
Introduction to Oracle9i: PL/SQL

Additional Practices

40054GC10
Production 1.0
June 2001
D32947

ORACLE®

Authors

Nagavalli Pataballa
Priya Nathan

Technical Contributors and Reviewers

Anna Atkinson
Bryan Roberts
Caroline Pereda
Cesljas Zarco
Chaya Rao
Coley William
Daniel Gabel
Dr. Christoph Burandt
Hakan Lindfors
Helen Robertson
John Hoff
Judy Brink
Lachlan Williams
Laszlo Czinkoczki
Laura Pezzini
Linda Boldt
Marco Verbeek
Natarajan Senthil
Priya Vennapusa
Robert Squires
Roger Abuzalaf
Ruediger Steffan
Sarah Jones
Stefan Lindblad
Sue Onraet
Susan Dee

Publisher

Sandya Krishna

Copyright © Oracle Corporation, 1999, 2000, 2001. All rights reserved.

This documentation contains proprietary information of Oracle Corporation. It is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited. If this documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions for commercial computer software and shall be deemed to be Restricted Rights software under Federal law, as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

This material or any portion of it may not be copied in any form or by any means without the express prior written permission of Oracle Corporation. Any other copying is a violation of copyright law and may result in civil and/or criminal penalties.

If this documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights," as defined in FAR 52.227-14, Rights in Data-General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them in writing to Education Products, Oracle Corporation, 500 Oracle Parkway, Box SB-6, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

All references to Oracle and Oracle products are trademarks or registered trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Additional Practices

Additional Practices Overview

These additional practices are provided as a supplement to the course *Introduction to Oracle9i: PL/SQL*. In these practices, you apply the concepts that you learned in *Introduction to Oracle9i: PL/SQL*.

The additional practices comprise of two parts:

Part A provides 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. In part A, you also create stored procedures, functions, packages, triggers, and use the Oracle-supplied packages with *iSQL*Plus* as the development environment. The tables used in this portion of the additional practices include EMPLOYEES, JOBS, JOB_HISTORY, and DEPARTMENTS.

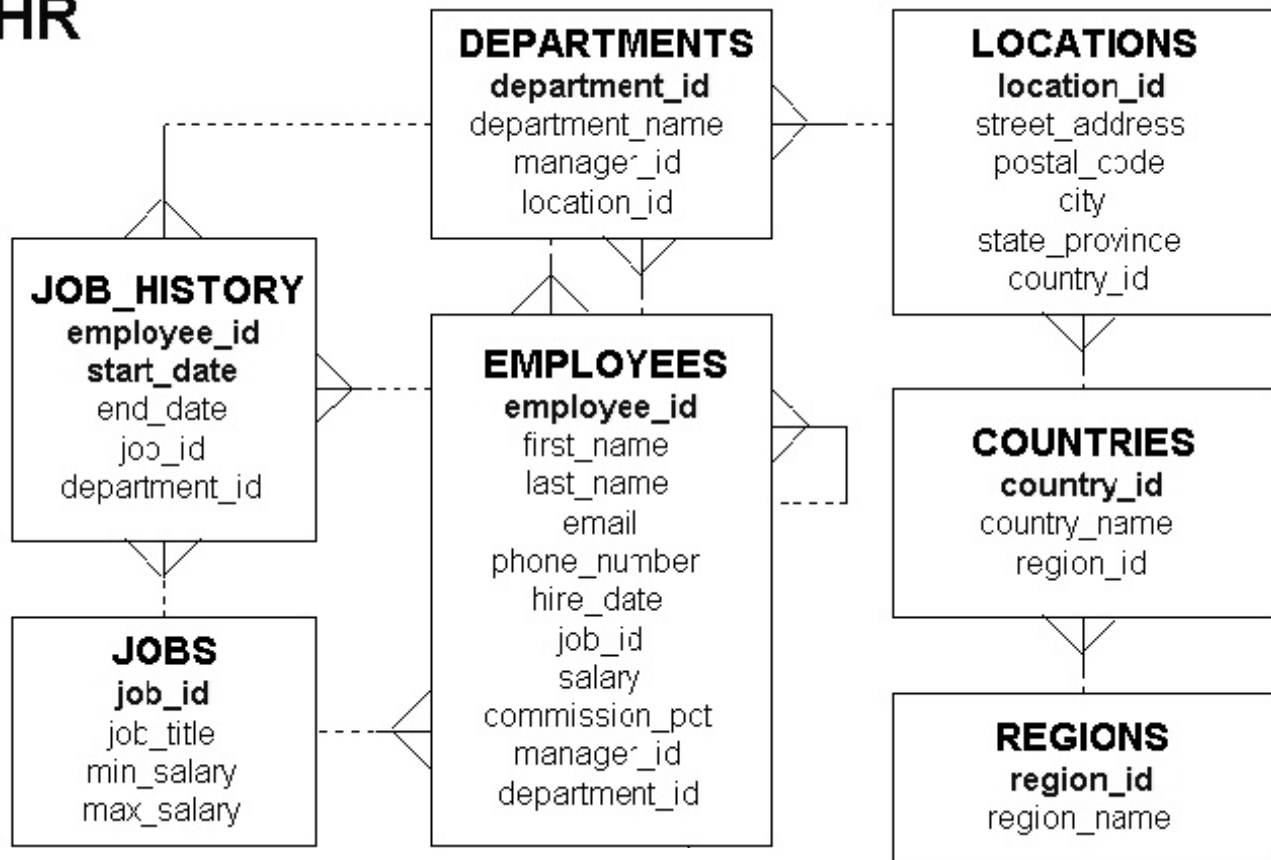
Part B is a case study which can be completed at the end of the course. This part supplements the practices for creating and managing program units. The tables used in the case study are based on a video database and contain the tables TITLE, TITLE_COPY, RENTAL, RESERVATION, and MEMBER.

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

Part A: ENTITY RELATIONSHIP DIAGRAM

Human Resources

HR



Part A

Note: These exercises can be used for extra practice when discussing how to declare variables and write executable statements.

1. Evaluate each of the following declarations. Determine which of them are not legal and explain why.
 - a. DECLARE
v_name, v_dept VARCHAR2(14);
 - b. DECLARE
v_test NUMBER(5);
 - c. DECLARE
V_MAXSALARY NUMBER(7,2) = 5000;
 - d. DECLARE
V_JOINDATE BOOLEAN := SYSDATE;
2. In each of the following assignments, determine the data type of the resulting expression.
 - a. v_email := v_firstname || to_char(v_empno);
 - b. v_confirm := to_date('20-JAN-1999', 'DD-MON-YYYY');
 - c. v_sal := (1000*12) + 500
 - d. v_test := FALSE;
 - e. v_temp := v_temp1 < (v_temp2/ 3);
 - f. v_var := sysdate;

Part A

```
3.  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;
2      v_custid := (v_custid *12) / 10;
      END;
/
```

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

- The value of V_CUSTID at position 1 is:
- The value of V_CUSTNAME at position 1 is:
- The value of V_NEW_CUSTID at position 2 is:
- The value of V_NEW_CUSTNAME at position 1 is:
- The value of V_CUSTID at position 2 is:
- The value of V_CUSTNAME at position 2 is:

Note: These exercises can be used for extra practice when discussing how to interact with the Oracle server and write control structures.

