Practice 1

Before you begin this practice, please ensure that you have seen both the viewlets on

*i*SQL*Plus usage.

The `labs` folder will be your working directory. You can save your scripts in the `labs`

folder. Please take the instructor's help to locate the `labs` folder for this course. The solutions for all practices are in the `soln` folder.

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

a.
```
BEGIN
END;
```

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

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

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

2.Create and execute a simple anonymous block that outputs "Hello World." Execute and save this script as `lab_01_02_soln.sql`.

Practice 2

Note: It is recommended to use *i*SQL*Plus for this practice.

1. Identify valid and invalid identifier names:
   a. `today`
   b. `last_name`
   c. `today's_date`
   d. `Number_of_days_in_February_this_year`
   e. `Isleap$year`
   f. `#number`
   g. `NUMBER#`
   h. `number1to7`

2. Identify valid and invalid variable declaration and initialization:
   a. `number_of_copies PLS_INTEGER;`
   b. printer_name constant `VARCHAR2(10);`
   c. `deliver_to VARCHAR2(10)`:=Johnson;
   d. `by_when DATE`:= `SYSDATE`+1;

3. Examine the following anonymous block and choose the appropriate statement.

```
SET SERVEROUTPUT ON
DECLARE
  fname VARCHAR2(20);
  lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE( FNAME ||' ' ||lname);
END;
/
```

a.The block will execute successfully and print 'fernandez'

b.The block will give an error because the `fname` variable is used without initializing.

c.The block will execute successfully and print 'null fernandez'

d.The block will give an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.

e.The block will give an error because the variable `FNAME` is not declared.

Practice 2 (continued)

4. Create an anonymous block. In *i*SQL*Plus, load the script `lab_01_02_soln.sql`, which you created in question 2 of practice 1.

a. Add a declarative section to this PL/SQL block. In the declarative section, declare the following variables:

1. Variable today of type DATE. Initialize today with SYSDATE.
2. Variable tomorrow of type today. Use %TYPE attribute to declare this variable.

b. In the executable section initialize the variable `tomorrow` with an expression, which calculates tomorrow's date (add one to the value in `today`). Print the value of `today` and `tomorrow` after printing 'Hello World'

c. Execute and save this script as `lab_02_04_soln.sql`. Sample output is shown below.

```
Hello World
TODAY IS : 12-JAN-04
TOMORROW IS : 13-JAN-04
PL/SQL procedure successfully completed.
```

BASIC_PERCENT
45

Next Page

5. Edit the `lab_02_04_soln.sql` script.

a. Add code to create two bind variables.
Create bind variables `basic_percent` and `pf_percent` of type `NUMBER`.

b. In the executable section of the PL/SQL block assign the values 45 and 12 to `basic_percent` and `pf_percent` respectively.

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

d. Execute and save your script file as `lab_02_05_soln.sql`. Sample output is shown below.

PF_PERCENT
12

Practice 3

Note: It is recommended to use *i*SQL*Plus for this practice.

PL/SQL Block

```
      DECLARE
 weight    NUMBER(3) := 600;
 message   VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   weight        NUMBER(3) := 1;
   message   VARCHAR2(255) := 'Product
11001';
   new_locn  VARCHAR2(50) := 'Europe';
  BEGIN
   weight := weight + 1;
   new_locn := 'Western ' || new_locn;

  END;
 weight := weight + 1;
 message := message || ' is in stock';
 new_locn := 'Western ' || new_locn;

END;
/
```

1 → BEGIN

2 → END;

1. Evaluate the PL/SQL block given above and determine the data type and value of each of the following variables according to the rules of scoping.

   a. The value of `weight` at position 1 is:

   b. The value of `new_locn` at position 1 is:

   c. The value of `weight` at position 2 is:

   d. The value of `message` at position 2 is:

   e. The value of `new_locn` at position 2 is:

Practice 3 (continued)

