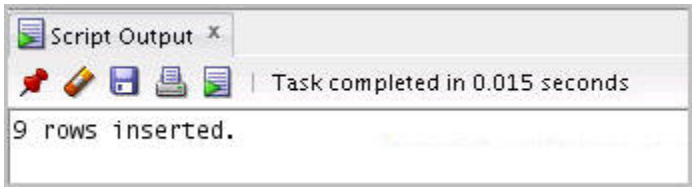
```
ALTER TABLE sales_report ADD CONSTRAINT ensure_json CHECK
(order_details IS JSON);
```



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

c. Populate the `SALES_REPORT` table with first name and last name data of the sales representatives referred to in the `ORDERS` table.

```
INSERT INTO sales_report(sales_rep_id, sales_rep_fname,
sales_rep_lname)
SELECT DISTINCT o.sales_rep_id, emp.first_name, emp.last_name
FROM hr.employees emp, oe.orders o
WHERE emp.employee_id = o.sales_rep_id;
```



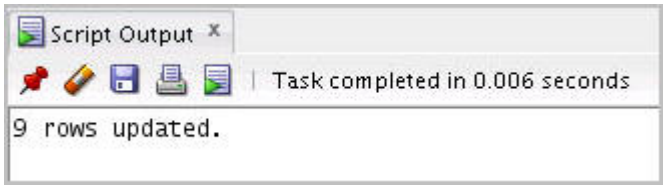
d. Update the `SALES_REPORT` table, to populate the `ORDER_DETAILS` column. Insert a group of JSON objects into the column, where each JSON object has details of the order handled by the sales representative. The JSON object should have information on `order_id` with the corresponding `customer_id` and order value.

```
SELECT * FROM sales_report;
```

Query Result | SQL | All Rows Fetched: 9 in 0.006 seconds

	SALES_REP_ID	SALES_REP_FNAME	SALES_REP_LNAME	ORDER_DETAILS
1	155	Oliver	Tuvault	(null)
2	156	Janette	King	(null)
3	163	Danielle	Greene	(null)
4	153	Christopher	Olsen	(null)
5	161	Sarath	Sewall	(null)
6	160	Louise	Doran	(null)
7	154	Nanette	Cambrault	(null)
8	158	Allan	McEwen	(null)
9	159	Lindsey	Smith	(null)

```
UPDATE sales_report sr
SET order_details = (SELECT JSON_ARRAYAGG(JSON_OBJECT(
                     'order_id' VALUE o.order_id,
                     'customer_id' VALUE o.customer_id,
                     'order_value' VALUE o.order_total))
FROM  oe.orders o
WHERE o.sales_rep_id = sr.sales_rep_id
group by sales_rep_id);
commit;
```

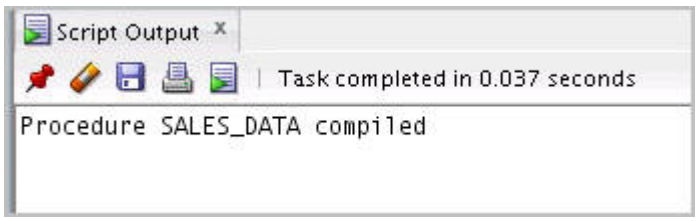


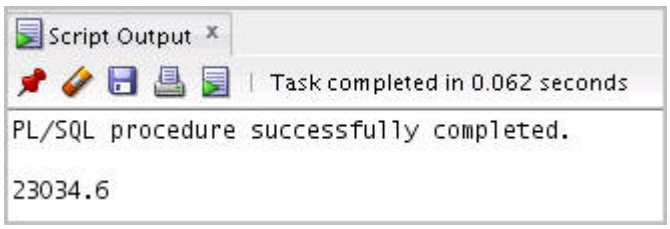
Alternatively, you can run the solution for task 4_a, 4_b from `sol_06.sql`.

Solution 6-2: JSON Data in PL/SQL Blocks

Create a procedure `SALES_DATA` that initializes a nested JSON object with properties – `sales_rep_fname`, `sales_rep_lname`, `order_details`. The order_details is further a JSON object with properties – `customer_id`, `order_value`. Access the `order_value` value of the JSON object using `get_number` method and display it to the output.

```
CREATE OR REPLACE PROCEDURE sales_data AS
input_data JSON_OBJECT_T;
ord_det JSON_OBJECT_T;
o_val number;
BEGIN
input_data := new JSON_OBJECT_T('{"sales_rep_fname" : "Janette",
                                 "sales_rep_lname" : "King",
                                 "order_details" : {
"customer_id" : "106","order_value" : "23034.6"}}');

ord_det := input_data.get_object('order_details');
o_val := ord_det.get_number('order_value');
DBMS_OUTPUT.PUT_LINE(o_val);
END sales_data;

EXECUTE sales_data;
```





Practices for Lesson 7: Using Advanced Interface Methods

Chapter 7

Practices for Lesson 7: Overview

Lesson Overview

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

Practice 7-1: Using Advanced Interface Methods

Overview

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

Use the OE connection.

Task

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

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

}
```

1. A shared library file called calc_tax.so was created for you. Copy the file from the /home/oracle/labs/labs directory into your /u01/app/oracle/product/12.2.0/dbhome_1/bin directory.
2. Connect to the sys connection, and create the alias library object. Name the library object c_code and define its path as:

```
CREATE OR REPLACE LIBRARY c_code
AS '/u01/app/oracle/product/12.2.0/dbhome_1/bin/calc_tax.so';
/
```

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

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

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

Calling Java from PL/SQL

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

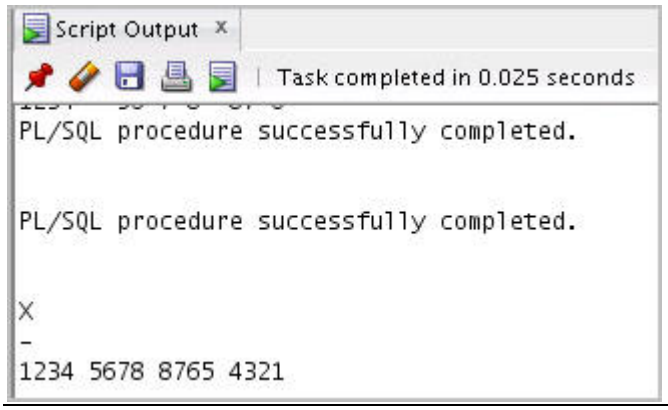
```
public class FormatCreditCardNo
{
public static final void formatCard(String[] cardno)
{
int count=0, space=0;
String oldcc=cardno[0];
String[] newcc= {""};
while (count<16)
{
newcc[0]+= oldcc.charAt(count);
space++;
if (space ==4)
{  newcc[0]+=" "; space=0;  }
count++;
}
cardno[0]=newcc [0];
}
}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

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

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

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

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



Solution 7-1: Using Advanced Interface Methods

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

Use the `OE` connection.

