ORACLE UNIVERSITY

Integrated Cloud Applications & Platform Services

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

Additional Material

D99739GC20

Edition 2.0 | March 2017 | D99762

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

ORACLE

Table of Contents

Additional Practices and Solutions for Lesson 1

Chapter 1

Practices for Lesson 1

There are no practices for this lesson.

Additional Practices and Solutions for Lesson 2

Chapter 2

Additional Practices for Lesson 2: Overview

Overview

These additional practices are provided as a supplement to the *Oracle Database: PL/SQL Fundamentals* course. In these practices, you apply the concepts that you learned in the course.

These additional practices provide supplemental practice in declaring variables, writing executable statements, interacting with the Oracle Server, writing control structures, and working with composite data types, cursors, and handle exceptions. The tables used in this portion of the additional practices include `employees`, `jobs`, `job_history`, and `departments`.

Practice 2: Evaluating Declarations

Overview

These paper-based exercises are used for extra practice in declaring variables and writing executable statements.

Evaluate each of the following declarations. Determine which of them are not legal and explain why.

```
1. DECLARE
   name,dept     VARCHAR2(14);
2. DECLARE
   test          NUMBER(5);
3. DECLARE
   MAXSALARY     NUMBER(7,2) = 5000;
4. DECLARE
   JOINDATE      BOOLEAN := SYSDATE;
```

Solution 2: Evaluating Declarations

Evaluate each of the following declarations. Determine which of them are not legal and explain why.

1.
```
DECLARE
name,dept     VARCHAR2(14);
```
This is illegal because only one identifier per declaration is allowed.

2.
```
DECLARE
test          NUMBER(5);
```
This is legal.

3.
```
DECLARE
MAXSALARY     NUMBER(7,2) = 5000;
```
This is illegal because the assignment operator is wrong. It should be :=.

4.
```
DECLARE
JOINDATE      BOOLEAN := SYSDATE;
```
This is illegal because there is a mismatch in the data types. A Boolean data type cannot be assigned a date value. The data type should be date.

Additional Practices and Solutions for Lesson 3

Chapter 3

Practice 3: Evaluating Expressions

In each of the following assignments, determine the data type of the resulting expression.

```
1. email := firstname || to_char(empno);

2. confirm := to_date('20-JAN-1999', 'DD-MON-YYYY');

3. sal := (1000*12) +  500

4. test := FALSE;

5. temp := temp1 < (temp2/ 3);

6. var := sysdate;
```

Solution 3: Evaluating Expressions

In each of the following assignments, determine the data type of the resulting expression.

1. `email := firstname || to_char(empno);`

   Character string

2. `confirm := to_date('20-JAN-1999', 'DD-MON-YYYY');`

   Date

3. `sal := (1000*12) +  500`

   Number

4. `test := FALSE;`

   Boolean

5. `temp := temp1 < (temp2/ 3);`

   Boolean

6. `var := sysdate;`

   Date

Additional Practices and Solutions for Lesson 4

Chapter 4

Practice 4: Evaluating Executable Statements

In this paper-based exercise, you evaluate the PL/SQL block, and then answer the questions that follow by determining the data type and value of each variable, according to the rules of scoping.

```
DECLARE
      v_custid    NUMBER(4) := 1600;
      v_custname VARCHAR2(300) := 'Women Sports Club';
      v_ new_custid   NUMBER(3) := 500;
BEGIN
DECLARE
      v_custid     NUMBER(4) := 0;
      v_custname  VARCHAR2(300) := 'Shape up Sports Club';
      v_new_custid  NUMBER(3) := 300;
      v_new_custname  VARCHAR2(300) := 'Jansports Club';
BEGIN
      v_custid := v_new_custid;
      v_custname := v_custname || ' ' ||  v_new_custname;
1 ---------->
END;
      v_custid := (v_custid *12) / 10;
2 ---------->
END;
```

Evaluate the preceding PL/SQL block and determine the *value* and *data type* of each of the following variables, according to the rules of scoping:

1. `v_custid` at position 1:

2. `v_custname` at position 1:

3. `v_new_custid` at position 1:

4. `v_new_custname` at position 1:

5. `v_custid` at position 2:

6. `v_custname` at position 2:

Solution 4: Evaluating Executable Statements

Evaluate the following PL/SQL block. Then, answer the questions that follow by determining the data type and value of each of the following variables, according to the rules of scoping.

```
DECLARE
      v_custid    NUMBER(4) := 1600;
      v_custname VARCHAR2(300) := 'Women Sports Club';
      v_ new_custid   NUMBER(3) := 500;
BEGIN
DECLARE
      v_custid     NUMBER(4) := 0;
      v_custname  VARCHAR2(300) := 'Shape up Sports Club';
      v_new_custid  NUMBER(3) := 300;
      v_new_custname  VARCHAR2(300) := 'Jansports Club';
BEGIN
      v_custid := v_new_custid;
      v_custname := v_custname || ' ' ||  v_new_custname;
1 ---------->
END;
      v_custid := (v_custid *12) / 10;
2 ---------->
END;
```

Evaluate the preceding PL/SQL block and determine the *value* and *data type* of each of the following variables, according to the rules of scoping:

1. `v_custid` at position 1:

   500, and the data type is `NUMBER`.

2. `v_custname` at position 1:

   Shape up Sports Club Jansports Club, and the data type is `VARCHAR2`.

3. `v_new_custid` at position 1:

   300, and the data type is `NUMBER` (or `INTEGER`).

4. `v_new_custname` at position 1:

   Jansports Club, and the data type is `VARCHAR2`.

5. `v_custid` at position 2:

   1920, and the data type is `NUMBER`.

6. `v_custname` at position 2:

   Women Sports Club, and the data type is `VARCHAR2`.

Additional Practices and Solutions for Lesson 5

Chapter 5

Practice 5-1: Using SQL Statements Within a PL/SQL

For this exercise, a temporary table is required to store the results.

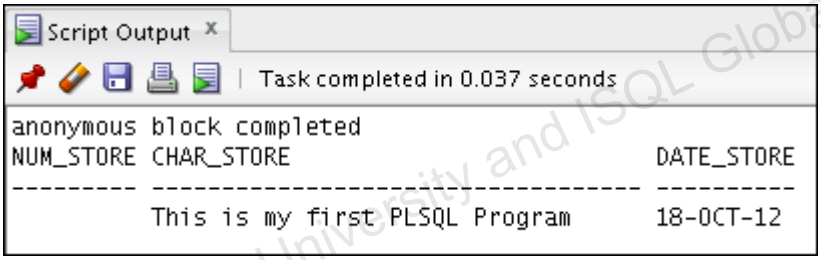
1. Run the `lab_ap_05.sql` script that creates the table described here:

Column Name	NUM_STORE	CHAR_STORE	DATE_STORE
Key Type			
Nulls/Unique			
FK Table			
FK Column			
Data Type	Number	VARCHAR2	Date
Length	7,2	35	

2. Write a PL/SQL block that performs the following:
   a. Declares two variables and assigns the following values to these variables:

Variable	Data type	Contents
V_MESSAGE	VARCHAR2(35)	This is my first PL/SQL program
V_ DATE_WRITTEN	DATE	Current date

   b. Stores the values from these variables in the appropriate `TEMP` table columns
3. Verify your results by querying the `TEMP` table. The output results should appear as follows:



Solution 5-1: Using SQL Statements Within a PL/SQL

For this exercise, a temporary table is required to store the results.

1. Run the `lab_ap_05.sql` script that creates the table described here:

Column Name	NUM_STORE	CHAR_STORE	DATE_STORE
Key Type			
Nulls/Unique			
FK Table			
FK Column			
Data Type	Number	VARCHAR2	Date
Length	7,2	35	

2. Write a PL/SQL block that performs the following:
   a. Declares two variables and assigns the following values to these variables:

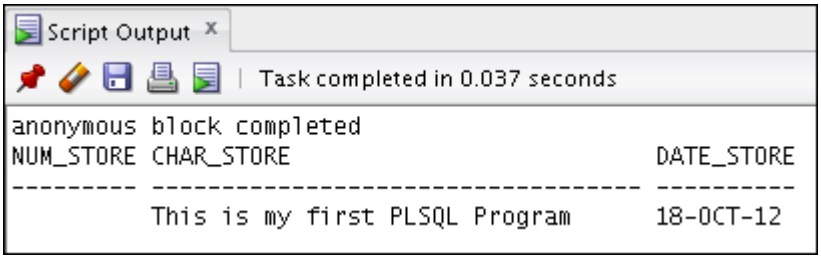
Variable	Data type	Contents
V_MESSAGE	VARCHAR2(35)	This is my first PL/SQL program
V_ DATE_WRITTEN	DATE	Current date

   b. Stores the values from these variables in the appropriate `TEMP` table columns

```
DECLARE
  V_MESSAGE VARCHAR2(35);
  V_DATE_WRITTEN DATE;
BEGIN
  V_MESSAGE := 'This is my first PLSQL Program';
  V_DATE_WRITTEN := SYSDATE;
  INSERT INTO temp(CHAR_STORE,DATE_STORE)
    VALUES (V_MESSAGE,V_DATE_WRITTEN);
END;
/
```

3. Verify your results by querying the `TEMP` table. The output results should look similar to the following:

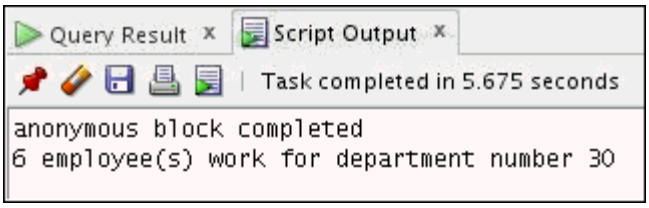
```
SELECT * FROM TEMP;
```



Practice 5-2: Using SQL Statements Within a PL/SQL

In this exercise, you use data from the `employees` table.

1. Write a PL/SQL block to determine how many employees work for a specified department. The PL/SQL block should:
   - Use a substitution variable to store a department number
   - Print the number of people working in the specified department
2. When the block is run, a substitution variable window appears. Enter a valid department number and click OK. The output results should look similar to the following:



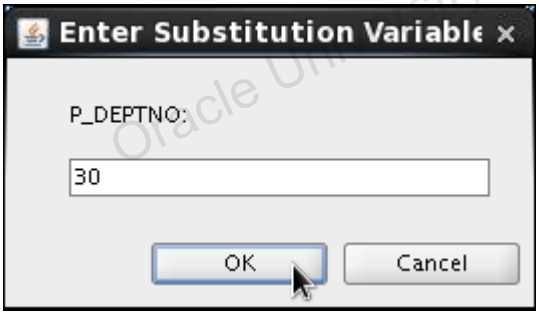
Solution 5-2: Using SQL Statements Within a PL/SQL

In this exercise, you use data from the `employees` table.

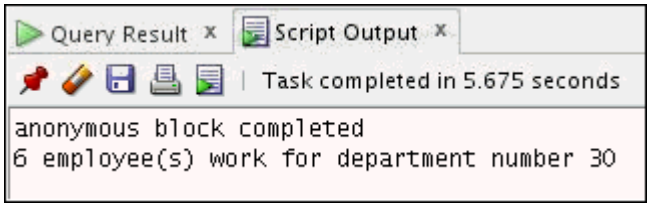
1. Write a PL/SQL block to determine how many employees work for a specified department. The PL/SQL block should:
   - Use a substitution variable to store a department number
   - Print the number of people working in the specified department

```
SET SERVEROUTPUT ON;
DECLARE
   V_HOWMANY NUMBER(3);
   V_DEPTNO DEPARTMENTS.department_id%TYPE := &P_DEPTNO;
BEGIN
   SELECT COUNT(*) INTO V_HOWMANY FROM employees
   WHERE department_id = V_DEPTNO;
   DBMS_OUTPUT.PUT_LINE (V_HOWMANY  || ' employee(s)
     work for department number ' ||V_DEPTNO);
END;
/
```

2. When the block is run, a substitution variable window appears. Enter a valid department number and click OK.



The output results should look similar to the following:



Additional Practices and Solutions for Lesson 6

Chapter 6

Practice 6-1: Writing Control Structures

In these practices, you use control structures to direct the logic of program flow.

1. Write a PL/SQL block to accept a year input and check whether it is a leap year.
   Hint: The year should be exactly divisible by 4 but not divisible by 100, or it should be divisible by 400.
2. Test your solution by using the following table. For example, if the year entered is 1990, the output should be “1990 is not a leap year.”

1990	Not a leap year
2000	Leap year
1996	Leap year
1886	Not a leap year
1992	Leap year
1824	Leap year

Solution 6-1: Writing Control Structures

1. Write a PL/SQL block to accept a year input and check whether it is a leap year.
   Hint: The year should be exactly divisible by 4 but not divisible by 100, or it should be divisible by 400.

```
SET SERVEROUTPUT ON;
DECLARE
   v_YEAR NUMBER(4) := &P_YEAR;
   v_REMAINDER1 NUMBER(5,2);
   v_REMAINDER2 NUMBER(5,2);
   v_REMAINDER3 NUMBER(5,2);
BEGIN
   v_REMAINDER1 := MOD(v_YEAR,4);
   v_REMAINDER2 := MOD(v_YEAR,100);
   v_REMAINDER3 := MOD(v_YEAR,400);
   IF ((v_REMAINDER1 = 0 AND v_REMAINDER2 <> 0 ) OR
       v_REMAINDER3 = 0) THEN
      DBMS_OUTPUT.PUT_LINE(v_YEAR || ' is a leap year');
   ELSE
      DBMS_OUTPUT.PUT_LINE(v_YEAR || ' is not a leap
      year');
   END IF;
END;
/
```

2. Test your solution by using the following table. For example, if the year entered is 1990, the output should be “1990 is not a leap year.”