Scope Example

```
DECLARE
  customer          VARCHAR2(50) := 'Womansport';
  credit_rating     VARCHAR2(50) := 'EXCELLENT';
BEGIN
   DECLARE
     customer    NUMBER(7) := 201;
     name        VARCHAR2(25) := 'Unisports';
   BEGIN
     credit_rating :='GOOD';
     …
   END;
  …
END;
/
```

2. In the PL/SQL block shown above, determine the values and data types for each of the following cases.

   a. The value of `customer` in the nested block is:

   b. The value of `name` in the nested block is:

   c. The value of `credit_rating` in the nested block is:

   d. The value of `customer` in the main block is:

   e. The value of `name` in the main block is:

   f. The value of `credit_rating` in the main block is:

Practice 3 (continued)

3. Use the same session that you used to execute the practices in Lesson 2. If you have opened a new session, then execute `lab_02_05_soln.sql`. Edit `lab_02_05_soln.sql`.

a.Use single line comment syntax to comment the lines that create the bind variables.

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

c.Declare two variables: `fname` of type `VARCHAR2` and size 15, and `emp_sal` of type `NUMBER` and size 10.

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

```
SELECT first_name, salary
  INTO fname, emp_sal FROM employees
  WHERE employee_id=110;
```

e.Change the line that prints 'Hello World' to print 'Hello' and the first name. You can comment the lines that display the dates and print the bind variables, if you want to.

f.Calculate the contribution of the employee towards provident fund (PF).
PF is 12% of the basic salary and basic salary is 45% of the salary. Use the bind variables for the calculation. Try and use only one expression to calculate the PF. Print the employee's salary and his contribution towards PF.

g.Execute and save your script as `lab_03_03_soln.sql`. Sample output is shown below.

```
Hello John
YOUR SALARY IS : 8200
YOUR CONTRIBUTION TOWARDS PF: 442.8
PL/SQL procedure successfully completed.
```

4. Accept a value at run time using the substitution variable. In this practice, you will modify the script that you created in exercise 3 to accept user input.

a.Load the script `lab_03_04.sql` file.

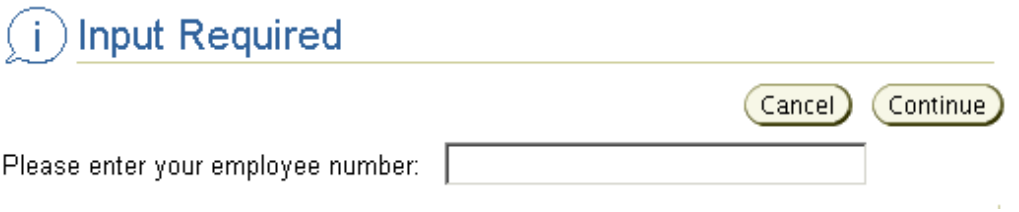
b.Include the `PROMPT` command to prompt the user with the following message:
'Please enter your employee number.'

c.Modify the declaration of the `empno` variable to accept the user input.

d.Modify the select statement to include the variable `empno`.

e.Execute and save your script as `lab_03_04_soln.sql`. Sample output is shown below.

Practice 3 (continued)



Enter 100 and click the Continue button.

```
Hello Steven
YOUR SALARY IS : 24000
YOUR CONTRIBUTION TOWARDS PF: 1296
PL/SQL procedure successfully completed.
```

5. Execute the script `lab_03_05.sql`. This script creates a table called `employee_details`.

   a. The `employee` and `employee_details` tables have the same data. You will update the data in the `employee_details` table. Do not update or change the data in the `employees` table.

   b. Open the script `lab_03_05b.sql` and observe the code in the file. Note that the code accepts the employee number and the department number from the user.

   c. You will use this as the skeleton script to develop the application, which was discussed in the lesson titled “Introduction.”

Practice 4

Note: It is recommended to use *i*SQL*Plus for this practice.

1.Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `max_deptno` variable. Display the maximum department ID.

a.Declare a variable `max_deptno` of type `NUMBER` in the declarative section.