4. Write a PL/SQL block to accept a year and check whether it is a leap year. For example, if the year entered is 1990, the output should be “1990 is not a leap year.”

Hint: The year should be exactly divisible by 4 but not divisible by 100, or it should be divisible by 400.

Part A

Test your solution with the following years:

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

```
old 2: V_YEAR NUMBER(4) := &P_YEAR;
new 2: V_YEAR NUMBER(4) := 1990;
1990 is not a leap year
PL/SQL procedure successfully completed.
```

5. a. For the exercises below, you will require a temporary table to store the results. You can either create the table yourself or run the labAp_5.sql script that will create the table for you. Create a table named TEMP with the following three columns:

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

b. Write a PL/SQL block that contains two variables, MESSAGE and DATE_WRITTEN.

Declare MESSAGE as VARCHAR2 data type with a length of 35 and DATE_WRITTEN as DATE data type. Assign the following values to the variables:

Variable	Contents
MESSAGE	'This is my first PL/SQL program'
DATE_WRITTEN	Current date

Store the values in appropriate columns of the TEMP table. Verify your results by querying the TEMP table.

NUM_STORE	CHAR_STORE	DATE_STORE
	This is my first PLSQL Program	20-MAR-01

Part A

6. Write a PL/SQL block to store a department number in a *iSQL*Plus* substitution variable and print the number of people working in that department.

Hint: Enable DBMS_OUTPUT in *iSQL*Plus* with SET SERVEROUTPUT ON.

```
old 3: V_DEPTNO DEPARTMENTS.department_id%TYPE := &P_DEPTNO;  
new 3: V_DEPTNO DEPARTMENTS.department_id%TYPE := 30;  
6 employee(s) work for department number 30  
PL/SQL procedure successfully completed.
```

7. Write a PL/SQL block to declare a variable called `v_salary` to store the salary of an employee. In the executable part of the program, do the following:
- Store an employee name in a *iSQL*Plus* substitution variable
 - Store his or her salary in the variable `v_salary`
 - If the salary is less than 3,000, give the employee a raise of 500 and display the message '<Employee Name>'s salary updated' in the window.
 - If the salary is more than 3,000, print the employee's salary in the format, '<Employee Name> earns'
 - Test the PL/SQL for the following last names:

LAST_NAME	SALARY
Pataballa	4800
Greenberg	12000
Ernst	6000

Note: Undefine the variable that stores the employee's name at the end of the script.

8. Write a PL/SQL block to store the salary of an employee in an *iSQL*Plus* substitution variable. In the executable part of the program do the following:
- Calculate the annual salary as salary * 12.
 - Calculate the bonus as indicated below:

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

- Display the amount of the bonus in the window in the following format:
'The bonus is \$.....'

Part A

- Test the PL/SQL for the following test cases:

SALARY	BONUS
5000	2000
1000	1000
15000	2000

Note: These exercises can be used for extra practice when discussing how to work with composite data types, cursors and handling exceptions.

9. Write a PL/SQL block to store an employee number, the new department number, and the percentage increase in the salary in *iSQL**Plus substitution variables. Update the department ID of the employee with the new department number, and update the salary with the new salary. Use the EMP table for the updates. Once the update is complete, display the message, 'Update complete' in the window. If no matching records are found, display 'No Data Found'. Test the PL/SQL for the following test cases:

EMPLOYEE_ID	NEW_DEPARTMEN T_ID	% INCREASE	MESSAGE
100	20	2	Updation Complete
10	30	5	No Data found
126	40	3	Updation Complete

Part A

10. Create a PL/SQL block to declare a cursor EMP_CUR to select the employee name, salary, and hire date from the EMPLOYEES table. Process each row from the cursor, and if the salary is greater than 15,000 and the hire date is greater than 01-FEB-1988, display the employee name, salary, and hire date in the window in the format shown in the sample output below:

```
Kochhar earns 17000 and joined the organization on 21-SEP-89
De Haan earns 17000 and joined the organization on 13-JAN-93
PL/SQL procedure successfully completed.
```

11. Create a PL/SQL block to retrieve the last name and department ID of each employee from the EMPLOYEES table for those employees whose EMPLOYEE_ID is less than 114. From the values retrieved from the EMPLOYEES table, populate two PL/SQL tables, one to store the records of the employee last names and the other to store the records of their department IDs. Using a loop, retrieve the employee name information and the salary information from the PL/SQL tables and display it in the window, using DBMS_OUTPUT.PUT_LINE. Display these details for the first 15 employees in the PL/SQL tables.

```
Employee Name: King Department_id: 90
Employee Name: Kochhar Department_id: 90
Employee Name: De Haan Department_id: 90
Employee Name: Hunold Department_id: 60
Employee Name: Ernst Department_id: 60
Employee Name: Austin Department_id: 60
Employee Name: Pataballa Department_id: 60
Employee Name: Lorentz Department_id: 60
Employee Name: Greenberg Department_id: 100
Employee Name: Favier Department_id: 100
Employee Name: Chen Department_id: 100
Employee Name: Sciarra Department_id: 100
Employee Name: Urman Department_id: 100
Employee Name: Popp Department_id: 100
Employee Name: Raphaely Department_id: 30
PL/SQL procedure successfully completed.
```

Part A

12. Create a PL/SQL block that declares a cursor called DATE_CUR. Pass a parameter of DATE data type to the cursor and print the details of all employees who have joined after that date.

```
DEFINE P_HIREDATE = 08-MAR-00

166 Ande 24-MAR-00
167 Banda 21-APR-00
173 Kumar 21-APR-00
PL/SQL procedure successfully completed.
```

Test the PL/SQL block for the following hire dates: 25-JUN-97, 28-SEP-98, 07-FEB-99

13. Create a PL/SQL block to promote clerks who earn more than 3,000 to the job title SR CLERK and increase their salary by 10%. Use the EMP table for this practice. Verify the results by querying on the EMP table. **Hint:** Use a cursor with FOR UPDATE and CURRENT OF syntax.
14. a. For the exercise below, you will require a table to store the results. You can create the ANALYSIS table yourself or run the labAp_14a.sql script that creates the table for you. Create a table called ANALYSIS with the following three columns:

Column Name	ENAME	YEARS	SAL
Key Type			
Nulls/Unique			
FK Table			
FK Column			
Datatype	VARCHAR2	Number	Number
Length	20	2	8,2

- b. Create a PL/SQL block to populate the ANALYSIS table with the information from the EMPLOYEES table. Use an *iSQL**Plus substitution variable to store an employee's last name. Query the EMPLOYEES table to find if the number of years that the employee has been with the organization is greater than five, and if the salary is less than 3,500, raise an exception. Handle the exception with an appropriate exception handler that inserts the following values into the ANALYSIS table: employee last name, number of years of service, and the current salary. Otherwise display Not due for a raise in the window. 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	Not due for a raise
Fripp	Not due for a raise
Khoo	Due for a raise

Part A

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

15. In this practice, create a program to add a new job into the JOBS table.

- a. Create a stored procedure called ADD_JOBS to enter a new order into the JOBS table.

The procedure should accept three parameters. The first and second parameters supplies 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.

- b. Disable the trigger SECURE_DML before invoking the procedure. Invoke the procedure to add a new job with job ID SY_ANAL, job title System Analyst, and minimum salary of 6,000.
- c. Verify that a row was added and remember the new job ID for use in the next exercise.

Commit the changes.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	6000	12000

16. In this practice, create a program to add a new row to the JOB_HISTORY table for an existing employee.

Note: Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the procedure in part b. Enable all these triggers after executing the procedure.

- a. Create a stored procedure called ADD_JOB_HIST to enter a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID that you created in question 15b.

Use the employee ID of the employee who is changing the job and the new job ID for the employee as parameters. Obtain the row corresponding to this employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table. Make hire date of this employee as the start date and today's date as end date for this row in the JOB_HISTORY table.

Change the hire date of this employee in the EMPLOYEES table to today's date. Update the job ID of this employee to the job ID passed as parameter (Use the job ID of the job created in question 15b) and salary equal to minimum salary for that job ID + 500.

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

- b. Disable triggers (See the note at the beginning of this question.)

Execute the procedure with employee ID 106 and job ID SY_ANAL as parameters.

Enable the triggers that you disabled.

- c. Query the tables to view your changes, and then commit the changes.

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	DEPARTMENT_ID
106	05-FEB-98	04-MAY-01	IT_PROG	60

JOB_ID	SALARY
SY_ANAL	6500

Part A

17. In this practice, create a program to update the minimum and maximum salaries for a job in the JOBS table.

- a. Create a stored procedure called UPD_SAL to update the minimum and maximum salaries for a specific job ID in the JOBS table.

Pass three parameters to the procedure: the job ID, a new minimum salary, and a new maximum salary for the job. Add exception handling to account for an invalid job ID in the JOBS table. Also, raise an exception if the maximum salary supplied is less than the minimum salary. Provide an appropriate message that will be displayed if the row in the JOBS table is locked and cannot be changed.

- b. Execute the procedure. You can use the following data to test your procedure:

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

```
ERROR ... MAX SAL SHOULD BE > MIN SAL
```

```
BEGIN upd_sal('SY_ANAL', 7000, 140); END;
```

```
*
```

```
ERROR at line 1:
```

```
ORA-20001: Data error. Max salary should be more than min salary
```

```
ORA-06512: at "SH9.UPD_SAL", line 32
```

```
ORA-06512: at line 1
```

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

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

- c. Query the JOBS table to view your changes, and then commit the changes.

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

Commit complete.

18. In this practice, create a procedure to monitor whether employees have exceeded their average salary limits.

- a. Add a column to the EMPLOYEES table by executing the following command: (labaddA_4.sql)

```
ALTER TABLE employees
```

```
ADD (sal_limit_indicate VARCHAR2(3) DEFAULT 'NO'
```

```
CONSTRAINT emp_sallimit_ck CHECK
```

```
(sal_limit_indicate IN ('YES', 'NO')));
```

- b. Write a stored procedure called CHECK_AVG_SAL. This checks each employee's average salary limit from the JOBS table against the salary that this employee has in the EMPLOYEES table and updates the SAL_LIMIT_INDICATE column in the EMPLOYEES table when this employee has exceeded his average salary limit.

Create a cursor to hold employee IDs, salaries, and their average salary limit. Find the average salary limit possible for an employee's job from the JOBS table. Compare the average salary limit possible for each employee to exact salaries and if the salary is more than the average salary limit, set the employee's SAL_LIMIT_INDICATE column to YES; otherwise, set it to NO.

Add exception handling to account for a record being locked.

Part A

- c. Execute the procedure, and then test the results.

Query the EMPLOYEES table to view your modifications, and then commit the changes.

JOB_ID	MIN_SALARY	SALARY	MAX_SALARY
SY_ANAL	7000	7000	14000

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

19. Create a program to retrieve the number of years of service for a specific employee.

- a. Create a stored function called GET_SERVICE_YRS 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.

- b. Invoke the function. You can use the following data:

```
EXECUTE DBMS_OUTPUT.PUT_LINE(get_service_yrs(999))
```

Hint: The above statement should produce an error message because there is no employee with employee ID 999.

```
EXECUTE DBMS_OUTPUT.PUT_LINE ('Approximately .... ' ||  
                                get_service_yrs(106) || ' years')
```

Hint: The above statement should be successful and return the number of years of service for employee with employee ID 106.

- c. Query the JOB_HISTORY and EMPLOYEES tables for the specified employee to verify that the modifications are accurate.

EMPLOYEE_ID	JOB_ID	DURATION
102	IT_PROG	5.52876712
101	AC_ACCOUNT	4.10136986
101	AC_MGR	3.38082192
201	MK_REP	3.83835616
114	ST_CLERK	1.77260274
122	ST_CLERK	.997260274
200	AD_ASST	5.75342466
176	SA_REP	.77260274
176	SA_MAN	.997260274
200	AC_ACCOUNT	4.50410959
106	IT_PROG	3.24556171

11 rows selected.

JOB_ID	DURATION
SY_ANAL	.000092719

Part A

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

- a. Create a stored function called `GET_JOB_COUNT` to retrieve the total number of different jobs on which an employee worked.

The function should accept one parameter to hold the employee ID. The function will return the number of different jobs that employee worked until now. This also includes the present job. Add exception handling to account for an invalid employee ID.

Hint: Verify distinct job IDs from the `JOB_HISTORY` table. Verify whether the current job ID is one of the job IDs on which the employee worked.

