A

Practice Solutions

Practice I: Solutions

1. Launch *i*SQL*Plus using the icon provided on your desktop.

   a. Log in to the database by using the username and database connect string details provided by your instructor (optionally, write the information here for your records):
   Username: **ora__**
   Password: **ora__**
   Database Connect String/Tnsname: **t1**

   b. Execute basic `SELECT` statements to query the data in the `DEPARTMENTS`, `EMPLOYEES`, and `JOBS` tables. Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides a description and some data from each table in the Human Resources schema.

```
SELECT * FROM departments;
SELECT * FROM employees;
```

2. Create a procedure called `HELLO` to display the text `Hello World`.

   a. Create a procedure called `HELLO`.

   b. In the executable section, use the `DBMS_OUTPUT.PUT_LINE` procedure to print `Hello World`, and save the code in the database.
   Note: If you get compile-time errors, then edit the PL/SQL to correct the code, and replace the `CREATE` keyword with the text `CREATE OR REPLACE`.

```
CREATE PROCEDURE hello IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World');
END;
/

Procedure created.
```

   c. Execute the `SET SERVEROUTPUT ON` command to ensure that the output from the `DBMS_OUTPUT.PUT_LINE` procedure will be displayed in *i*SQL*Plus.

```
SET SERVEROUTPUT ON
```

   d. Create an anonymous block to invoke the stored procedure.

```
BEGIN
  hello;
END;
/

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

Practice I: Solutions (continued)

3. Create a function called TOTAL_SALARY to compute the sum of all employee salaries.

   a. Create a function called TOTAL_SALARY that returns a NUMBER.

   b. In the executable section, execute a query to store the total salary of all employees in a local variable that you declare in the declaration section. Return the value stored in the local variable. Compile the code.

```
CREATE FUNCTION total_salary RETURN NUMBER IS
  total employees.salary%type;
BEGIN
  SELECT SUM(salary) INTO total
  FROM employees;
  RETURN total;
END;
/

Function created.
```

   c. Use an anonymous block to invoke the function. To display the result computed by the function, use the DBMS_OUTPUT.PUT_LINE procedure.
   Hint: Either nest the function call inside the DBMS_OUTPUT.PUT_LINE parameter, or store the function result in a local variable of the anonymous block and use the local variable in the DBMS_OUTPUT.PUT_LINE procedure.

```
DECLARE
  total number := total_salary;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Total Salary: '|| total);
END;
/
-- OR ...
BEGIN
  DBMS_OUTPUT.PUT_LINE('Total Salary: '|| total_salary);
END;
/

Total Salary: 691400
PL/SQL procedure successfully completed.

Total Salary: 691400
PL/SQL procedure successfully completed.
```

Practice I: Solutions (continued)

If you have time, complete the following exercise:

4. Launch SQL*Plus using the icon that is provided on your desktop.

   a. Invoke the procedure and function that you created in exercises 2 and 3.

```
SET SERVEROUTPUT ON
EXECUTE hello;

Hello World
PL/SQL procedure successfully completed.

EXECUTE DBMS_OUTPUT.PUT_LINE('Total Salary: '|| total_salary);

Total Salary: 691400
PL/SQL procedure successfully completed.
```

   b. Create a new procedure called `HELLO_AGAIN` to print `Hello World again`.

```
CREATE PROCEDURE hello_again IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World again');
END;
/

Procedure created.
```

   c. Invoke the `HELLO_AGAIN` procedure with an anonymous block.

```
SET SERVEROUTPUT ON
BEGIN
  hello_again;
END;
/

Hello World again
PL/SQL procedure successfully completed.
```

Practice 1: Solutions

Note: You can find table descriptions and sample data in Appendix B, "Table Descriptions and Data." Click the Save Script button to save your subprograms as `.sql` files in your local file system.
Remember to enable `SERVEROUTPUT` if you have previously disabled it.

1. Create and invoke the `ADD_JOB` procedure and consider the results.

   a. Create a procedure called `ADD_JOB` to insert a new job into the `JOBS` table. Provide the ID and title of the job using two parameters.

```
CREATE OR REPLACE PROCEDURE add_job (
  jobid jobs.job_id%TYPE,
  jobtitle jobs.job_title%TYPE) IS
BEGIN
  INSERT INTO jobs (job_id, job_title)
  VALUES (jobid, jobtitle);
  COMMIT;
END add_job;
/

Procedure created.
```

   b. Compile the code, and invoke the procedure with `IT_DBA` as job ID and `Database Administrator` as job title. Query the `JOBS` table to view the results.

```
EXECUTE add_job ('IT_DBA', 'Database Administrator')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';

PL/SQL Procedure Successfully Completed.
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Database Administrator		

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

```
EXECUTE add_job ('ST_MAN', 'Stock Manager')

BEGIN add_job ('ST_MAN', 'Stock Manager'); END;

*

ERROR at line 1:
ORA-00001: unique constraint (ORA1.JOB_ID_PK) violated
ORA-06512: at "ORA1.ADD_JOB", line 5
ORA-06512: at line 1
```

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

Practice 1: Solutions (continued)

2. Create a procedure called `UPD_JOB` to modify a job in the `JOBS` table.

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

```
CREATE OR REPLACE PROCEDURE upd_job(
  jobid IN jobs.job_id%TYPE,
  jobtitle IN jobs.job_title%TYPE) IS
BEGIN
  UPDATE jobs
  SET    job_title = jobtitle
  WHERE  job_id = jobid;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202, 'No job updated.');
  END IF;
END upd_job;
/

Procedure created.
```

   b. Compile the code; invoke the procedure to change the job title of the job ID `IT_DBA` to `Data Administrator`. Query the `JOBS` table to view the results.

```
EXECUTE upd_job ('IT_DBA', 'Data Administrator')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';

PL/SQL Procedure Successfully Completed.
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Data Administrator		

   Also check the exception handling by trying to update a job that does not exist. (You can use the job ID `IT_WEB` and the job title `Web Master`.)

```
EXECUTE upd_job ('IT_WEB', 'Web Master')

BEGIN upd_job ('IT_WEB', 'Web Master'); END;

*

ERROR at line 1:
ORA-20202: No job updated.
ORA-06512: at "ORA1.UPD_JOB", line 9
ORA-06512: at line 1
```

Practice 1: Solutions (continued)

3. Create a procedure called DEL_JOB to delete a job from the JOBS table.

   a. Create a procedure called DEL_JOB to delete a job. Include the necessary exception handling if no job is deleted.

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

Procedure created.
```

   b. Compile the code; invoke the procedure using job ID IT_DBA. Query the JOBS table to view the results.

```
EXECUTE del_job ('IT_DBA')
SELECT * FROM jobs WHERE job_id = 'IT_DBA';

PL/SQL procedure successfully completed.

no rows selected
```

   Also, check the exception handling by trying to delete a job that does not exist. (Use the IT_WEB job ID.) You should get the message that you used in the exception-handling section of the procedure as output.

```
EXECUTE del_job ('IT_WEB')

BEGIN del_job ('IT_WEB'); END;

*

ERROR at line 1:
ORA-20203: No jobs deleted.
ORA-06512: at "ORA1.DEL_JOB", line 6
ORA-06512: at line 1
```

4. Create a procedure called GET_EMPLOYEE to query the EMPLOYEES table, retrieving the salary and job ID for an employee when provided with the employee ID.

   a. Create a procedure that returns a value from the SALARY and JOB_ID columns for a specified employee ID. Compile the code and remove the syntax errors.

Practice 1: Solutions (continued)

```
CREATE OR REPLACE PROCEDURE get_employee
    (empid IN  employees.employee_id%TYPE,
     sal   OUT employees.salary%TYPE,
     job   OUT employees.job_id%TYPE) IS
BEGIN
  SELECT  salary, job_id
  INTO    sal, job
  FROM    employees
  WHERE   employee_id = empid;
END get_employee;
/

Procedure created.
```

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

```
VARIABLE salary NUMBER
VARIABLE job    VARCHAR2(15)
EXECUTE get_employee(120, :salary, :job)
PRINT salary job

PL/SQL procedure successfully completed.
```

SALARY
8000

JOB
ST_MAN

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

```
EXECUTE get_employee(300, :salary, :job)

BEGIN get_employee(300, :salary, :job); END;

*

ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "ORA1.GET_EMPLOYEE", line 6
ORA-06512: at line 1
```

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

Practice 2: Solutions

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

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

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

Function created.
```

   b. Create a VARCHAR2 host variable called TITLE, allowing a length of 35 characters. Invoke the function with SA_REP job ID to return the value in the host variable. Print the host variable to view the result.

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

PL/SQL procedure successfully completed.
```

TITLE
Sales Representative

2. Create a function called GET_ANNUAL_COMP to return the annual salary computed from an employee's monthly salary and commission passed as parameters.

   a. Develop and store the function GET_ANNUAL_COMP, accepting parameter values for monthly salary and commission. Either or both values passed can be NULL, but the function should still return a non-NULL annual salary. Use the following basic formula to calculate the annual salary:

```
(salary*12) + (commission_pct*salary*12)
```

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

Function created.
```

Practice 2: Solutions (continued)

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

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

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

6 rows selected.

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

   a. Create a function VALID_DEPTID to validate a specified department ID and return a BOOLEAN value of TRUE if the department exists.

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

Function created.
```

Practice 2: Solutions (continued)

b. Create the ADD_EMPLOYEE procedure to add an employee to the EMPLOYEES table. The row should be added to the EMPLOYEES table if the VALID_DEPTID function returns TRUE; otherwise, alert the user with an appropriate message. Provide the following parameters (with defaults specified in parentheses): first_name, last_name, email, job (SA_REP), mgr (145), sal (1000), comm (0), and deptid (30). Use the EMPLOYEES_SEQ sequence to set the employee_id column, and set hire_date to TRUNC(SYSDATE).

```
CREATE OR REPLACE PROCEDURE add_employee(
   first_name employees.first_name%TYPE,
   last_name  employees.last_name%TYPE,
   email      employees.email%TYPE,
   job        employees.job_id%TYPE         DEFAULT 'SA_REP',
   mgr        employees.manager_id%TYPE     DEFAULT 145,
   sal        employees.salary%TYPE         DEFAULT 1000,
   comm       employees.commission_pct%TYPE DEFAULT 0,
   deptid     employees.department_id%TYPE  DEFAULT 30) IS
BEGIN
 IF valid_deptid(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, first_name, last_name, email,
     job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
 ELSE
   RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try again.');
 END IF;
END add_employee;
/

Procedure created.
```


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

Note: If the database server time is not between 8:00 and 18:00, the Secure_employees trigger will be fired on performing any DML operation on the EMPLOYEES table. Disable the aforesaid trigger to overcome this problem.

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

BEGIN add_employee('Jane', 'Harris', 'JAHARRIS', deptid=> 15); END;

*

ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.ADD_EMPLOYEE", line 17
ORA-06512: at line 1
```

Practice 2: Solutions (continued)

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

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

PL/SQL procedure successfully completed.
```

Practice 3: Solutions

1. Create a package specification and body called JOB_PKG, containing a copy of your ADD_JOB, UPD_JOB, and DEL_JOB procedures, as well as your GET_JOB function.
   Tip: Consider saving the package specification and body in two separate files (for example, p3q1_s.sql and p3q1_b.sql for the package specification and body, respectively). Include a SHOW ERRORS statement after the CREATE PACKAGE statement in each file. Alternatively, place all code in one file.
   Note: Use the code in your previously saved script files when creating the package.

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

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

Package created.