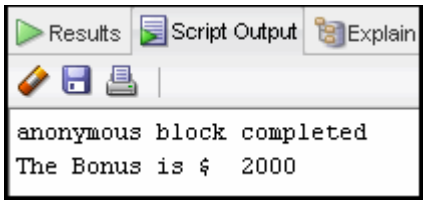
1990	Not a leap year
2000	Leap year
1996	Leap year
1886	Not a leap year
1992	Leap year
1824	Leap year

Practice 6-2: Writing Control Structures

1. Write a PL/SQL block to store the monthly salary of an employee in a substitution variable. The PL/SQL block should:
   - Calculate the annual salary as salary * 12
   - Calculate the bonus as indicated in the following table:

Annual Salary	Bonus
>= 20,000	2,000
19,999–10,000	1,000
<= 9,999	500

   - Display the amount of the bonus in the Script Output window in the following format:



2. Test the PL/SQL for the following test cases:

Monthly Salary	Bonus
3000	2000
1200	1000
800	500

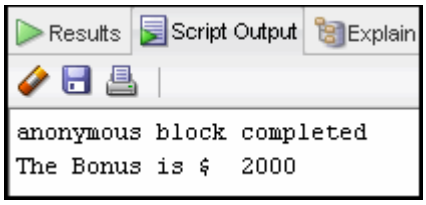
Solution 6-2: Writing Control Structures

1. Write a PL/SQL block to store the monthly salary of an employee in a substitution variable. The PL/SQL block should:

   - Calculate the annual salary as salary * 12
   - Calculate the bonus as indicated in the following table:

Annual Salary	Bonus
>= 20,000	2,000
19,999–10,000	1,000
<= 9,999	500

   - Display the amount of the bonus in the Script Output window in the following format:




```
SET SERVEROUTPUT ON;
DECLARE
   V_SAL        NUMBER(7,2) := &B_SALARY;
   V_BONUS      NUMBER(7,2);
   V_ANN_SALARY NUMBER(15,2);
BEGIN
   V_ANN_SALARY := V_SAL * 12;
   IF V_ANN_SALARY >= 20000 THEN
      V_BONUS := 2000;
   ELSIF V_ANN_SALARY <= 19999 AND V_ANN_SALARY >=10000 THEN
      V_BONUS := 1000;
   ELSE
      V_BONUS := 500;
   END IF;
   DBMS_OUTPUT.PUT_LINE ('The Bonus is $  ' ||
     TO_CHAR(V_BONUS));
END;
/
```

2. Test the PL/SQL for the following test cases:

Monthly Salary	Bonus
3000	2000
1200	1000
800	500

Additional Practices and Solutions for Lessons 7 and 8

Chapter 7

Additional Practices for Lessons Titled “Working with Composite Data Types” and “Using Explicit Cursors”

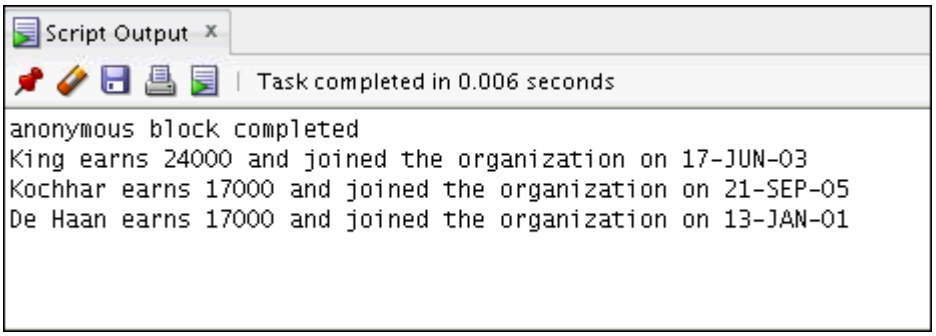
Overview

In the following exercises, you practice using associative arrays (this topic is covered in the lesson titled “Working with Composite Data Types”) and explicit cursors (this topic is covered in the lesson titled “Using Explicit Cursors”). In the first exercise, you define and use an explicit cursor to fetch data. In the second exercise, you combine the use of associative arrays with an explicit cursor to output data that meets a certain criteria.

Practice 7/8-1: Fetching Data with an Explicit Cursor

In this practice, you create a PL/SQL block to perform the following:

1. Declare a cursor named `EMP_CUR` to select the employee's last name, salary, and hire date from the `EMPLOYEES` table.
2. Process each row from the cursor, and if the salary is greater than 15,000 and the hire date is later than 01-FEB-1988, display the employee name, salary, and hire date in the format shown in the following sample output:



Solution 7/8-1: Fetching Data with an Explicit Cursor

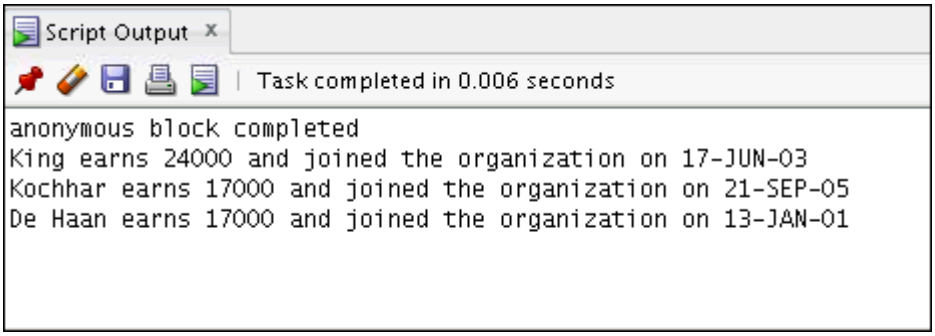
In this practice, you create a PL/SQL block to perform the following:

1. Declare a cursor named `EMP_CUR` to select the employee's last name, salary, and hire date from the `EMPLOYEES` table.

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR C_EMP_CUR IS
    SELECT  last_name,salary,hire_date FROM EMPLOYEES;
  V_ENAME VARCHAR2(25);
  v_SAL   NUMBER(7,2);
  V_HIREDATE DATE;
```

2. Process each row from the cursor, and if the salary is greater than 15,000 and the hire date is later than 01-FEB-1988, display the employee name, salary, and hire date in the format shown in the following sample output:

```
BEGIN
  OPEN C_EMP_CUR;
  FETCH C_EMP_CUR INTO V_ENAME,V_SAL,V_HIREDATE;
  WHILE C_EMP_CUR%FOUND
  LOOP
    IF V_SAL > 15000 AND V_HIREDATE >=
       TO_DATE('01-FEB-1988','DD-MON-YYYY') THEN
         DBMS_OUTPUT.PUT_LINE (V_ENAME || ' earns '
         || TO_CHAR(V_SAL)|| ' and joined the organization on '
         || TO_DATE(V_HIREDATE,'DD-Mon-YYYY'));
    END IF;
    FETCH C_EMP_CUR INTO V_ENAME,V_SAL,V_HIREDATE;
  END LOOP;
  CLOSE C_EMP_CUR;
END;
/
```



Practice 7/8-2: Using Associative Arrays and Explicit Cursors

In this practice, you create a PL/SQL block to retrieve and output the last name and department ID of each employee from the `EMPLOYEES` table for those employees whose `EMPLOYEE_ID` is less than 115.

In the PL/SQL block, use a cursor `FOR` loop strategy instead of the `OPEN` / `FETCH` / `CLOSE` cursor methods used in the previous practice.

1. In the declarative section:
   - Create two associative arrays. The unique key column for both arrays should be of the `BINARY INTEGER` data type. One array holds the employee's last name and the other holds the department ID.
   - Declare a cursor that selects the last name and department ID for employees whose ID is less than 115
   - Declare the appropriate counter variable to be used in the executable section
2. In the executable section, use a cursor `FOR` loop (covered in the lesson titled "Using Explicit Cursors") to access the cursor values, assign them to the appropriate associative arrays, and output those values from the arrays. The correct output should return 15 rows, in the following format:

```
Script Output
Task completed in 0.003 seconds
anonymous block completed
Employee: King is in department number: 90
Employee: Kochhar is in department number: 90
Employee: De Haan is in department number: 90
Employee: Hunold is in department number: 60
Employee: Ernst is in department number: 60
Employee: Austin is in department number: 60
Employee: Pataballa is in department number: 60
Employee: Lorentz is in department number: 60
Employee: Greenberg is in department number: 100
Employee: Faviet is in department number: 100
Employee: Chen is in department number: 100
Employee: Sciarra is in department number: 100
Employee: Urman is in department number: 100
Employee: Popp is in department number: 100
Employee: Raphaely is in department number: 30
```

Solution 7/8-2: Using Associative Arrays and Explicit Cursors

In this practice, you create a PL/SQL block to retrieve and output the last name and department ID of each employee from the `EMPLOYEES` table for those employees whose `EMPLOYEE_ID` is less than 115.

In the PL/SQL block, use a cursor `FOR` loop strategy instead of the `OPEN` / `FETCH` / `CLOSE` cursor methods used in the previous practice.

1. In the declarative section:
   - Create two associative arrays. The unique key column for both arrays should be of the `BINARY INTEGER` data type. One array holds the employee's last name and the other holds the department ID.
   - Declare a counter variable to be used in the executable section.
   - Declare a cursor that selects the last name and department ID for employees whose ID is less than 115.

```
SET SERVEROUTPUT ON;
DECLARE
      TYPE Table_Ename IS table of employees.last_name%TYPE
            INDEX BY BINARY_INTEGER;
      TYPE Table_dept IS table of employees.department_id%TYPE
            INDEX BY BINARY_INTEGER;
      Tename Table_Ename;
      Tdept  Table_dept;
      i BINARY_INTEGER :=0;
      CURSOR Namedept IS SELECT last_name,department_id
  FROM employees WHERE employee_id < 115;
```

2. In the executable section, use a cursor `FOR` loop (covered in the lesson titled "Using Explicit Cursors") to access the cursor values, assign them to the appropriate associative arrays, and output those values from the arrays.

```
BEGIN
  FOR emprec in Namedept
  LOOP
    i         := i +1;
    Tename(i) := emprec.last_name;
     Tdept(i)  := emprec.department_id;
    DBMS_OUTPUT.PUT_LINE ('Employee: ' || Tename(i) ||
' is in department number: ' || Tdept(i));
  END LOOP;
END;
/
```

The correct output should return 15 rows, similar to the following:

```
anonymous block completed
Employee: King is in department number: 90
Employee: Kochhar is in department number: 90
Employee: De Haan is in department number: 90
Employee: Hunold is in department number: 60
Employee: Ernst is in department number: 60
Employee: Austin is in department number: 60
Employee: Pataballa is in department number: 60
Employee: Lorentz is in department number: 60
Employee: Greenberg is in department number: 100
Employee: Faviet is in department number: 100
Employee: Chen is in department number: 100
Employee: Sciarra is in department number: 100
Employee: Urman is in department number: 100
Employee: Popp is in department number: 100
Employee: Raphaely is in department number: 30
```

Additional Practices and Solutions for Lesson 8

Chapter 8

Practices for Lesson 8

Practices of this lesson are included in Practice 7.

Additional Practices and Solutions for Lesson 9

Chapter 9

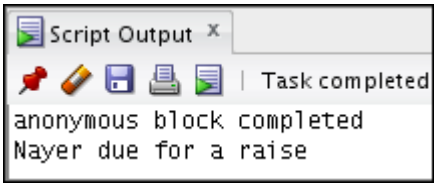
Practice 9-1: Handling Exceptions

For this exercise, you must first create a table to store some results. Run the `lab_ap_09.sql` script that creates the table for you. The script looks like the following:

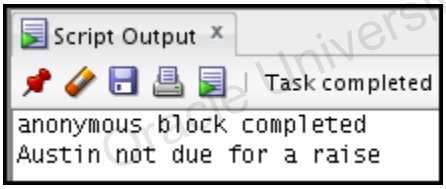
```
CREATE TABLE analysis
       (ename Varchar2(20), years Number(2), sal Number(8,2)
    );
```

In this practice, you write a PL/SQL block that handles an exception, as follows:

1. Declare variables for the employee last name, salary, and hire date. Use a substitution variable for the employee last name. Then, query the `EMPLOYEES` table for the `last_name`, `salary`, and `hire_date` of the specified employee.
2. If the employee has been with the organization for more than five years, and if that employee's salary is less than 3,500, raise an exception. In the exception handler, perform the following:
   - Output the following information: employee last name and the message "due for a raise," similar to the following:

   - Insert the last name, years of service, and salary into the `ANALYSIS` table.
3. If there is no exception, output the employee last name and the message "not due for a raise," similar to the following:

4. Verify the results by querying the `ANALYSIS` table. Use the following test cases to test the PL/SQL block.

LAST_NAME	MESSAGE
Austin	Not due for a raise
Nayer	Due for a raise
Fripp	Not due for a raise
Khoo	Due for a raise

Solution 9-1: Handling Exceptions

For this exercise, you must first create a table to store some results. Run the `lab_ap_09.sql` script that creates the table for you. The script looks similar to the following:

```
CREATE TABLE analysis
       (ename Varchar2(20), years Number(2), sal Number(8,2)
    );
```

In this practice, you write a PL/SQL block that handles an exception, as follows:

1. Declare variables for the employee last name, salary, and hire date. Use a substitution variable for the employee last name. Then, query the `EMPLOYEES` table for the `last_name`, `salary`, and `hire_date` of the specified employee.
2. If the employee has been with the organization for more than five years, and if that employee's salary is less than 3,500, raise an exception. In the exception handler, perform the following:
   - Output the following information: employee last name and the message "due for a raise."
   - Insert the employee name, years of service, and salary into the `ANALYSIS` table.
3. If there is no exception, output the employee last name and the message "not due for a raise."

```
SET SERVEROUTPUT ON;
DECLARE
    E_DUE_FOR_RAISE EXCEPTION;
    V_HIREDATE EMPLOYEES.HIRE_DATE%TYPE;
    V_ENAME EMPLOYEES.LAST_NAME%TYPE := INITCAP( '& B_ENAME');
    V_SAL EMPLOYEES.SALARY%TYPE;
    V_YEARS NUMBER(2);
BEGIN
    SELECT
SALARY,HIRE_DATE,MONTHS_BETWEEN(SYSDATE,hire_date)/12 YEARS
    INTO V_SAL,V_HIREDATE,V_YEARS
    FROM employees WHERE last_name = V_ENAME;
         IF V_SAL < 3500 AND V_YEARS > 5 THEN
              RAISE E_DUE_FOR_RAISE;
         ELSE
              DBMS_OUTPUT.PUT_LINE (V_ENAME|| ' not due for a
raise');
         END IF;
    EXCEPTION
         WHEN E_DUE_FOR_RAISE THEN
         BEGIN
              DBMS_OUTPUT.PUT_LINE (V_ENAME || ' due for a
raise');
              INSERT INTO ANALYSIS(ENAME,YEARS,SAL)
```

```
                    VALUES (V_ENAME,V_YEARS,V_SAL);
            END;
END;
    /
```

4. Verify the results by querying the `ANALYSIS` table. Use the following test cases to test the PL/SQL block.

LAST_NAME	MESSAGE
Austin	Not due for a raise
Nayer	Due for a raise
Fripp	Not due for a raise
Khoo	Due for a raise