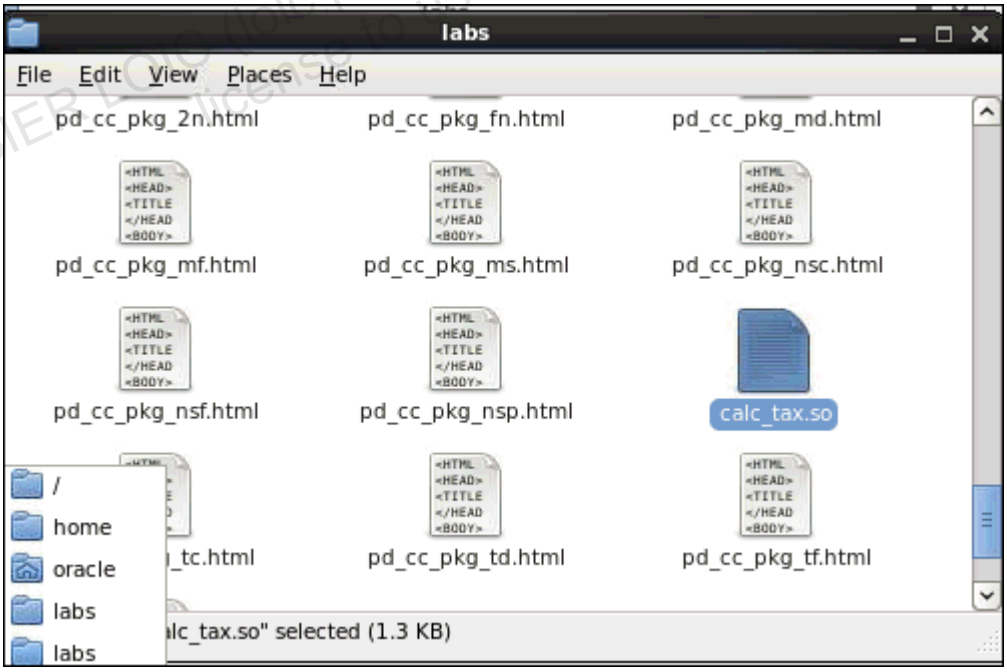
Using External C Routines

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

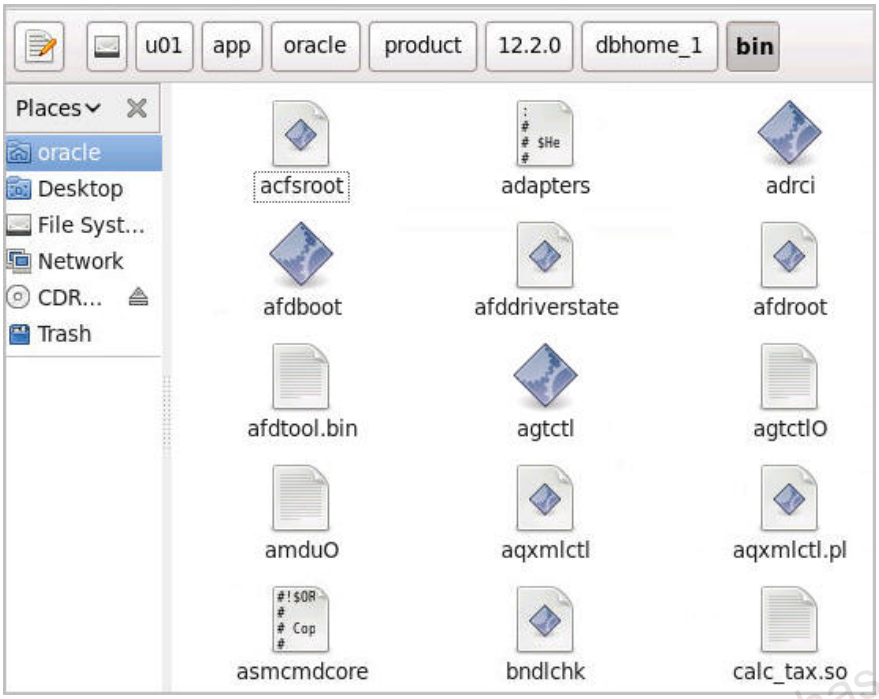
```
#include <ctype.h>
int calc_tax(int n)
{

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

}
```

1. A shared library file called `calc_tax.so` was created for you. Copy the file from the `/home/oracle/labs/labs` directory into your `/u01/app/oracle/product/12.2.0/dbhome_1/bin` directory.

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

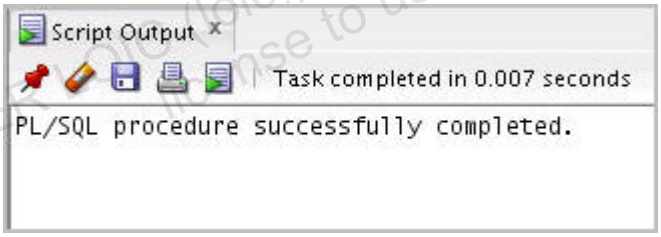


   Navigate to the `/u01/app/oracle/product/12.2.0/dbhome_1/bin` folder. Right-click the `BIN` directory and select Paste from the shortcut menu.



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

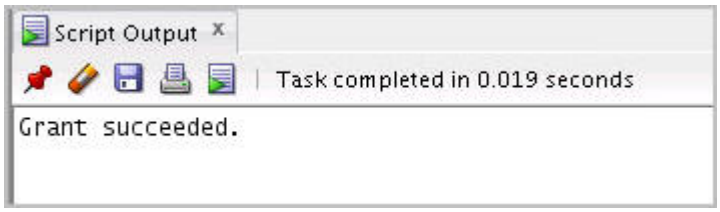
```
-- Use SYS connection
CREATE OR REPLACE LIBRARY c_code
AS '/u01/app/oracle/product/12.2.0/dbhome_1/bin/calc_tax.so';
/
```



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

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

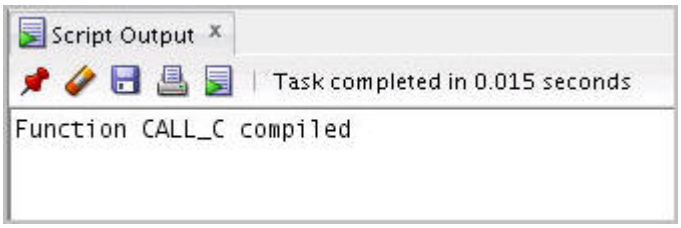
```
-- Use SYS connection
GRANT EXECUTE ON c_code TO OE;
```



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

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

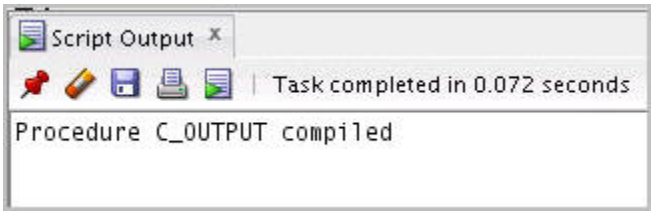
```
-- Use OE Connection
CREATE OR REPLACE FUNCTION call_c
(x BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
LIBRARY sys.c_code
NAME "calc_tax";
/
```



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

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

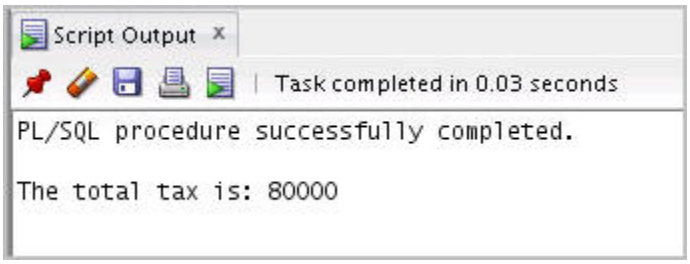
```
-- Use OE connection
CREATE OR REPLACE PROCEDURE c_output
  (p_in IN BINARY_INTEGER)
IS
  i BINARY_INTEGER;
BEGIN
  i := call_c(p_in);
  DBMS_OUTPUT.PUT_LINE('The total tax is: ' || i);
END c_output;
/
```



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

6. Set `SERVEROUTPUT ON` and execute the `C_OUTPUT` procedure.

```
SET SERVEROUTPUT ON

EXECUTE c_output(1000000)
```



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

Calling Java from PL/SQL

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

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

7. Load the `.java` source file.

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

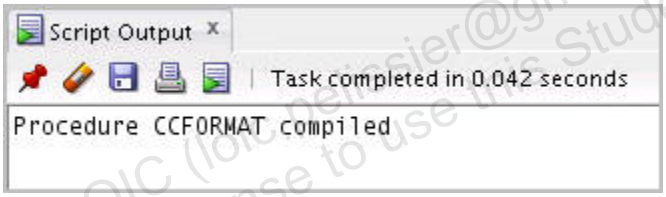
   Alternatively, you can copy and paste the commands in the Linux terminal for task 7 from `sol_07.sql`.