No errors.
```


   Note: Consider whether you still need the stand-alone procedures and functions you just packaged.

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

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

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

Practice 3: Solutions (continued)

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

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

END job_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```


c. Invoke your ADD_JOB package procedure by passing the values IT_SYSAN and Systems Analyst as parameters.

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

PL/SQL procedure successfully completed.
```

d. Query the JOBS table to see the result.

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

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst		

Practice 3: Solutions (continued)

2. Create and invoke a package that contains private and public constructs.

   a. Create a package specification and package body called `EMP_PKG` that contains your `ADD_EMPLOYEE` and `GET_EMPLOYEE` procedures as public constructs, and include your `VALID_DEPTID` function as a private construct.

   Package specification:

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   Package body:

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

Practice 3: Solutions (continued)

Package body (continued):

```
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

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

Package body created.

No errors.
```

Practice 3: Solutions (continued)

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

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

BEGIN emp_pkg.add_employee('Jane', 'Harris','JAHARRIS', deptid => 15);
END;

*

ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.EMP_PKG", line 31
ORA-06512: at line 1
```

c. Invoke the `GET_EMPLOYEE` package procedure by using department ID 80 for employee `David Smith` with e-mail `DASMITH`.

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

PL/SQL procedure successfully completed.
```

Note: If you are using SQL Developer, your compile time errors are displayed in the Message Log. If you are using SQL*Plus or iSQL*Plus to create your stored code, use the SQL*Plus `SHOW ERRORS` to view compile errors.

Practice 4: Solutions

1. Copy and modify the code for the EMP_PKG package that you created in Practice 3, Exercise 2, and overload the ADD_EMPLOYEE procedure.

   a. In the package specification, add a new procedure called ADD_EMPLOYEE, which accepts three parameters: the first name, last name, and department ID. Compile the changes.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. Implement the new ADD_EMPLOYEE procedure in the package body so that it formats the e-mail address in uppercase characters, using the first letter of the first name concatenated with the first seven letters of the last name. The procedure should call the existing ADD_EMPLOYEE procedure to perform the actual INSERT operation using its parameters and formatted e-mail to supply the values. Compile the changes.

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

Practice 4: Solutions (continued)

```
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

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

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

Practice 4: Solutions (continued)

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

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

PL/SQL procedure successfully completed.
```

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

a. In the specification, add a GET_EMPLOYEE function that accepts the parameter called emp_id based on the employees.employee_id%TYPE type, and a second GET_EMPLOYEE function that accepts a parameter called family_name of the employees.last_name%TYPE type. Both functions should return an EMPLOYEES%ROWTYPE. Compile the changes.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Practice 4: Solutions (continued)

b. In the package body, implement the first GET_EMPLOYEE function to query an employee by his or her ID, and the second to use the equality operator on the value supplied in the family_name parameter. Compile the changes.

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

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

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

Practice 4: Solutions (continued)

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

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

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

END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 4: Solutions (continued)

c. Add a utility procedure PRINT_EMPLOYEE to the package that accepts an EMPLOYEES%ROWTYPE as a parameter and displays the department_id, employee_id, first_name, last_name, job_id, and salary for an employee on one line, using DBMS_OUTPUT. Compile the changes.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS

Prackage created.

No Errors.

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

Practice 4: Solutions (continued)

```
PROCEDURE add_employee(
  first_name employees.first_name%TYPE,
  last_name employees.last_name%TYPE,
  email employees.email%TYPE,
  job employees.job_id%TYPE DEFAULT 'SA_REP',
  mgr employees.manager_id%TYPE DEFAULT 145,
  sal employees.salary%TYPE DEFAULT 1000,
  comm employees.commission_pct%TYPE DEFAULT 0,
  deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
  IF valid_deptid(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, first_name, last_name, email,
      job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
  ELSE
    RAISE_APPLICATION_ERROR (-20204,
                             'Invalid department ID. Try again.');
  END IF;
END add_employee;

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

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

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

Practice 4: Solutions (continued)

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

  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.salary);
  END;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```


d. Use an anonymous block to invoke the `EMP_PKG.GET_EMPLOYEE` function with an employee ID of `100`, and family name of `'Joplin'`. Use the `PRINT_EMPLOYEE` procedure to display the results for each row returned.

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

90 100 Steven King AD_PRES 24000
30 209 Samuel Joplin SA_REP 1000

PL/SQL procedure successfully completed.
```

Note: The employee ID `209` for Samuel Joplin is allocated by using an Oracle sequence object. You may receive a different value when you execute the PL/SQL block shown in this solution.

Practice 4: Solutions (continued)

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

   a. In the package specification, create a procedure called INIT_DEPARTMENTS with no parameters.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. In the package body, implement the INIT_DEPARTMENTS procedure to store all department IDs in a private PL/SQL index-by table named valid_departments containing BOOLEAN values. Use the department_id column value as the index to create the entry in the index-by table to indicate its presence, and assign the entry a value of TRUE. Declare the valid_departments variable and its type definition boolean_tabtype before all procedures in the body.

Practice 4: Solutions (continued)

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

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
  RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    ...
  END valid_deptid;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    ...
  END add_employee;

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE) IS
    email employees.email%type;
  BEGIN
    ...
  END;

  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE) IS
  BEGIN
    ...
  END get_employee;

  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype IS
    emprec employees%rowtype;
  BEGIN
    ...
  END;
```

Practice 4: Solutions (continued)

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

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

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

Package body created.
No errors.
```

c. In the body, create an initialization block that calls the `INIT_DEPARTMENTS` procedure to initialize the table. Compile the changes.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  ...
  PROCEDURE init_departments IS
  BEGIN
    FOR rec IN (SELECT department_id FROM departments)
    LOOP
      valid_departments(rec.department_id) := TRUE;
    END LOOP;
  END;
BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS
Package body created.

No errors
```

Practice 4: Solutions (continued)

4. Change the VALID_DEPTID validation processing to use the private PL/SQL table of department IDs.

   a. Modify VALID_DEPTID to perform its validation by using the PL/SQL table of department ID values. Compile the changes.

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

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

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

Package body created.

No errors.
```


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

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

BEGIN emp_pkg.add_employee('James', 'Bond', 15); END;

*
ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.EMP_PKG", line 32
ORA-06512: at "ORA1.EMP_PKG", line 43
ORA-06512: at line 1
```

Practice 4: Solutions (continued)

The insert operation to add the employee fails with an exception, because department 15 does not exist.

c. Insert a new department with ID 15 and name Security, and commit the changes.

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

1 row created.

Commit complete.
```

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

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

BEGIN emp_pkg.add_employee('James', 'Bond', 15); END;

*
ERROR at line 1:
ORA-20204: Invalid department ID. Try again.
ORA-06512: at "ORA1.EMP_PKG", line 32
ORA-06512: at "ORA1.EMP_PKG", line 43
ORA-06512: at line 1
```

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

e. Execute the `EMP_PKG.INIT_DEPARTMENTS` procedure to update the internal PL/SQL table with the latest departmental data.

```
EXECUTE EMP_PKG.INIT_DEPARTMENTS

PL/SQL procedure successfully completed.
```

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

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

PL/SQL procedure successfully completed.
```

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

Practice 4: Solutions (continued)

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

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

1 row deleted.
1 row deleted.
Commit complete.
PL/SQL procedure successfully completed.
```

5. Reorganize the subprograms in the package specification body so that they are in alphabetical sequence.

   a. Edit the package specification and reorganize subprograms alphabetically. In *i*SQL*Plus, load and compile the package specification. What happens?

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS
```

It compiles successfully.
Note: The package may already have its subprograms in alphabetical sequence.

Practice 4: Solutions (continued)

b. Edit the package body and reorganize all subprograms alphabetically. In *i*SQL*Plus, load and compile the package specification. What happens?

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

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204, 'Invalid department ID. Try
again.');
    END IF;
  END add_employee;

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

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

Practice 4: Solutions (continued)

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

FUNCTION get_employee(family_name employees.last_name%type)
  return employees%rowtype IS
  emprec employees%rowtype;
BEGIN
  SELECT * INTO emprec
  FROM employees
  WHERE last_name = family_name;
  RETURN emprec;
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(emprec employees%rowtype) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                       emprec.employee_id||' '||
                       emprec.first_name||' '||
                       emprec.last_name||' '||
                       emprec.job_id||' '||
                       emprec.salary);
END;

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

Practice 4: Solutions (continued)

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

Warning: Package Body created with compilation errors.

Errors for PACKAGE BODY EMP_PKG:

LINE/COL ERROR
-------- -----------------------------------------------------------------
16/5     PL/SQL: Statement ignored
16/8     PLS-00313: 'VALID_DEPTID' not declared in this scope
```

It does not compile successfully because the `VALID_DEPTID` function is referenced before it is declared.

c. Fix the compilation error by using a forward declaration in the body for the offending subprogram reference. Load and re-create the package body. What happens? Save the package code in a script file.

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

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

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;
  ...
```

Practice 4: Solutions (continued)

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

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

A forward declaration for the `VALID_DEPTID` function enables the package body to be compiled successfully.

If you have time, complete the following exercise:

6. Wrap the `EMP_PKG` package body and re-create it.

   a. Query the data dictionary to view the source for the `EMP_PKG` body.

```
SELECT text
FROM user_source
WHERE name = 'EMP_PKG'
AND type = 'PACKAGE BODY'
ORDER BY line;
```

TEXT
PACKAGE BODY emp_pkg IS
TYPE boolean_tabtype IS TABLE OF BOOLEAN
INDEX BY BINARY_INTEGER;
valid_departments boolean_tabtype;
BEGIN
init_departments;
END emp_pkg;

100 rows selected.

Practice 4: Solutions (continued)

b. Start a command window and execute the `WRAP` command-line utility to wrap the body of the `EMP_PKG` package. Give the output file name a `.plb` extension.
Hint: Copy the file (which you saved in step 5c) containing the package body to a file called `emp_pkg_b.sql`.

```
WRAP INAME=emp_pkg_b.sql

PL/SQL Wrapper: Release 10.2.0.1.0- Production on Tue Nov 14 03:49:53
2006

Copyright (c) 1993, 2004,Oracle.  All Rights Reserved.

Processing emp_pkg_b.sql to emp_pkg_b.plb
```

c. Using *i*SQL*Plus, load and execute the `.plb` file containing the wrapped source.

```
CREATE OR REPLACE PACKAGE BODY emp_pkg wrapped
0
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
 :
 :
be 4 0
67 3 0
15 2 0
133 6 0
5 1 0
5d 3 0
193 1 8
0

/
SHOW ERRORS

Package body created.

No errors.
```

Practice 4: Solutions (continued)

d. Query the data dictionary to display the source for the `EMP_PKG` package body again. Are the original source code lines readable?

```
SELECT text
FROM user_source
WHERE name = 'EMP_PKG'
AND type = 'PACKAGE BODY'
ORDER BY line;
```

TEXT
PACKAGE BODY emp_pkg wrapped 0 abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd abcd 3 b 9200000 1 4 0 44 2 :e:
121 2 11b 11c 1 129 1 133 1 13b 1 143 1 14c 1 150 2 14f 150 1 14b 2 158 15b 1
1fc 227 1 4 0 235 0 1 14 b 24 0 1 1 1 1 1 1 1 8 1 1 0 0 0 0 0 0 0 0 0 ca 4 0 36 3 u
b5 4 0 2d 3 0 1b5 a 0 174 7 0 143 6 0 20e b 0 ab 1 4 23 1 3 10b 5 0 3f 3 0 163 1 7
132 1 6 f7 1 5 49 3 0 197 9 0 101 5 0 53 3 0 1ff 1 b 14 1 2 e 1 0 ac 4 0 24 3 0 f8 5
0 164 7 0 3 0 1 1b4 1 a 200 b 0 be 4 0 67 3 0 15 2 0 133 6 0 5 1 0 5d 3 0 193 1 8
0

The source code for the body is no longer readable. You can view the wrapped source, but the orginal source code is not shown. For this reason, make sure you always have a secure copy of your source code in files outside the database when using the `WRAP` utility.