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

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

d.Execute and save your script as `lab_04_01_soln.sql`. Sample output is shown below.

```
The maximum department_id is : 270
PL/SQL procedure successfully completed.
```

2. Modify the PL/SQL block you created in exercise 1 to insert a new department into the `departments` table.

a.Load the script `lab_04_01_soln.sql`. Declare two variables: `dept_name` of type `departments.department_name`.
Bind variable `dept_id` of type `NUMBER`.
Assign 'Education' to `dept_name` in the declarative section.

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

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

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

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

f.Execute and save your script as `lab_04_02_soln.sql`. Sample output is shown below.

```
The maximum department_id is : 270
SQL%ROWCOUNT gives 1
PL/SQL procedure successfully completed.
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		

3. In exercise 2, you have set `location_id` to null. Create a PL/SQL block that updates the `location_id` to 3000 for the new department. Use the bind variable `dept_id` to update the row.
Note: Skip step a if you have not started a new *i*SQL*Plus session for this practice.

a. If you have started a new *i*SQL*Plus session, delete the department that you have added to the `departments` table and execute the script `lab_04_02_soln.sql`.

b. Start the executable block with the keyword `BEGIN`. Include the `UPDATE` statement to set the `location_id` to 3000 for the new department. Use the bind variable `dept_id` in your `UPDATE` statement.

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

d. Finally, include a `DELETE` statement to delete the department that you added.

e. Execute and save your script as `lab_04_03_soln.sql`. Sample output is shown below.

PL/SQL procedure successfully completed.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Education		3000

1 row deleted.

4. Load the script `lab_03_05b.sql` to the *i*SQL*Plus workspace.

a. Observe that the code has nested blocks. You will see the declarative section of the outer block. a. Look for the comment "INCLUDE EXECUTABLE SECTION OF OUTER BLOCK HERE" and start an executable section

b. Include a single `SELECT` statement, which retrieves the `employee_id` of the employee working in the 'Human Resources' department. Use the `INTO` clause to store the retrieved value in the variable `emp_authorization`.

c. Save your script as `lab_04_04_soln.sql`.

Practice 5

1. Execute the command in the file `lab_05_01.sql` to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.

   a. Insert the numbers 1 to 10, excluding 6 and 8.

   b. Commit before the end of the block.

   c. Execute a `SELECT` statement to verify that your PL/SQL block worked.
   You should see the following output.

RESULTS
1
2
3
4
5
7
9
10

8 rows selected.

2. Execute the script `lab_05_02.sql`. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size 50. Create a PL/SQL block that inserts an asterisk in the `stars` column for every $1000 of the employee's salary. Save your script as `lab_05_02_soln.sql`.

   a.Use the `DEFINE` command to define a variable called `empno` and initialize it to 176.

   b.Start the declarative section of the block and pass the value of `empno` to the PL/SQL block through an *i*SQL*Plus substitution variable. Declare a variable `asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `sal` of type `emp.salary`.

   c. In the executable section, write logic to append an asterisk (*) to the string for every $1000 of the salary amount. For example, if the employee earns $8000, the string of asterisks should contain eight asterisks. If the employee earns $12500, the string of asterisks should contain 13 asterisks.

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

Practice 5 (continued)

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

f.Execute and save your script as `lab_05_02_soln.sql`. The output is shown below.

EMPLOYEE_ID	SALARY	STARS
176	8600	*********

3.Load the script `lab_04_04_soln.sql`, which you created in question 4 of
Practice 4.

a.Look for the comment "INCLUDE SIMPLE IF STATEMENT HERE" and include a simple `IF` statement to check if the values of `emp_id` and `emp_authorization` are the same.

b.Save your script as `lab_05_03_soln.sql`.

Practice 6

1.Write a PL/SQL block to print information about a given country.

a.Declare a PL/SQL record based on the structure of the `countries` table.

b.Use the `DEFINE` command to define a variable `countryid`. Assign CA to `countryid`. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.

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

d.In the executable section, get all the information from the `countries` table by using `countryid`. Display selected information about the country. A sample output is shown below.