- b. Invoke the function. You can use the following data:

```
EXECUTE DBMS_OUTPUT.PUT_LINE('Employee worked on ' ||  
                               get_job_count(176) || ' different jobs.')
```

```
Employee worked on 2 different jobs.  
PL/SQL procedure successfully completed.
```

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

21. Create a package specification and body called `EMP_JOB_PKG` that contains your

`ADD_JOBS`, `ADD_JOB_HIST`, and `UPD_SAL` procedures as well as your `GET_SERVICE_YRS` function.

- a. Make all the constructs public. Consider whether you still need the stand-alone procedures and functions that you just packaged.
- b. Disable all the triggers before invoking the procedure and enable them after invoking the procedure, as suggested in question 16b.

Invoke your `ADD_JOBS` procedure to create a new job with ID `PR_MAN`, job title `Public Relations Manager`, and salary of 6,250.

Invoke your `ADD_JOB_HIST` procedure to modify the job of employee with employee ID 110 to job ID `PR_MAN`.

Hint: All of the above calls to the functions should be successful.

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

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-97	04-MAY-01	FI_ACCOUNT	100

JOB_ID	SALARY
PR_MAN	6750

Part A

Note: These exercises can be used for extra practice when discussing how to use Oracle-supplied packages:

22. In this practice, use an Oracle-supplied package to schedule your GET_JOB_COUNT function to run semiannually.

- a. Create an anonymous block to call the DBMS_JOB Oracle-supplied package.

Invoke the package function DBMS_JOB.SUBMIT and pass the following four parameters: a variable to hold the job number, the name of the subprogram you want to submit, SYSDATE as the date when the job will run, and an interval of ADDMONTHS(SYSDATE + 6) for semiannual submission.

Note: To force the job to run immediately, call DBMS_JOB.RUN(your_job_number) after calling DBMS_JOB.SUBMIT, . This executes the job waiting in the queue.

Execute the anonymous block.

- b. Check your results by querying the EMPLOYEES and JOB_HISTORY tables and querying the USER_JOBS dictionary view to see the status of your job submission.

Your output should appear similar to the following output:

JOB	WHAT	SCHEMA_USER	LAST_DATE	NEXT_DATE	INTERVAL
1	BEGIN DBMS_OUTPUT.PUT_LINE (get_job_count(110)); END;	SH9	04-MAY-01	04-NOV-01	ADD_MONTHS(SYSDATE, 6)

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

23. In this practice, create a trigger to ensure that the job ID of any new employee being hired to department 80 (the Sales department) is a sales manager or representative.

- a. Disable all the previously created triggers as discussed in question 16b.
b. Create a trigger called CHK_SALES_JOB.

Fire the trigger before every row that is changed after insertions and updates to the JOB_ID column in the EMPLOYEES table. Check that the new employee has a job ID of SA_MAN or SA_REP in the EMPLOYEES table. Add exception handling and provide an appropriate message so that the update fails if the new job ID is not that of a sales manager or representative.

- c. Test the trigger. You can use the following data:

```
UPDATE employees
  SET job_id = 'AD_VP'
  WHERE employee_id = 106;
```

```
UPDATE employees
  SET job_id = 'AD_VP'
  WHERE employee_id = 179;
```

```
UPDATE employees
  SET job_id = 'SA_MAN'
  WHERE employee_id = 179;
```

Hint: The middle statement should produce the error message specified in your trigger.

Part A

- d. Query the EMPLOYEES table to view the changes. Commit the changes.

JOB_ID	DEPARTMENT_ID	SALARY
SA_MAN	80	6200

- e. Enable all the triggers that you previously disabled, as discussed in question 16b.

24. In this practice, 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.

- a. Create a trigger called CHECK_SAL_RANGE.

Fire the trigger before every row that is changed when data is updated in the MIN_SALARY and MAX_SALARY columns in the JOBS table. For any minimum or maximum salary value that is changed, check that 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. Include exception handling to cover a salary range change that affects the record of any existing employee.

- b. Test the trigger. You can use the following data:

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

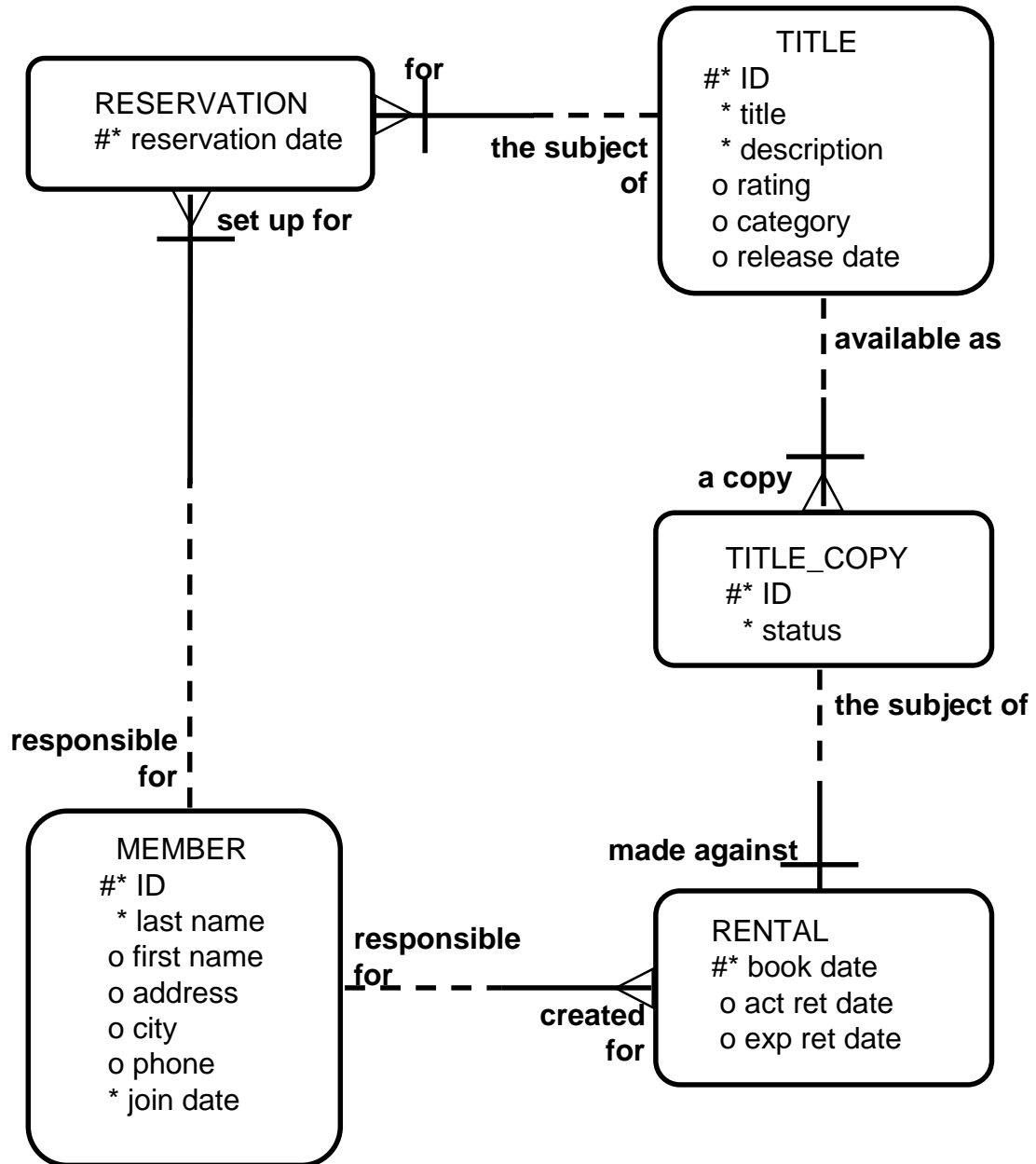
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

