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(
```

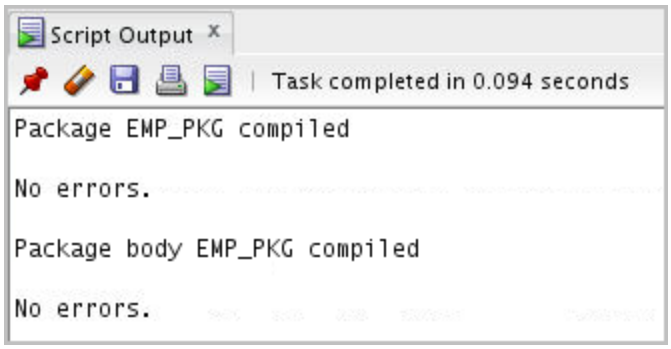
```
    p_first_name employees.first_name%TYPE,
    p_last_name employees.last_name%TYPE,
    p_deptid employees.department_id%TYPE);

  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
```

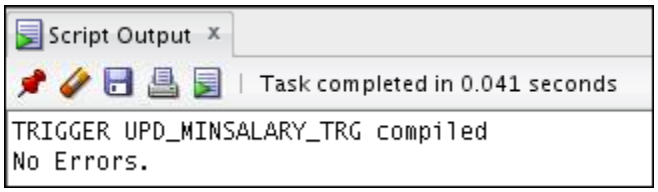
```
      WHERE employee_id = rec_emp.employee_id;
    END LOOP;
  END set_salary;

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



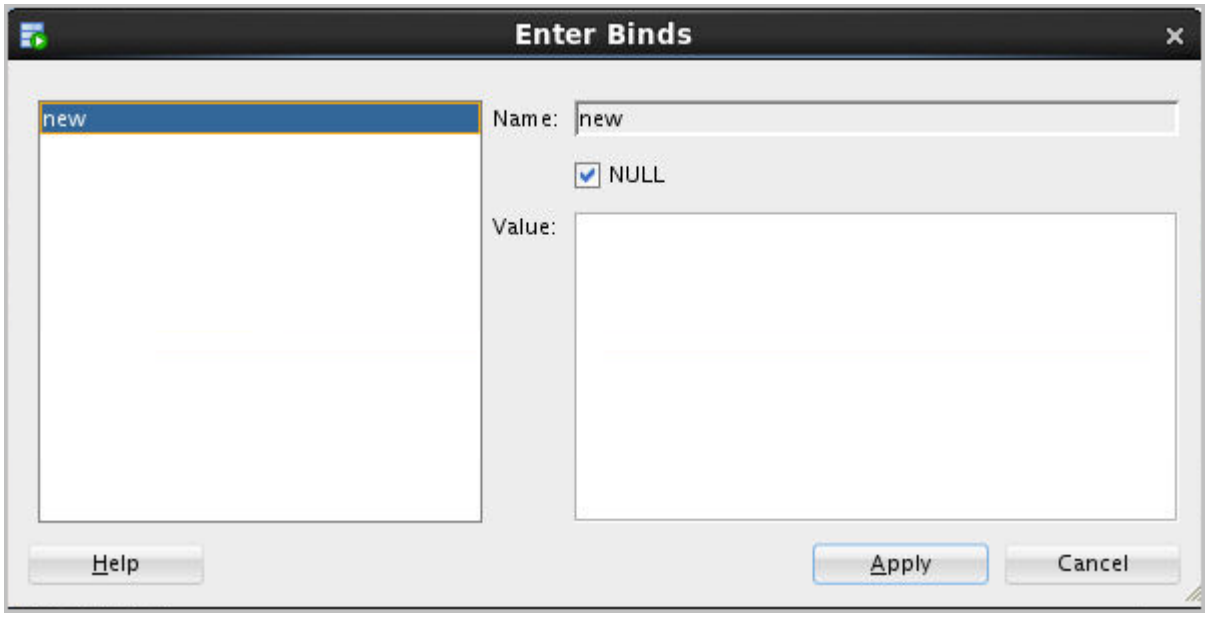
b. Create a row trigger named `UPD_MINSALARY_TRG` on the `JOBS` table that invokes the `EMP_PKG.SET_SALARY` procedure, when the minimum salary in the `JOBS` table is updated for a specified job ID.