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

   Use the `OE` connection.

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

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



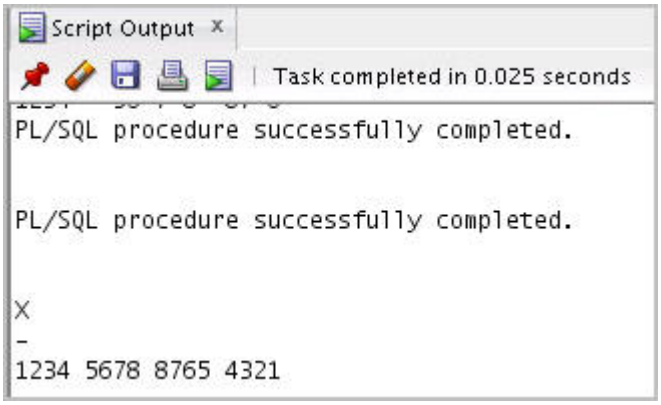
   Alternatively, you can run the solution for task 8 from `sol_07.sql`.

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

   Use the `OE` connection.

```
EXECUTE ccformat(:x);

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

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

EXECUTE ccformat(:x)

PRINT x
```



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

Practices for Lesson 8: Performance and Tuning

Chapter 8

Practices for Lesson 8: Overview

Lesson Overview

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

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

Practice 8-1: Performance and Tuning

Overview

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

Task

Writing Better Code

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

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

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

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

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

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

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

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

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

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

Using Efficient Data Types

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

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

   c. Have the function return `TRUE` if the customer has credit cards. The function should return `FALSE` if the customer does not have credit cards. Pass an uninitialized nested table into the function. The function places the credit card information into this uninitialized parameter.
3. Test your modified code with the following data:

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

EXECUTE credit_card_pkg.display_card_info(120)
```

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

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

```
/
```

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

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

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

Collecting Exception Information

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

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

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

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

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

c. Note the output:_____

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

```
    DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
  bulk_errors EXCEPTION;
  PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
  v_cards.Delete(3);
```

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

e. Note the output:_____

f. Why is the output different?

Timing Performance of `SIMPLE_INTEGER` and `PLS_INTEGER`

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

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

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

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

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

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

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

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

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

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

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

EXECUTE p()

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

EXECUTE p()
```

c. Note the output:_____

d. Explain the output.

Solution 8-1: Performance and Tuning

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

Writing Better Code

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

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

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


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

```
  -- continued on next page.


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

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

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

Using Efficient Data Types

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

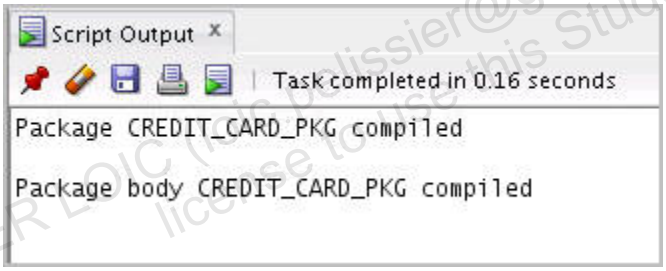
```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS

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

END credit_card_pkg;  -- package spec
/
```

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

```
-- note: If you did not complete lesson 4 practice, you will
need
-- to run solution scripts for tasks 1_a, 1_b, 1_c from
sol_04.sql
-- in order to have the supporting structures in place.

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

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
```

```
IS

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

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

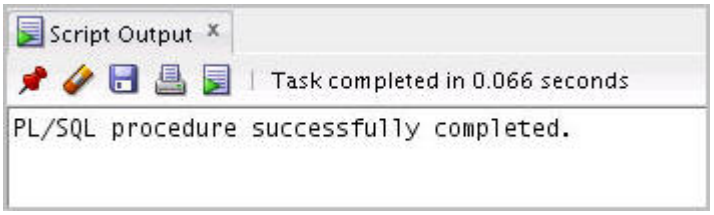
```
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
  BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
      FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
          DBMS_OUTPUT.PUT('Card Type: ' ||
            v_card_info(idx).card_type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



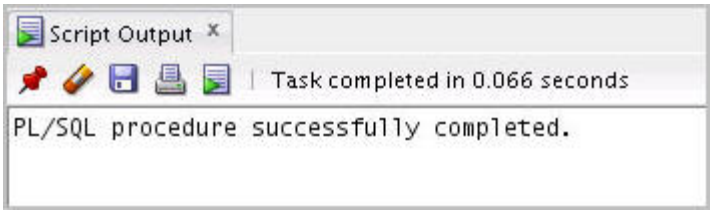
Alternatively, run the code from task 2_c of `sol_08.sql`.

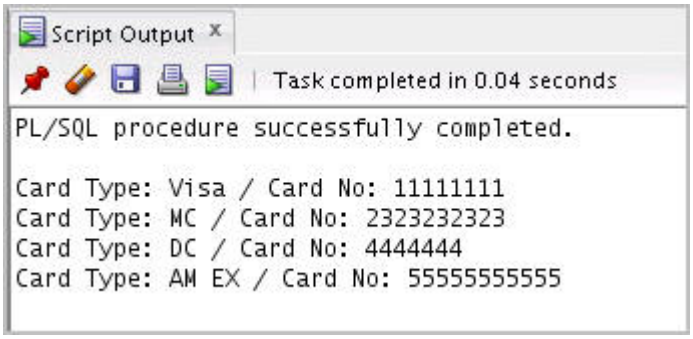
3. Test your modified code with the following data:

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



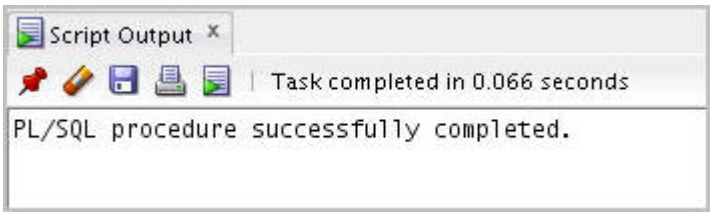

```
EXECUTE credit_card_pkg.display_card_info(120)
```

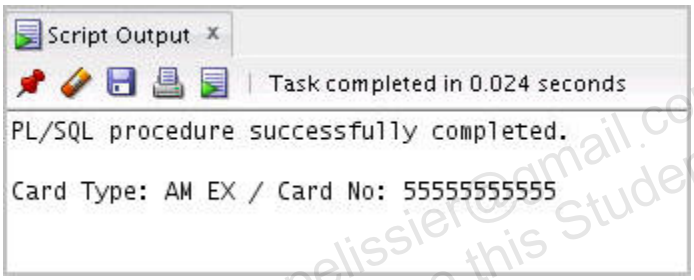




Note: If you did not complete Practice 4, your results will be:

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



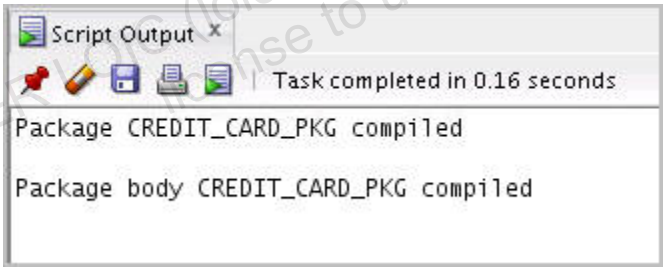


4. You must modify the `UPDATE_CARD_INFO` procedure to return information (by using the `RETURNING` clause) about the credit cards being updated. Assume that this information will be used by another application developer on your team, who is writing a graphical reporting utility on customer credit cards.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

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

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

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

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

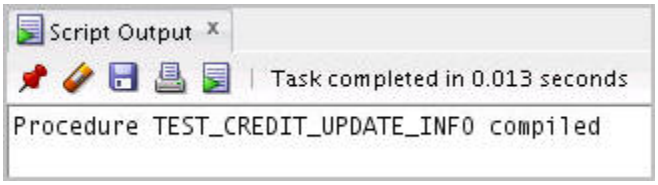
```
        RETURNING credit_cards INTO o_card_info;
    END IF;
  END update_card_info;