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

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

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

Part B: Entity Relationship Diagram



Part B

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

Create the package using *iSQL*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 entity relationship diagram is shown on the facing page.

Part B

1. Run the script `buildvid1.sql` to create all of the required tables and sequences needed for this exercise.

Run the script `buildvid2.sql` to populate all the tables created through by the script `buildvid1.sql`

2. Create a package named VIDEO 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 of 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 out 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 to this procedure the SQLCODE number, 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.

Part B

You can use the following data to test your routines:

```
EXECUTE video.new_member  
      ('Haas', 'James', 'Chestnut Street', 'Boston', '617-123-  
      4567')
```

PL/SQL procedure successfully completed.

```
EXECUTE video.new_member  
      ('Biri', 'Allan', 'Hiawatha Drive', 'New York', '516-123-4567')
```

PL/SQL procedure successfully completed.

```
EXECUTE DBMS_OUTPUT.PUT_LINE(video.new_rental(110, 98))
```

09-MAR-01

PL/SQL procedure successfully completed.

```
EXECUTE DBMS_OUTPUT.PUT_LINE(video.new_rental(109, 93))
```

09-MAR-01

PL/SQL procedure successfully completed.

```
EXECUTE DBMS_OUTPUT.PUT_LINE(video.new_rental(107, 98))
```

Movie reserved. Expected back on: 05-MAR-01

PL/SQL procedure successfully completed.

```
EXECUTE DBMS_OUTPUT.PUT_LINE(video.new_rental('Biri', 97))
```

Warning! More than one member by this name.

111 Biri, Allan

108 Biri, Ben

PL/SQL procedure successfully completed.

```
EXECUTE DBMS_OUTPUT.PUT_LINE(video.new_rental(97, 97))
```

```
BEGIN DBMS_OUTPUT.PUT_LINE(video.new_rental(97, 97)); END;
```

*

ERROR at line 1:

ORA-20002: NEW_RENTAL has

attempted to use a foreign key value that is invalid

ORA-06512: at "PLPU.VIDEO", line 13

ORA-06512: at "PLPU.VIDEO", line 120

ORA-06512: at line 1

Part B

```
EXECUTE video.return_movie(98, 1, 'AVAILABLE')
```

```
Put this movie on hold -- reserved by member #107  
PL/SQL procedure successfully completed.
```

```
EXECUTE video.return_movie(95, 3, 'AVAILABLE')
```

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

```
EXECUTE video.return_movie(111, 1, 'RENTED')
```

```
BEGIN video.return_movie(111, 1, 'RENTED'); END;
```

```
*
```

```
ERROR at line 1:
```

```
ORA-20999: Unhandled error in RETURN_MOVIE. Please contact your application  
administrator with the following information: ORA-01403: no data found
```

```
ORA-06512: at "PLPU.VIDEO", line 16
```

```
ORA-06512: at "PLPU.VIDEO", line 80
```

```
ORA-06512: at line 1
```

Part B

3. The business hours for the video store are 8:00 a.m. to 10:00 p.m., Sunday through Friday, and 8:00 a.m. to 12:00 a.m. on Saturday. To ensure that the tables can only be modified 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 trigger.

Note: In order for your trigger to fail, you 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 p.m. to 8:00 a.m.

Additional Practice Solutions

Part A: Additional Practice 1 and 2 Solutions

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

a. DECLARE

```
v_name, v_dept          VARCHAR2(14);
```

This is illegal because only one identifier per declaration is allowed.

b. DECLARE

```
v_test                  NUMBER(5);
```

This is legal.

c. DECLARE

```
V_MAXSALARY             NUMBER(7,2) = 5000;
```

This is illegal because the assignment operator is wrong. It should be :=.

d. DECLARE

```
V_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.

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

a. `v_email := v_firstname || to_char(v_empno);`

Character string

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

Date

c. `v_sal := (1000*12) + 500`

Number

d. `v_test := FALSE;`

Boolean

e. `v_temp := v_temp1 < (v_temp2/ 3);`

Boolean

f. `v_var := sysdate;`

Date

Part A: Additional Practice 3 Solutions

3. 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;
```

```
  END;
```

```
v_custid := (v_custid *12) / 10;
```

```
END;
```

```
/
```

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

a. The value of V_CUSTID at position 1 is:

300, and the data type is NUMBER

b. The value of V_CUSTNAME at position 1 is:

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

c. The value of V_NEW_CUSTID at position 1 is:

500, and the data type is NUMBER (or INTEGER)

d. The value of V_NEW_CUSTNAME at position 1 is:

Jansports Club, and the data type is VARCHAR2

e. The value of V_CUSTID at position 2 is:

1920, and the data type is NUMBER

f. The value of V_CUSTNAME at position 2 is:

Women Sports Club, and the data type is VARCHAR2

Part A: Additional Practice 3 and 4 Solutions

4. Write a PL/SQL block to accept a year and check whether it is a leap year. For example, if the year entered is 1990, the output should be “1990 is not a leap year”.

Hint: The year should be exactly divisible by 4 but not divisible by 100, or it should be divisible by 400.