Practice 5: Solutions

1. Create a procedure called EMPLOYEE_REPORT that generates an employee report in a file in the operating system, using the UTL_FILE package. The report should generate a list of employees who have exceeded the average salary of their department.

   a. Your program should accept two parameters. The first parameter is the output directory. The second parameter is the name of the text file that is written.
   Note: Use the directory location value UTL_FILE. Add an exception-handling section to handle errors that may be encountered when using the UTL_FILE package.

   The following is a sample output from the report file:

```
Employees who earn more than average salary:
REPORT GENERATED ON 26-FEB-04
Hartstein                      20      $13,000.00
Raphaely                       30      $11,000.00
Marvis                         40      $6,500.00
...
*** END OF REPORT ***
```

```
CREATE OR REPLACE PROCEDURE employee_report(
  dir IN VARCHAR2, filename IN VARCHAR2) IS
  f UTL_FILE.FILE_TYPE;
  CURSOR avg_csr IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                    FROM  employees inner
                   GROUP BY outer.department_id)
    ORDER BY department_id;
BEGIN
  f := UTL_FILE.FOPEN(dir, filename,'w');
  UTL_FILE.PUT_LINE(f, 'Employees who earn more than average salary: ');
  UTL_FILE.PUT_LINE(f, 'REPORT GENERATED ON ' ||SYSDATE);
  UTL_FILE.NEW_LINE(f);
  FOR emp IN avg_csr
  LOOP
    UTL_FILE.PUT_LINE(f,
    RPAD(emp.last_name, 30) || ' ' ||
    LPAD(NVL(TO_CHAR(emp.department_id,'9999'),'-'), 5) || ' ' ||
    LPAD(TO_CHAR(emp.salary, '$99,999.00'), 12));
  END LOOP;
  UTL_FILE.NEW_LINE(f);
  UTL_FILE.PUT_LINE(f, '*** END OF REPORT ***');
  UTL_FILE.FCLOSE(f);
END employee_report;
/

Procedure created.
```

Practice 5: Solutions (continued)

b. Invoke the program, using the second parameter with a name such as `sal_rptxx.txt`, where `xx` represents your user number (for example, 01, 15, and so on).

```
EXECUTE employee_report('UTL_FILE','sal_rpt01.txt')

PL/SQL Procedure sucessfully completed.
```

Note: The data displays the employee's last name, department ID, and salary. Ask your instructor to provide instructions on how to obtain the report file from the server using the `Putty PSFTP` utility.

After you use `PSTFP` to retrieve your generated file, it should contain something similar to the following example:

```
Employees who earn more than average salary:
REPORT GENERATED ON  16-FEB-04

Hartstein                       20   $13,000.00
Raphaely                        30   $11,000.00
Mavris                          40    $6,500.00
Weiss                           50    $8,000.00
Kaufling                        50    $7,900.00
Fripp                           50    $8,200.00
Vollman                         50    $6,500.00
Hunold                          60    $9,000.00
Baer                            70   $10,000.00
Russell                         80   $14,000.00
Bernstein                       80    $9,500.00
Olsen                           80    $8,000.00
  :
  :
Errazuriz                       80   $12,000.00
Zlotkey                         80   $10,500.00
Cambrault                       80   $11,000.00
King                            90   $24,000.00
Kochhar                         90   $17,000.00
De Haan                         90   $17,000.00
Greenberg                      100   $12,000.00
Faviet                         100    $9,000.00
Chen                           100    $8,200.00
Sciarra                        100    $7,700.00
Urman                          100    $7,800.00
Popp                           100    $6,900.00
Higgins                        110   $12,000.00
Gietz                          110    $8,300.00
Grant                            -    $7,000.00

*** END OF REPORT ***
```

Practice 5: Solutions (continued)

2. Create a new procedure called `WEB_EMPLOYEE_REPORT` that generates the same data as the `EMPLOYEE_REPORT`.

   a. First, execute `SET SERVEROUTPUT ON`, and then execute `htp.print('hello')` followed by executing `OWA_UTIL.SHOWPAGE`. The exception messages generated can be ignored.

```
SET SERVEROUTPUT ON
EXECUTE HTP.PRINT('hello')
EXECUTE OWA_UTIL.SHOWPAGE

BEGIN htp.print('hello'); END;

*

ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.OWA_UTIL", line 325
ORA-06512: at "SYS.HTP", line 1322
ORA-06512: at "SYS.HTP", line 1397
ORA-06512: at "SYS.HTP", line 1684
ORA-06512: at line 1
PL/SQL procedure successfully completed.
```

   These steps are performed to ensure that the messages are not generated again. However, remember that the HTP package is intended to be used in the Oracle HTTP Server context, not *i*SQL*Plus.

   b. Write the `WEB_EMPLOYEE_REPORT` procedure using the `HTP` package to generate an HTML report of employees with a salary greater than the average for their departments. If you know HTML, create an HTML table; otherwise, create simple lines of data.
   Hint: Copy the cursor definition and the `FOR` loop from the `EMPLOYEE_REPORT` procedure for the basic structure for your Web report.

```
CREATE OR REPLACE PROCEDURE web_employee_report IS
  CURSOR avg_csr IS
    SELECT last_name, department_id, salary
    FROM employees outer
    WHERE salary > (SELECT AVG(salary)
                    FROM  employees inner
                    GROUP BY outer.department_id)
    ORDER BY department_id;
```

Practice 5: Solutions (continued)

```
BEGIN
  htp.htmlopen;
  htp.headopen;
  htp.title('Employee Salary Report');
  htp.headclose;
  htp.bodyopen;
  htp.header(1, 'Employees who earn more than average salary');
  htp.print('REPORT GENERATED ON' || to_char(SYSDATE, 'DD-MON-YY'));
  htp.br;
  htp.hr;
  htp.tableOpen;
  htp.tablerowOpen;
  htp.tableHeader('Last Name');
  htp.tableHeader('Department');
  htp.tableHeader('Salary');
  htp.tablerowclose;

  FOR emp IN avg_csr
  LOOP
    htp.tablerowOpen;
    htp.tabledata(emp.last_name);
    htp.tabledata(NVL(TO_CHAR(emp.department_id,'9999'),'-'));
    htp.tabledata(TO_CHAR(emp.salary, '$99,999.00'));
    htp.tablerowclose;
  END LOOP;

  htp.tableclose;
  htp.hr;
  htp.print('*** END OF REPORT ***');
  htp.bodyclose;
  htp.htmlclose;
END web_employee_report;
/
show errors


Procedure created.

No errors.
```


c. Execute the procedure using *i*SQL*Plus to generate the HTML data into a server buffer, and execute the `OWA_UTIL.SHOWPAGE` procedure to display contents of the buffer. Remember that `SERVEROUTPUT` should be `ON` before you execute the code.

```
EXECUTE web_employee_report
EXECUTE owa_util.showpage

PL/SQL procedure successfully completed.

:
```

Practice 5: Solutions (continued)

```
<HTML>
<HEAD>
<TITLE>Employee Salary Report</TITLE>
</HEAD>
<BODY>
<H1>Employees who earn more than average salary</H1>
REPORT GENERATED ON16-FEB-04
<BR>
<HR>
<TABLE >
<TR>
<TH>Last Name</TH>
<TH>Department</TH>
<TH>Salary</TH>
</TR>
<TR>
<TD>Hartstein</TD>
<TD> 20</TD>
<TD> $13,000.00</TD>
</TR>
<TR>
<TD>Raphaely</TD>
<TD> 30</TD>
<TD> $11,000.00</TD>
</TR>
<TR>
<TD>Mavris</TD>
<TD> 40</TD>
<TD> $6,500.00</TD>
</TR>
<TR>
<TD>Weiss</TD>
<TD> 50</TD>
<TD> $8,000.00</TD>
</TR>
<TR>
<TD>Kaufling</TD>
<TD> 50</TD>
<TD> $7,900.00</TD>
</TR>
<TR>
<TD>Fripp</TD>
<TD> 50</TD>
<TD> $8,200.00</TD>
</TR>
<TR>
<TD>Vollman</TD>
<TD> 50</TD>
<TD> $6,500.00</TD>
</TR>
```