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



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

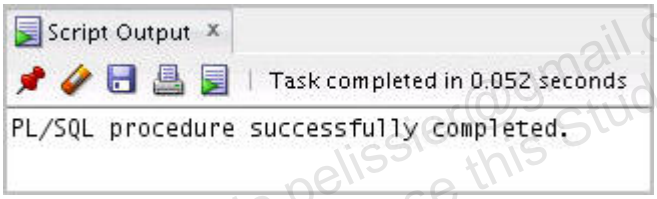
c. You can test your modified code with the following procedure (contained in task 4_c of `lab_08.sql`):

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

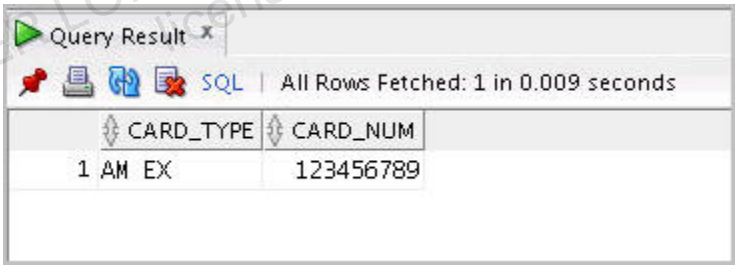


d. Test your code with the following statements that are set in boldface:

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



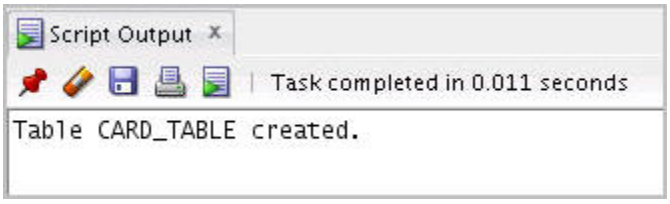

```
SELECT * FROM TABLE(SELECT credit_cards FROM customers WHERE
customer_id = 125);
```



Collecting Exception Information

5. Now you test exception handling with the `SAVE EXCEPTIONS` clause.
   a. Run task 5_a of the `lab_08.sql` file to create a test table:

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



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

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

   c. Note the output:

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

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

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

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

e. Note the output:

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

f. Why is the output different?

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

Timing Performance of `SIMPLE_INTEGER` and `PLS_INTEGER`

6. Now you compare the performance between the `PLS_INTEGER` and `SIMPLE_INTEGER` data types with native compilation:
   a. Run task 6_a of `lab_08.sql` to create a testing procedure that contains conditional compilation:

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

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

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

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

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

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

  t1 := DBMS_UTILITY.GET_CPU_TIME();
  DBMS_OUTPUT.PUT_LINE(
```

```
    RPAD(LOWER($$PLSQL_Code_Type), 15)||
    RPAD(LOWER(My_Integer_t_Name), 15)||
    TO_CHAR((t1-t0), '9999')||' centiseconds');
END p;
/
```

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

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

EXECUTE p()

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

EXECUTE p()
```

c. Note the output:

First run:

```
Procedure P altered.

PL/SQL procedure successfully completed.

native          simple_integer    12 centiseconds
```

Second run:

```
Procedure P altered.

PL/SQL procedure successfully completed.

native          pls_integer      141 centiseconds
```

d. Explain the output.

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

Practices for Lesson 9: Improving Performance with Caching

Chapter 9

Practices for Lesson 9: Overview

Lesson Overview

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

Practice 9-1: Improving Performance with Caching

Overview

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

Use the `OE` connection to complete this practice.

Task

Examining SQL and PL/SQL Result Caching

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

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

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

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

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

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

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

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

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

4. Because the function is called frequently, and because the content of the data returned does not change frequently, this code is a good candidate for PL/SQL result caching.
   Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

Solution 9-1: Improving Performance with Caching

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

Use the `OE` connection.

Examining SQL and PL/SQL Result Caching

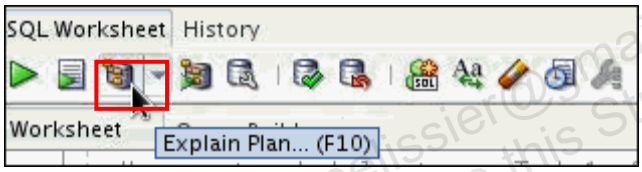
1. Use SQL Developer to connect to the `OE` schema. Examine the Explain Plan for the following query, which is found in the `lab_09.sql` file. To view the Explain Plan, click the Execute Explain Plan button on the toolbar in the Code Editor window.

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

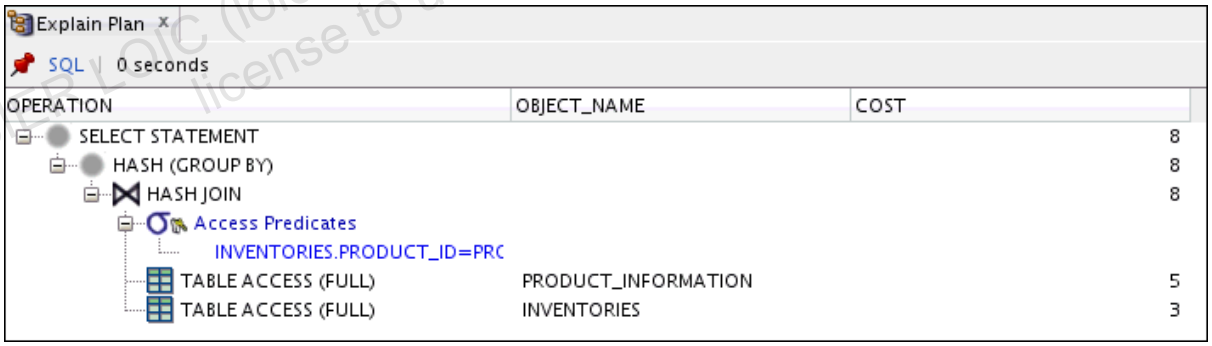
   ```
   SELECT count(*),
          round(avg(quantity_on_hand)) AVG_AMT,
          product_id, product_name
   FROM inventories natural join product_information
   GROUP BY product_id, product_name;
   ```

   Select the `OE` connection.