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
```



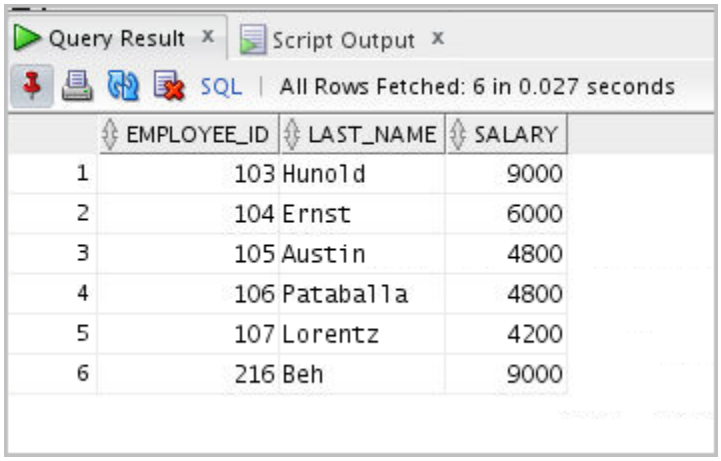
Note: The trigger compilation might ask for values of bind variables while compiling. You may encounter a wizard as the one below. Click Apply.

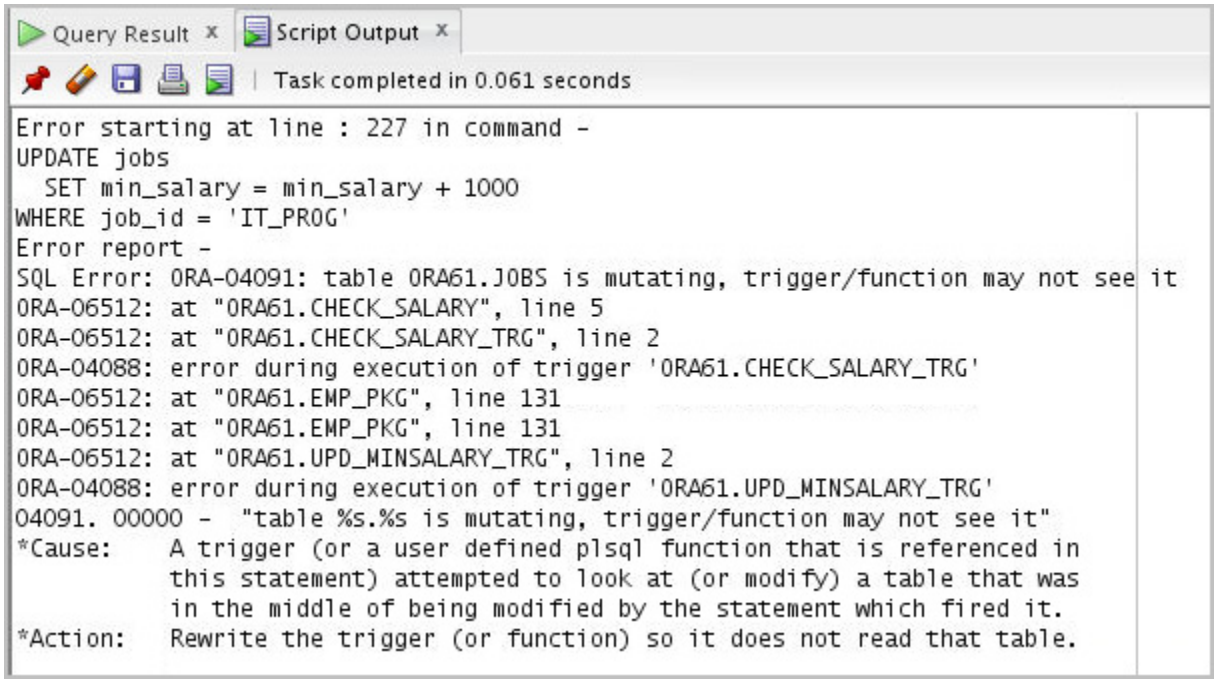


c. Write a query to display the employee ID, last name, job ID, current salary, and minimum salary for employees who are programmers—that is, their `JOB_ID` is `'IT_PROG'`. Then, update the minimum salary in the `JOBS` table to increase it by $1,000. What happens?

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';
```