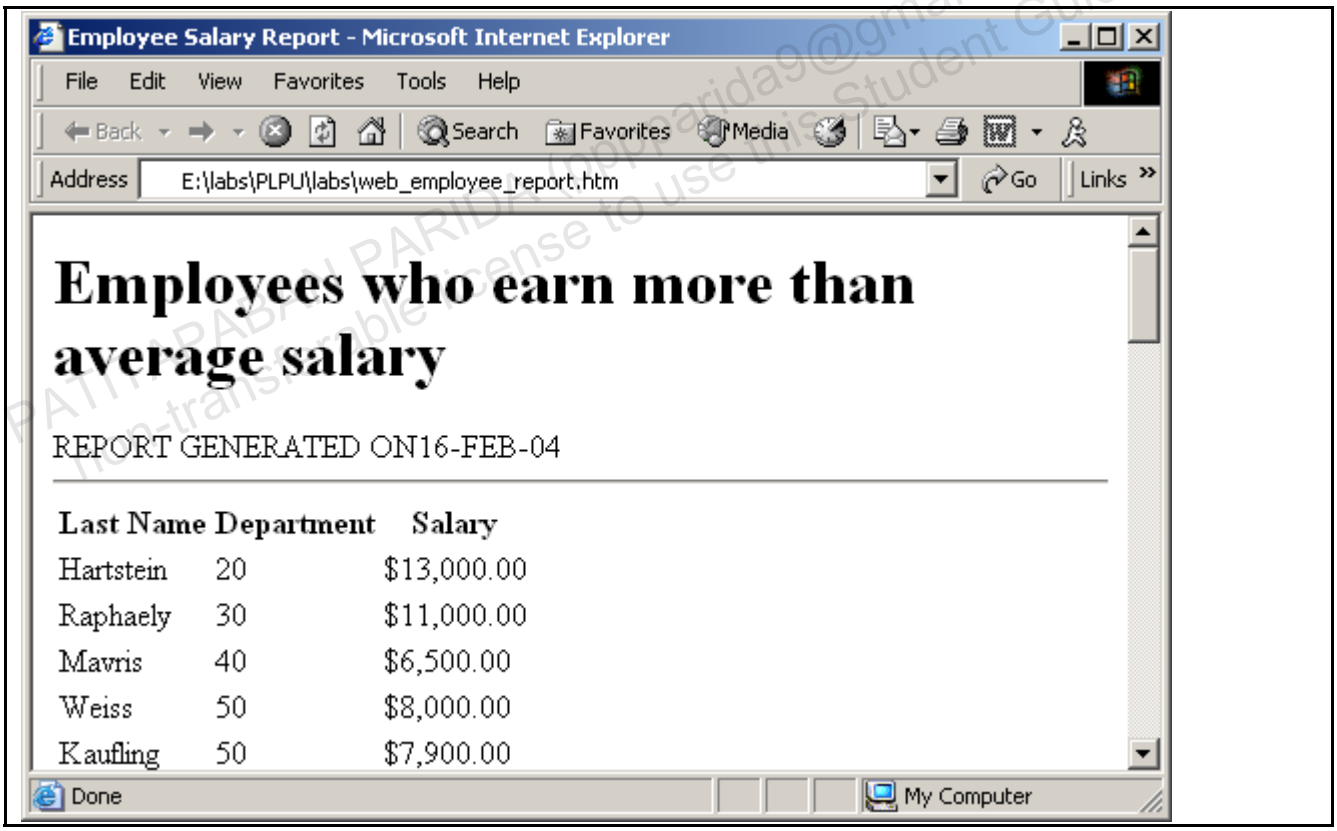
Practice 5: Solutions (continued)

```
<TR>
<TD>Hunold</TD>
<TD> 60</TD>
<TD> $9,000.00</TD>
</TR>
<TR>
<TD>Baer</TD>
<TD> 70</TD>
<TD> $10,000.00</TD>
</TR>
<TR> <TD>Russell</TD> <TD> 80</TD> <TD> $14,000.00</TD> </TR> <TR>
<TD>Bernstein</TD> <TD> 80</TD> <TD>
$9,500.00</TD> </TR> <TR> <TD>Olsen</TD> <TD> 80</TD> <TD> $8,000.00</TD>
</TR> <TR> <TD>Vishney</TD> <TD> 80</TD> <TD> $10,500.00</TD>
</TR> <TR> <TD>Sewall</TD> <TD> 80</TD> <TD> $7,000.00</TD> </TR> <TR>
<TD>Doran</TD> <TD> 80</TD> <TD>
$7,500.00</TD> </TR> <TR> <TD>Smith</TD> <TD> 80</TD> <TD> $8,000.00</TD>
</TR> <TR> <TD>McEwen</TD> <TD> 80</TD> <TD> $9,000.00</TD>
</TR> <TR> <TD>Sully</TD> <TD> 80</TD> <TD> $9,500.00</TD> </TR> <TR>
<TD>King</TD> <TD> 80</TD> <TD>
$10,000.00</TD> </TR> <TR> <TD>Tuvault</TD> <TD> 80</TD> <TD>
$7,000.00</TD> </TR> <TR> <TD>Cambrault</TD> <TD> 80</TD> <TD>
$7,500.00</TD>
</TR> <TR> <TD>Bates</TD> <TD> 80</TD> <TD> $7,300.00</TD> </TR> <TR>
<TD>Smith</TD> <TD> 80</TD> <TD>
$7,400.00</TD> </TR> <TR> <TD>Fox</TD> <TD> 80</TD> <TD> $9,600.00</TD>
</TR> <TR> <TD>Bloom</TD> <TD> 80</TD> <TD> $10,000.00</TD> </TR>
<TR> <TD>Ozer</TD> <TD> 80</TD> <TD> $11,500.00</TD> </TR> <TR>
<TD>Ande</TD> <TD> 80</TD> <TD> $6,400.00</TD> </TR> <TR> <TD>Lee</TD>
<TD>
80</TD> <TD> $6,800.00</TD> </TR> <TR> <TD>Marvins</TD> <TD> 80</TD> <TD>
$7,200.00</TD> </TR> <TR>
<TD>Greene</TD> <TD> 80</TD> <TD> $9,500.00</TD> </TR> <TR>
<TD>Livingston</TD> <TD> 80</TD> <TD> $8,400.00</TD> </TR> <TR>
<TD>Taylor</TD> <TD>
80</TD> <TD> $8,600.00</TD> </TR> <TR> <TD>Hutton</TD> <TD> 80</TD> <TD>
$8,800.00</TD> </TR>
<TR> <TD>Abel</TD> <TD> 80</TD> <TD> $11,000.00</TD> </TR> <TR>
<TD>Hall</TD> <TD> 80</TD> <TD> $9,000.00</TD> </TR> <TR> <TD>Tucker</TD>
<TD>
80</TD> <TD> $10,000.00</TD> </TR> <TR> <TD>Partners</TD> <TD> 80</TD>
<TD> $13,500.00</TD> </TR> <TR>
<TD>Errazuriz</TD> <TD> 80</TD> <TD> $12,000.00</TD> </TR> <TR>
<TD>Zlotkey</TD> <TD> 80</TD> <TD>
$10,500.00</TD> </TR> <TR> <TD>Cambrault</TD> <TD> 80</TD> <TD>
$11,000.00</TD> </TR> <TR> <TD>King</TD> <TD> 90</TD> <TD>
$24,000.00</TD> </TR>
<TR> <TD>Kochhar</TD> <TD> 90</TD> <TD> $17,000.00</TD> </TR> <TR> <TD>De
Haan</TD> <TD> 90</TD> <TD>
$17,000.00</TD> </TR> <TR> <TD>Greenberg</TD> <TD> 100</TD> <TD>
$12,000.00</TD> </TR>
```

Practice 5: Solutions (continued)

```
<TR> <TD>Faviet</TD> <TD> 100</TD> <TD> $9,000.00</TD>
</TR> <TR> <TD>Chen</TD> <TD> 100</TD> <TD> $8,200.00</TD> </TR> <TR>
<TD>Sciarra</TD> <TD> 100</TD> <TD>
$7,700.00</TD> </TR> <TR> <TD>Urman</TD> <TD> 100</TD> <TD>
$7,800.00</TD> </TR> <TR> <TD>Popp</TD> <TD> 100</TD> <TD> $6,900.00</TD>
</TR>
<TR> <TD>Higgins</TD> <TD> 110</TD> <TD> $12,000.00</TD> </TR> <TR>
<TD>Gietz</TD> <TD> 110</TD> <TD>
$8,300.00</TD> </TR> <TR> <TD>Grant</TD> <TD>-</TD> <TD> $7,000.00</TD>
</TR> </TABLE> <HR> *** END OF REPORT *** </BODY> </HTML>
PL/SQL procedure successfully completed.
```

d. Create an HTML file called `web_employee_report.htm` containing the output result text that you select and copy from the opening `<HTML>` tag to the closing `</HTML>` tag. Paste the copied text into the file and save it to disk. Double-click the file to display the results in your default browser.



Practice 5: Solutions (continued)

3. Your boss wants to run the employee report frequently. You create a procedure that uses the DBMS_SCHEDULER package to schedule the EMPLOYEE_REPORT procedure for execution. You should use parameters to specify a frequency, and an optional argument to specify the number of minutes after which the scheduled job should be terminated.

   a. Create a procedure called SCHEDULE_REPORT that provides the following two parameters:
   - interval to specify a string indicating the frequency of the scheduled job
   - minutes to specify the total life in minutes (default of 10) for the scheduled job, after which it is terminated. The code will divide the duration by the quantity (24 × 60) when it is added to the current date and time to specify the termination time.

   When the procedure creates a job, with the name of EMPSAL_REPORT by calling DBMS_SCHEDULER.CREATE_JOB, the job should be enabled and scheduled for the PL/SQL block to start immediately. You must schedule an anonymous block to invoke the EMPLOYEE_REPORT procedure so that the file name can be updated with a new time, each time the report is executed. EMPLOYEE_REPORT is given the directory name supplied by your instructor for task 1, and the file name parameter is specified in the following format:
   sal_rptxx_hh24-mi-ss.txt, where xx is your assigned user number and hh24-mi-ss represents the hours, minutes, and seconds

   Use the following local PL/SQL variable to construct a PL/SQL block:

```
plsql_block VARCHAR2(200) :=
 'BEGIN'||
 ' EMPLOYEE_REPORT(''UTL_FILE'',''||
 '''sal_rptxx_''||to_char(sysdate,''HH24-MI-SS'')||''.txt'');'||
 'END;';
```

   This code is provided to help you because it is a nontrivial PL/SQL string to construct. In the PL/SQL block, **xx** is your student number.

```
CREATE OR REPLACE PROCEDURE schedule_report(
  interval VARCHAR2, minutes NUMBER := 10) IS
  plsql_block VARCHAR2(200) :=
    'BEGIN'||
    ' EMPLOYEE_REPORT(''UTL_FILE'',''||
       '''sal_rpt01_''||to_char(sysdate,''HH24-MI-SS'')||''.txt''); '||
    'END;';
BEGIN
```

Practice 5: Solutions (continued)

```
  DBMS_SCHEDULER.CREATE_JOB(
     job_name => 'EMPSAL_REPORT',
     job_type => 'PLSQL_BLOCK',
     job_action => plsql_block,
     start_date => SYSDATE,
     repeat_interval => interval,
     end_date => SYSDATE + minutes/(24*60),
     enabled => TRUE);
END;
/
SHOW ERRORS

Procedure created.

No errors.
```

b. Test the SCHEDULE_REPORT procedure by executing it with a parameter specifying a frequency of every 2 minutes and a termination time 10 minutes after it starts.
Note: You will have to connect to the database server by using PSFTP to check whether your files are created.

```
EXECUTE schedule_report('FREQUENCY=MINUTELY;INTERVAL=2', 10)

PL/SQL procedure successfully completed.
```

c. During and after the process, you can query job_name and enabled columns from the USER_SCHEDULER_JOBS table to check whether the job still exists.

```
SELECT job_name, enabled
FROM user_scheduler_jobs;
```

Note: This query should return no rows after 10 minutes have elapsed.

Practice 6: Solutions

1. Create a package called TABLE_PKG that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.

   a. Create a package specification with the following procedures:

```
PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2)
PROCEDURE add_row(table_name VARCHAR2, col_values VARCHAR2,
  cols VARCHAR2 := NULL)
PROCEDURE upd_row(table_name VARCHAR2, set_values VARCHAR2,
  conditions VARCHAR2 := NULL)
PROCEDURE del_row(table_name VARCHAR2,
  conditions VARCHAR2 := NULL)
PROCEDURE remove(table_name VARCHAR2)
```

   Ensure that subprograms manage optional default parameters with NULL values.

```
CREATE OR REPLACE PACKAGE table_pkg IS
  PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2);
  PROCEDURE add_row(table_name VARCHAR2, col_values VARCHAR2,
    cols VARCHAR2 := NULL);
  PROCEDURE upd_row(table_name VARCHAR2, set_values VARCHAR2,
    conditions VARCHAR2 := NULL);
  PROCEDURE del_row(table_name VARCHAR2, conditions VARCHAR2 := NULL);
  PROCEDURE remove(table_name VARCHAR2);
END table_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

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

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

  PROCEDURE make(table_name VARCHAR2, col_specs VARCHAR2) IS
    stmt VARCHAR2(200) := 'CREATE TABLE '|| table_name ||
                          ' (' || col_specs || ')';
  BEGIN
    execute(stmt);
  END;
```

Practice 6: Solutions (continued)

```
  PROCEDURE add_row(table_name VARCHAR2, col_values VARCHAR2,
    cols VARCHAR2 := NULL) IS
    stmt VARCHAR2(200) := 'INSERT INTO '|| table_name;
  BEGIN
    IF cols IS NOT NULL THEN
       stmt := stmt || ' (' || cols || ')';
    END IF;
    stmt := stmt || ' VALUES (' || col_values || ')';
    execute(stmt);
  END;

  PROCEDURE upd_row(table_name VARCHAR2, set_values VARCHAR2,
    conditions VARCHAR2 := NULL) IS
    stmt VARCHAR2(200) := 'UPDATE '|| table_name || ' SET ' ||
set_values;
  BEGIN
    IF conditions IS NOT NULL THEN
       stmt := stmt || ' WHERE ' || conditions;
    END IF;
    execute(stmt);
  END;

  PROCEDURE del_row(table_name VARCHAR2, conditions VARCHAR2 := NULL) IS
    stmt VARCHAR2(200) := 'DELETE FROM '|| table_name;
  BEGIN
    IF conditions IS NOT NULL THEN
       stmt := stmt || ' WHERE ' || conditions;
    END IF;
    execute(stmt);
  END;

  PROCEDURE remove(table_name VARCHAR2) IS
    csr_id INTEGER;
    stmt VARCHAR2(100) := 'DROP TABLE '||table_name;
  BEGIN
    csr_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_OUTPUT.PUT_LINE(stmt);
    DBMS_SQL.PARSE(csr_id, stmt, DBMS_SQL.NATIVE);
    -- Parse executes DDL statements,no EXECUTE is required.
    DBMS_SQL.CLOSE_CURSOR(csr_id);
  END;

END table_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 6: Solutions (continued)

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

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

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

PL/SQL procedure successfully completed.
```

d. Describe the MY_CONTACTS table structure.

```
DESCRIBE my_contacts
```

Name	Null?	Type
ID		NUMBER(4)
NAME		VARCHAR2(40)

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

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

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

PL/SQL procedure successfully completed.
```

f. Query the MY_CONTACTS table contents.