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

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

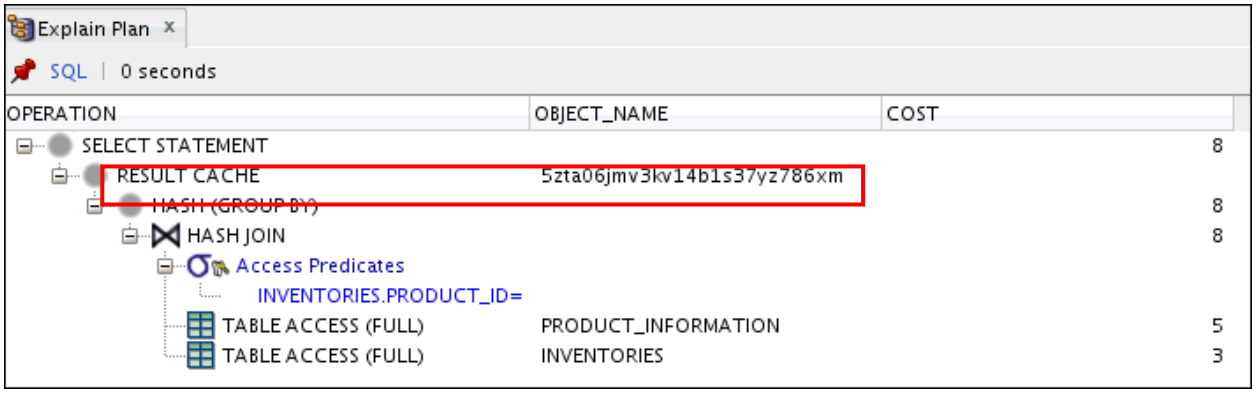
   

2. Add the `RESULT_CACHE` hint to the query and reexamine the Explain Plan results.

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

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

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

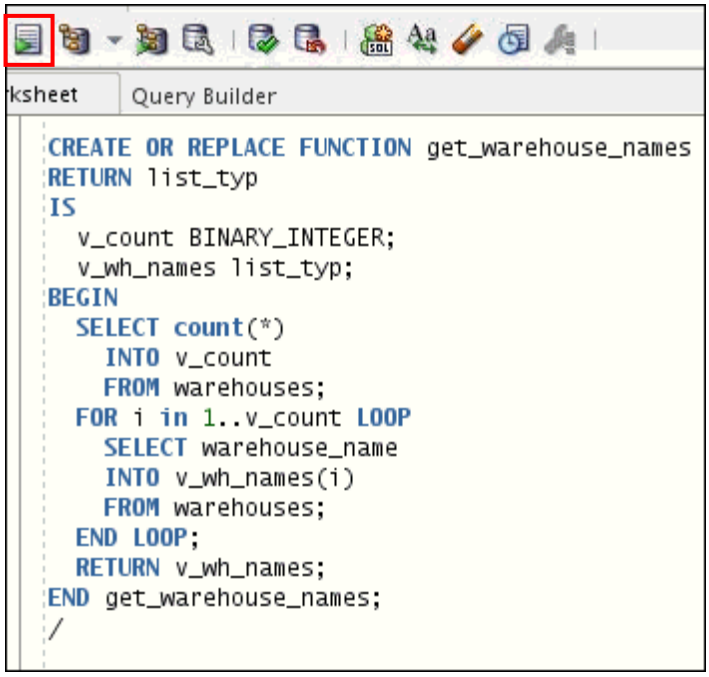


Results: Note that result caching is used in the Explain Plan.

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

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

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

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



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

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

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

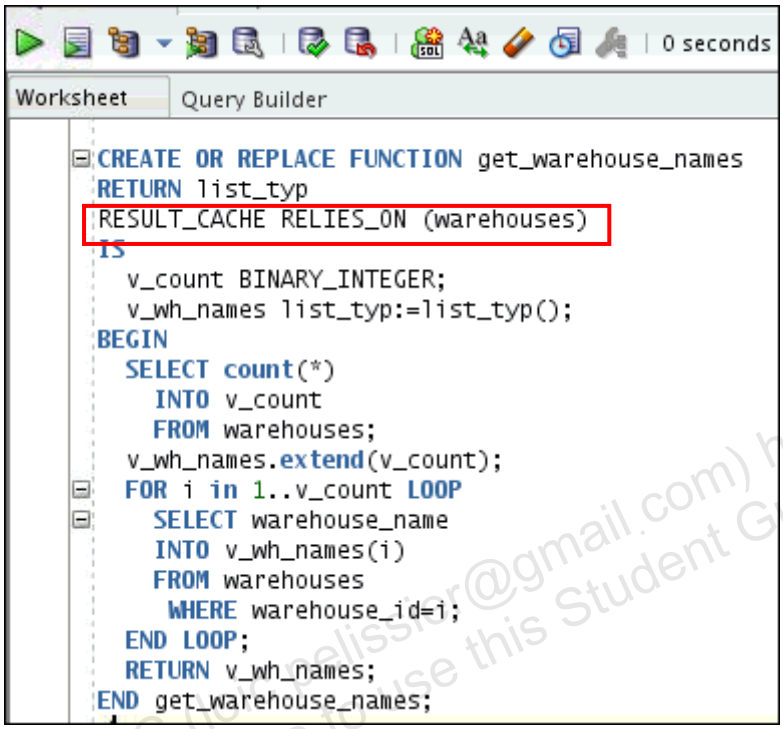
   Insert the following line after RETURN list_typ:

   `RESULT_CACHE RELIES_ON (warehouses)`

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

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

Click the Run Script button to recompile the code.




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

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

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

Practices for Lesson 10: Analyzing PL/SQL Code

Chapter 10

Practices for Lesson 10: Overview

Lesson Overview

In this practice, you will perform the following:

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

Practice 10-1: Analyzing PL/SQL Code

Overview

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

Use your OE connection.

Task

Finding Coding Information

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

   Use the OE connection.

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

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

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

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

Using PL/Scope

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

   Use the OE connection.

   a. Enable your session to collect identifiers.

   b. Recompile your CREDIT_CARD_PKG code.

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

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

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

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

```
                RPAD(Type, 20)|| RPAD(Usage, 20)
                IDENTIFIER_USAGE_CONTEXTS
    FROM v
    START WITH USAGE_CONTEXT_ID = 0
    CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
    ORDER SIBLINGS BY Line, Col;
```

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

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

   b. Issue the following statements to view the metadata generated from the GET_TABLE_MD function:
      You can run task 3_b of the lab_10.sql script.

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

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

Solution 10-1: Analyzing PL/SQL Code

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

Use your OE connection.

Finding Coding Information

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

   Use the OE connection.

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

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

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

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

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

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

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

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

```
PL/SQL procedure successfully completed.

Programs that reference tables or views
OE.ADD_ORDER_ITEMS,OE.PORDER
OE.ALLOCATE_NEW_PROJ_LIST,OE.DEPARTMENT
OE.CHANGE_CREDIT,OE.CUSTOMERS
OE.CREDIT_CARD_PKG,OE.CUSTOMERS
OE.GET_EMAIL,OE.CUSTOMERS
OE.GET_WAREHOUSE_NAMES,OE.WAREHOUSES
OE.LIST_PRODUCTS_DYNAMIC,OE.PRODUCT_INFORMATION
OE.LIST_PRODUCTS_STATIC,OE.PRODUCT_INFORMATION
OE.LOAD_PRODUCT_IMAGE,OE.PRODUCT_INFORMATION
OE.LOB_TXT,OE.LOB_TEXT
OE.MANAGE_DEPT_PROJ,OE.DEPARTMENT
OE.ORDERS_CTX_PKG,OE.CUSTOMERS
OE.ORD_COUNT,OE.ORDERS
OE.PRINT_CUSTOMERS,OE.CUSTOMERS
OE.PRINT_CUSTOMERS,OE.ORDERS
OE.PRINT_EMPLOYEES,OE.CUSTOMERS
```

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

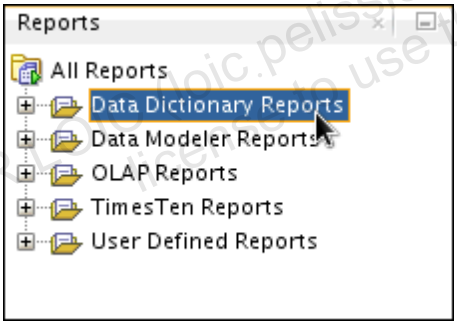
```
SET SERVEROUTPUT ON
EXECUTE query_code_pkg.find_text_in_code('ORDERS')
```

```
PL/SQL procedure successfully completed.

Checking for presence of ORDERS:
CUSTOMER_TYP-12,    , cust_orders        order_list_typ

GET_TABLE_MD-11,                       (v_hdl ,'NAME','ORDERS');

JSON_PUT_PRACTICE-8,                                    FROM orders o

ORDERS_APP_PKG-1,PACKAGE orders_app_pkg

ORDERS_APP_PKG-1,PACKAGE BODY orders_app_pkg

ORDERS_CTX_PKG-1,PACKAGE orders_ctx_pkg IS

ORDERS_CTX_PKG-1,PACKAGE BODY orders_ctx_pkg IS
```