```
SELECT * FROM analysis;
```

	ENAME	YEARS	SAL
1	Nayer	9	3200
2	Khoo	11	3100

Additional Practices 10

Chapter 10

Additional Practices 10

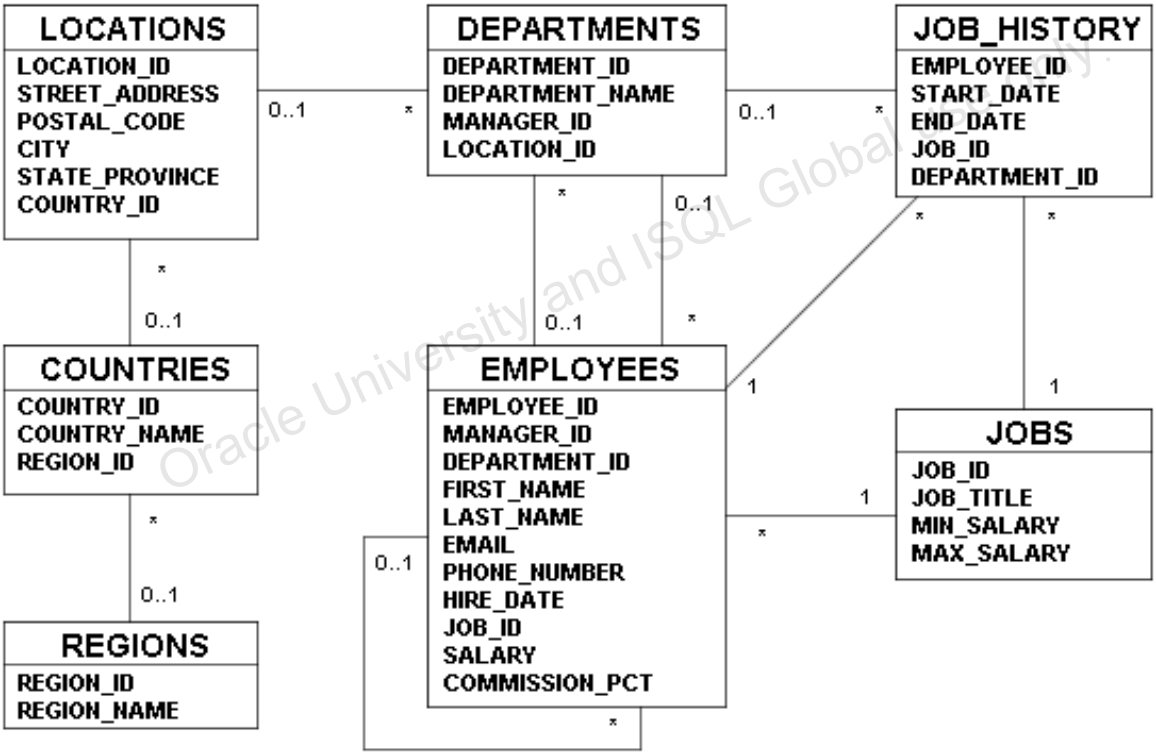
Overview

The additional practices are provided as a supplement to the course *Oracle Database: Develop PL/SQL Program Units*. In these practices, you apply the concepts that you learned in the course. The additional practices comprise two lessons.

Lesson 1 provides supplemental exercises to create stored procedures, functions, packages, and triggers, and to use the Oracle-supplied packages with SQL Developer or SQL*Plus as the development environment. The tables used in this portion of the additional practice include `EMPLOYEES`, `JOBS`, `JOB_HISTORY`, and `DEPARTMENTS`.

An entity relationship diagram is provided at the start of each practice. Each entity relationship diagram displays the table entities and their relationships. More detailed definitions of the tables and the data contained in them is provided in the appendix titled "Additional Practices: Table Descriptions and Data."

The Human Resources (HR) Schema Entity Relationship Diagram



Practice 10-1: Creating a New SQL Developer Database Connection

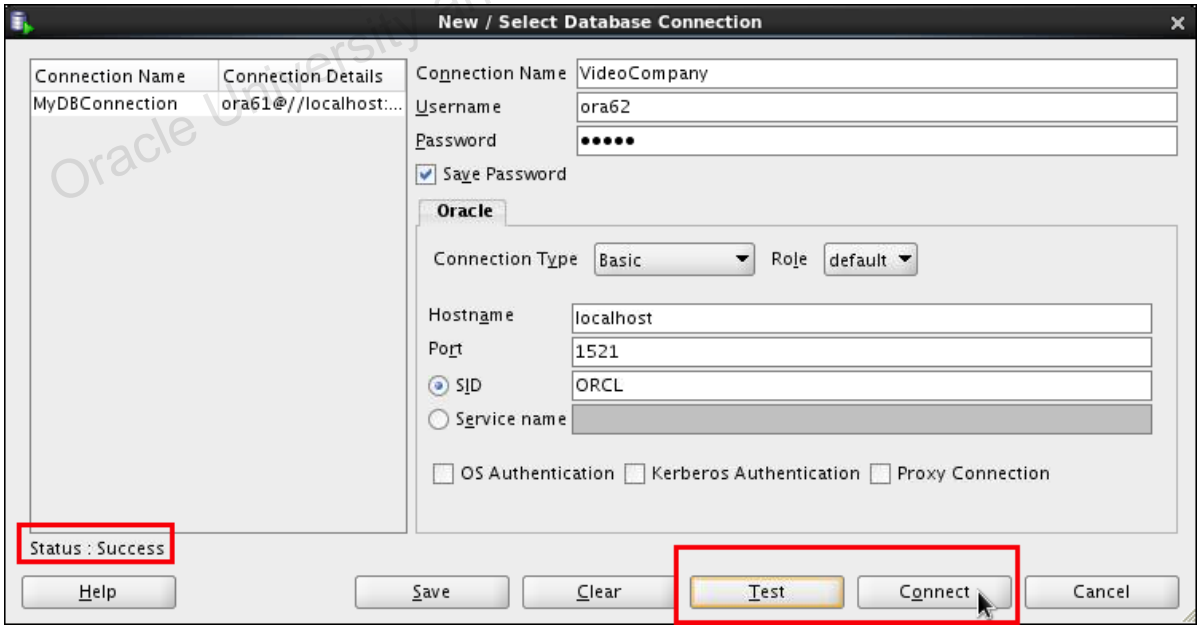
Overview

In this practice, you start SQL Developer using your connection information and create a new database connection.

Start up SQL Developer using the user ID and password that are provided to you by the instructor such as `ora62`.

Task

1. Start up SQL Developer using the user ID and password that are provided to you by the instructor such as `ora62`.
2. Create a database connection using the following information:
   a. Connection Name: `VideoCompany`
   b. Username: `ora62`
   c. Password: `ora62`
   d. Select the Save Password check box.
   e. Hostname: Enter the host name for your PC or alternatively mention localhost
   f. Port: 1521
   g. SID: `ORCL`
3. Test the new connection. If the Status shows as Success, connect to the database using this new connection:
   a. Click the Test button in the New/Select Database Connection window. If the status shows as Success, click the Connect button.



Solution 10-1: Creating a New SQL Developer Database Connection

In this practice, you start SQL Developer using your connection information and create a new database connection.

1. Start up SQL Developer using the user ID and password that are provided to you by the instructor, such as `ora62`.

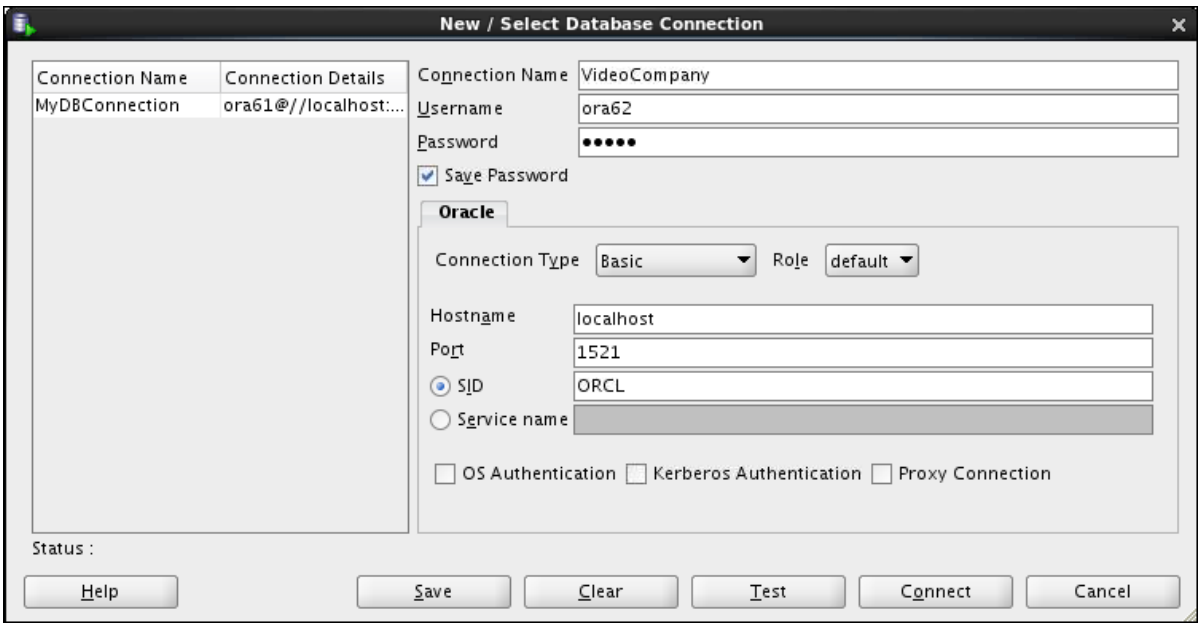
   Click the SQL Developer icon on your desktop.

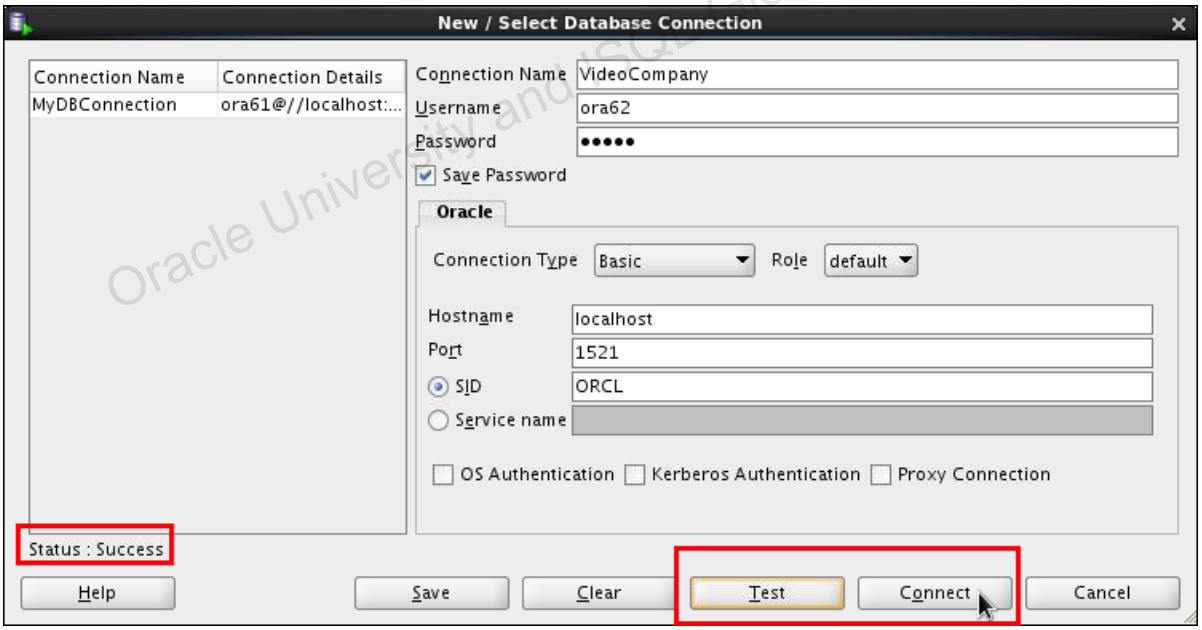
2. Create a database connection using the following information:
   a. Connection Name: `VideoCompany`
   b. Username: `ora62`
   c. Password: `ora62`
   d. Hostname: Enter the host name for your PC or let the default `localhost` remain.
   e. Port: 1521
   f. SID: `ORCL`

   Right-click the Connections icon on the Connections tabbed page, and then select the New Connection option from the shortcut menu. The New/Select Database Connection window is displayed. Use the preceding information provided to create the new database connection.

   Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Substitute the username, password, host name, and service name with the appropriate information as provided by your instructor. The following is a sample of the newly created database connection for student `ora62`:



3. Test the new connection. If the status shows as Success, connect to the database using this new connection:
   a. Click the Test button in the New/Select Database Connection window. If the status shows as Success, click the Connect button.



Practice 10-2: Adding a New Job to the `JOBS` Table

Overview

In this practice, you create a subprogram to add a new job into the `JOBS` table.

Tasks

1. Create a stored procedure called `NEW_JOB` to enter a new order into the `JOBS` table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.
2. Enable `SERVEROUTPUT`, and then invoke the procedure to add a new job with job ID '`SY_ANAL`', job title '`System Analyst`', and minimum salary of 6000.
3. Check whether a row was added and note the new job ID for use in the next exercise. Commit the changes.

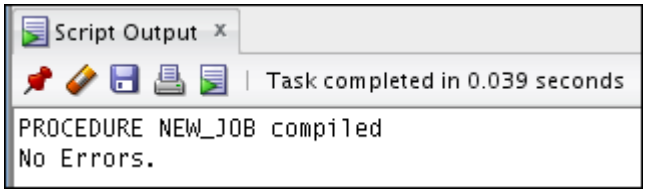
Solution 10-2: Adding a New Job to the JOBS Table

In this practice, you create a subprogram to add a new job into the JOBS table.

1. Create a stored procedure called NEW_JOB to enter a new order into the JOBS table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.

Open the /home/oracle/labs/plpu/solns/sol_ap1.sql script. Uncomment and select the code under Task 1 of Additional Practice 1-2. Click the Run Script icon (or press F5) on the SQL Worksheet toolbar to create and compile the procedure. Make sure that you have selected the new VideoCompany connection. The code, connection prompt, and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE new_job(
  p_jobid  IN jobs.job_id%TYPE,
  p_title  IN jobs.job_title%TYPE,
  v_minsal IN jobs.min_salary%TYPE) IS
  v_maxsal  jobs.max_salary%TYPE := 2 * v_minsal;
BEGIN
  INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
  VALUES (p_jobid, p_title, v_minsal, v_maxsal);
  DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
  DBMS_OUTPUT.PUT_LINE (p_jobid || '  ' || p_title ||' '||
                        v_minsal || '  ' || v_maxsal);
END new_job;
/
SHOW ERRORS
```



2. Enable `SERVEROUTPUT`, and then invoke the procedure to add a new job with job ID '`SY_ANAL`', job title '`System Analyst`', and minimum salary of 6000.