```
Country Id: CA Country Name: Canada Region: 2
PL/SQL procedure successfully completed.
```

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

2.Create a PL/SQL block to retrieve the name of some departments from the `departments` table and print each department name on the screen, incorporating an `INDEX BY` table. Save the script as `lab_06_02_soln.sql`.

a.Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the name of the departments.

b.Declare two variables: `loop_count` and `deptno` of type `NUMBER`. Assign 10 to `loop_count` and 0 to `deptno`.

c.Using a loop, retrieve the name of 10 departments and store the names in the `INDEX BY` table. Start with `department_id` 10. Increase `deptno` by 10 for every iteration of the loop. The following table shows the `department_id` for which you should retrieve the `department_name` and store in the `INDEX BY` table.

Practice 6 (continued)

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

d.Using another loop, retrieve the department names from the `INDEX BY` table and display them.

e.Execute and save your script as `lab_06_02_soln`.sql. The output is shown below.

```
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
PL/SQL procedure successfully completed.
```

Practice 6 (continued)

3. Modify the block that you created in question 2 to retrieve all information about each department from the `departments` table and display the information. Use an `INDEX BY` table of records.

a.Load the script `lab_06_02_soln.sql`.

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

c.Modify the select statement to retrieve all department information currently in the `departments` table and store it in the `INDEX BY` table.

d.Using another loop, retrieve the department information from the `INDEX BY` table and display the information. A sample output is shown below.

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

4.Load the script `lab_05_03_soln.sql`.

a.Look for the comment “DECLARE AN INDEX BY TABLE OF TYPE VARCHAR2(50). CALL IT ename_table_type” and include the declaration.

b.Look for the comment “DECLARE A VARIABLE ename_table OF TYPE ename_table_type” and include the declaration.

c.Save your script as `lab_06_04_soln.sql`.

Practice 7

1. Create a PL/SQL block that determines the top *n* salaries of the employees.

a. Execute the script `lab_07_01.sql` to create a new table, `top_salaries`, for storing the salaries of the employees.

b. Accept a number *n* from the user where *n* represents the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter 5.

Note: Use the `DEFINE` command to define a variable `p_num` to provide the value for *n*. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.

c. In the declarative section, declare two variables: `num` of type `NUMBER` to accept the substitution variable `p_num`, `sal` of type `employees.salary`. Declare a cursor, `emp_cursor`, that retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.

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

e. After inserting into the `top_salaries` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `employees` table.

SALARY
24000
17000
14000
13500
13000

f. Test a variety of special cases, such as *n* = 0 or where *n* is greater than the number of employees in the `employees` table. Empty the `top_salaries` table after each test.

2. Create a PL/SQL block that does the following:

a. Use the `DEFINE` command to define a variable `p_deptno` to provide the department ID.

b. In the declarative section, declare a variable `deptno` of type `NUMBER` and assign the value of `p_deptno`.

c. Declare a cursor, `emp_cursor`, that retrieves the `last_name`, `salary`, and `manager_id` of the employees working in the department specified in `deptno`.

Practice 7 (continued)

d.In the executable section use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5000 and if the manager ID is either 101 or 124, display the message <<*last_name*>> Due for a raise. Otherwise, display the message <<*last_name*>> Not due for a raise.

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

Department ID	**Message**
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise Mourgas Not Due for a raise . . . . . . Rajs Due for a raise
80	Russel Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . . . .

Practice 7 (continued)

3. Write a PL/SQL block, which declares and uses cursors with parameters.

In a loop, use a cursor to retrieve the department number and the department name from the `departments` table for a department whose `department_id` is less than 100. Pass the department number to another cursor as a parameter to retrieve from the `employees` table the details of employee last name, job, hire date, and salary of those employees whose `employee_id` is less than 120 and who work in that department.

a. In the declarative section, declare a cursor `dept_cursor` to retrieve `department_id`, `department_name` for those departments with `department_id` less than 100. Order by `department_id`.

b. Declare another cursor `emp_cursor` that takes the department number as parameter and retrieves `last_name`, `job_id`, `hire_date`, and `salary` of those employees with `employee_id` of less than 120 and who work in that department.

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

d. Open the `dept_cursor`, use a simple loop and fetch values into the variables declared. Display the department number and department name.

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

Note: You may want to print a line after you have displayed the details of each department. Use appropriate attributes for the exit condition. Also determine whether a cursor is already open before opening the cursor.

f. Close all the loops and cursors, and end the executable section. Execute the script.

Practice 7 (continued)

The sample output is shown below.