```
SELECT *
FROM my_contacts;
```

ID	NAME
1	Geoff Gallus
2	Nancy
3	Sunitha Patel
4	Valli Pataballa

g. Execute the DEL_ROW package procedure to delete a contact with ID value 1.

```
EXECUTE table_pkg.del_row('my_contacts', 'id=1')

PL/SQL procedure successfully completed.
```

Practice 6: Solutions (continued)

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

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

```
EXEC table_pkg.upd_row('my_contacts','name=''Nancy Greenberg'''','id=2')

PL/SQL procedure successfully completed.
```

i. Select the data from the `MY_CONTACTS` table again to view the changes.

```
SELECT *
FROM my_contacts;
```

ID	NAME
2	Nancy Greenberg
3	Sunitha Patel
4	Valli Pataballa

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

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

PL/SQL procedure successfully completed.

ERROR:
ORA-04043: object my_contacts does not exist
```


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

a. In the specification, create a package procedure called `MAKE` that accepts the name of a PL/SQL program unit to be compiled.

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

Package created.

No errors.
```

Practice 6: Solutions (continued)

b. In the body, the MAKE procedure should call a private function called GET_TYPE to determine the PL/SQL object type from the data dictionary, and return the type name (use PACKAGE for a package with a body) if the object exists; otherwise, it should return a NULL. If the object exists, MAKE dynamically compiles it with the ALTER statement.

```
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;
END compile_pkg;
/
SHOW ERRORS
```

Practice 6: Solutions (continued)

```
Package body created.

No errors.
```

c. Use the COMPILE_PKG.MAKE procedure to compile the EMPLOYEE_REPORT procedure, the EMP_PKG package, and a nonexistent object called EMP_DATA.

```
EXECUTE compile_pkg.make('employee_report')
EXECUTE compile_pkg.make('emp_pkg')
EXECUTE compile_pkg.make('emp_data')

ALTER PROCEDURE employee_report COMPILE
PL/SQL procedure successfully completed.

ALTER PACKAGE emp_pkg COMPILE
PL/SQL procedure successfully completed

BEGIN compile_pkg.make('emp_data'); END;

*

ERROR at line 1:
ORA-20001: Subprogram 'emp_data' does not exist
ORA-06512: at "ORA1.COMPILE_PKG", line 39
ORA-06512: at line 1
```

3. Add a procedure to the COMPILE_PKG that uses the DBMS_METADATA to obtain a DDL statement that can regenerate a named PL/SQL subprogram, and writes the DDL to a file by using the UTL_FILE package.

a. In the package specification, create a procedure called REGENERATE that accepts the name of a PL/SQL component to be regenerated. Declare a public VARCHAR2 variable called dir initialized with the directory alias value 'UTL_FILE'. Compile the specification.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  dir VARCHAR2(100) := 'UTL_FILE';
  PROCEDURE make(name VARCHAR2);
  PROCEDURE regenerate(name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Note: Initialize the correct path name in the dir variable value for your course.

Practice 6: Solutions (continued)

b. In the package body, implement the `REGENERATE` procedure so that it uses the `GET_TYPE` function to determine the PL/SQL object type from the supplied name. If the object exists, then obtain the DDL used to create the component using the procedure `DBMS_METADATA.GET_DDL`, which must be provided with the object name in uppercase text.
Save the DDL statement in a file by using the `UTL_FILE.PUT` procedure. Write the file in the directory path stored in the public variable called `dir` (from the specification). Construct a file name (in lowercase characters) by concatenating the `USER` function, an underscore, and the object name with a `.sql` extension. For example: `ora1_myobject.sql`. Compile the body.

```
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;
```

Practice 6: Solutions (continued)

```
  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 regenerate (name VARCHAR2) IS
    file UTL_FILE.FILE_TYPE;
    filename VARCHAR2(100) := LOWER(USER ||'_'|| name ||'.sql');
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      file := UTL_FILE.FOPEN(dir, filename, 'w');
      UTL_FILE.PUT(file,
        DBMS_METADATA.GET_DDL(proc_type, UPPER(name)));
      UTL_FILE.FCLOSE(file);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Object with '''|| name ||''' does not exist');
    END IF;
  END regenerate;

END compile_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Execute the COMPILE_PKG.REGENERATE procedure by using the name of the TABLE_PKG created in the first task of this practice.

```
EXECUTE compile_pkg.regenerate('TABLE_PKG')
```

Note: If required, you can execute the following statement to set the directory for the file:

```
EXECUTE compile_pkg.dir := '<utl_file_dir>';
```

Practice 6: Solutions (continued)

d. Use Putty FTP to get the generated file from the server to your local directory. Edit the file to place a / terminator character at the end of a `CREATE` statement (if required). Cut and paste the results into the *i*SQL*Plus buffer and execute the statement.

Here is a sample Putty FTP session:

```
psftp> open esslin05
login as: teach7
Using username "teach7".
Password: ******
Remote working directory is /home1/teach7
psftp> cd UTL_FILE
Remote directory is now /home1/teach7/UTL_FILE
psftp> lcd E:\labs\PLPU\labs
New local directory is E:\labs\PLPU\labs
psftp> get ora1_emp_pkg.sql
remote:/home1/teach7/UTL_FILE/ora1_emp_pkg.sql => local:ora1_emp_pkg.sql
psftp> exit
```

Practice 7: Solutions

1. Update EMP_PKG with a new procedure to query employees in a specified department.

   a. In the specification, declare a get_employees procedure, with its parameter called dept_id based on the employees.department_id column type. Define an index-by PL/SQL type as a TABLE OF EMPLOYEES%ROWTYPE.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. In the body of the package, define a private variable called emp_table based on the type defined in the specification to hold employee records. Implement the get_employees procedure to bulk fetch the data into the table.

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

Practice 7: Solutions (continued)

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

PROCEDURE add_employee(
  first_name employees.first_name%TYPE,
  last_name employees.last_name%TYPE,
  email employees.email%TYPE,
  job employees.job_id%TYPE DEFAULT 'SA_REP',
  mgr employees.manager_id%TYPE DEFAULT 145,
  sal employees.salary%TYPE DEFAULT 1000,
  comm employees.commission_pct%TYPE DEFAULT 0,
  deptid employees.department_id%TYPE DEFAULT 30) IS
BEGIN
  IF valid_deptid(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, first_name, last_name, email,
      job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
  ELSE
    RAISE_APPLICATION_ERROR (-20204,
                             'Invalid department ID. Try again.');
  END IF;
END add_employee;

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

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

Practice 7: Solutions (continued)

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

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

PROCEDURE get_employees(dept_id employees.department_id%type) IS
BEGIN
  SELECT * BULK COLLECT INTO emp_table
  FROM EMPLOYEES
  WHERE department_id = 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(emprec employees%rowtype) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                       emprec.employee_id||' '||
                       emprec.first_name||' '||
                       emprec.last_name||' '||
                       emprec.job_id||' '||
                       emprec.salary);
END;
```

Practice 7: Solutions (continued)

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

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Create a new procedure in the specification and body, called `show_employees`, which does not take arguments and displays the contents of the private PL/SQL table variable (if any data exists).
Hint: Use the `print_employee` procedure.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
```

Practice 7: Solutions (continued)

```
  PROCEDURE print_employee(emprec employees%rowtype);
  PROCEDURE show_employees;
END emp_pkg;
/
SHOW ERRORS

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

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

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS
  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

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

Practice 7: Solutions (continued)

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

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

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

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

Practice 7: Solutions (continued)

```
  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.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(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package created.

No errors.

Package body created.

No errors.
```

Practice 7: Solutions (continued)

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

```
EXECUTE emp_pkg.get_employees(30)
EXECUTE emp_pkg.show_employees

PL/SQL procedure successfully completed.

Employees in Package table
30 114 Den Raphaely PU_MAN 11000
30 115 Alexander Khoo PU_CLERK 3100
30 116 Shelli Baida PU_CLERK 2900
30 117 Sigal Tobias PU_CLERK 2800
30 118 Guy Himuro PU_CLERK 2600
30 119 Karen Colmenares PU_CLERK 2500
30 209 Samuel Joplin SA_REP 1000
PL/SQL procedure successfully completed.


EXECUTE emp_pkg.get_employees(60)
EXECUTE emp_pkg.show_employees

PL/SQL procedure successfully completed.

Employees in Package table
60 103 Alexander Hunold IT_PROG 9000
60 104 Bruce Ernst IT_PROG 6000
60 105 David Austin IT_PROG 4800
60 106 Valli Pataballa IT_PROG 4800
60 107 Diana Lorentz IT_PROG 4200
PL/SQL procedure successfully completed.
```


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

   a. First, load and execute the `E:\labs\PLPU\labs\lab_07_02_a.sql` script to create a log table called `LOG_NEWEMP`, and a sequence called `log_newemp_seq`.

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

CREATE SEQUENCE log_newemp_seq;

Table created.

Sequence created.
```

Practice 7: Solutions (continued)

b. In the package body, modify the `add_employee` procedure, which performs the actual `INSERT` operation to have a local procedure called `audit_newemp`. The `audit_newemp` procedure must use an autonomous transaction to insert a log record into the `LOG_NEWEMP` table. Store the `USER`, the current time, and the new employee name in the log table row. Use `log_newemp_seq` to set the `entry_id` column.
Note: Remember to perform a `COMMIT` operation in a procedure with an autonomous transaction.

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

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

  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS

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

  BEGIN
    IF valid_deptid(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, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;
```

Practice 7: Solutions (continued)

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

  ...

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

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

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

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

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

Practice 7: Solutions (continued)

```
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS

    PROCEDURE audit_newemp IS
      PRAGMA AUTONOMOUS_TRANSACTION;
      user_id VARCHAR2(30) := USER;
    BEGIN
      INSERT INTO log_newemp (entry_id, user_id, log_time, name)
      VALUES (log_newemp_seq.NEXTVAL, user_id, sysdate,
              first_name||' '||last_name);
      COMMIT;
    END audit_newemp;
  BEGIN
    IF valid_deptid(deptid) THEN
      audit_newemp;
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;
  ...
  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

BEGIN
  init_departments;
END emp_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 7: Solutions (continued)

d. Invoke the `add_employee` procedure for these new employees: `Max Smart` in department 20 and `Clark Kent` in department 10. What happens?

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

PL/SQL procedure successfully completed.