   Uncomment and select the code under Task 2 of Additional Practice 1-2. When prompted to select a connection, select the new `VideoCompany` connection. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE new_job ('SY_ANAL', 'System Analyst', 6000)
```

```
Script Output x
Task completed in 1.065 seconds
anonymous block completed
New row added to JOBS table:
SY_ANAL  System Analyst 6000  12000
```

3. Check whether a row was added and note the new job ID for use in the next exercise. Commit the changes.

   Run Uncomment and select the code under Task 3 of Additional Practice 1-2. The code and the results are displayed as follows:

```
SELECT *
FROM   jobs
WHERE  job_id = 'SY_ANAL';
COMMIT;
```

```
Script Output x
Task completed in 0.047 seconds
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
SY_ANAL    System Analyst                           6000      12000

commited.
```

Practice 10-3: Adding a New Row to the `JOB_HISTORY` Table

Overview

In this Additional Practice, you add a new row to the `JOB_HISTORY` table for an existing employee.

Tasks

1. Create a stored procedure called `ADD_JOB_HIST` to add a new row into the `JOB_HISTORY` table for an employee who is changing his job to the new job ID (`'SY_ANAL'`) that you created in Task 2 of Practice 1-2.
   a. The procedure should provide two parameters, one for the employee ID who is changing the job, and the second for the new job ID.
   b. Read the employee ID from the `EMPLOYEES` table and insert it into the `JOB_HISTORY` table.
   c. Make the hire date of this employee as start date and today's date as end date for this row in the `JOB_HISTORY` table.
   d. Change the hire date of this employee in the `EMPLOYEES` table to today's date.
   e. Update the job ID of this employee to the job ID passed as parameter (use the `'SY_ANAL'` job ID) and salary equal to the minimum salary for that job ID + 500. **Note:** Include exception handling to handle an attempt to insert a nonexistent employee.
2. Disable all triggers on the `EMPLOYEES`, `JOBS`, and `JOB_HISTORY` tables before invoking the `ADD_JOB_HIST` procedure.
3. Enable `SERVEROUTPUT`, and then execute the procedure with employee ID 106 and job ID `'SY_ANAL'` as parameters.
4. Query the `JOB_HISTORY` and `EMPLOYEES` tables to view your changes for employee 106, and then commit the changes.
5. Re-enable the triggers on the `EMPLOYEES`, `JOBS`, and `JOB_HISTORY` tables.

Solution 10-3: Adding a New Row to the JOB_HISTORY Table

In this Additional Practice, you add a new row to the JOB_HISTORY table for an existing employee.

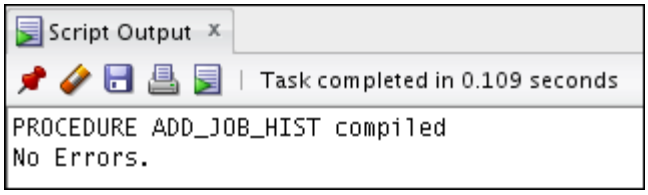
1. Create a stored procedure called ADD_JOB_HIST to add a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID ('SY_ANAL') that you created in exercise 2.
   a. The procedure should provide two parameters, one for the employee ID who is changing the job, and the second for the new job ID.
   b. Read the employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table.
   c. Make the hire date of this employee as start date and today's date as end date for this row in the JOB_HISTORY table.
   d. Change the hire date of this employee in the EMPLOYEES table to today's date.
   e. Update the job ID of this employee to the job ID passed as parameter (use the 'SY_ANAL' job ID) and salary equal to the minimum salary for that job ID + 500.

      Note: Include exception handling to handle an attempt to insert a nonexistent employee.

      Uncomment and select the code under Task 1 of Additional Practice 1-3. The code and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE add_job_hist(
  p_emp_id    IN employees.employee_id%TYPE,
  p_new_jobid IN jobs.job_id%TYPE) IS
BEGIN
  INSERT INTO job_history
    SELECT employee_id, hire_date, SYSDATE, job_id,
department_id
    FROM   employees
    WHERE  employee_id = p_emp_id;
  UPDATE employees
    SET  hire_date = SYSDATE,
         job_id = p_new_jobid,
         salary = (SELECT min_salary + 500
                   FROM   jobs
                   WHERE  job_id = p_new_jobid)
   WHERE employee_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
                        ' details to the JOB_HISTORY table');
  DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
                        p_emp_id|| ' to '|| p_new_jobid);
```

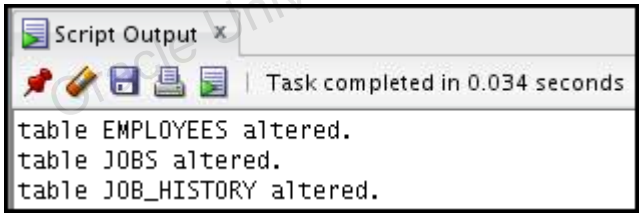
```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20001, 'Employee does not
exist!');
END add_job_hist;
/
SHOW ERRORS
```



2. Disable all triggers on the `EMPLOYEES`, `JOBS`, and `JOB_HISTORY` tables before invoking the `ADD_JOB_HIST` procedure.

Uncomment and select the code under Task 2 of Additional Practice 1-3. The code and the results are displayed as follows:

```
ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;
ALTER TABLE job_history DISABLE ALL TRIGGERS;
```



3. Enable `SERVEROUTPUT`, and then execute the procedure with employee ID 106 and job ID '`SY_ANAL`' as parameters.

Uncomment and select the code under Task 3 of Additional Practice 1-3. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE add_job_hist(106, 'SY_ANAL')
```

```
Script Output
Task completed in 0.101 seconds
anonymous block completed
Added employee 106 details to the JOB_HISTORY table
Updated current job of employee 106 to SY_ANAL
```

4. Query the `JOB_HISTORY` and `EMPLOYEES` tables to view your changes for employee 106, and then commit the changes.

Uncomment and select the code under Task 4 of Additional Practice 1-3. The code and the results are displayed as follows:

```
SELECT *  FROM  job_history
WHERE  employee_id = 106;

SELECT job_id, salary FROM employees
WHERE  employee_id = 106;

COMMIT;
```

```
Script Output
Task completed in 0.038 seconds
EMPLOYEE_ID START_DATE END_DATE  JOB_ID     DEPARTMENT_ID
----------- ---------- ---------- ---------- -------------
        106 05-FEB-06  21-NOV-12 IT_PROG              60

JOB_ID       SALARY
---------- --------
SY_ANAL        6500

commited.
```

5. Re-enable the triggers on the `EMPLOYEES`, `JOBS` and `JOB_HISTORY` tables.

Uncomment and select the code under Task 5 of Additional Practice 1-3. The code and the results are displayed as follows:

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE jobs ENABLE ALL TRIGGERS;
ALTER TABLE job_history ENABLE ALL TRIGGERS;
```

Script Output

Task completed in 0.082 seconds

```
table EMPLOYEES altered.
table JOBS altered.
table JOB_HISTORY altered.
```

Practice 10-4: Updating the Minimum and Maximum Salaries for a Job

Overview

In this Additional Practice, you create a program to update the minimum and maximum salaries for a job in the `JOBS` table.

Tasks

1. Create a stored procedure called `UPD_JOBSAL` to update the minimum and maximum salaries for a specific job ID in the `JOBS` table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the `JOBS` table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the `JOBS` table is locked.

   Hint: The resource locked/busy error number is –54.
2. Enable `SERVEROUTPUT`, and then execute the `UPD_JOBSAL` procedure by using a job ID of '`SY_ANAL`', a minimum salary of 7000 and a maximum salary of 140.

   Note: This should generate an exception message.
3. Disable triggers on the `EMPLOYEES` and JOBS tables.
4. Execute the `UPD_JOBSAL` procedure using a job ID of '`SY_ANAL`', a minimum salary of 7000, and a maximum salary of 14000.
5. Query the `JOBS` table to view your changes, and then commit the changes.
6. Enable the triggers on the `EMPLOYEES` and `JOBS` tables.

Solution 10-4: Updating the Minimum and Maximum Salaries for a Job

In this Additional Practice, you create a program to update the minimum and maximum salaries for a job in the JOBS table.

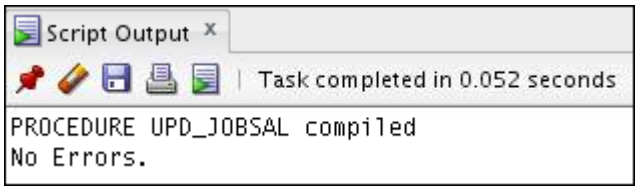
1. Create a stored procedure called UPD_JOBSAL to update the minimum and maximum salaries for a specific job ID in the JOBS table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the JOBS table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the JOBS table is locked.

   Hint: The resource locked/busy error number is –54.

   Uncomment and select the code under Task 1 of Additional Practice 1-4. The code and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE upd_jobsal(
  p_jobid       IN jobs.job_id%type,
  p_new_minsal  IN jobs.min_salary%type,
  p_new_maxsal  IN jobs.max_salary%type) IS
  v_dummy            PLS_INTEGER;
  e_resource_busy  EXCEPTION;
  e_sal_error        EXCEPTION;
  PRAGMA           EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
  IF (p_new_maxsal < p_new_minsal) THEN
    RAISE e_sal_error;
  END IF;
  SELECT 1 INTO v_dummy
    FROM jobs
    WHERE job_id = p_jobid
    FOR UPDATE OF min_salary NOWAIT;
  UPDATE jobs
    SET min_salary =  p_new_minsal,
        max_salary =  p_new_maxsal
    WHERE job_id  = p_jobid;
EXCEPTION
  WHEN e_resource_busy THEN
    RAISE_APPLICATION_ERROR (-20001,
      'Job information is currently locked, try later.');
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20001, 'This job ID does not
exist');
```

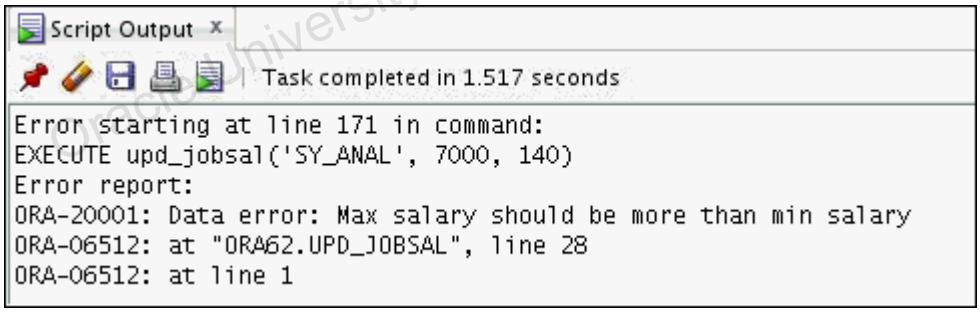
```
  WHEN e_sal_error THEN
    RAISE_APPLICATION_ERROR(-20001,
     'Data error: Max salary should be more than min salary');
END upd_jobsal;
/
SHOW ERRORS
```



2. Enable `SERVEROUTPUT`, and then execute the `UPD_JOBSAL` procedure by using a job ID of '`SY_ANAL`', a minimum salary of 7000 and a maximum salary of 140.
   Note: This should generate an exception message.

   Uncomment and select the code under Task 2 of Additional Practice 1-4. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE upd_jobsal('SY_ANAL', 7000, 140)
```



3. Disable triggers on the `EMPLOYEES` and `JOBS` tables.

   Uncomment and select the code under Task 3 of Additional Practice 1-4. The code and the results are displayed as follows:

```
ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;
```

```
Script Output
Task completed in 0.03 seconds
table EMPLOYEES altered.
table JOBS altered.
```

4. Execute the `UPD_JOBSAL` procedure using a job ID of '`SY_ANAL`', a minimum salary of 7000, and a maximum salary of 14000.

Uncomment and select the code under Task 4 of Additional Practice 1-4. The code and the results are displayed as follows:

```
EXECUTE upd_jobsal('SY_ANAL', 7000, 14000)
```

```
Script Output
Task completed in 1.912 seconds
anonymous block completed
```

5. Query the `JOBS` table to view your changes, and then commit the changes.

Uncomment and select the code under Task 5 of Additional Practice 1-4. The code and the results are displayed as follows:

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

```
Script Output
Task completed in 0.003 seconds
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
SY_ANAL    System Analyst                           7000      14000
```

6. Enable the triggers on the `EMPLOYEES` and `JOBS` tables.

Uncomment and select the code under Task 6 of Additional Practice 1-4. The code and the results are displayed as follows:

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE jobs ENABLE ALL TRIGGERS;
```

Script Output

Task completed in 0.016 seconds

```
table EMPLOYEES altered.
table JOBS altered.
```

Practice 10-5: Monitoring Employees Salaries

Overview

In this Additional Practice, you create a procedure to monitor whether employees have exceeded their average salaries for their job type.

Tasks

1. Disable the `SECURE_EMPLOYEES` trigger.
2. In the `EMPLOYEES` table, add an `EXCEED_AVGSAL` column to store up to three characters and a default value of `NO`. Use a check constraint to allow the values `YES` or `NO`.
3. Create a stored procedure called `CHECK_AVGSAL` that checks whether each employee's salary exceeds the average salary for the `JOB_ID`.
   a. The average salary for a job is calculated from the information in the `JOBS` table.
   b. If the employee's salary exceeds the average for his or her job, then update the `EXCEED_AVGSAL` column in the `EMPLOYEES` table to a value of `YES`; otherwise, set the value to `NO`.
   c. Use a cursor to select the employee's rows using the `FOR UPDATE` option in the query.
   d. Add exception handling to account for a record being locked.
      Hint: The resource locked/busy error number is `-54`.
   e. Write and use a local function called `GET_JOB_AVGSAL` to determine the average salary for a job ID specified as a parameter.
4. Execute the `CHECK_AVGSAL` procedure. To view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary and the `exceed_avgsal` indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

   Note: These exercises can be used for extra practice when discussing how to create functions.

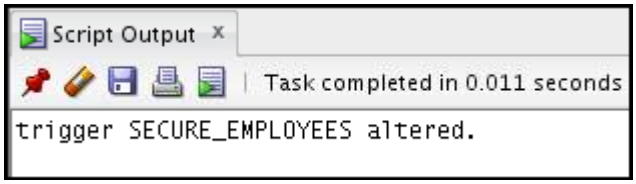
Solution 10-5: Monitoring Employees Salaries

In this practice, you create a procedure to monitor whether employees have exceeded their average salaries for their job type.

1. Disable the `SECURE_EMPLOYEES` trigger.

   Uncomment and select the code under Task 1 of Additional Practice 1-5. The code and the results are displayed as follows:

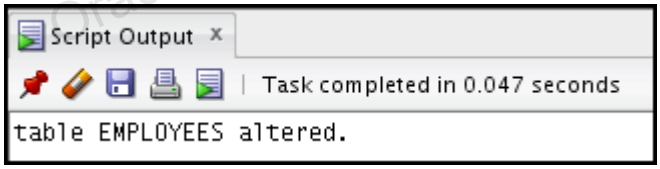
```
ALTER TRIGGER secure_employees DISABLE;
```



2. In the `EMPLOYEES` table, add an `EXCEED_AVGSAL` column to store up to three characters and a default value of `NO`. Use a check constraint to allow the values `YES` or `NO`.

   Uncomment and select the code under Task 2 of Additional Practice 1-5. The code and the results are displayed as follows:

```
ALTER TABLE employees ADD (exceed_avgsal VARCHAR2(3) DEFAULT
'NO'
    CONSTRAINT employees_exceed_avgsal_ck
    CHECK (exceed_avgsal IN ('YES', 'NO')));
```



3. Create a stored procedure called `CHECK_AVGSAL` that checks whether each employee's salary exceeds the average salary for the `JOB_ID`.
   a. The average salary for a job is calculated from the information in the `JOBS` table.
   b. If the employee's salary exceeds the average for his or her job, then update the `EXCEED_AVGSAL` column in the `EMPLOYEES` table to a value of `YES`; otherwise, set the value to `NO`.
   c. Use a cursor to select the employee's rows using the `FOR UPDATE` option in the query.
   d. Add exception handling to account for a record being locked.
      Hint: The resource locked/busy error number is `-54`.
   e. Write and use a local function called `GET_JOB_AVGSAL` to determine the average salary for a job ID specified as a parameter.

Uncomment and select the code under Task 3 of Additional Practice 1-5. The code and the results are displayed as follows::

```
CREATE OR REPLACE PROCEDURE check_avgsal IS
  emp_exceed_avgsal_type employees.exceed_avgsal%type;
  CURSOR  c_emp_csr IS
    SELECT  employee_id, job_id, salary
    FROM employees
    FOR UPDATE;
  e_resource_busy  EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_resource_busy, -54);
  FUNCTION get_job_avgsal (jobid VARCHAR2) RETURN NUMBER IS
    avg_sal employees.salary%type;
  BEGIN
    SELECT (max_salary + min_salary)/2 INTO avg_sal
    FROM jobs
    WHERE job_id = jobid;
    RETURN avg_sal;
  END;