```
Department Number : 10 Department Name : Administration
------------------------------------------------------------------------------
Department Number : 20 Department Name : Marketing
------------------------------------------------------------------------------
Department Number : 30 Department Name : Purchasing
Raphaely PU_MAN 07-DEC-94 11000
Khoo PU_CLERK 18-MAY-95 3100
Baida PU_CLERK 24-DEC-97 2900
Tobias PU_CLERK 24-JUL-97 2800
Himuro PU_CLERK 15-NOV-98 2600
Colmenares PU_CLERK 10-AUG-99 2500
------------------------------------------------------------------------------
Department Number : 40 Department Name : Human Resources
------------------------------------------------------------------------------
Department Number : 50 Department Name : Shipping
------------------------------------------------------------------------------
Department Number : 60 Department Name : IT
Hunold IT_PROG 03-JAN-90 9000
Ernst IT_PROG 21-MAY-91 6000
Austin IT_PROG 25-JUN-97 4800
Pataballa IT_PROG 05-FEB-98 4800
Lorentz IT_PROG 07-FEB-99 4200
------------------------------------------------------------------------------
Department Number : 70 Department Name : Public Relations
------------------------------------------------------------------------------
Department Number : 80 Department Name : Sales
------------------------------------------------------------------------------
Department Number : 90 Department Name : Executive
King AD_PRES 17-JUN-87 24000
Kochhar AD_VP 21-SEP-89 17000
De Haan AD_VP 13-JAN-93 17000
------------------------------------------------------------------------------
PL/SQL procedure successfully completed.
```

Practice 7 (continued)

4.Load the script `lab_06_04_soln.sql`.

a.Look for the comment "DECLARE A CURSOR CALLED emp_records TO HOLD salary, first_name, and last_name of employees" and include the declaration. Create the cursor such that it retrieves the salary, first_name, and last_name of employees in the department specified by the user (substitution variable `emp_deptid`). Use the `FOR UPDATE` clause.

b.Look for the comment "INCLUDE EXECUTABLE SECTION OF INNER BLOCK HERE" and start the executable block.

c.Only employees working in the departments with `department_id` 20, 60, 80,100, and 110 are eligible for raises this quarter. Check if the user has entered any of these department IDs. If the value does not match, display the message "SORRY, NO SALARY REVISIONS FOR EMPLOYEES IN THIS DEPARTMENT." If the value matches, then, open the cursor `emp_records`.

d.Start a simple loop and fetch the values into `emp_sal`, `emp_fname`, and `emp_lname`. Use `%NOTFOUND` for the exit condition.

e.Include a `CASE` expression. Use the following table as reference for the conditions in the `WHEN` clause of the `CASE` expression.
Note: In your `CASE` expression use the constants such as `c_range1`, `c_hike1` which are already declared.

salary	**Hike percentage**
< 6500	20
> 6500 < 9500	15
> 9500 <12000	8
>12000	3

For example, if the salary of the employee is less than 6500, then increase the salary by 20 percent. In every `WHEN` clause, concatenate the `first_name` and `last_name` of the employee and store it in the `INDEX BY` table. Increment the value in variable `i` so that you can store the string in the next location. Include an `UPDATE` statement with the `WHERE CURRENT OF` clause.

f.Close the loop. Use the `%ROWCOUNT` attribute and print the number of records that were modified. Close the cursor.

g.Include a simple loop to print the names of all the employees whose salaries were revised.

Note: You already have the names of these employees in the `INDEX BY` table. Look for the comment "CLOSE THE INNER BLOCK" and include an `END IF` statement and an `END` statement.

f.Save your script as `lab_07_04_soln.sql`.

Practice 8

1. The purpose of this example is to show the usage of predefined exceptions. Write a PL/SQL block to select the name of the employee with a given salary value.
   a. Delete all records in the `messages` table. Use the `DEFINE` command to define a variable `sal` and initialize it to 6000.
   b. In the declarative section declare two variables: `ename` of type `employees.last_name` and `emp_sal` of type `employees.salary`. Pass the value of the substitution variables to `emp_sal`.
   c. In the executable section, retrieve the last names of employees whose salaries are equal to the value in `emp_sal`.
      Note: Do not use explicit cursors.
      If the salary entered returns only one row, insert into the `messages` table the employee's name and the salary amount.
   d. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "No employee with a salary of <*salary*>."
   e. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the `messages` table the message "More than one employee with a salary of <*salary*>."
   f. Handle any other exception with an appropriate exception handler and insert into the `messages` table the message "Some other error occurred."
   g. Display the rows from the `messages` table to check whether the PL/SQL block has executed successfully. Sample output is shown below.

Practice 8 (continued)

2. The purpose of this example is to show how to declare exceptions with a standard Oracle server error. Use the Oracle server error ORA-02292 (integrity constraint violated – child record found).
   a. In the declarative section, declare an exception `childrecord_exists`. Associate the declared exception with the standard Oracle server error – 02292.
   b. In the executable section, display 'Deleting department 40.....'. Include a `DELETE` statement to delete the department with `department_id` 40.