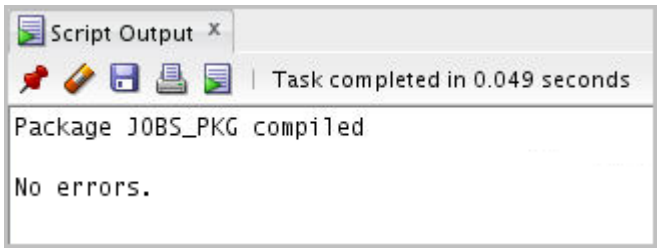
The update of the `min_salary` column for job `'IT_PROG'` fails because the `UPD_MINSALARY_TRG` trigger on the `JOBS` table attempts to update the employees' salaries by calling the `EMP_PKG.SET_SALARY` procedure. The `SET_SALARY` procedure causes the `CHECK_SALARY_TRG` trigger to fire (a cascading effect).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

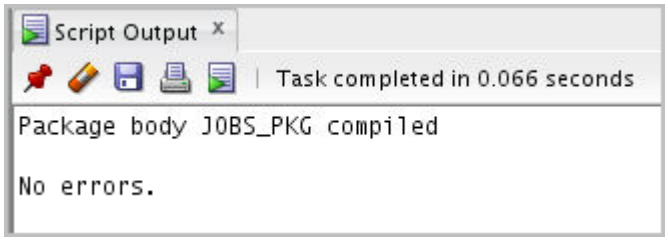
1) Declare a private PL/SQL index-by table called `jobs_tab_type` that is indexed by a string type based on the `JOBS.JOB_ID%TYPE`.
2) Declare a private variable called `jobstab` based on the `jobs_tab_type`.
3) The `INITIALIZE` procedure reads the rows in the `JOBS` table by using a cursor loop, and 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;
```

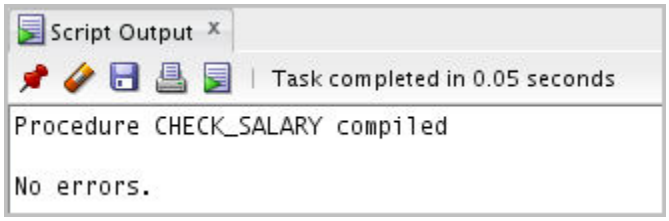
```
END jobs_pkg;
/
SHOW ERRORS
```



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.

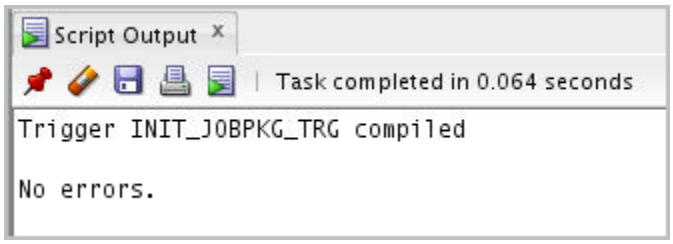
```
CREATE OR REPLACE PROCEDURE check_salary (p_the_job VARCHAR2,
p_the_salary NUMBER) IS
  v_minsal jobs.min_salary%type;
  v_maxsal jobs.max_salary%type;
BEGIN