BEGIN
  FOR emprec IN c_emp_csr
  LOOP
    emp_exceed_avgsal_type := 'NO';
    IF emprec.salary >= get_job_avgsal(emprec.job_id) THEN
      emp_exceed_avgsal_type := 'YES';
    END IF;
    UPDATE employees
      SET exceed_avgsal = emp_exceed_avgsal_type
      WHERE CURRENT OF c_emp_csr;
  END LOOP;
EXCEPTION
  WHEN e_resource_busy THEN
    ROLLBACK;
    RAISE_APPLICATION_ERROR (-20001, 'Record is busy, try
later.');
END check_avgsal;
/
SHOW ERRORS
```

```
Script Output
Task completed in 0.053 seconds
PROCEDURE CHECK_AVGSAL compiled
No Errors.
```

4. Execute the `CHECK_AVGSAL` procedure. To view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary and the `exceed_avgsal` indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

   Note: These exercises can be used for extra practice when discussing how to create functions.

   Uncomment and select the code under Task 4 of Additional Practice 1-5. The code and the results are displayed as follows:

```
EXECUTE check_avgsal

SELECT e.employee_id, e.job_id, (j.max_salary-j.min_salary/2)
job_avgsal,
       e.salary, e.exceed_avgsal avg_exceeded
FROM   employees e, jobs j
WHERE  e.job_id = j.job_id
and e.exceed_avgsal = 'YES';

COMMIT;
```

Script Output

Task completed in 0.035 seconds

```
anonymous block completed
EMPLOYEE_ID JOB_ID     JOB_AVGSAL   SALARY AVG_EXCEEDED
----------- ---------- ---------- -------- ------------
        113 FI_ACCOUNT       6900     6900 YES
        112 FI_ACCOUNT       6900     7800 YES
        111 FI_ACCOUNT       6900     7700 YES
        110 FI_ACCOUNT       6900     8200 YES
        109 FI_ACCOUNT       6900     9000 YES
        206 AC_ACCOUNT       6900     8300 YES
        174 SA_REP           9008    11000 YES
        170 SA_REP           9008     9600 YES
        169 SA_REP           9008    10000 YES
        168 SA_REP           9008    11500 YES
        163 SA_REP           9008     9500 YES
        162 SA_REP           9008    10500 YES
        157 SA_REP           9008     9500 YES
        156 SA_REP           9008    10000 YES
        151 SA_REP           9008     9500 YES
        150 SA_REP           9008    10000 YES
        122 ST_MAN           5750     7900 YES
        121 ST_MAN           5750     8200 YES
        120 ST_MAN           5750     8000 YES
        137 ST_CLERK         3996     3600 YES
        192 SH_CLERK         4250     4000 YES
        185 SH_CLERK         4250     4100 YES
        184 SH_CLERK         4250     4200 YES
        103 IT_PROG          8000     9000 YES
        201 MK_MAN          10500    13000 YES
        203 HR_REP           7000     6500 YES
        204 PR_REP           8250    10000 YES

 27 rows selected

commited.
```

Practice 10-6: Retrieving the Total Number of Years of Service for an Employee

Overview

In this practice, you create a subprogram to retrieve the number of years of service for a specific employee.

Tasks

1. Create a stored function called `GET_YEARS_SERVICE` to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.
2. Invoke the `GET_YEARS_SERVICE` function in a call to `DBMS_OUTPUT.PUT_LINE` for an employee with ID 999.
3. Display the number of years of service for employee 106 with `DBMS_OUTPUT.PUT_LINE` invoking the `GET_YEARS_SERVICE` function. Make sure that you enable `SERVEROUTPUT`.
4. Query the `JOB_HISTORY` and `EMPLOYEES` tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those that you get when you run these queries.

Solution 10-6: Retrieving the Total Number of Years of Service for an Employee

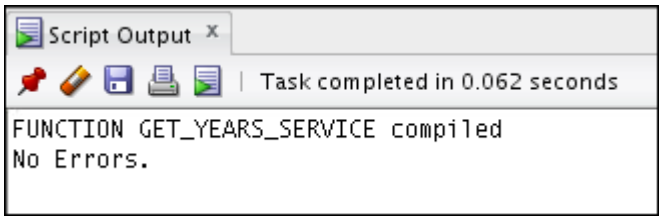
In this practice, you create a subprogram to retrieve the number of years of service for a specific employee.

1. Create a stored function called `GET_YEARS_SERVICE` to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.

Uncomment and select the code under Task 1 of Additional Practice 1-6. The code and the results are displayed as follows:

```
CREATE OR REPLACE FUNCTION get_years_service(
  p_emp_empid_type IN employees.employee_id%TYPE) RETURN NUMBER
IS
  CURSOR c_jobh_csr IS
    SELECT MONTHS_BETWEEN(end_date, start_date)/12
v_years_in_job
    FROM   job_history
    WHERE  employee_id = p_emp_empid_type;
  v_years_service NUMBER(2) := 0;
  v_years_in_job  NUMBER(2) := 0;
BEGIN
  FOR jobh_rec IN c_jobh_csr
  LOOP
    EXIT WHEN c_jobh_csr%NOTFOUND;
    v_years_service := v_years_service +
jobh_rec.v_years_in_job;
  END LOOP;
  SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO
v_years_in_job
  FROM   employees
  WHERE  employee_id = p_emp_empid_type;
  v_years_service := v_years_service + v_years_in_job;
  RETURN ROUND(v_years_service);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_empid_type ||' does not
exist.');
    RETURN NULL;
END get_years_service;
```

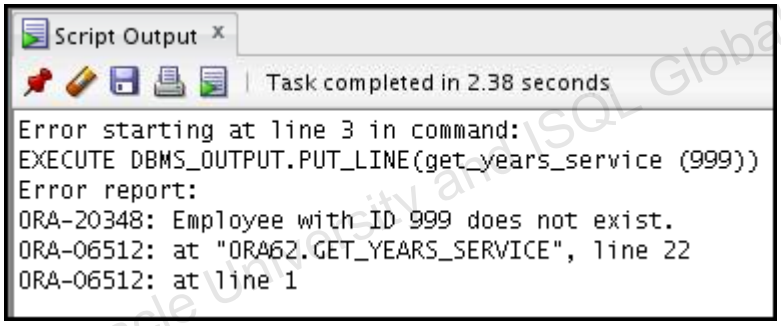
```
/
SHOW ERRORS
```



2. Invoke the `GET_YEARS_SERVICE` function in a call to `DBMS_OUTPUT.PUT_LINE` for an employee with ID 999.

Uncomment and select the code under Task 2 of Additional Practice 1-6. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON
EXECUTE DBMS_OUTPUT.PUT_LINE(get_years_service (999))
```



3. Display the number of years of service for employee 106 with `DBMS_OUTPUT.PUT_LINE` invoking the `GET_YEARS_SERVICE` function. Make sure that you enable `SERVEROUTPUT`.

Uncomment and select the code under Task 1 of Additional Practice 1-6. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

BEGIN
  DBMS_OUTPUT.PUT_LINE (
    'Employee 106 has worked ' || get_years_service(106) || '
years');
END;
/
```

```
Script Output
Task completed in 0.005 seconds
anonymous block completed
Employee 106 has worked 7 years
```

4. Query the `JOB_HISTORY` and `EMPLOYEES` tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those you get when you run these queries.

Uncomment and select the code under Task 4 of Additional Practice 1-6. The code and the results are displayed as follows:

```
SELECT employee_id, job_id,
       MONTHS_BETWEEN(end_date, start_date)/12 duration
FROM   job_history;

SELECT job_id, MONTHS_BETWEEN(SYSDATE, hire_date)/12 duration
FROM   employees
WHERE  employee_id = 106;
```

```
Script Output
Task completed in 0.008 seconds
EMPLOYEE_ID JOB_ID     DURATION
----------- ---------- --------
        102 IT_PROG    5.52956989247311827956989247311827956989
        101 AC_ACCOUNT 4.09946236559139784946236559139784946237
        101 AC_MGR     3.38172043010752688172043010752688172043
        201 MK_REP     3.83870967741935483870967741935483870968
        114 ST_CLERK   1.76881720430107526881720430107526881 72
        122 ST_CLERK   0.99731182795698924731182795698924731 18283
        200 AD_ASST        5.75
        176 SA_REP     0.76881720430107526881720430107526881 72042
        176 SA_MAN     0.99731182795698924731182795698924731 18283
        200 AC_ACCOUNT 4.49731182795698924731182795698924731183
        106 IT_PROG    6.79549258263639984070091596973317403425

 11 rows selected

JOB_ID     DURATION
---------- --------
SY_ANAL           0
```

Practice 10-7: Retrieving the Total Number of Different Jobs for an Employee

Overview

In this practice, you create a program to retrieve the number of different jobs that an employee worked on during his or her service.

Tasks

1. Create a stored function called `GET_JOB_COUNT` to retrieve the total number of different jobs on which an employee worked.
   a. The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job.
   b. Add exception handling to account for an invalid employee ID.
      Hint: Use the distinct job IDs from the `JOB_HISTORY` table, and exclude the current job ID, if it is one of the job IDs on which the employee has already worked.
   c. Write a `UNION` of two queries and count the rows retrieved into a PL/SQL table.
   d. Use a `FETCH` with `BULK COLLECT INTO` to obtain the unique jobs for the employee.
2. Invoke the function for the employee with the ID of 176. Make sure that you enable `SERVEROUTPUT`.

   Note: These exercises can be used for extra practice when discussing how to create packages.

Solution 10-7: Retrieving the Total Number of Different Jobs for an Employee

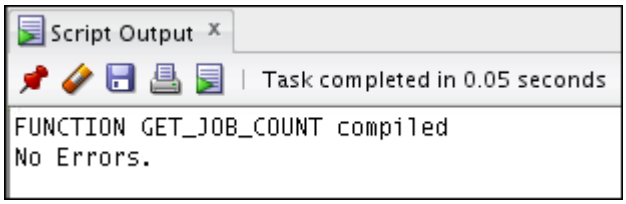
In this practice, you create a program to retrieve the number of different jobs that an employee worked on during his or her service.

1. Create a stored function called `GET_JOB_COUNT` to retrieve the total number of different jobs on which an employee worked.
   a. The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job.
   b. Add exception handling to account for an invalid employee ID.
      Hint: Use the distinct job IDs from the `JOB_HISTORY` table, and exclude the current job ID if it is one of the job IDs on which the employee has already worked.
   c. Write a `UNION` of two queries and count the rows retrieved into a PL/SQL table.
   d. Use a `FETCH` with `BULK COLLECT INTO` to obtain the unique jobs for the employee.

   Uncomment and select the code under Task 1 of Additional Practice 1-7. The code and the results are displayed as follows:

```
CREATE OR REPLACE FUNCTION get_job_count(
  p_emp_empid_type IN employees.employee_id%TYPE) RETURN NUMBER
IS
  TYPE jobs_table_type IS TABLE OF jobs.job_id%type;
  v_jobtab jobs_table_type;
  CURSOR c_empjob_csr IS
    SELECT job_id
    FROM job_history
    WHERE employee_id = p_emp_empid_type
      UNION
    SELECT job_id
    FROM employees
    WHERE employee_id = p_emp_empid_type;
BEGIN
  OPEN c_empjob_csr;
  FETCH c_empjob_csr BULK COLLECT INTO v_jobtab;
  CLOSE c_empjob_csr;
IF (v_jobtab.count = 0) THEN
  RAISE NO_DATA_FOUND;
ELSE
  RETURN v_jobtab.count;
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
```

```
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_empid_type ||' does not
exist!');
    RETURN NULL;
END get_job_count;
/
SHOW ERRORS
```



2. Invoke the function for the employees with the ID of 176 and 16. Make sure that you enable `SERVEROUTPUT`.

   Note: These exercises can be used for extra practice when discussing how to create packages.

   Uncomment and select the code under Task 2 of Additional Practice 1-7. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

BEGIN
   DBMS_OUTPUT.PUT_LINE('Employee 176 worked on ' ||
     get_job_count(176) || ' different jobs.');
  DBMS_OUTPUT.PUT_LINE('Employee 16 worked on ' ||
     get_job_count(16) || ' different jobs.');
END;
/
```

Script Output

Task completed in 0.011 seconds

```
Error starting at line : 408 in command -
BEGIN
   DBMS_OUTPUT.PUT_LINE('Employee 176 worked on ' ||
      get_job_count(176) || ' different jobs.');
  DBMS_OUTPUT.PUT_LINE('Employee 16 worked on ' ||
      get_job_count(16) || ' different jobs.');
END;
Error report -
ORA-20348: Employee with ID 16 does not exist!
ORA-06512: at "ORA62.GET_JOB_COUNT", line 25
ORA-06512: at line 4

Employee 176 worked on 2 different jobs.
```

Practice 10-8: Creating a New Package that Contains the Newly Created Procedures and Functions

Overview

In this practice, you create a package called `EMPJOB_PKG` that contains your `NEW_JOB`, `ADD_JOB_HIST`, `UPD_JOBSAL` procedures, as well as your `GET_YEARS_SERVICE` and `GET_JOB_COUNT` functions.

Tasks

1. Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.
2. Create the package body with the subprogram implementation; remember to remove, from the subprogram implementations, any types that you moved into the package specification.
3. Invoke your `EMPJOB_PKG.NEW_JOB` procedure to create a new job with the ID `PR_MAN`, the job title Public Relations Manager, and the salary 6250. Make sure that you enable `SERVEROUTPUT`.
4. Invoke your `EMPJOB_PKG.ADD_JOB_HIST` procedure to modify the job of employee ID 110 to job ID `PR_MAN`.

   Note: You need to disable the `UPDATE_JOB_HISTORY` trigger before you execute the `ADD_JOB_HIST` procedure, and re-enable the trigger after you have executed the procedure.
5. Query the `JOBS`, `JOB_HISTORY`, and `EMPLOYEES` tables to verify the results.

   Note: These exercises can be used for extra practice when discussing how to create database triggers.

Solution 10-8: Creating a New Package that Contains the Newly Created Procedures and Functions

In this practice, you create a package called `EMPJOB_PKG` that contains your `NEW_JOB`, `ADD_JOB_HIST`, `UPD_JOBSAL` procedures, as well as your `GET_YEARS_SERVICE` and `GET_JOB_COUNT` functions.

1. Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.

   Uncomment and select the code under Task 1 of Additional Practice 1-8. The code and the results are displayed as follows:

```
CREATE OR REPLACE PACKAGE empjob_pkg IS
  TYPE jobs_table_type IS TABLE OF jobs.job_id%type;

  PROCEDURE add_job_hist(
     p_emp_id IN employees.employee_id%TYPE,
     p_new_jobid IN jobs.job_id%TYPE);

  FUNCTION get_job_count(
     p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

  FUNCTION get_years_service(
     p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

  PROCEDURE new_job(
    p_jobid IN jobs.job_id%TYPE,
    p_title IN jobs.job_title%TYPE,
    p_minsal IN jobs.min_salary%TYPE);

  PROCEDURE upd_jobsal(
    p_jobid IN jobs.job_id%type,
    p_new_minsal IN jobs.min_salary%type,
    p_new_maxsal IN jobs.max_salary%type);
END empjob_pkg;
/
SHOW ERRORS
```

```
Script Output
Task completed in 0.074 seconds
PACKAGE EMPJOB_PKG compiled
No Errors.
```

2. Create the package body with the subprogram implementation; remember to remove from the subprogram implementations any types that you moved into the package specification.