```
Deleting department 40.....
Cannot delete this department. There are employees in this department
(child records exist.)
PL/SQL procedure successfully completed.
```

   c. Include an exception section to handle the `childrecord_exists` exception and display the appropriate message. Sample output is shown below.
3. Load the script `lab_07_04_soln.sql`.
   a. Observe the declarative section of the outer block. Note that the `no_such_employee` exception is declared.
   b. Look for the comment "RAISE EXCEPTION HERE." If the value of `emp_id` is not between 100 and 206, then raise the `no_such_employee` exception.
   c. Look for the comment "INCLUDE EXCEPTION SECTION FOR OUTER BLOCK" and handle the exceptions `no_such_employee` and `too_many_rows`. Display appropriate messages when the exceptions occur. The `employees` table has only one employee working in the `HR` department and therefore the code is written accordingly. The `too_many_rows` exception is handled to indicate that the select statement retrieves more than one employee working in the `HR` department.
   d. Close the outer block.
   e. Save your script as `lab_08_03_soln.sql`.
   f. Execute the script. Enter the employee number and the department number and observe the output. Enter different values and check for different conditions. The sample output for employee ID 203 and department ID 100 is shown below.

```
NUMBER OF RECORDS MODIFIED : 6
The following employees' salaries are updated
Nancy Greenberg
Daniel Faviet
John Chen
Ismael Sciarra
Jose Manuel Urman
Luis Popp
PL/SQL procedure successfully completed.
```

Practice 1

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

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_DBA	Database Administrator		

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

_____

_____

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

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`.)

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

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.

Practice 1 (continued)

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.
   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.

SALARY
8000

JOB
ST_MAN

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

_____

_____

Practice 2

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

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

   b. Create a VARCHAR2 host variable called 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.

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 GET_ANNUAL_COMP function, 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)
```

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

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

6 rows selected.

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

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

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

Practice 3

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 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.

      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.
   c. Invoke your ADD_JOB package procedure by passing the values IT_SYSAN and SYSTEMS ANALYST as parameters.
   d. Query the JOBS table to see the result.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_SYSAN	Systems Analyst		

2. Create and invoke a package that contains private and public constructs.
   a. Create a package specification and 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.
   b. Invoke the EMP_PKG.ADD_EMPLOYEE procedure, using department ID 15 for employee Jane Harris with the e-mail ID JAHARRIS. Because department ID 15 does not exist, you should get an error message as specified in the exception handler of your procedure.
   c. Invoke the ADD_EMPLOYEE package procedure by using department ID 80 for employee David Smith with the e-mail ID DASMITH.