PL/SQL procedure successfully completed.
```

Both insert operations complete successfully, and the log table has two log records, as shown in the next step.

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

```
SELECT department_id, first_name, last_name
FROM employees
WHERE last_name IN ('Smart','Kent');
```

DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	FIRST_NAME
10	222	Kent	Clark
20	221	Smart	Max

```
SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA1	18-FEB-04	Max Smart
2	ORA1	18-FEB-04	Clark Kent

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

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

```
ROLLBACK;

Rollback complete.
```

Practice 7: Solutions (continued)

```
SELECT department_id, first_name, last_name
FROM employees
WHERE last_name IN ('Smart','Kent');

no rows selected

SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA1	18-FEB-04	Max Smart
2	ORA1	18-FEB-04	Clark Kent

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

If you have time, complete the following exercise:

3. Modify the EMP_PKG package to use AUTHID of CURRENT_USER and test the behavior with any other student.
   Note: Verify that the LOG_NEWEMP table exists from Exercise 2 in this practice.

   a. First, grant the EXECUTE privilege on your EMP_PKG package to another student.

```
Assume you are ORA1 and the other student is ORA2. You enter:
GRANT EXECUTE ON EMP_PKG TO ORA2;

Grant succeeded.
```

   b. Ask the other student to invoke your add_employee procedure to insert the employee Jaco Pastorius in department 10. Remember to prefix the package name with the owner of the package. The call should operate with definer's rights.

```
User ORA2 enters:
EXECUTE ora1.emp_pkg.add_employee('Jaco', 'Pastorius', 10)

PL/SQL procedure successfully completed.
```

Practice 7: Solutions (continued)

c. Now, execute a query of the employees in department 10. In which user's employee table did the new record get inserted?

```
User ORA1 executes:
SELECT department_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Jennifer	Whalen
10	Jaco	Pastorius

```
User ORA2 executes:
SELECT department_id, first_name, last_name
FROM departments
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Jennifer	Whalen

The new employee is added to the table in the `ORA1` schema—that is, in the table of the owner of the `EMP_PKG` package.

d. Now, modify your package `EMP_PKG` specification to use an `AUTHID CURRENT_USER`. Compile the body of `EMP_PKG`.

```
User ORA1 executes:
CREATE OR REPLACE PACKAGE emp_pkg AUTHID CURRENT_USER IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
```

Practice 7: Solutions (continued)

```
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
  PROCEDURE show_employees;
END emp_pkg;
/
SHOW ERRORS

ALTER PACKAGE emp_pkg COMPILE BODY;

Package created.

No errors.

Package body altered.
```

e. Ask the same student to execute the `add_employee` procedure again to add employee `Joe Zawinal` in department 10.

Note: Make sure that the user `ORA2` has executed the Practice 7-2a and created the `log_newemp` table before executing `emp_pkg.add_employee`.

```
User ORA2 executes:

EXECUTE ora1.emp_pkg.add_employee('Joe', 'Zawinal', 10)

PL/SQL procedure successfully completed.
```

f. Query your employees in department 10. In which table was the new employee added?

```
User ORA1 executes:
SELECT department_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Jennifer	Whalen
10	Jaco	Pastorius

```
User ORA2 executes:
SELECT department_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

DEPARTMENT_ID	FIRST_NAME	LAST_NAME
10	Joe	Zawinal
10	Jennifer	Whalen

Practice 7: Solutions (continued)

The new employee is added to the user `ORA2` employee table. That is, the new employee is added to the table that is owned by the caller (invoker's rights) of the package procedure.

Practice 7: Solutions (continued)

g. Write a query to display the records added in the LOG_NEWEMP tables. Ask the other student to query his or her own copy of the table.

User ORA1 executes:

```
SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
1	ORA1	18-FEB-04	Max Smart
2	ORA1	18-FEB-04	Clark Kent
3	ORA2	18-FEB-04	Jaco Pastorius

User ORA2 executes:

```
SELECT *
FROM log_newemp;
```

ENTRY_ID	USER_ID	LOG_TIME	NAME
3	ORA2	18-FEB-04	Joe Zawinal
1	ORA2	18-FEB-04	Max Smart
2	ORA2	18-FEB-04	Clark Kent

The log records created by the `audit_emp` procedure (which executes the autonomous transaction) are stored in the log table of the owner of the package when the package procedure is executed with the definer's (owner) rights. The log records are stored in the caller's log table when the package procedure is executed with invoker's (caller) rights.

Practice 8: Solutions

1. Answer the following questions.

   a. Can a table or a synonym be invalidated?

      A table or a synonym can never be invalidated; however, dependent objects can be invalidated.

   b. Consider the following dependency example:

      The stand-alone procedure MY_PROC depends on the MY_PROC_PACK package procedure. The MY_PROC_PACK procedure's definition is changed by recompiling the package body. The MY_PROC_PACK procedure's declaration is not altered in the package specification.

      In this scenario, is the stand-alone procedure MY_PROC invalidated?

      No, it is not invalidated because the stand-alone procedure MY_PROC depends on the MY_PROC_PACK package procedure, which has not been altered. Although the package body is recompiled, the package specification is not invalidated and does not need to be recompliled.

2. 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 lesson titled "Creating Stored Functions." You can run the solution scripts for Practice 2 if you need to create the procedure and function.

   a. Load and execute the utldtree.sql script, which is located in the E:\lab\PLPU\Labs folder.

      When you execute the script, the following results are displayed (you can ignore the error messages):

```
drop sequence deptree_seq
              *

ERROR at line 1:
ORA-02289: sequence does not exist
Sequence created.


drop table deptree_temptab
           *

ERROR at line 1:
ORA-00942: table or view does not exist
Table created.

Procedure created.
```

Practice 8: Solutions (continued)

```
drop view deptree
*

ERROR at line 1:
ORA-00942: table or view does not exist

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

set echo off

  from deptree_temptab d, dba_objects o
       *

ERROR at line 5:
ORA-00942: table or view does not exist

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

set echo off
View created.


drop view ideptree
*

ERROR at line 1:
ORA-00942: table or view does not exist
View created.
```

b. Execute the `deptree_fill` procedure for the `add_employee` procedure.

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

PL/SQL procedure successfully completed.
```

c. Query the `IDEPTREE` view to see your results.

```
SELECT * FROM IDEPTREE;
```

DEPENDENCIES
PROCEDURE ORA1.ADD_EMPLOYEE

Practice 8: Solutions (continued)

d. Execute the `deptree_fill` procedure for the `valid_deptid` function.

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

PL/SQL procedure successfully completed.
```

e. Query the `IDEPTREE` view to see your results.

```
SELECT * FROM IDEPTREE;
```

DEPENDENCIES
FUNCTION ORA1.VALID_DEPTID
PROCEDURE ORA1.ADD_EMPLOYEE

If you have time, complete the following exercise:

3. Dynamically validate invalid objects.

a. Make a copy of your `EMPLOYEES` table, called `EMPS`.

```
CREATE TABLE emps AS
  SELECT * FROM employees;

Table created.
```

b. Alter your `EMPLOYEES` table and add the `TOTSAL` column with the `NUMBER(9,2)` data type.

```
ALTER TABLE employees
  ADD (totsal NUMBER(9,2));

Table altered.
```

c. Create and save a query (`lab8_soln_3c.sql`) to display the name, type, and status of all invalid objects.

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';
```

Practice 8: Solutions (continued)

OBJECT_NAME	OBJECT_TYPE	STATUS
EMP_DETAILS_VIEW	VIEW	INVALID
SECURE_EMPLOYEES	TRIGGER	INVALID
UPDATE_JOB_HISTORY	TRIGGER	INVALID
TOTAL_SALARY	FUNCTION	INVALID
GET_EMPLOYEE	PROCEDURE	INVALID
GET_ANNUAL_COMP	FUNCTION	INVALID
ADD_EMPLOYEE	PROCEDURE	INVALID
EMP_PKG	PACKAGE	INVALID
EMP_PKG	PACKAGE BODY	INVALID
EMPLOYEE_REPORT	PROCEDURE	INVALID
WEB_EMPLOYEE_REPORT	PROCEDURE	INVALID

11 rows selected.

d. In `compile_pkg` (created in Practice 6 in the lesson titled "Dynamic SQL and Metadata"), 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;
```

Practice 8: Solutions (continued)

```
  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;
/
```

Practice 8: Solutions (continued)

```
SHOW ERRORS

Package created.

No errors.

Package body created.

No errors.
```

e. Execute the `compile_pkg.recompile` procedure.

```
EXECUTE compile_pkg.recompile

PL/SQL procedure successfully completed.
```

f. Run the script file that you created in step 3c (`lab8_soln_3c.sql`) to check the status column value. Do you still have objects with an `INVALID` status?

```
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE status = 'INVALID';

no rows selected
```

No rows are returned. There are no objects with an `INVALID` status.

Practice 9: Solutions

1. Create a table called PERSONNEL by executing the E:\labs\PLPU\labs\lab_09_01.sql script. The table contains the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
last_name	VARCHAR2	35
review	CLOB	N/A
picture	BLOB	N/A

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

Table created.
```

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

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

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

1 row created.

1 row created.
```

Practice 9: Solutions (continued)

3. Examine and execute the `E:\labs\PLPU\labs\lab_09_03.sql` script. The script creates a table named `REVIEW_TABLE`. This table contains annual review information for each employee. The script also contains two statements to insert review details for two employees.

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

INSERT INTO review_table
VALUES (2034,
       'Very good performance this year. '||
       'Recommended to increase salary by $500');
INSERT INTO review_table
VALUES (2035,
       'Excellent performance this year. '||
       'Recommended to increase salary by $1000');

COMMIT;

Table created.

1 row created.

1 row created.

Commit complete.
```

4. Update the `PERSONNEL` table.

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

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

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

1 row updated.
```

Practice 9: Solutions (continued)

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

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

```
DECLARE
  lobloc CLOB;
  text VARCHAR2(2000);
  amount NUMBER ;
  offset INTEGER;
BEGIN
  SELECT ann_review INTO text
  FROM review_table
  WHERE employee_id = 2035;
  SELECT review INTO lobloc
  FROM personnel
  WHERE last_name = 'Bond' FOR UPDATE;
  offset := 1;
  amount := length(text);
  DBMS_LOB.WRITE (lobloc, amount, offset, text);
  COMMIT;
END;
/

PL/SQL procedure successfully completed.
```


If you have time, complete the following exercise:

5. Create a procedure that adds a locator to a binary file into the PICTURE column of the COUNTRIES table. The binary file is a picture of the country flag. The image files are named after the country IDs. You need to load an image file locator into all rows in the Europe region (REGION_ID = 1) in the COUNTRIES table. A DIRECTORY object called COUNTRY_PIC referencing the location of the binary files has to be created for you.

   a. Add the image column to the COUNTRIES table using:
   ALTER TABLE countries ADD (picture BFILE);

```
ALTER TABLE countries ADD (picture BFILE);

Table altered.
```

Alternatively, use the E:\labs\PLPU\labs\ Lab_09_05_a.sql file.