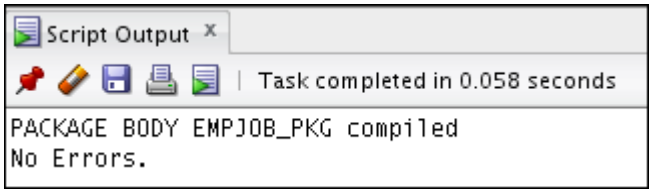
Uncomment and select the code under Task 2 of Additional Practice 1-8. The code and the results are displayed as follows:

```
CREATE OR REPLACE PACKAGE BODY empjob_pkg IS
  PROCEDURE add_job_hist(
    p_emp_id IN employees.employee_id%TYPE,
    p_new_jobid IN jobs.job_id%TYPE) IS
  BEGIN
    INSERT INTO job_history
      SELECT employee_id, hire_date, SYSDATE, job_id,
department_id
      FROM employees
      WHERE employee_id = p_emp_id;
    UPDATE employees
      SET hire_date = SYSDATE,
          job_id = p_new_jobid,
          salary = (SELECT min_salary + 500
                    FROM jobs
          WHERE job_id = p_new_jobid)
      WHERE employee_id = p_emp_id;
    DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
       ' details to the JOB_HISTORY table');
    DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
       p_emp_id|| ' to '|| p_new_jobid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR (-20001, 'Employee does not
exist!');
  END add_job_hist;

  FUNCTION get_job_count(
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
    v_jobtab jobs_table_type;
    CURSOR c_empjob_csr IS
      SELECT job_id
```

```
      FROM job_history
      WHERE employee_id = p_emp_id
      UNION
      SELECT job_id
      FROM employees
      WHERE employee_id = p_emp_id;
  BEGIN
    OPEN c_empjob_csr;
    FETCH c_empjob_csr BULK COLLECT INTO v_jobtab;
    CLOSE c_empjob_csr;
    RETURN v_jobtab.count;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20348,
        'Employee with ID '|| p_emp_id ||' does not exist!');
      RETURN 0;
  END get_job_count;

  FUNCTION get_years_service(
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
    CURSOR c_jobh_csr IS
      SELECT MONTHS_BETWEEN(end_date, start_date)/12
v_years_in_job
      FROM job_history
      WHERE employee_id = p_emp_id;
    v_years_service NUMBER(2) := 0;
    v_years_in_job NUMBER(2) := 0;
  BEGIN
    FOR jobh_rec IN c_jobh_csr
    LOOP
      EXIT WHEN c_jobh_csr%NOTFOUND;
      v_years_service := v_years_service +
jobh_rec.v_years_in_job;
    END LOOP;
    SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO
v_years_in_job
    FROM employees
    WHERE employee_id = p_emp_id;
    v_years_service := v_years_service + v_years_in_job;
    RETURN ROUND(v_years_service);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20348,
```

```
      'Employee with ID '|| p_emp_id ||' does not exist.');
    RETURN 0;
END get_years_service;

PROCEDURE new_job(
  p_jobid IN jobs.job_id%TYPE,
  p_title IN jobs.job_title%TYPE,
  p_minsal IN jobs.min_salary%TYPE) IS
  v_maxsal jobs.max_salary%TYPE := 2 * p_minsal;
BEGIN
  INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
  VALUES (p_jobid, p_title, p_minsal, v_maxsal);
  DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
  DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_title ||' '||
                        p_minsal || ' ' || v_maxsal);
END new_job;

PROCEDURE upd_jobsal(
  p_jobid IN jobs.job_id%type,
  p_new_minsal IN jobs.min_salary%type,
  p_new_maxsal IN jobs.max_salary%type) IS
  v_dummy PLS_INTEGER;
  e_resource_busy EXCEPTION;
  e_sal_error EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
  IF (p_new_maxsal < p_new_minsal) THEN
    RAISE e_sal_error;
  END IF;
  SELECT 1 INTO v_dummy
  FROM jobs
  WHERE job_id = p_jobid
  FOR UPDATE OF min_salary NOWAIT;
  UPDATE jobs
    SET min_salary = p_new_minsal,
        max_salary = p_new_maxsal
  WHERE job_id = p_jobid;
EXCEPTION
  WHEN e_resource_busy THEN
    RAISE_APPLICATION_ERROR (-20001,
      'Job information is currently locked, try later.');
  WHEN NO_DATA_FOUND THEN
```

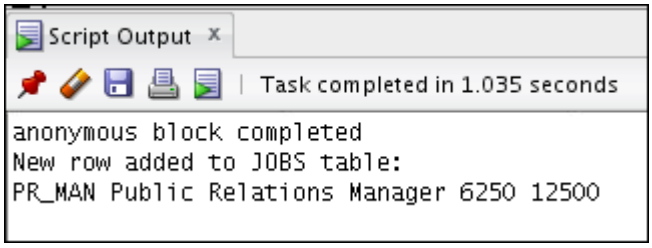
```
      RAISE_APPLICATION_ERROR(-20001, 'This job ID does not
exist');
    WHEN e_sal_error THEN
      RAISE_APPLICATION_ERROR(-20001,
        'Data error: Max salary should be more than min
salary');
  END upd_jobsal;
END empjob_pkg;
/
SHOW ERRORS
```



3. Invoke your `EMPJOB_PKG.NEW_JOB` procedure to create a new job with the ID `PR_MAN`, the job title Public Relations Manager, and the salary 6250. Make sure that you enable `SERVEROUTPUT`.

   Uncomment and select the code under Task 3 of Additional Practice 1-8. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE empjob_pkg.new_job('PR_MAN', 'Public Relations Manager',
6250)
```



4. Invoke your `EMPJOB_PKG.ADD_JOB_HIST` procedure to modify the job of employee ID 110 to job ID `PR_MAN`.
   Note: You need to disable the `UPDATE_JOB_HISTORY` trigger before you execute the `ADD_JOB_HIST` procedure, and re-enable the trigger after you have executed the procedure.

   Uncomment and select the code under Task 4 of Additional Practice 1-8. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON
ALTER TRIGGER update_job_history DISABLE;
EXECUTE empjob_pkg.add_job_hist(110, 'PR_MAN')
ALTER TRIGGER update_job_history ENABLE;
```

```
Script Output
Task completed in 0.02 seconds
trigger UPDATE_JOB_HISTORY altered.
anonymous block completed
Added employee 110 details to the JOB_HISTORY table
Updated current job of employee 110 to PR_MAN

trigger UPDATE_JOB_HISTORY altered.
```

5. Query the `JOBS`, `JOB_HISTORY`, and `EMPLOYEES` tables to verify the results.

   Note: These exercises can be used for extra practice when discussing how to create database triggers.

   Uncomment and select the code under Task 5 of Additional Practice 1-8. The code and the results are displayed as follows:

```
SELECT * FROM jobs WHERE job_id = 'PR_MAN';
SELECT * FROM job_history WHERE employee_id = 110;
SELECT job_id, salary FROM employees WHERE employee_id = 110;
```

```
Script Output
Task completed in 0.019 seconds
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
PR_MAN     Public Relations Manager                 6250      12500

EMPLOYEE_ID START_DATE END_DATE  JOB_ID     DEPARTMENT_ID
----------- ---------- --------- ---------- -------------
        110 28-SEP-05  22-NOV-12 FI_ACCOUNT           100

JOB_ID       SALARY
---------- --------
PR_MAN         6750
```

Practice 10-9: Creating a Trigger to Ensure that the Employees' Salaries Are Within the Acceptable Range

Overview

In this practice, you create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.

Tasks

1. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row that is updated in the `MIN_SALARY` and `MAX_SALARY` columns in the `JOBS` table.
   a. For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the `EMPLOYEES` table falls within the new range of salaries specified for this job ID.
   b. Include exception handling to cover a salary range change that affects the record of any existing employee.
2. Test the trigger using the `SY_ANAL` job, setting the new minimum salary to 5000, and the new maximum salary to 7000. Before you make the change, write a query to display the current salary range for the `SY_ANAL` job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the `JOBS` table for the specified job ID.
3. Using the `SY_ANAL` job, set the new minimum salary to 7,000, and the new maximum salary to 18000. Explain the results.

Solution 10-9: Creating a Trigger to Ensure that the Employees' Salaries are Within the Acceptable Range

In this practice, you create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.

1. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row that is updated in the `MIN_SALARY` and `MAX_SALARY` columns in the `JOBS` table.
   a. For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the `EMPLOYEES` table falls within the new range of salaries specified for this job ID.
   b. Include exception handling to cover a salary range change that affects the record of any existing employee.

   Uncomment and select the code under Task 1 of Additional Practice 1-9. The code and the results are displayed as follows:

```
CREATE OR REPLACE TRIGGER check_sal_range
BEFORE UPDATE OF min_salary, max_salary ON jobs
FOR EACH ROW
DECLARE
  v_minsal employees.salary%TYPE;
  v_maxsal employees.salary%TYPE;
  e_invalid_salrange EXCEPTION;
BEGIN
  SELECT MIN(salary), MAX(salary) INTO v_minsal, v_maxsal
  FROM employees
  WHERE job_id = :NEW.job_id;
  IF (v_minsal < :NEW.min_salary) OR (v_maxsal >
:NEW.max_salary) THEN
    RAISE e_invalid_salrange;
  END IF;
EXCEPTION
  WHEN e_invalid_salrange THEN
    RAISE_APPLICATION_ERROR(-20550,
     'Employees exist whose salary is out of the specified
range. '||
     'Therefore the specified salary range cannot be updated.');
END check_sal_range;
/
SHOW ERRORS
```

```
Script Output
Task completed in 0.063 seconds
TRIGGER CHECK_SAL_RANGE compiled
No Errors.
```

2. Test the trigger using the `SY_ANAL` job, setting the new minimum salary to 5000, and the new maximum salary to 7000. Before you make the change, write a query to display the current salary range for the `SY_ANAL` job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the `JOBS` table for the specified job ID.

Uncomment and select the code under Task 2 of Additional Practice 1-9. The code and the results are displayed as follows:

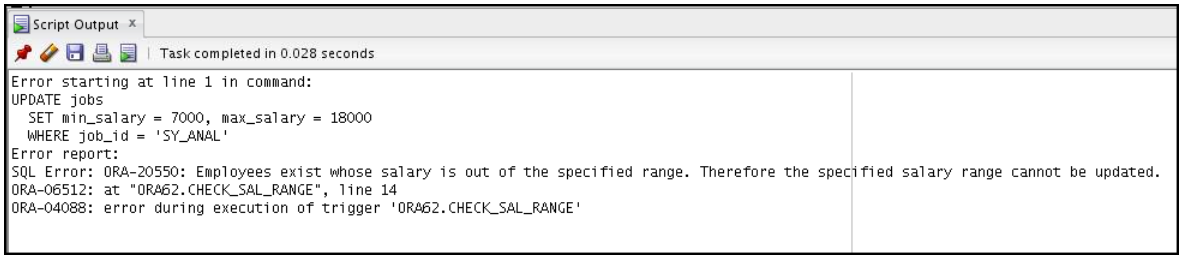
```
SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';

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

UPDATE jobs
  SET min_salary = 5000, max_salary = 7000
  WHERE job_id = 'SY_ANAL';

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

```
Script Output
Task completed in 0.018 seconds
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
SY_ANAL    System Analyst                           7000      14000

EMPLOYEE_ID LAST_NAME                   SALARY
----------- ------------------------- --------
        106 Pataballa                    6500

1 rows updated.
JOB_ID     JOB_TITLE                          MIN_SALARY MAX_SALARY
---------- ---------------------------------- ---------- ----------
SY_ANAL    System Analyst                           5000       7000
```

3. Using the `SY_ANAL` job, set the new minimum salary to 7,000, and the new maximum salary to 18000. Explain the results.

Uncomment and select the code under Task 3 of Additional Practice 1-9. The code and the results are displayed as follows:

```
UPDATE jobs
  SET min_salary = 7000, max_salary = 18000
  WHERE job_id = 'SY_ANAL';
```



The update fails to change the salary range due to the functionality provided by the `CHECK_SAL_RANGE` trigger because employee 106 who has the `SY_ANAL` job ID has a salary of `6500`, which is less than the minimum salary for the new salary range specified in the `UPDATE` statement.

Additional Practices 11

Chapter 11

Additional Practices 11

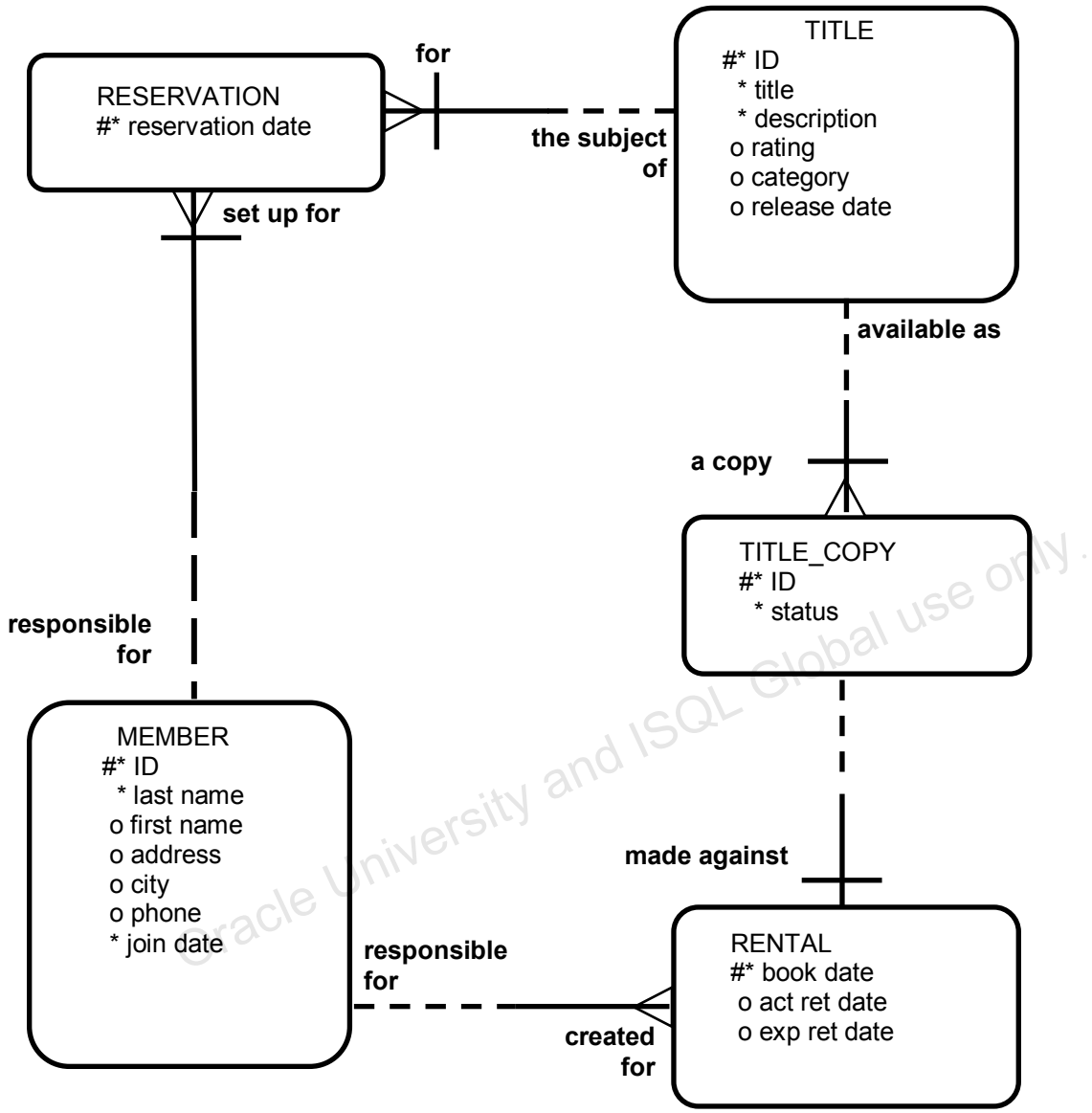
Overview

In this case study, you create a package named `VIDEO_PKG` that contains procedures and functions for a video store application. This application enables customers to become a member of the video store. Any member can rent movies, return rented movies, and reserve movies. Additionally, you create a trigger to ensure that any data in the video tables is modified only during business hours.

Create the package by using SQL*Plus and use the `DBMS_OUTPUT` Oracle-supplied package to display messages.

The video store database contains the following tables: `TITLE`, `TITLE_COPY`, `RENTAL`, `RESERVATION`, and `MEMBER`.

The video store database entity relationship diagram



Practice 11-1: Creating the `VIDEO_PKG` Package

Overview

In this practice, you create a package named `VIDEO_PKG` that contains procedures and functions for a video store application.

Task

1. Load and execute the `/home/oracle/labs/plpu/labs/buildvid1.sql` script to create all the required tables and sequences that are needed for this exercise.
2. Load and execute the `/home/oracle/labs/plpu/labs/buildvid2.sql` script to populate all the tables created through the buildvid1.sql script.
3. Create a package named `VIDEO_PKG` with the following procedures and functions:
   a. **`NEW_MEMBER`:** A public procedure that adds a new member to the `MEMBER` table. For the member ID number, use the sequence `MEMBER_ID_SEQ`; for the join date, use `SYSDATE`. Pass all other values to be inserted into a new row as parameters.
   b. **`NEW_RENTAL`:** An overloaded public function to record a new rental. Pass the title ID number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as `AVAILABLE` in the `TITLE_COPY` table for one copy of this title, then update this `TITLE_COPY` table and set the status to `RENTED`. If there is no copy available, the function must return `NULL`. Then, insert a new record into the `RENTAL` table identifying the booked date as today's date, the copy ID number, the member ID number, the title ID number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return `NULL`, and display a list of the customers' names that match and their ID numbers.
   c. **`RETURN_MOVIE`:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title ID, the copy ID, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the `RENTAL` table and set the actual return date to today's date. Update the status in the `TITLE_COPY` table based on the status parameter passed into the procedure.
   d. **`RESERVE_MOVIE`:** A private procedure that executes only if all the video copies requested in the `NEW_RENTAL` procedure have a status of `RENTED`. Pass the member ID number and the title ID number to this procedure. Insert a new record into the `RESERVATION` table and record the reservation date, member ID number, and title ID number. Print a message indicating that a movie is reserved and its expected date of return.
   e. **`EXCEPTION_HANDLER`:** A private procedure that is called from the exception handler of the public programs. Pass the `SQLCODE` number to this procedure, and the name of the program (as a text string) where the error occurred. Use `RAISE_APPLICATION_ERROR` to raise a customized error. Start with a unique key violation (-1) and foreign key violation (-2292). Allow the exception handler to raise a generic error for any other errors.
4. Use the following scripts located in the `/home/oracle/labs/plpu/soln` directory to test your routines:
   a. Add two members using the code under Task 4_a from `sol_ap2.sql` script.

b. Add new video rentals using the code under Task 4_b from `sol_ap2.sql` script.

c. Return movies using the code under Task 4_c from `sol_ap2.sql` script.

5. The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 12:00 PM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.

   a. Create a stored procedure called `TIME_CHECK` that checks the current time against business hours. If the current time is not within business hours, use the `RAISE_APPLICATION_ERROR` procedure to give an appropriate message.

   b. Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your `TIME_CHECK` procedure from each of these triggers.

   c. Test your triggers.

      Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

Solution 11-1: Creating the `VIDEO_PKG` Package

In this practice, you create a package named `VIDEO_PKG` that contains procedures and functions for a video store application.

1. Load and execute the `/home/oracle/labs/plpu/labs/buildvid1.sql` script to create all the required tables and sequences that are needed for this exercise.

   Run the `/home/oracle/labs/plpu/labs/buildvid1.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
SET ECHO OFF
/* Script to build the Video Application (Part 1 -
buildvid1.sql)
   for the Oracle Introduction to Oracle with Procedure Builder
course.
   Created by: Debby Kramer Creation date: 12/10/95
   Last updated: 11/21/12
   Modified by Supriya Ananth on 21-NOV-2012
    For the course Oracle Database: PL/SQL Program Units
    This part of the script creates tables and sequences that
are used
    by Task 4 of the Additional Practices of the course.
*/

DROP TABLE rental CASCADE CONSTRAINTS;
DROP TABLE reservation CASCADE CONSTRAINTS;
DROP TABLE title_copy CASCADE CONSTRAINTS;
DROP TABLE title CASCADE CONSTRAINTS;
DROP TABLE member CASCADE CONSTRAINTS;

PROMPT Please wait while tables are created....

CREATE TABLE MEMBER
  (member_id  NUMBER (10)        CONSTRAINT member_id_pk PRIMARY
KEY
 , last_name  VARCHAR2(25)
    CONSTRAINT member_last_nn NOT NULL
 , first_name VARCHAR2(25)
 , address    VARCHAR2(100)
 , city       VARCHAR2(30)
 , phone      VARCHAR2(25)
 , join_date  DATE DEFAULT SYSDATE
    CONSTRAINT join_date_nn NOT NULL)
```

```
/

CREATE TABLE TITLE
  (title_id   NUMBER(10)
     CONSTRAINT title_id_pk PRIMARY KEY
 , title      VARCHAR2(60)
     CONSTRAINT title_nn NOT NULL
 , description VARCHAR2(400)
     CONSTRAINT title_desc_nn NOT NULL
 , rating      VARCHAR2(4)
     CONSTRAINT title_rating_ck CHECK (rating IN
('G','PG','R','NC17','NR'))
 , category     VARCHAR2(20) DEFAULT 'DRAMA'
     CONSTRAINT title_categ_ck CHECK (category IN
('DRAMA','COMEDY','ACTION',
'CHILD','SCIFI','DOCUMENTARY'))
 , release_date DATE)
/

CREATE TABLE TITLE_COPY
  (copy_id   NUMBER(10)
 , title_id  NUMBER(10)
    CONSTRAINT copy_title_id_fk
       REFERENCES title(title_id)
 , status    VARCHAR2(15)
     CONSTRAINT copy_status_nn NOT NULL
     CONSTRAINT copy_status_ck CHECK (status IN ('AVAILABLE',
'DESTROYED',
                                 'RENTED', 'RESERVED'))
 , CONSTRAINT copy_title_id_pk  PRIMARY KEY(copy_id, title_id))
/

CREATE TABLE RENTAL
  (book_date DATE DEFAULT SYSDATE
 , copy_id   NUMBER(10)
 , member_id NUMBER(10)
    CONSTRAINT rental_mbr_id_fk REFERENCES member(member_id)
 , title_id  NUMBER(10)
 , act_ret_date DATE
 , exp_ret_date DATE DEFAULT SYSDATE+2
 , CONSTRAINT rental_copy_title_id_fk FOREIGN KEY (copy_id,
title_id)
              REFERENCES title_copy(copy_id,title_id)
```

```
 , CONSTRAINT rental_id_pk PRIMARY KEY(book_date, copy_id,
title_id, member_id))
/

CREATE TABLE RESERVATION
  (res_date  DATE
 , member_id NUMBER(10)
 , title_id  NUMBER(10)
 , CONSTRAINT res_id_pk PRIMARY KEY(res_date, member_id,
title_id))
/

PROMPT Tables created.

DROP SEQUENCE title_id_seq;
DROP SEQUENCE member_id_seq;

PROMPT Creating Sequences...

CREATE SEQUENCE member_id_seq
  START WITH 100
  NOCACHE
/

CREATE SEQUENCE title_id_seq
  START WITH 91
  NOCACHE
/

PROMPT Sequences created.

PROMPT Run buildvid2.sql now to populate the above tables.
```

Script Output

Task completed in 1.53 seconds

```
Error starting at line 12 in command:
DROP TABLE rental CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:

Error starting at line 13 in command:
DROP TABLE reservation CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:

Error starting at line 14 in command:
DROP TABLE title_copy CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:

Error starting at line 15 in command:
DROP TABLE title CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
*Cause:
*Action:

Error starting at line 16 in command:
DROP TABLE member CASCADE CONSTRAINTS
Error report:
SQL Error: ORA-00942: table or view does not exist
00942. 00000 -  "table or view does not exist"
```

```
*Cause:
*Action:
Please wait while tables are created....
table MEMBER created.
table TITLE created.
table TITLE_COPY created.
table RENTAL created.
table RESERVATION created.
Tables created.

Error starting at line 80 in command:
DROP SEQUENCE title_id_seq
Error report:
SQL Error: ORA-02289: sequence does not exist
02289. 00000 -  "sequence does not exist"
*Cause:    The specified sequence does not exist, or the user does
           not have the required privilege to perform this operation.
*Action:   Make sure the sequence name is correct, and that you have
           the right to perform the desired operation on this sequence.

Error starting at line 81 in command:
DROP SEQUENCE member_id_seq
Error report:
SQL Error: ORA-02289: sequence does not exist
02289. 00000 -  "sequence does not exist"
*Cause:    The specified sequence does not exist, or the user does
           not have the required privilege to perform this operation.
*Action:   Make sure the sequence name is correct, and that you have
           the right to perform the desired operation on this sequence.
Creating Sequences...
sequence MEMBER_ID_SEQ created.
sequence TITLE_ID_SEQ created.
Sequences created.
Run buildvid2.sql now to populate the above tables.
```

2. Load and execute the `/home/oracle/labs/plpu/labs/buildvid2.sql` script to populate all the tables created through the buildvid1.sql script.

   Run the `/home/oracle/labs/plpu/labs/buildvid2.sql` script. The code, the connection prompt, and the results are displayed as follows:

```
/* Script to build the Video Application (Part 2 -
buildvid2.sql)
   This part of the script populates the tables that are created
using buildvid1.sql
   These are used by Part B of the Additional Practices of the
course.
   You should run the script buildvid1.sql before running this
script to create the above tables.
*/
```

```
INSERT INTO member VALUES  (member_id_seq.NEXTVAL, 'Velasquez',
'Carmen', '283 King Street', 'Seattle', '587-99-6666', '03-MAR-
90');
INSERT INTO member  VALUES   (member_id_seq.NEXTVAL, 'Ngao',
'LaDoris', '5 Modrany',  'Bratislava', '586-355-8882', '08-MAR-
90');
INSERT INTO member VALUES  (member_id_seq.NEXTVAL,'Nagayama',
'Midori', '68 Via Centrale', 'Sao Paolo', '254-852-5764', '17-
JUN-91');
INSERT INTO member VALUES  (member_id_seq.NEXTVAL,'Quick-To-
See','Mark', '6921 King Way', 'Lagos', '63-559-777', '07-APR-
90');
INSERT INTO member VALUES  (member_id_seq.NEXTVAL, 'Ropeburn',
'Audry', '86 Chu Street',  'Hong Kong', '41-559-87', '04-MAR-
90');
INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Urguhart',
'Molly', '3035 Laurier Blvd.',  'Quebec', '418-542-9988','18-
JAN-91');
INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Menchu',
'Roberta', 'Boulevard de Waterloo 41', 'Brussels', '322-504-
2228', '14-MAY-90');
INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Biri', 'Ben',
'398 High St.', 'Columbus', '614-455-9863', '07-APR-90');
INSERT INTO member VALUES (member_id_seq.NEXTVAL, 'Catchpole',
'Antoinette', '88 Alfred St.', 'Brisbane', '616-399-1411', '09-
FEB-92');

COMMIT;


INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Willie and Christmas Too', 'All
of Willie''s friends made a Christmas list for Santa, but Willie
has yet to create his own wish list.', 'G', 'CHILD', '05-OCT-
95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Alien Again', 'Another
installment of science fiction history. Can the heroine save the
planet from the alien life form?', 'R', 'SCIFI',          '19-
MAY-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'The Glob', 'A meteor crashes near
a small American town and unleashes carivorous goo in this
classic.', 'NR', 'SCIFI', '12-AUG-95');
```

```
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'My Day Off', 'With a little luck
and a lot of ingenuity, a teenager skips school for a day in New
York.', 'PG', 'COMEDY', '12-JUL-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Miracles on Ice', 'A six-year-old
has doubts about Santa Claus. But she discovers that miracles
really do exist.', 'PG', 'DRAMA', '12-SEP-95');
INSERT INTO TITLE (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL,  'Soda Gang', 'After discovering a
cached of drugs, a young couple find themselves pitted against a
vicious gang.', 'NR', 'ACTION', '01-JUN-95');
INSERT INTO title (title_id, title, description, rating,
category, release_date)
VALUES (TITLE_ID_SEQ.NEXTVAL, 'Interstellar Wars', 'Futuristic
interstellar action movie. Can the rebels save the humans from
the evil Empire?', 'PG', 'SCIFI','07-JUL-77');


COMMIT;

INSERT INTO title_copy VALUES (1,92, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,93, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,93, 'RENTED');
INSERT INTO title_copy VALUES (1,94, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (3,95, 'RENTED');
INSERT INTO title_copy VALUES (1,96, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,97, 'AVAILABLE');
COMMIT;

INSERT INTO reservation VALUES (sysdate-1, 101, 93);
INSERT INTO reservation VALUES (sysdate-2, 106, 102);


COMMIT;

INSERT INTO rental VALUES (sysdate-1, 2, 101, 93, null,
sysdate+1);
INSERT INTO rental VALUES (sysdate-2, 3, 102, 95, null,
sysdate);
```

```
INSERT INTO rental VALUES (sysdate-4, 1, 106, 97, sysdate-2,
sysdate-2);
INSERT INTO rental VALUES (sysdate-3, 1, 101, 92, sysdate-2,
sysdate-1);

COMMIT;

PROMPT ** Tables built and data loaded **
```

```
1 rows inserted.
1 rows inserted.
commited.
1 rows inserted.
1 rows inserted.
1 rows inserted.
1 rows inserted.
1 rows inserted.
commited.
** Tables built and data loaded **
```

3. Create a package named `VIDEO_PKG` with the following procedures and functions:
   a. **`NEW_MEMBER`:** A public procedure that adds a new member to the `MEMBER` table. For the member ID number, use the sequence `MEMBER_ID_SEQ`; for the join date, use `SYSDATE`. Pass all other values to be inserted into a new row as parameters.
   b. **`NEW_RENTAL`:** An overloaded public function to record a new rental. Pass the title ID number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as `AVAILABLE` in the `TITLE_COPY` table for one copy of this title, then update this `TITLE_COPY` table and set the status to `RENTED`. If there is no copy available, the function must return `NULL`. Then, insert a new record into the `RENTAL` table identifying the booked date as today's date, the copy ID number, the member ID number, the title ID number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return `NULL`, and display a list of the customers' names that match and their ID numbers.
   c. **`RETURN_MOVIE`:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title ID, the copy ID, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the `RENTAL` table and set the actual return date to today's date. Update the status in the `TITLE_COPY` table based on the status parameter passed into the procedure.
   d. **`RESERVE_MOVIE`:** A private procedure that executes only if all the video copies requested in the `NEW_RENTAL` procedure have a status of `RENTED`. Pass the member ID number and the title ID number to this procedure. Insert a new record into the `RESERVATION` table and record the reservation date, member ID number, and title ID number. Print a message indicating that a movie is reserved and its expected date of return.
   e. **`EXCEPTION_HANDLER`:** A private procedure that is called from the exception handler of the public programs. Pass the SQLCODE number to this procedure, and the name of the program (as a text string) where the error occurred. Use `RAISE_APPLICATION_ERROR` to raise a customized error. Start with a unique key violation (-1) and foreign key violation (-2292). Allow the exception handler to raise a generic error for any other errors.