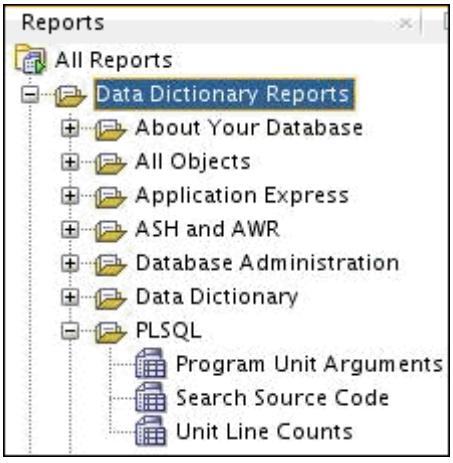
Alternatively, you can execute the solutions for tasks 1_b and 1_c from `sol_10.sql`.

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

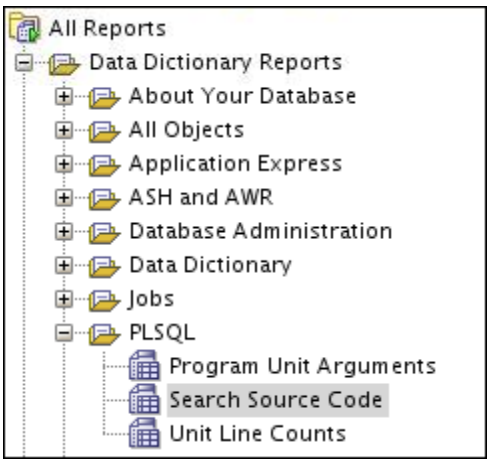
Navigate to the Reports tabbed page in Oracle SQL Developer.

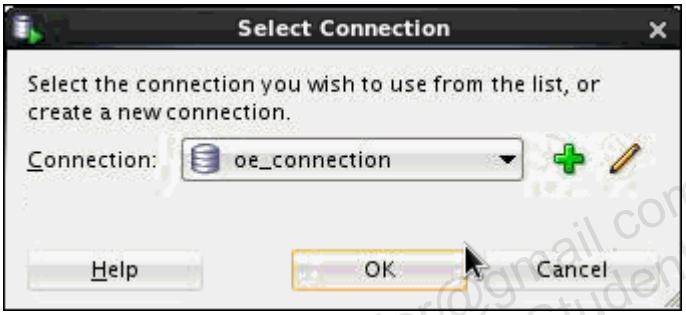


Expand the Data Dictionary Reports node and expand the PL/SQL node.

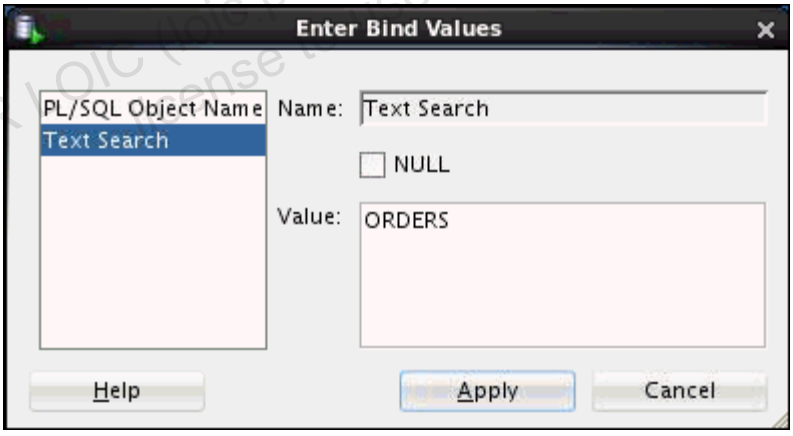


Select Search Source Code, and then select your `OE` connection and click OK.





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



	Owner	PL/SQL Object Name	Type	Line	Text
1	OE	CUSTOMER_TYP	TYPE	12	, cust_orders order_list_typ
2	OE	GET_TABLE_MD	FUNCTION	11	(v_hdl ,'NAME','ORDERS');
3	OE	JSON_PUT_PRACTICE	PROCEDURE	8	FROM orders o
4	OE	ORDERS_APP_PKG	PACKAGE	1	PACKAGE orders_app_pkg
5	OE	ORDERS_APP_PKG	PACKAGE BODY	1	PACKAGE BODY orders_app_pkg
6	OE	ORDERS_CTX_PKG	PACKAGE	1	PACKAGE orders_ctx_pkg IS
7	OE	ORDERS_CTX_PKG	PACKAGE BODY	1	PACKAGE BODY orders_ctx_pkg IS
8	OE	ORDERS_CTX_PKG	PACKAGE BODY	8	DBMS_SESSION.SET_CONTEXT('orders_ctx', 'customer_id', custnum);
9	OE	ORDERS_ITEMS_TRG	TRIGGER	1	TRIGGER orders_items_trg INSTEAD OF INSERT ON NESTED
10	OE	ORDERS_ITEMS_TRG	TRIGGER	2	TABLE order_item_list OF oc_orders FOR EACH ROW
11	OE	ORDERS_TRG	TRIGGER	1	TRIGGER orders_trg INSTEAD OF INSERT
12	OE	ORDERS_TRG	TRIGGER	2	ON oc_orders FOR EACH ROW
13	OE	ORDERS_TRG	TRIGGER	4	INSERT INTO ORDERS (order_id, order_mode, order_total,
14	OE	ORD_COUNT	FUNCTION	3	RESULT_CACHE RELIES_ON (orders)
15	OE	ORD_COUNT	FUNCTION	8	FROM orders
16	OE	PRINT_CUSTOMERS	PROCEDURE	11	from oe.customers c, oe.orders o
17	OE	PRINT_EMPLOYEES	PROCEDURE	12	from oe.customers c, oe.orders o

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

Using PL/Scope

Use the `OE` connection.

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

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

```
session SET altered.
```

   b. Recompile your `CREDIT_CARD_PKG` code.

```
ALTER PACKAGE credit_card_pkg COMPILE;
```

```
package CREDIT_CARD_PKG altered.
```

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

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

	PLSCOPE_SETTINGS
1	IDENTIFIERS:ALL

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

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

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

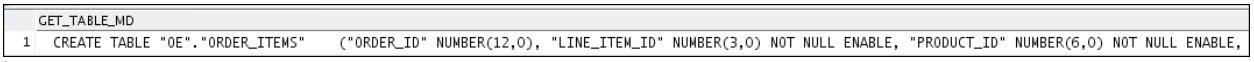
	IDENTIFIER_USAGE_CONTEXTS		
1	Credit_Card_Pkg.....	package	definition
2	Cust_Card_Info....	function	definition
3	P_Cust_Id.....	formal in	declaration
4	Number.....	number datatype	reference
5	P_Card_Info.....	formal in out	declaration
6	Typ_Cr_Card_Ns	nested table	reference
7	Boolean.....	boolean datatype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