Practice 4

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 that accepts three parameters: the first name, last name, and department ID. Save and compile the changes.
   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. Save and compile the changes.
   c. Invoke the new ADD_EMPLOYEE procedure using the name Samuel Joplin to be added to department 30.
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 the parameter called family_name of type employees.last_name%TYPE. Both functions should return an EMPLOYEES%ROWTYPE. Save and compile the changes.
   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. Save and compile the changes.
   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. Save and compile the changes.
   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.
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.
   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 (continued)

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

4. Change `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. Save and compile the changes.

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

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

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

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

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

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

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

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

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

c. Fix the compilation error 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.

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.

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_pkb_b.sql`.

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

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

Practice 5

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

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

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

   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.

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.

   b. Write the `WEB_EMPLOYEE_REPORT` procedure by 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.

   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

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 divides 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. The `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.

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

c. During and after the process, you can query the `job_name` and enabled columns from the `USER_SCHEDULER_JOBS` table to check whether the job still exists.
Note: This query should return no rows after 10 minutes have elapsed.

Practice 6

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.

   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.

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

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

   d. Describe the MY_CONTACTS table structure.

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

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');

   f. Query the MY_CONTACTS table contents.

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

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

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

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

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

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

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

   b. In the body, the MAKE procedure should call a private function named GET_TYPE to determine the PL/SQL object type from the data

Practice 6 (continued)

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 statement 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.

   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 statement used to create the component using the `DBMS_METADATA.GET_DDL` procedure, 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.

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

   d. Use Putty FTP to get the generated file from the server to your local directory. Edit the file to insert 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.

Practice 7

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.

   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.

   c. Create a new procedure in the specification and body, called show_employees, that 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.

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

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

   a. First, load and execute the 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.

   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.

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

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

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

   f. Execute a ROLLBACK statement to undo the insert operations that have not been committed. Use the same queries from Exercise 2e: the first to check whether the employee rows for Smart and Kent have been removed, and the second to check the

Practice 7 (continued)

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 whether the `LOG_NEWEMP` table exists from Exercise 2 in this practice.

a. Grant the `EXECUTE` privilege on your `EMP_PKG` package to another student.

b. Ask the other student to invoke your `add_employee` procedure to insert 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.

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

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

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

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

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.

Practice 8

1. Answer the following questions:

   a. Can a table or a synonym 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?

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.

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

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

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

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

If you have time, complete the following exercise:

3. Dynamically validate invalid objects.

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

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

   c. Create and save a query to display the name, type, and status of all invalid objects.

   d. In the `compile_pkg` (created in Practice 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.

   e. Execute the `compile_pkg.recompile` procedure.

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

Practice 9

1. Create a table called PERSONNEL by executing the script file E:\labs\PLPU\labs\lab_09_01.sql. 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

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.
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.
4. Update the PERSONNEL table.
   a. Populate the CLOB for the first row, using this subquery in an UPDATE statement:

      SELECT ann_review
      FROM review_table
      WHERE employee_id = 2034;

   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;

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);

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

   b. Create a PL/SQL procedure called load_country_image that uses 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

Practice 10

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

   a. 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.

   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.

2. Test the CHECK_SAL_TRG using the following cases:

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

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

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

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

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

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

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

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

Practice 11

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 employees' 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 employees' salaries to the minimum for their jobs if their current salaries are less than the new minimum value.

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

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

2. To resolve the mutating table issue, you create a 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);
```

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 the

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.

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.

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

e. Test the code changes by executing the query to display the employees who are programmers, 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 jobs?

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

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

b. To correct the problem encountered when adding or updating an employee, 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.

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.

Practice 12

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.

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

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

   d. Switch compilation to use interpreted compilation.

2. In the 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.

   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.

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: an input parameter specifying a department ID, and an IN OUT parameter specifying a PL/SQL table of employee rows.
   Hint: You must declare a TYPE in the package specification for the PL/SQL table parameter's data type.

   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.

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

5. Recompile the TEST_PKG that you created two steps earlier (in Exercise 3). What compiler warnings are displayed, if any?

6. Write a PL/SQL anonymous block to compile the