Practice 9: Solutions (continued)

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

```
CREATE OR REPLACE PROCEDURE load_country_image (dir IN VARCHAR2) IS
  file           BFILE;
  filename       VARCHAR2(40);
  rec_number     NUMBER;
  file_exists    BOOLEAN;
  CURSOR country_csr IS
    SELECT country_id
    FROM countries
    WHERE region_id = 1
    FOR UPDATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('LOADING LOCATORS TO IMAGES...');
  FOR rec IN country_csr
  LOOP
    filename := rec.country_id || '.gif';
    file := BFILENAME(dir, filename);
    file_exists := (DBMS_LOB.FILEEXISTS(file) = 1);
    IF file_exists THEN
     DBMS_LOB.FILEOPEN(file);
     UPDATE countries
       SET picture = file
       WHERE CURRENT OF country_csr;
     DBMS_OUTPUT.PUT_LINE('Set Locator to file: '|| filename ||
       ' Size: ' || DBMS_LOB.GETLENGTH(file));
     DBMS_LOB.FILECLOSE(file);
     rec_number := country_csr%ROWCOUNT;
    ELSE
     DBMS_OUTPUT.PUT_LINE('File ' || filename ||' does not exist');
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('TOTAL FILES UPDATED: ' || rec_number);
  EXCEPTION
    WHEN OTHERS THEN
      DBMS_LOB.FILECLOSE(file);
      DBMS_OUTPUT.PUT_LINE('Error: '|| to_char(SQLCODE) || SQLERRM);
END load_country_image;
/
SHOW ERRORS

Procedure created.

No errors.
```

Practice 9: Solutions (continued)

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

```
SET SERVEROUTPUT ON
EXECUTE load_country_image('COUNTRY_PIC')

LOADING LOCATORS TO IMAGES...
Set Locator to file: BE.gif Size: 1397
Set Locator to file: CH.gif Size: 1202
Set Locator to file: DE.gif Size: 1271
Set Locator to file: DK.gif Size: 1327
Set Locator to file: FR.gif Size: 1337
Set Locator to file: IT.gif Size: 1322
Set Locator to file: NL.gif Size: 1205
Set Locator to file: UK.gif Size: 2489
TOTAL FILES UPDATED: 8
PL/SQL procedure successfully completed.
```

Practice 10: Solutions

1. The rows in the JOBS table store a minimum salary and a maximum salary allowed for different JOB_ID values. You are asked to write code to ensure that employees' salaries fall within the range allowed for their job type, for insert and update operations.

   a. Write a procedure called CHECK_SALARY that accepts two parameters, one for an employee's job ID string and the other for the salary. The procedure uses the job ID to determine the minimum and maximum salary for the specified job. If the salary parameter does not fall within the salary range of the job, inclusive of the minimum and maximum, then it should raise an application exception, with the message `Invalid salary <sal>. Salaries for job <jobid> must be between <min> and <max>`. Replace the various items in the message with values supplied by parameters and variables populated by queries. Save the file.

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

Procedure created.

No errors.
```

   b. Create a trigger called CHECK_SALARY_TRG on the EMPLOYEES table that fires before an INSERT or UPDATE operation on each row. The trigger must call the CHECK_SALARY procedure to carry out the business logic. The trigger should pass the new job ID and salary to the procedure parameters.

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

Practice 10: Solutions (continued)

```
Trigger created.

No errors.
```

2. Test the CHECK_SAL_TRG using the following cases:

   a. Using your EMP_PKG.ADD_EMPLOYEE procedure, add employee Eleanor Beh in department 30. What happens and why?

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

BEGIN emp_pkg.add_employee('Eleanor', 'Beh', 30); END;

*

ERROR at line 1:
ORA-20100: Invalid salary $1000. Salaries for job SA_REP must be between
$6000 and $12000
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
ORA-06512: at "ORA1.EMP_PKG", line 33
ORA-06512: at "ORA1.EMP_PKG", line 50
ORA-06512: at line 1
```

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

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

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

UPDATE employees
       *

ERROR at line 1:
ORA-20100: Invalid salary $2000. Salaries for job PU_CLERK must be
between $2500 and $5500
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

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

UPDATE employees
         *

ERROR at line 1:
ORA-20100: Invalid salary $3100. Salaries for job HR_REP must be between
$4000 and $9000
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

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

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

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

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

1 row updated.
```

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

3. Update the `CHECK_SALARY_TRG` trigger to fire only when the job ID or salary values have actually changed.

   a. Implement the business rule using a `WHEN` clause to check whether the `JOB_ID` or `SALARY` values have changed.
   Note: Make sure that the condition handles the `NULL` in the `OLD.column_name` values if an `INSERT` operation is performed; otherwise, an insert operation will fail.

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

Practice 10: Solutions (continued)

```
SHOW ERRORS

Trigger created.

No errors.
```

b. Test the trigger by executing `EMP_PKG.ADD_EMPLOYEE` procedure with the following parameter values: `first_name='Eleanor', last name='Beh', email='EBEH', job='IT_PROG', sal=5000`.

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

PL/SQL procedure successfully completed.
```

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

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

UPDATE employees
         *

ERROR at line 1:
ORA-20100: Invalid salary $11000. Salaries for job IT_PROG must be
between $4000 and $10000
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

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

Practice 10: Solutions (continued)

d. Update the salary to $9,000 for `Eleanor Beh`.
Hint: Use an `UPDATE` statement with a subquery in the `WHERE` clause. What happens?

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

1 row updated
```

The update operation is successful because the salary is valid for the employee's job type.

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

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

UPDATE employees
         *

ERROR at line 1:
ORA-20100: Invalid salary $9000. Salaries for job ST_MAN must be between
$5500 and $8500
ORA-06512: at "ORA1.CHECK_SALARY", line 9
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
```

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

Practice 10: Solutions (continued)

4. You are asked to prevent employees from being deleted during business hours.

   a. Write a statement trigger called DELETE_EMP_TRG on the EMPLOYEES table to prevent rows from being deleted during weekday business hours, which are from 9:00 a.m. to 6:00 p.m.

```
CREATE OR REPLACE TRIGGER delete_emp_trg
BEFORE DELETE ON employees
DECLARE
  the_day VARCHAR2(3) := TO_CHAR(SYSDATE, 'DY');
  the_hour PLS_INTEGER := TO_NUMBER(TO_CHAR(SYSDATE, 'HH24'));
BEGIN
   IF (the_hour BETWEEN 9 AND 18) AND (the_day NOT IN ('SAT','SUN')) THEN
     RAISE_APPLICATION_ERROR(-20150,
      'Employee records cannot be deleted during the week 9am and 6pm');
   END IF;
END;
/
SHOW ERRORS

Trigger created.

No errors.
```

   b. Attempt to delete employees with JOB_ID of SA_REP who are not assigned to a department.
   Note: This is employee Grant with ID 178.

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

DELETE FROM employees
         *

ERROR at line 1:
ORA-20150: Employee records cannot be deleted during the week 9am and 6pm
ORA-06512: at "ORA1.DELETE_EMP_TRG", line 6
ORA-04088: error during execution of trigger 'ORA1.DELETE_EMP_TRG'
```

Practice 11: Solutions

1. Employees receive an automatic increase in salary if the minimum salary for a job is increased to a value larger than their current salary. 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 employee's salary, 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 (from Practice 7) by adding a procedure called SET_SALARY that updates the employees' salaries. The procedure accepts two parameters: the job ID for those salaries that may have to be updated, and the new minimum salary for the job ID. The procedure sets all the employee salaries to the minimum for their job if their current salary is less than the new minimum value.

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30);
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    deptid employees.department_id%TYPE);
  PROCEDURE get_employee(
    empid IN employees.employee_id%TYPE,
    sal OUT employees.salary%TYPE,
    job OUT employees.job_id%TYPE);
  FUNCTION get_employee(emp_id employees.employee_id%type)
    return employees%rowtype;
  FUNCTION get_employee(family_name employees.last_name%type)
    return employees%rowtype;
  PROCEDURE get_employees(dept_id employees.department_id%type);
  PROCEDURE init_departments;
  PROCEDURE print_employee(emprec employees%rowtype);
  PROCEDURE set_salary(jobid VARCHAR2, min_salary NUMBER);
END emp_pkg;
/
SHOW ERRORS
```

Practice 11: Solutions (continued)

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

  FUNCTION valid_deptid(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN;
  PROCEDURE add_employee(
    first_name employees.first_name%TYPE,
    last_name employees.last_name%TYPE,
    email employees.email%TYPE,
    job employees.job_id%TYPE DEFAULT 'SA_REP',
    mgr employees.manager_id%TYPE DEFAULT 145,
    sal employees.salary%TYPE DEFAULT 1000,
    comm employees.commission_pct%TYPE DEFAULT 0,
    deptid employees.department_id%TYPE DEFAULT 30) IS

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

  BEGIN
    IF valid_deptid(deptid) THEN
      audit_newemp;
      INSERT INTO employees(employee_id, first_name, last_name, email,
        job_id,manager_id,hire_date,salary,commission_pct,department_id)
      VALUES (employees_seq.NEXTVAL, first_name, last_name, email,
        job, mgr, TRUNC(SYSDATE), sal, comm, deptid);
    ELSE
      RAISE_APPLICATION_ERROR (-20204,
                               'Invalid department ID. Try again.');
    END IF;
  END add_employee;

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

Practice 11: Solutions (continued)

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

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

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

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

Practice 11: Solutions (continued)

```
  PROCEDURE print_employee(emprec employees%rowtype) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(emprec.department_id ||' '||
                         emprec.employee_id||' '||
                         emprec.first_name||' '||
                         emprec.last_name||' '||
                         emprec.job_id||' '||
                         emprec.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(deptid IN departments.department_id%TYPE)
   RETURN BOOLEAN IS
    dummy PLS_INTEGER;
  BEGIN
    RETURN valid_departments.exists(deptid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
    RETURN FALSE;
  END valid_deptid;

  PROCEDURE set_salary(jobid VARCHAR2, min_salary NUMBER) IS
    CURSOR empcsr IS
      SELECT employee_id
      FROM employees
      WHERE job_id = jobid AND salary < min_salary;
  BEGIN
    FOR emprec IN empcsr
    LOOP
      UPDATE employees
        SET salary = min_salary
      WHERE employee_id = emprec.employee_id;
    END LOOP;
  END set_salary;

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

Practice 11: Solutions (continued)

```
Package created.

No errors.

Package body created.

No 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.

```
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

Trigger created.

No errors.
```

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

```
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';
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	6000
105	Austin	4800
106	Pataballa	4800
107	Lorentz	4200
226	Beh	9000

```
6 rows selected.
```

Practice 11: Solutions (continued)

```
UPDATE jobs
       *

ERROR at line 1:
ORA-04091: table ORA1.JOBS is mutating, trigger/function may not see it
ORA-06512: at "ORA1.CHECK_SALARY", line 5
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
ORA-06512: at "ORA1.EMP_PKG", line 140
ORA-06512: at "ORA1.UPD_MINSALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.UPD_MINSALARY_TRG'
```

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). `CHECK_SALARY_TRG` calls the `CHECK_SALARY` procedure, which attempts to read the `JOBS` table data, thus encountering the mutating table exception on the `JOBS` table, which is the table that is subject to the original `UPDATE` operation.

2. To resolve the mutating table issue, you create JOBS_PKG to maintain in memory a copy of the rows in the JOBS table. Then the CHECK_SALARY procedure is modified to use the package data rather than issue a query on a table that is mutating to avoid the exception. However, a BEFORE INSERT OR UPDATE statement trigger must be created 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(jobid VARCHAR2) RETURN NUMBER;
FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
PROCEDURE set_minsalary(jobid VARCHAR2,min_salary NUMBER);
PROCEDURE set_maxsalary(jobid VARCHAR2,max_salary NUMBER);
```

```
CREATE OR REPLACE PACKAGE jobs_pkg IS
  PROCEDURE initialize;
  FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER;
  FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER;
  PROCEDURE set_minsalary(jobid VARCHAR2, min_salary NUMBER);
  PROCEDURE set_maxsalary(jobid VARCHAR2, max_salary NUMBER);
END jobs_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Practice 11: Solutions (continued)

b. Implement the body of the JOBS_PKG where:
You declare a private PL/SQL index-by table called jobs_tabtype that is indexed by a string type based on JOBS.JOB_ID%TYPE.
You declare a private variable called jobstab based on jobs_tabtype.

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.
The GET_MINSALARY function uses a jobid parameter as an index to the jobstab and returns the min_salary for that element.
The GET_MAXSALARY function uses a jobid parameter as an index to the jobstab and returns the max_salary for that element.
The SET_MINSALARY procedure uses its jobid as an index to the jobstab to set the min_salary field of its element to the value in the min_salary parameter.
The SET_MAXSALARY procedure uses its jobid as an index to the jobstab to set the max_salary field of its element to the value in the max_salary parameter.

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

  PROCEDURE initialize IS
  BEGIN
    FOR jobrec IN (SELECT * FROM jobs)
    LOOP
      jobstab(jobrec.job_id) := jobrec;
    END LOOP;
  END initialize;

  FUNCTION get_minsalary(jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(jobid).min_salary;
  END get_minsalary;

  FUNCTION get_maxsalary(jobid VARCHAR2) RETURN NUMBER IS
  BEGIN
    RETURN jobstab(jobid).max_salary;
  END get_maxsalary;

  PROCEDURE set_minsalary(jobid VARCHAR2, min_salary NUMBER) IS
  BEGIN
    jobstab(jobid).max_salary := min_salary;
  END set_minsalary;
```

Practice 11: Solutions (continued)

```
  PROCEDURE set_maxsalary(jobid VARCHAR2, max_salary NUMBER) IS
  BEGIN
    jobstab(jobid).max_salary := max_salary;
  END set_maxsalary;

END jobs_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

c. Copy the CHECK_SALARY procedure from Practice 10, 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.