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

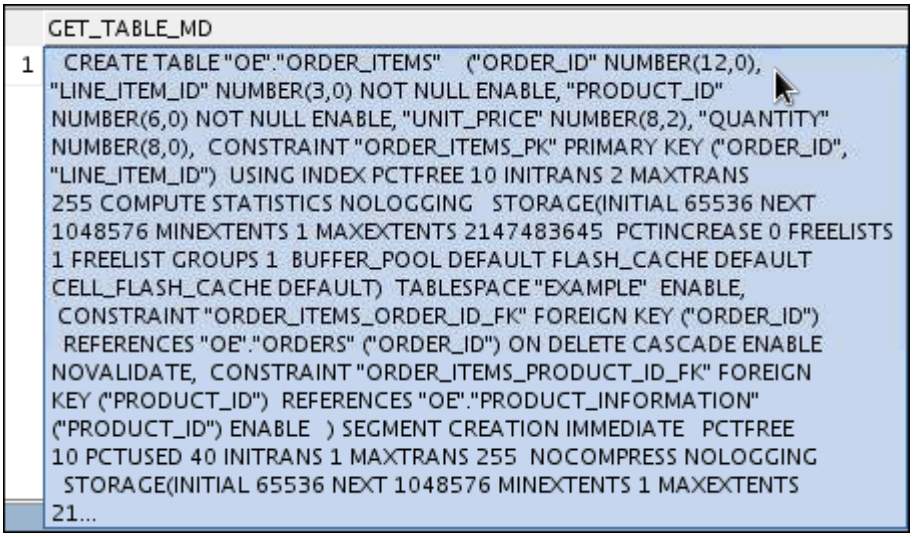
```
CREATE OR REPLACE FUNCTION get_table_md RETURN CLOB IS
 v_hdl  NUMBER; -- returned by 'OPEN'
 v_th   NUMBER; -- returned by 'ADD_TRANSFORM'
 v_doc  CLOB;
BEGIN
 -- specify the OBJECT TYPE
 v_hdl := DBMS_METADATA.OPEN('TABLE');
 -- use FILTERS to specify the objects desired
 DBMS_METADATA.SET_FILTER(v_hdl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

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

```
set pagesize 0
set long  1000000

SELECT get_table_md FROM dual;
```



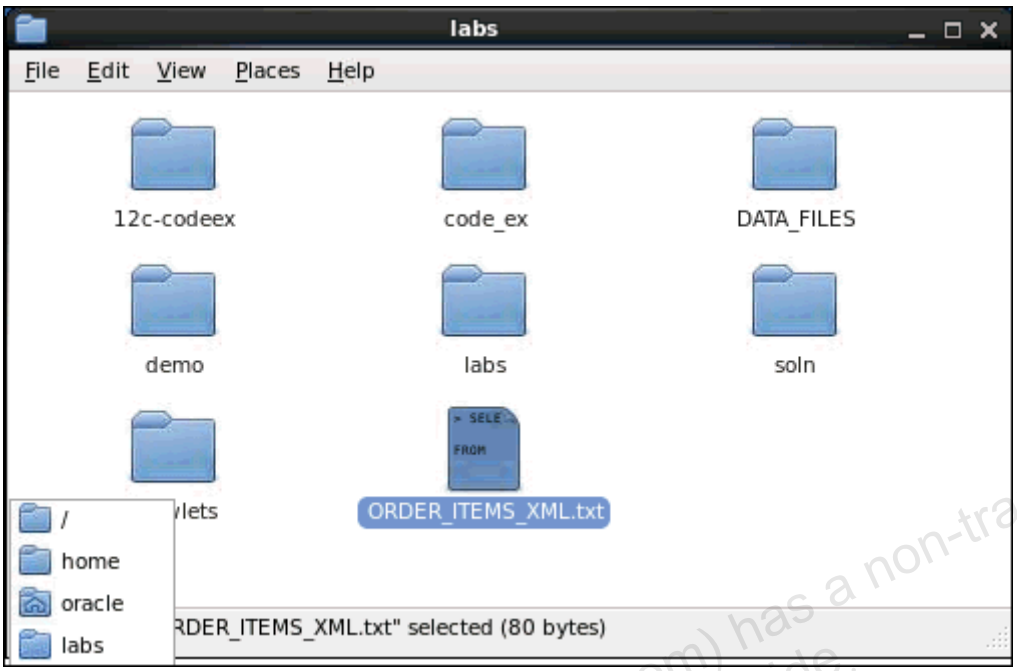
   Move the cursor over the get_table_md column to see the detailed view of the record.



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

```
SET PAGESIZE 0
SET LONG 1000000
SPOOL /home/oracle/labs/ORDER_ITEMS_XML.txt

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

SPOOL OFF
```

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



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

Chapter 11

Practices for Lesson 11: Overview

Lesson Overview

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

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

Overview

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

Task

Use your `OE` connection.

1. Generate profiling data for your `CREDIT_CARD_PKG`.
   a. Re-create `CREDIT_CARD_PKG` by running the `/home/oracle/labs/lab_11.sql` script.
   b. You must identify the location of the profiler files. Create a `DIRECTORY` object to identify this information, and grant the necessary privileges. Use the `SYS` connection.
   c. Use `DBMS_HPROF.START_PROFILING` to start the profiler for your session.
   d. Run your `CREDIT_CARD_PKG.UPDATE_CARD_INFO` with the following data.
      ```
      credit_card_pkg.update_card_info
          (154, 'Discover', '123456789');
      ```
   e. Use `DBMS_HPROF.STOP_PROFILING` to stop the profiler.
2. Run the `dbmshptab.sql` script, located in the `/u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin` folder, to set up the profiler tables.
3. Use `DBMS_HPROF.ANALYZE` to analyze the raw data and write the information to the profiler tables.
   a. Get `RUN_ID`.
   b. Query the `DBMSHP_RUNS` table to find top-level information for `RUN_ID` that you retrieved.
   c. Query the `DBMSHP_FUNCTION_INFO` table to find information about each function profiled.
4. Use the `plshprof` command-line utility to generate simple HTML reports directly from the raw profiler data.
   a. Open a command window.
   b. Change the working directory to `/home/oracle/labs/labs`.
   c. Run the `plshprof` utility.
5. Open the report in your browser and review the data.

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

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

Use your OE connection.

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

      Use the OE connection.

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

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

      Use the SYS connection.

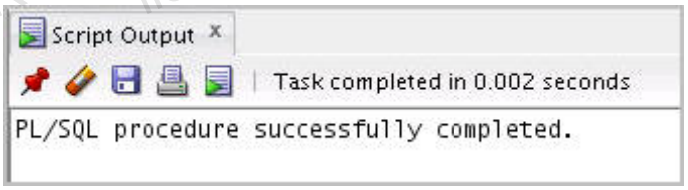
      ```
      CREATE DIRECTORY profile_data AS '/home/oracle/labs/labs';
      GRANT READ, WRITE, EXECUTE ON DIRECTORY profile_data TO OE;
      GRANT EXECUTE ON DBMS_HPROF TO OE;
      ```

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

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

      Use the OE connection.

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

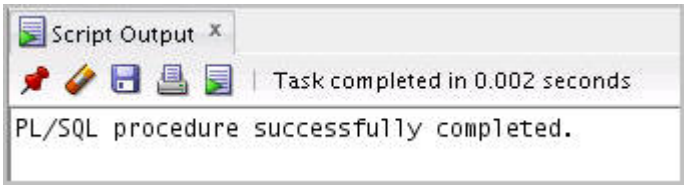
      

   d. Run your CREDIT_CARD_PKG.UPDATE_CARD_INFO with the following data.

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

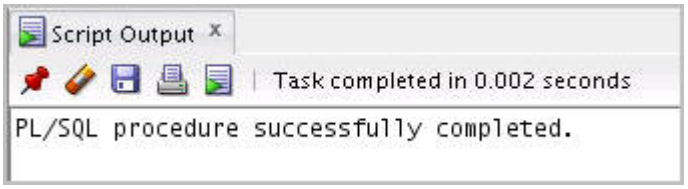
      Use the OE connection.

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



e. Use `DBMS_HPROF.STOP_PROFILING` to stop the profiler.
Use the `OE` connection.

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



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

2. Run the `dbmshptab.sql` script, located in the `/u01/app/oracle/product/12.2.0/dbhome_1/rdbms/admin` folder, to set up the profiler tables.