Uncomment and run the code under Task 3 from `/home/oracle/labs/plpu/solns/sol_ap2.sql` script. The code, the connection prompt, and the results are displayed as follows:

VIDEO_PKG **Package Specification**

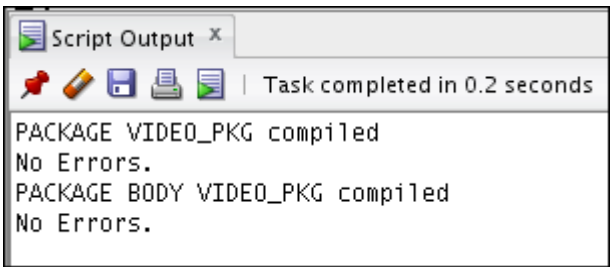
```
CREATE OR REPLACE PACKAGE video_pkg IS
  PROCEDURE new_member
    (p_lname       IN member.last_name%TYPE,
     p_fname       IN member.first_name%TYPE   DEFAULT NULL,
     p_address     IN member.address%TYPE      DEFAULT NULL,
     p_city        IN member.city%TYPE         DEFAULT NULL,
     p_phone       IN member.phone%TYPE        DEFAULT NULL);

  FUNCTION new_rental
    (p_memberid   IN rental.member_id%TYPE,
     p_titleid    IN rental.title_id%TYPE)
    RETURN DATE;

  FUNCTION new_rental
    (p_membername IN member.last_name%TYPE,
     p_titleid    IN rental.title_id%TYPE)
    RETURN DATE;

  PROCEDURE return_movie
    (p_titleid    IN rental.title_id%TYPE,
     p_copyid     IN rental.copy_id%TYPE,
     p_sts        IN title_copy.status%TYPE);
END video_pkg;
/
SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY video_pkg IS
  PROCEDURE exception_handler(errcode IN  NUMBER, p_context IN
VARCHAR2) IS
  BEGIN
    IF errcode = -1 THEN
      RAISE_APPLICATION_ERROR(-20001,
        'The number is assigned to this member is already in
use, '||
        'try again.');
    ELSIF errcode = -2291 THEN
      RAISE_APPLICATION_ERROR(-20002, p_context ||
        ' has attempted to use a foreign key value that is
invalid');
    ELSE
      RAISE_APPLICATION_ERROR(-20999, 'Unhandled error in ' ||
```

```
        p_context || '. Please contact your application '||
        'administrator with the following information: '
        || CHR(13) || SQLERRM);
    END IF;
  END exception_handler;

  PROCEDURE reserve_movie
    (p_memberid  IN  reservation.member_id%TYPE,
     p_titleid   IN  reservation.title_id%TYPE) IS
    CURSOR c_rented_csr IS
      SELECT exp_ret_date
        FROM rental
        WHERE title_id = p_titleid
        AND act_ret_date IS NULL;
  BEGIN
    INSERT INTO reservation (res_date, member_id, title_id)
    VALUES (SYSDATE, p_memberid, p_titleid);
    COMMIT;
    FOR rented_rec IN c_rented_csr LOOP
      DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on: '
        || rented_rec.exp_ret_date);
      EXIT WHEN c_rented_csr%found;
    END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'RESERVE_MOVIE');
  END reserve_movie;

PROCEDURE return_movie(
   p_titleid IN rental.title_id%TYPE,
   p_copyid IN rental.copy_id%TYPE,
   p_sts IN title_copy.status%TYPE) IS
   v_dummy VARCHAR2(1);
   CURSOR c_res_csr IS
      SELECT *
      FROM reservation
      WHERE title_id = p_titleid;
  BEGIN
    SELECT '' INTO v_dummy
      FROM title
      WHERE title_id = p_titleid;
    UPDATE rental
      SET act_ret_date = SYSDATE
```

```
    WHERE title_id = p_titleid
    AND copy_id = p_copyid AND act_ret_date IS NULL;
  UPDATE title_copy
    SET status = UPPER(p_sts)
    WHERE title_id = p_titleid AND copy_id = p_copyid;
  FOR res_rec IN c_res_csr LOOP
    IF c_res_csr%FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Put this movie on hold -- '||
        'reserved by member #' || res_rec.member_id);
    END IF;
  END LOOP;
EXCEPTION
  WHEN OTHERS THEN
    exception_handler(SQLCODE, 'RETURN_MOVIE');
END return_movie;

FUNCTION new_rental(
  p_memberid  IN  rental.member_id%TYPE,
  p_titleid   IN  rental.title_id%TYPE) RETURN DATE IS
  CURSOR c_copy_csr IS
    SELECT * FROM title_copy
    WHERE title_id = p_titleid
    FOR UPDATE;
  v_flag   BOOLEAN  := FALSE;
BEGIN
  FOR copy_rec IN c_copy_csr LOOP
    IF copy_rec.status = 'AVAILABLE' THEN
      UPDATE title_copy
        SET status = 'RENTED'
        WHERE CURRENT OF c_copy_csr;
      INSERT INTO rental(book_date, copy_id, member_id,
                         title_id, exp_ret_date)
      VALUES (SYSDATE, copy_rec.copy_id, p_memberid,
                         p_titleid, SYSDATE + 3);
      v_flag := TRUE;
      EXIT;
    END IF;
  END LOOP;
  COMMIT;
  IF v_flag THEN
    RETURN (SYSDATE + 3);
  ELSE
    reserve_movie(p_memberid, p_titleid);
```

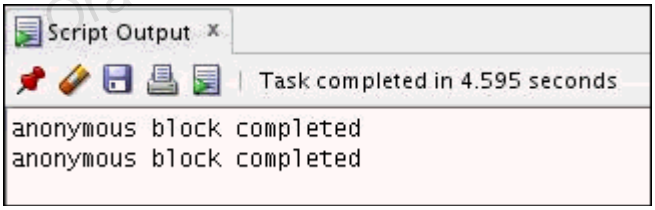
```
    RETURN NULL;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
    RETURN NULL;
END new_rental;

FUNCTION new_rental(
  p_membername IN member.last_name%TYPE,
  p_titleid    IN rental.title_id%TYPE) RETURN DATE IS
  CURSOR c_copy_csr IS
    SELECT * FROM title_copy
      WHERE title_id = p_titleid
      FOR UPDATE;
  v_flag  BOOLEAN  := FALSE;
  v_memberid  member.member_id%TYPE;
  CURSOR c_member_csr IS
    SELECT member_id, last_name, first_name
      FROM member
      WHERE LOWER(last_name) = LOWER(p_membername)
      ORDER BY last_name, first_name;
BEGIN
  SELECT member_id INTO v_memberid
    FROM member
    WHERE lower(last_name) = lower(p_membername);
  FOR copy_rec IN c_copy_csr LOOP
    IF copy_rec.status = 'AVAILABLE' THEN
      UPDATE title_copy
        SET status = 'RENTED'
        WHERE CURRENT OF c_copy_csr;
      INSERT INTO rental (book_date, copy_id, member_id,
                          title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, v_memberid,
                          p_titleid, SYSDATE + 3);
      v_flag := TRUE;
      EXIT;
    END IF;
  END LOOP;
  COMMIT;
  IF v_flag THEN
    RETURN(SYSDATE + 3);
  ELSE
```

```
      reserve_movie(v_memberid, p_titleid);
      RETURN NULL;
    END IF;
  EXCEPTION
    WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE(
       'Warning! More than one member by this name.');
      FOR member_rec IN c_member_csr LOOP
        DBMS_OUTPUT.PUT_LINE(member_rec.member_id || CHR(9) ||
          member_rec.last_name || ', ' ||
member_rec.first_name);
      END LOOP;
      RETURN NULL;
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_RENTAL');
      RETURN NULL;
  END new_rental;

  PROCEDURE new_member(
    p_lname       IN member.last_name%TYPE,
    p_fname       IN member.first_name%TYPE   DEFAULT NULL,
    p_address     IN member.address%TYPE      DEFAULT NULL,
    p_city        IN member.city%TYPE         DEFAULT NULL,
    p_phone       IN member.phone%TYPE        DEFAULT NULL) IS
  BEGIN
    INSERT INTO member(member_id, last_name, first_name,
                       address, city, phone, join_date)
      VALUES(member_id_seq.NEXTVAL, p_lname, p_fname,
             p_address, p_city, p_phone, SYSDATE);
    COMMIT;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_MEMBER');
  END new_member;
END video_pkg;
/
SHOW ERRORS
```



4. Use the following scripts located in the `/home/oracle/labs/plpu/soln` directory to test your routines. Make sure you enable `SERVEROUTPUT`:
   a. Add two members using the code under Task 4_a.
   Uncomment and run the code under Task 4_a. The code and the results are displayed as follows:

```
EXECUTE video_pkg.new_member('Haas', 'James', 'Chestnut Street',
'Boston', '617-123-4567')
EXECUTE  video_pkg.new_member('Biri', 'Allan',  'Hiawatha
Drive', 'New York', '516-123-4567')
```



   b. Add new video rentals using the code under Task 4_b.
   Uncomment and run the code under Task 4_b. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(110, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(109, 93))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(107, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental('Biri', 97))
```

```
Script Output
Task completed in 1.148 seconds
anonymous block completed


anonymous block completed
14-DEC-12

anonymous block completed


anonymous block completed
Warning! More than one member by this name.
110 Biri, Allan
107 Biri, Ben
```

c. Return movies using the code under Task 4_c.

Uncomment and run the code under Task 4_c. The code and the results are displayed as follows:

```
SET SERVEROUTPUT ON

EXECUTE video_pkg.return_movie(92, 3, 'AVAILABLE')
EXECUTE video_pkg.return_movie(95, 3, 'AVAILABLE')
EXECUTE video_pkg.return_movie(93, 1, 'RENTED')
```

```
Script Output
Task completed in 0.267 seconds
anonymous block completed
anonymous block completed
anonymous block completed
Put this movie on hold -- reserved by member #101
```

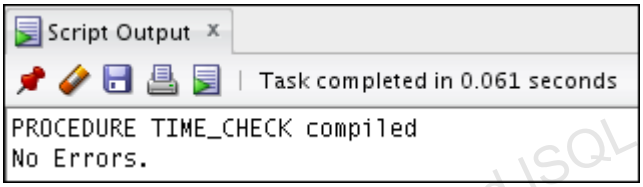
5. The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 12:00 PM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.

   a. Create a stored procedure called `TIME_CHECK` that checks the current time against business hours. If the current time is not within business hours, use the `RAISE_APPLICATION_ERROR` procedure to give an appropriate message.

   Uncomment and run the code under task 5_a. The code and the results are displayed as follows:

```
CREATE OR REPLACE PROCEDURE time_check IS
```

```
BEGIN
  IF ((TO_CHAR(SYSDATE,'D') BETWEEN 1 AND 6) AND
      (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT
BETWEEN
       TO_DATE('08:00', 'hh24:mi') AND TO_DATE('22:00',
'hh24:mi')))
       OR ((TO_CHAR(SYSDATE, 'D') = 7)
       AND  (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT
BETWEEN
       TO_DATE('08:00', 'hh24:mi') AND TO_DATE('24:00',
'hh24:mi'))) THEN
    RAISE_APPLICATION_ERROR(-20999,
       'Data changes restricted to office hours.');
  END IF;
END time_check;
/
SHOW ERRORS
```

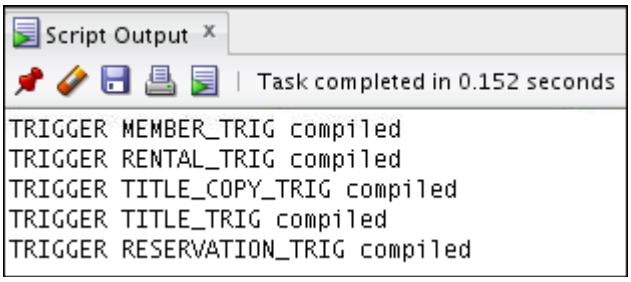


b. Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your `TIME_CHECK` procedure from each of these triggers.

Uncomment and run the code under Task 5_b. The code and the result are displayed as follows:

```
CREATE OR REPLACE TRIGGER member_trig
  BEFORE INSERT OR UPDATE OR DELETE ON member
CALL time_check
/

CREATE OR REPLACE TRIGGER rental_trig
  BEFORE INSERT OR UPDATE OR DELETE ON rental
CALL time_check
/

CREATE OR REPLACE TRIGGER title_copy_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title_copy
CALL time_check
/
```

```
CREATE OR REPLACE TRIGGER title_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title
CALL time_check
/
CREATE OR REPLACE TRIGGER reservation_trig
  BEFORE INSERT OR UPDATE OR DELETE ON reservation
CALL time_check
/
```



c. Test your triggers.

Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

Uncomment and run the code under Task 5_c. The code and the result are displayed as follows:

```
-- First determine current timezone and time
SELECT SESSIONTIMEZONE,
       TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY HH24:MI') CURR_DATE
FROM DUAL;

-- Change your time zone using [+|-]HH:MI format such that --
the current time returns a time between 6pm and 8am

ALTER SESSION SET TIME_ZONE='-07:00';

-- Add a new member (for a sample test)

EXECUTE  video_pkg.new_member('Elias', 'Elliane',  'Vine
Street', 'California', '789-123-4567')

BEGIN video_pkg.new_member('Elias', 'Elliane',  'Vine Street',
'California', '789-123-4567'); END;
```

```
-- Restore the original time zone for your session.
ALTER SESSION SET TIME_ZONE='-00:00';
```