Test your solution with the following years:

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

```
SET SERVEROUTPUT ON
DEFINE p_year = 1990
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;
/
SET SERVEROUTPUT OFF
```

Part A: Additional Practice 5 Solutions

5. a. For the exercises below, you will require a temporary table to store the results. You can either create the table yourself or run the `labAp_5.sql` script that will create the table for you. Create a table named `TEMP` with the following three columns:

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

```
CREATE TABLE temp
(num_store NUMBER(7,2),
char_store VARCHAR2(35),
date_store DATE);
```

- b. Write a PL/SQL block that contains two variables, `MESSAGE` and `DATE_WRITTEN`. Declare `MESSAGE` as `VARCHAR2` data type with a length of 35 and `DATE_WRITTEN` as `DATE` data type. Assign the following values to the variables:

Variable	Contents
<code>MESSAGE</code>	This is my first PL/SQL program'
<code>DATE_WRITTEN</code>	Current date

Store the values in appropriate columns of the `TEMP` table. Verify your results by querying the `TEMP` table.

```
DECLARE
    MESSAGE VARCHAR2(35);
    DATE_WRITTEN DATE;
BEGIN
    MESSAGE := 'This is my first PLSQL Program';
    DATE_WRITTEN := SYSDATE;
    INSERT INTO temp(char_store,date_store)
    VALUES (MESSAGE,DATE_WRITTEN);
END;
/
SELECT * FROM TEMP;
```


Part A: Additional Practice 6 and 7 Solutions

6. Write a PL/SQL block to store a department number in a *iSQL*Plus* substitution variable and print the number of people working in that department.

Hint: Enable DBMS_OUTPUT in *iSQL*Plus* with SET SERVEROUTPUT ON.

```
SET SERVEROUTPUT ON
DEFINE P_DEPTNO = 30
DECLARE
  V_COUNT NUMBER(3);
  V_DEPTNO DEPARTMENTS.department_id%TYPE := &P_DEPTNO;
BEGIN
  SELECT COUNT(*) INTO V_COUNT FROM employees
  WHERE department_id = V_DEPTNO;
  DBMS_OUTPUT.PUT_LINE (V_COUNT || ' employee(s) work for department
  number ' || V_DEPTNO);
END;
/
SET SERVEROUTPUT OFF
```

7. Write a PL/SQL block to declare a variable called `v_salary` to store the salary of an employee. In the executable part of the program, do the following:
- Store an employee name in a *iSQL*Plus* substitution variable
 - Store his or her salary in the `v_salary` variable
 - If the salary is less than 3,000, give the employee a raise of 500 and display the message '<Employee Name>'s salary updated' in the window.
 - If the salary is more than 3,000, print the employee's salary in the format, '<Employee Name> earns'
 - Test the PL/SQL for the following last names

Note: Undefine the variable that stores the employee's name at the end of the script.

```
SET SERVEROUTPUT ON
DEFINE P_LASTNAME = Pataballa
DECLARE
  V_SALARY NUMBER(7,2);
  V_LASTNAME EMPLOYEES.LAST_NAME%TYPE;
BEGIN
  SELECT salary INTO V_SALARY
  FROM employees
  WHERE last_name = INITCAP('&P_LASTNAME') FOR UPDATE of salary;
```

Part A: Additional Practice 7 and 8 Solutions

```
V_LASTNAME := INITCAP('&P_LASTNAME');  
    IF V_SALARY < 3000 THEN  
        UPDATE employees SET salary = salary + 500  
        WHERE last_name = INITCAP('&P_LASTNAME') ;  
        DBMS_OUTPUT.PUT_LINE (V_LASTNAME || ''s salary updated');  
    ELSE  
        DBMS_OUTPUT.PUT_LINE (V_LASTNAME || ' earns ' ||  
            TO_CHAR(V_SALARY));  
    END IF;  
END;  
/  
SET SERVEROUTPUT OFF  
UNDEFINE P_LASTNAME
```

8. Write a PL/SQL block to store the salary of an employee in an iSQL*Plus substitution variable. In the executable part of the program do the following:

- Calculate the annual salary as salary * 12.
- Calculate the bonus as indicated below:

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

- Display the amount of the bonus in the window in the following format:
‘The bonus is \$.....’

```
SET SERVEROUTPUT ON  
DEFINE P_SALARY = 5000  
DECLARE  
    V_SALARY    NUMBER(7,2) := &P_SALARY;  
    V_BONUS     NUMBER(7,2);  
    V_ANN_SALARY NUMBER(15,2);
```

Part A: Additional Practice 8 and 9 Solutions

```
BEGIN
    V_ANN_SALARY := V_SALARY * 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;
/
SET SERVEROUTPUT OFF
```

9. Write a PL/SQL block to store an employee number, the new department number and the percentage increase in the salary in *iSQL**Plus substitution variables. Update the department ID of the employee with the new department number, and update the salary with the new salary. Use the EMP table for the updates. Once the update is complete, display the message, 'Update complete' in the window. If no matching records are found, display the message, 'No Data Found'. Test the PL/SQL for the following test cases:

```
SET SERVEROUTPUT ON
DEFINE P_EMPNO = 100
DEFINE P_NEW_DEPTNO = 10
DEFINE P_PER_INCREASE = 2
DECLARE
    V_EMPNO emp.EMPLOYEE_ID%TYPE := &P_EMPNO;
    V_NEW_DEPTNO emp.DEPARTMENT_ID%TYPE := &P_NEW_DEPTNO;
    V_PER_INCREASE NUMBER(7,2) := &P_PER_INCREASE;
BEGIN
    UPDATE emp
    SET department_id = V_NEW_DEPTNO,
        salary = salary + (salary *
V_PER_INCREASE/100)
    WHERE employee_id = V_EMPNO;
    IF SQL%ROWCOUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE ('No Data Found');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('Update Complete');
    END IF;
END;
/
SET SERVEROUTPUT OFF
```

Part A: Additional Practice 10 Solutions

10. Create a PL/SQL block to declare a cursor EMP_CUR to select the employee name, salary, and hire date from the EMPLOYEES table. Process each row from the cursor, and if the salary is greater than 15,000 and the hire date is greater than 01-FEB-1988, display the employee name, salary, and hire date in the window.

```
SET SERVEROUTPUT ON
DECLARE
    CURSOR EMP_CUR IS
        SELECT  last_name,salary,hire_date FROM EMPLOYEES;
        V_ENAME VARCHAR2(25);
        V_SAL    NUMBER(7,2);
        V_HIREDATE DATE;
BEGIN
    OPEN EMP_CUR;
    FETCH EMP_CUR INTO V_ENAME,V_SAL,V_HIREDATE;
    WHILE 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 EMP_CUR INTO V_ENAME,V_SAL,V_HIREDATE;
        END LOOP;
    CLOSE EMP_CUR;
END;
/
SET SERVEROUTPUT OFF
```