```
@/u01/app/oracle/product/12.2.0/dbhome_1/rdbms/admin/dbmshptab.sql
```

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

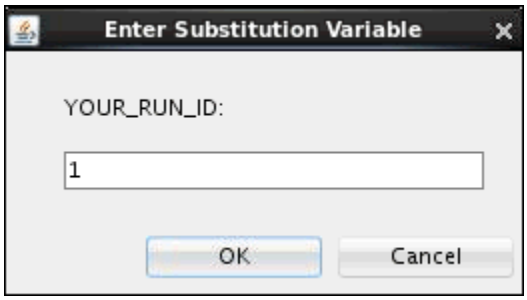
3. Use `DBMS_HPROF.ANALYZE` to analyze the raw data and write the information to the profiler tables.
   a. Get `RUN_ID`.

```
SET SERVEROUTPUT ON

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

```
Run ID: 1
```

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

```
SET VERIFY OFF

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



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

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

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

	OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	FUNCTION_ELAPSED_TIME
1	(null)	(null)	(null)	__anonymous_block	PLSQL	8	467
2	(null)	(null)	(null)	__plsql_vm	PLSQL	8	39
3	OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	2	202
4	SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1	0
5	SYS	DBMS_OUTPUT	PACKAGE BODY	GET_LINE	PLSQL	5	15
6	OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line17	SQL	2	5179
7	OE	CREDIT_CARD_PKG	PACKAGE BODY	__static_sql_exec_line9	SQL	2	771

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

4. Use the `plshprof` command-line utility to generate simple HTML reports directly from the raw profiler data.
   a. Open a command window.
   b. Change the working directory to `/home/oracle/labs/labs`.
   c. Run the `plshprof` utility.

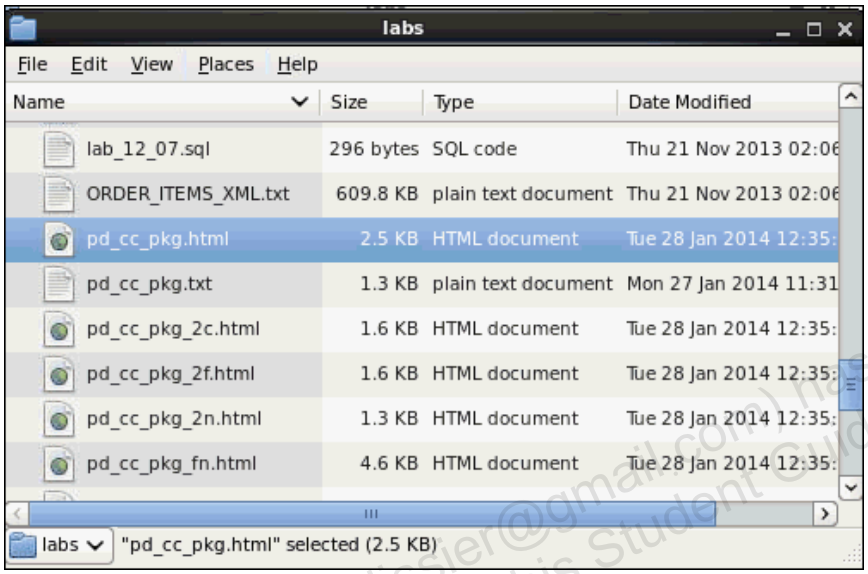
```
--at your command window, change your working directory to
/home/oracle/labs/labs
cd /home/oracle/labs/labs
plshprof  -output pd_cc_pkg  pd_cc_pkg.txt
```

```
[oracle@edvmr1p0 labs]$ plshprof -output pd_cc_pkg pd_cc_pkg.txt
PLSHPROF: Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Prod
uction
[7 symbols processed]
[Report written to 'pd_cc_pkg.html']
```

Note : The number of symbols should be equal to the number of values returned in Step 3c

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

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



PL/SQL Elapsed Time (microsecs) Analysis

6673 microsecs (elapsed time) & 28 function calls

The PL/SQL Hierarchical Profiler produces a collection of reports that present inf formats. The following reports have been found to be the most generally useful as

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

In addition, the following reports are also available:

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

Practices for Lesson 12: Securing Applications through PL/SQL

Chapter 12

Practices for Lesson 12: Overview

Lesson Overview

In this practice, you:

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

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

Overview

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

Task

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

```
DESCRIBE orders
```

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

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

Run this step to check the ORDERS table.

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

1. Use your SYS connection. Examine and then run the lab_12.sql script.
   This script creates the sales representative ID accounts with appropriate privileges to access the database.
2. Set up an application context:
   a. Connect to the database as SYS before creating this context.
   b. Create an application context named sales_orders_ctx.
   c. Associate this context to oe.sales_orders_pkg.
3. Connect as OE.
   a. Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
 PROCEDURE set_app_context;
 FUNCTION the_predicate
  (p_schema VARCHAR2, p_name VARCHAR2)
   RETURN VARCHAR2;
```

```
END sales_orders_pkg;    -- package spec
/
```

b. Create this package specification and the package body in the OE schema.

c. When you create the package body, set up two constants as follows:

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

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

4. Connect as SYS and define the policy.

a. Use DBMS_RLS.ADD_POLICY to define the policy.

b. Use these specifications for the parameter values:

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

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

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

```
CONNECT sr153/oracle@pdborcl

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

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

CONNECT sr154/oracle@pdborcl

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

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

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

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

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

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

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

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

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

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

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

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

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

GRANT create session
    , alter session
TO sr153, sr154;

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

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

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

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

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

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

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

```
context SALES_ORDERS_CTX created.
```

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

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

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

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

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

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

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

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

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

END sales_orders_pkg; -- package body
/
```

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

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

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

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

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

```
PL/SQL procedure successfully completed.
```

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

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

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

```
TRIGGER SET_ID_ON_LOGON compiled
```

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

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

```
CONNECT sr153/oracle

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

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

CONNECT sr154/oracle

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

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

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

SALES_REP_ID   COUNT(*)
------------ ----------
         153          5
SQL>
```

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

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

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

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

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

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

Chapter 13

Practices for Lesson 13: Overview

Lesson Overview

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

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

Overview

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

Use the `OE` connection for this practice.

Task

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

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

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

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

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

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

a. Execute the following statements and note the results.

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

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

b. Has SQL injection occurred?

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

a. Execute the following statements and note the results:

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

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

b. Has SQL injection occurred?

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

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

Use the `OE` connection for this practice.

Understanding SQL Injection

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

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

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

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

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

Rewriting Code to Protect Against SQL Injection

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

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

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

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

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

```
PROCEDURE GET_INCOME_LEVEL compiled
```

a. Execute the following statements and note the results.

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

```
PL/SQL procedure successfully completed.

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

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

```
PL/SQL procedure successfully completed.

SQL statement: SELECT income_level FROM customers WHERE cust_email = 'x' union select username
Income level is: AM145
Income level is: AM147
Income level is: AM148
Income level is: AM149
Income level is: ANONYMOUS
```

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

b. Has SQL injection occurred?

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

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

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

a. Execute the following statements and note the results.

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

```
PL/SQL procedure successfully completed.

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

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

```
PL/SQL procedure successfully completed.
```

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

b. Has SQL injection occurred?
No

Practices for Lesson 14: Advanced Security Mechanisms

Chapter 14

Practices for Lesson 14: Overview

Practice Overview

There are no practices for this lesson.