```
CREATE OR REPLACE PROCEDURE check_salary (the_job VARCHAR2, the_salary
NUMBER) IS
  minsal jobs.min_salary%type;
  maxsal jobs.max_salary%type;
BEGIN
  /*
  ** Commented out to avoid mutating trigger exception on the JOBS table
  SELECT min_salary, max_salary INTO minsal, maxsal
  FROM jobs
  WHERE job_id = UPPER(the_job);
  */
  minsal := jobs_pkg.get_minsalary(UPPER(the_job));
  maxsal := jobs_pkg.get_maxsalary(UPPER(the_job));
  IF the_salary NOT BETWEEN minsal AND maxsal THEN
    RAISE_APPLICATION_ERROR(-20100,
      'Invalid salary $'||the_salary||'. '||
      'Salaries for job '|| the_job ||
      ' must be between $'|| minsal ||' and $' || maxsal);
  END IF;
END;
/
SHOW ERRORS

Procedure created.

No errors.
```

Practice 11: Solutions (continued)

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.

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

Trigger created.

No 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 `1000` in the `JOBS` table, followed by 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 job?

```
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
103	Hunold	9000
104	Ernst	6000
105	Austin	4800
106	Pataballa	4800
107	Lorentz	4200
226	Beh	9000

```
6 rows selected.

1 row updated.
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
104	Ernst	6000
105	Austin	5000
106	Pataballa	5000
107	Lorentz	5000
226	Beh	9000

```
6 rows selected.
```

The employees with last names `Austin`, `Pataballa`, and `Lorentz` have all had their salaries updated. No exception ocurred 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`’, `sal => 6500`). What happens?

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

BEGIN emp_pkg.add_employee('Steve', 'Morse', 'SMORSE', sal => 6500); END;

*

ERROR at line 1:
ORA-01403: no data found
ORA-01403: no data found
ORA-06512: at "ORA1.JOBS_PKG", line 16
ORA-06512: at "ORA1.CHECK_SALARY", line 11
ORA-06512: at "ORA1.CHECK_SALARY_TRG", line 2
ORA-04088: error during execution of trigger 'ORA1.CHECK_SALARY_TRG'
ORA-06512: at "ORA1.EMP_PKG", line 33
ORA-06512: at line 1
```

The problem here is that the `CHECK_SALARY` procedure attempts to read the value of package state variables that have not yet been initialized. This is because it had been modified to read the miniumum and maximum salaries from `JOBS_PK`, which should store the data in a PL/SQL table. When `CHECK_SALARY` attempts to call `JOBS_PKG.GET_MINSALARY` and `JOBS_PKG.GET_MAXSALARY`, these return `NO_DATA_FOUND` exceptions that cause the trigger and the insert operation to fail. This can be resolved with a `BEFORE` statement trigger that calls `JOBS_PKG.INITIALIZE` to ensure that the `JOBS_PKG` state is set before you read it. This is done in the next exercise (3b).

Practice 11: Solutions (continued)

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

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

Trigger created.
```

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', sal => 6500)

PL/SQL procedure successfully completed.

SELECT employee_id, first_name, last_name, salary, job_id, department_id
FROM employees
WHERE last_name = 'Morse';
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	DEPARTMENT_ID
241	Steve	Morse	6500	SA_REP	30

Practice 12: Solutions

1. Alter the PLSQL_COMPILER_FLAGS parameter to enable native compilation for your session, and compile any subprogram that you have written.

   a. Execute the ALTER SESSION command to enable native compilation.

```
ALTER SESSION SET PLSQL_COMPILER_FLAGS = 'NATIVE';

Session altered.
```

   b. Compile the EMPLOYEE_REPORT procedure. What occurs during compilation?

```
ALTER PROCEDURE employee_report COMPILE;

Procedure altered.
```

A shared library is generated in a directory specified by the database parameter, plsql_native_library_dir. The library name is prefixed with the object name and user compiling it, as in the following: EMPLOYEE_REPORT__ORA1__P__50344.so.

   c. Execute the EMPLOYEE_REPORT with the value 'UTL_FILE' as the first parameter, and 'native_salrepXX.txt' where XX is your student number.

```
EXECUTE employee_report('UTL_FILE', 'native_salrep01.txt')

PL/SQL procedure successfully completed.
```

   d. Switch compilation to use interpreted compilation

```
ALTER SESSION SET PLSQL_COMPILER_FLAGS = 'INTERPRETED';

Session altered.
```

2. In COMPILE_PKG (from Practice 6), add an overloaded version of the procedure called MAKE, which will compile a named procedure, function, or package.

   a. In the specification, declare a MAKE procedure that accepts two string arguments, one for the name of the PL/SQL construct and the other for the type of PL/SQL program, such as PROCEDURE, FUNCTION, PACKAGE, or PACKAGE BODY.

```
CREATE OR REPLACE PACKAGE compile_pkg IS
  dir VARCHAR2(100) := 'UTL_FILE';
  PROCEDURE make(name VARCHAR2);
  PROCEDURE make(name VARCHAR2, objtype VARCHAR2);
  PROCEDURE regenerate(name VARCHAR2);
END compile_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

Practice 12: Solutions (continued)

b. In the body, write the MAKE procedure to call the DBMS_WARNINGS package to suppress the PERFORMANCE category. However, save the current compiler warning settings before you alter them. Then write an EXECUTE IMMEDIATE statement to compile the PL/SQL object using an appropriate ALTER...COMPILE statement with the supplied parameter values. Finally, restore the compiler warning settings that were in place for the calling environment before the procedure is invoked.

```
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;
```

Practice 12: Solutions (continued)

```
  PROCEDURE make(name VARCHAR2, objtype VARCHAR2) IS
    stmt       VARCHAR2(100);
    warn_value varchar2(200);
  BEGIN
      stmt := 'ALTER '|| objtype ||' '|| name ||' COMPILE';
      warn_value := dbms_warning.get_warning_setting_string;
      dbms_warning.add_warning_setting_cat(
         'PERFORMANCE', 'DISABLE', 'SESSION');
      execute(stmt);
      dbms_warning.set_warning_setting_string(
         warn_value, 'SESSION');
  END make;

  PROCEDURE regenerate (name VARCHAR2) IS
    file UTL_FILE.FILE_TYPE;
    filename VARCHAR2(100) := LOWER(USER ||'_'|| name ||'.sql');
    proc_type  VARCHAR2(30) := get_type(name);
  BEGIN
    IF proc_type IS NOT NULL THEN
      file := UTL_FILE.FOPEN(dir, filename, 'w');
      UTL_FILE.PUT(file,
        DBMS_METADATA.GET_DDL(proc_type, UPPER(name)));
      UTL_FILE.FCLOSE(file);
    ELSE
      RAISE_APPLICATION_ERROR(-20001,
         'Object with '''|| name ||''' does not exist');
    END IF;

  END regenerate;

END compile_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

Practice 12: Solutions (continued)

3. Write a new PL/SQL package called TEST_PKG containing a procedure called GET_EMPLOYEES that uses an IN OUT argument.

   a. In the specification, declare the GET_EMPLOYEES procedure with two parameters, one input parameter specifying a department ID, and an IN OUT parameter specifying a PL/SQL table of employee rows.
   Hint: You have to declare a TYPE in the package specification for the PL/SQL table parameter's data type.

```
CREATE OR REPLACE PACKAGE test_pkg IS
  TYPE emp_tabtype IS TABLE OF employees%ROWTYPE;
  PROCEDURE get_employees(dept_id NUMBER, emps IN OUT emp_tabtype);
END test_pkg;
/
SHOW ERRORS

Package created.

No errors.
```

   b. In the package body, implement the GET_EMPLOYEES procedure to retrieve all the employee rows for a specified department into the PL/SQL table IN OUT parameter.
   Hint: Use the SELECT … BULK COLLECT INTO syntax to simplify the code.

```
CREATE OR REPLACE PACKAGE BODY test_pkg IS
  PROCEDURE get_employees(dept_id NUMBER, emps IN OUT emp_tabtype) IS
  BEGIN
     SELECT * BULK COLLECT INTO emps
     FROM employees
     WHERE department_id = dept_id;
  END get_employees;
END test_pkg;
/
SHOW ERRORS

Package body created.

No errors.
```

4. Use the ALTER SESSION statement to set the PLSQL_WARNINGS so that all compiler warning categories are enabled.

```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:ALL';

Session altered.
```

Practice 12: Solutions (continued)

5. Recompile the TEST_PKG created in an earlier task. What compiler warnings are displayed, if any?

```
ALTER PACKAGE test_pkg COMPILE;
SHOW ERRORS

SP2-0809: Package altered with compilation warnings
Errors for PACKAGE TEST_PKG:
```

LINE/COL	ERROR
3/43	PLW-07203: parameter 'EMPS' may benefit from use of the NOCOPY co mpiler hint

6. Write a PL/SQL anonymous block to compile the TEST_PKG package by using the overloaded COMPILE_PKG.MAKE procedure with two parameters. The anonymous block should display the current session warning string value before and after it invokes the COMPILE_PKG.MAKE procedure. Do you see any warning messages? Confirm your observations by executing the SHOW ERRORS PACKAGE command for the TEST_PKG.

```
BEGIN
  dbms_output.put_line('Warning level before compilation: '||
            dbms_warning.get_warning_setting_string);
  compile_pkg.make('TEST_PKG', 'PACKAGE');
  dbms_output.put_line('Warning level after compilation: '||
            dbms_warning.get_warning_setting_string);
END;
/
SHOW ERRORS PACKAGE test_pkg;

Warning level before compilation: ENABLE:ALL
ALTER PACKAGE TEST_PKG COMPILE
Warning level after compilation: ENABLE:ALL
PL/SQL procedure successfully completed.

No errors.
```

Note: The current warning level setting should be the same before and after the call to the COMPILE_PKG.MAKE procedure, which alters the settings to suppress warnings and restores the original setting before returning to the caller.