Part A: Additional Practice 11 Solutions

11. Create a PL/SQL block to retrieve the last name and department ID of each employee from the EMPLOYEES table for those employees whose EMPLOYEE_ID is less than 114. From the values retrieved from the EMPLOYEES table, populate two PL/SQL tables, one to store the records of the employee last names and the other to store the records of their department IDs. Using a loop, retrieve the employee name information and the salary information from the PL/SQL tables and display it in the window, using DBMS_OUTPUT.PUT_LINE. Display these details for the first 15 employees in the PL/SQL tables.

```
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;
```

```
    V_Tename Table_Ename;
```

```
    V_Tdept Table_dept;
```

```
    i BINARY_INTEGER :=0;
```

```
    CURSOR C_Namedept IS SELECT last_name,department_id from employees
                          WHERE employee_id < 115;
```

```
    V_COUNT NUMBER := 15;
```

```
BEGIN
```

```
    FOR emprec in C_Namedept
```

```
    LOOP
```

```
        i := i +1;
```

```
        V_Tename(i) := emprec.last_name;
```

```
        V_Tdept(i) := emprec.department_id;
```

```
    END LOOP;
```

```
    FOR i IN 1..v_count
```

```
    LOOP
```

```
        DBMS_OUTPUT.PUT_LINE ('Employee Name: ' || V_Tename(i) ||
                               ' Department_id: ' || V_Tdept(i));
```

```
    END LOOP;
```

```
END;
```

```
/
```

```
SET SERVEROUTPUT OFF
```

Part A: Additional Practice 12 Solutions

12. Create a PL/SQL block that declares a cursor called DATE_CUR. Pass a parameter of DATE data type to the cursor and print the details of all employees who have joined after that date. Test the PL/SQL block for the following hire dates: 08-MAR-00, 25-JUN-97, 28-SEP-98, 07-FEB-99.

```
SET SERVEROUTPUT ON
DEFINE P_HIREDATE = 08-MAR-00
DECLARE
    CURSOR DATE_CURSOR(JOIN_DATE DATE) IS
        SELECT employee_id,last_name,hire_date FROM employees
        WHERE HIRE_DATE >JOIN_DATE ;
    V_EMPNO    employees.employee_id%TYPE;
    V_ENAME    employees.last_name%TYPE;
    V_HIREDATE employees.hire_date%TYPE;
    V_DATE employees.hire_date%TYPE :=    '&P_HIREDATE';
BEGIN
    OPEN DATE_CURSOR(V_DATE);
    LOOP
        FETCH DATE_CURSOR INTO V_EMPNO,V_ENAME,V_HIREDATE;
        EXIT WHEN DATE_CURSOR%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (V_EMPNO || ' ' || V_ENAME || ' ' ||
                                V_HIREDATE);

        END LOOP;
    END;
/
SET SERVEROUTPUT OFF;
```

Part A: Additional Practice 13 Solutions

13. Create a PL/SQL block to promote clerks who earn more than 3,000 to SR. CLERK and increase their salary by 10%. Use the EMP table for this practice. Verify the results by querying on the EMP table.

Hint: Use a cursor with FOR UPDATE and CURRENT OF syntax.

```
DECLARE
    CURSOR C_Senior_Clerk IS
        SELECT employee_id, job_id FROM emp
        WHERE job_id = 'ST_CLERK' AND salary > 3000
        FOR UPDATE OF job_id;
BEGIN
    FOR V_Emrec IN C_Senior_Clerk
    LOOP
        UPDATE emp
        SET job_id = 'SR_CLERK',
            salary = 1.1 * salary
        WHERE CURRENT OF C_Senior_Clerk;
    END LOOP;
    COMMIT;
END;
/
SELECT * FROM emp;
```

Part A: Additional Practice 14 Solutions

14. a. For the exercise below, you will require a table to store the results. You can create the ANALYSIS table yourself or run the labAp_14a.sql script that creates the table for you. Create a table called ANALYSIS with the following three columns:

Column Name	ENAME	YEARS	SAL
Key Type			
Nulls/Unique			
FK Table			
FK Column			
Datatype	VARCHAR2	Number	Number
Length	20	2	8,2

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

- b. Create a PL/SQL block to populate the ANALYSIS table with the information from the EMPLOYEES table. Use an iSQL*Plus substitution variable to store an employee's last name. Query the EMPLOYEES table to find if the number of years that the employee has been with the organization is greater than five, and if the salary is less than 3,500, raise an exception. Handle the exception with an appropriate exception handler that inserts the following values into the ANALYSIS table: employee last name, number of years of service, and the current salary. Otherwise display Not due for a raise in the window. Verify the results by querying the ANALYSIS table. Use the following test cases to test the PL/SQL block:

```
SET SERVEROUTPUT ON
DEFINE P_ENAME = Austin
DECLARE
  DUE_FOR_RAISE EXCEPTION;
  V_HIREDATE EMPLOYEES.HIRE_DATE%TYPE;
  V_ENAME EMPLOYEES.LAST_NAME%TYPE := INITCAP( '& P_ENAME' );
  V_SAL EMPLOYEES.SALARY%TYPE;
  V_YEARS NUMBER(2);
```


Part A: Additional Practice 14 Solutions (continued)

```
BEGIN
    SELECT LAST_NAME,SALARY,HIRE_DATE
    INTO   V_ENAME,V_SAL,V_HIREDATE
    FROM employees WHERE last_name =  V_ENAME;
    V_YEARS := MONTHS_BETWEEN(SYSDATE,V_HIREDATE)/12;
    IF V_SAL < 3500 AND V_YEARS > 5  THEN
        RAISE DUE_FOR_RAISE;
    ELSE
        DBMS_OUTPUT.PUT_LINE ('Not due for a raise');
    END IF;
EXCEPTION
    WHEN DUE_FOR_RAISE THEN
        INSERT INTO ANALYSIS(ENAME,YEARS,SAL)
        VALUES (V_ENAME,V_YEARS,V_SAL);
END;
/
```

Part A: Additional Practice 15 Solutions

15. In this practice, create a program to add a new job into the JOBS table.

- a. Create a stored procedure called ADD_JOBS to enter a new order into the JOBS table.

The procedure should accept three parameters. The first and second parameters supplies 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.

```
CREATE OR REPLACE PROCEDURE add_jobs
(p_jobid    IN jobs.job_id%TYPE,
 p_jobtitle IN jobs.job_title%TYPE,
 p_minsal   IN jobs.min_salary%TYPE
)
IS
    v_maxsal jobs.max_salary%TYPE;
BEGIN
    v_maxsal := 2 * p_minsal;
    INSERT INTO jobs
        (job_id, job_title, min_salary, max_salary)
    VALUES
        (p_jobid, p_jobtitle, p_minsal, v_maxsal);
    DBMS_OUTPUT.PUT_LINE ('Added the following row
                           into the JOBS table ...');
    DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_jobtitle ||
                           ' ' || p_minsal || ' ' || v_maxsal);
END add_jobs;
/
```

- b. Disable the trigger SECURE_DML before invoking the procedure. Invoke the procedure to add a new job with job ID SY_ANAL, job title System Analyst, and minimum salary of 6,000.

```
ALTER TRIGGER secure_employees DISABLE;
EXECUTE add_jobs ('SY_ANAL', 'System Analyst', 6000)
```

- c. Verify that a row was added and remember the new job ID for use in the next exercise.