  -- 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
```

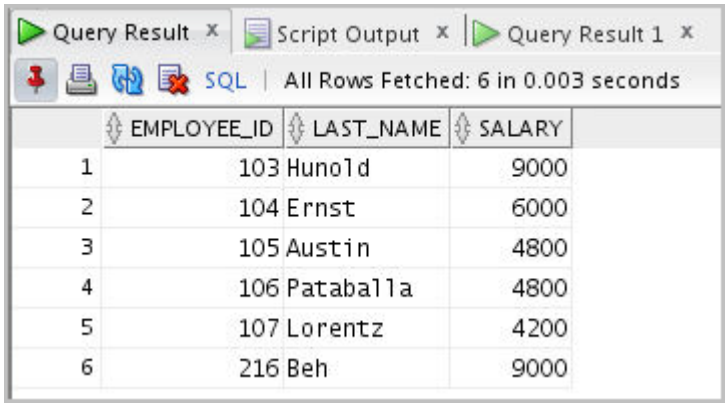


e. Test the code changes by executing the query to display the employees who are programmers, and then issue an update statement to increase the minimum salary of the `IT_PROG` job type by 1,000 in the `JOBS` table. Follow this up with a query on the employees with the `IT_PROG` job type to check the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

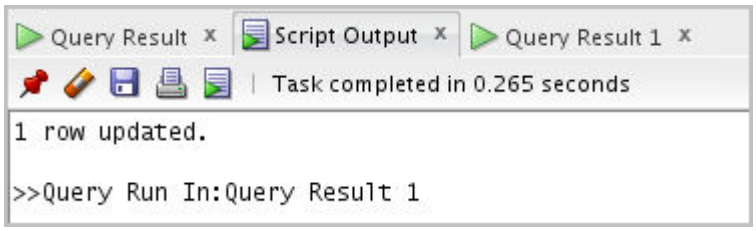
Uncomment and select the code under Task 2_e. Click the Run Script (F5) icon on the SQL Worksheet toolbar to run the script. The code and the results are shown below.

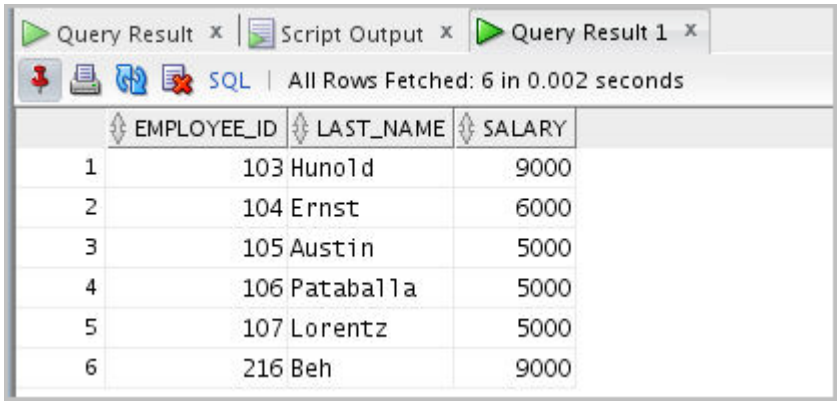
```
SELECT employee_id, last_name, salary
FROM employees
WHERE job_id = 'IT_PROG';

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';
```



	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





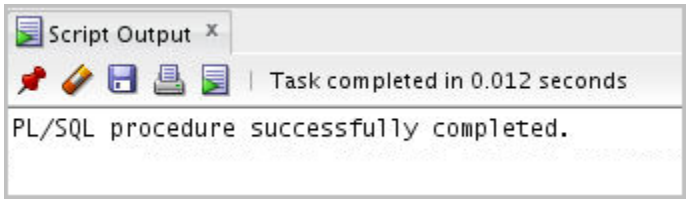
	EMPLOYEE_ID	LAST_NAME	SALARY
1	103	Hunold	9000
2	104	Ernst	6000
3	105	Austin	5000
4	106	Pataballa	5000
5	107	Lorentz	5000
6	216	Beh	9000

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.

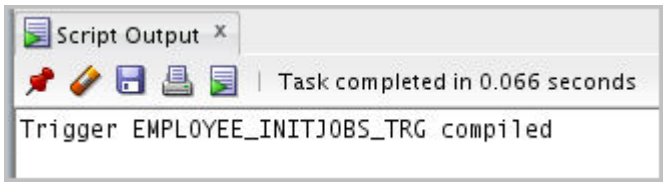
```
EXECUTE emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', p_sal
=> 6500)
```



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.