Commit the changes.

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

Part A: Additional Practice 16 Solutions

16. In this practice, create a program to add a new row to the JOB_HISTORY table, for an existing employee.

Note: Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the procedure in part b. Enable all these triggers after executing the procedure.

- a. Create a stored procedure called ADD_JOB_HIST to enter a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID that you created in question 15b.

Use the employee ID of the employee who is changing the job and the new job ID for the employee as parameters. Obtain the row corresponding to this employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table. Make hire date of this employee as start date and today's date as end date for this row in the JOB_HISTORY table.

Change the hire date of this employee in the EMPLOYEES table to today's date. Update the job ID of this employee to the job ID passed as parameter (Use the job ID of the job created in question 15b) and salary equal to minimum salary for that job ID + 500.

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

```
CREATE OR REPLACE PROCEDURE add_job_hist
(p_empid IN employees.employee_id%TYPE,
 p_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_empid;
    UPDATE employees
        SET    hire_date = SYSDATE,
              job_id = p_jobid,
              salary = (SELECT min_salary+500
                        FROM    jobs
                        WHERE   job_id = p_jobid)
        WHERE employee_id = p_empid;
    DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_empid ||
                          ' details to the JOB_HISTORY table');
    DBMS_OUTPUT.PUT_LINE ('Updated current job of employee '
                          || p_empid || ' to ' || p_jobid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR (-20001, 'Employee does not exist!');
END add_job_hist;
/
```

Part A: Additional Practice 16 Solutions (continued)

- b. Disable triggers. (See the note at the beginning of this question.)

Execute the procedure with employee ID 106 and job ID SY_ANAL as parameters.

Enable the triggers that you disabled.

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

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

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

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

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

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

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

- c. Query the tables to view your changes, and then commit the changes.

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

```
SELECT job_id, salary FROM employees  
WHERE employee_id = 106;
```

Part A: Additional Practice 17 Solutions

17. In this practice, create a program to update the minimum and maximum salaries for a job in the JOBS table.

- a. Create a stored procedure called UPD_SAL to update the minimum and maximum salaries for a specific job ID in the JOBS table.

Pass three parameters to the procedure: the job ID, a new minimum salary, and a new maximum salary for the job. Add exception handling to account for an invalid job ID in the JOBS table. Also, raise an exception if the maximum salary supplied is less than the minimum salary. Provide an appropriate message that will be displayed if the row in the JOBS table is locked and cannot be changed.

```
CREATE OR REPLACE PROCEDURE upd_sal
(p_jobid    IN jobs.job_id%type,
 p_minsal   IN jobs.min_salary%type,
 p_maxsal   IN jobs.max_salary%type)
IS
    v_dummy          VARCHAR2(1);
    e_resource_busy   EXCEPTION;
    sal_error         EXCEPTION;
    PRAGMA            EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
    IF (p_maxsal < p_minsal) THEN
        DBMS_OUTPUT.PUT_LINE('ERROR. MAX SAL SHOULD BE > MIN SAL');
        RAISE sal_error;
    END IF;
    SELECT ''
        INTO v_dummy
        FROM jobs
        WHERE job_id = p_jobid
        FOR UPDATE OF min_salary NOWAIT;
    UPDATE jobs
        SET     min_salary = p_minsal,
               max_salary = p_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 sal_error THEN
 RAISE_APPLICATION_ERROR(-20001, 'Data error..Max salary
 should be more than min salary');
END upd_sal;
/
```


```

Part A: Additional Practice 17 and 18 Solutions

- b. Execute the procedure. You can use the following data to test your procedure:

```
EXECUTE upd_sal ('SY_ANAL', 7000, 140)
EXECUTE upd_sal ('SY_ANAL', 7000, 14000)
```

- c. Query the JOBS table to view your changes, and then commit the changes.

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

18. In this practice, create a procedure to monitor whether employees have exceeded their average salary limits.

- a. Add a column to the EMPLOYEES table by executing the following command:

(labaddA_4.sql)

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

- b. Write a stored procedure called CHECK_AVG_SAL which checks each employee's average salary limit from the JOBS table against the salary that this employee has in the EMPLOYEES table and updates the SAL_LIMIT_INDICATE column in the EMPLOYEES table when this employee has exceeded his average salary limit.

Create a cursor to hold employee Ids, salaries, and their average salary limit. Find the average salary limit possible for an employee's job from the JOBS table. Compare the average salary limit possible per employee to their salary and if the salary is more than the average salary limit, set the employee's SAL_LIMIT_INDICATE column to YES; otherwise, set it to NO. Add exception handling to account for a record being locked.

Part A: Additional Practice 18 Solutions (continued)

```
CREATE OR REPLACE PROCEDURE check_avg_sal
IS
    v_avg_sal NUMBER;
    CURSOR emp_sal_cur IS
        SELECT employee_id, job_id, salary
        FROM employees
        FOR UPDATE;
    e_resource_busy EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_resource_busy, -54);
BEGIN
    FOR r_emp IN emp_sal_cur LOOP
        SELECT (max_salary + min_salary)/2
        INTO v_avg_sal
        FROM jobs
        WHERE jobs.job_id = r_emp.job_id;
        IF r_emp.salary >= v_avg_sal THEN
            UPDATE employees
            SET sal_limit_indicate = 'YES'
            WHERE CURRENT OF emp_sal_cur;
        ELSE
            UPDATE employees
            SET sal_limit_indicate = 'NO'
            WHERE employee_id = r_emp.employee_id;
        END IF;
    END LOOP;
EXCEPTION
    WHEN e_resource_busy THEN
        ROLLBACK;
        RAISE_APPLICATION_ERROR (-20001,
            'Record is busy, try later.');
```

END check_avg_sal;

/

- c. Execute the procedure, and then test the results.

```
EXECUTE check_avg_sal
```

Query the EMPLOYEES table to view your modifications, and then commit the changes.

```
SELECT e.job_id, j.min_salary, e.salary, j.max_salary
FROM   employees e, jobs j
WHERE  e.job_id = j.job_id
AND    employee_id = 106;
```

Part A: Additional Practice 19 Solutions

19. Create a program to retrieve the number of years of service for a specific employee.
- a. Create a stored function called GET_SERVICE_YRS 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.

```
CREATE OR REPLACE FUNCTION get_service_yrs
(p_empid IN employees.employee_id%TYPE)
RETURN number
IS
    CURSOR emp_yrs_cur IS
        SELECT (end_date - start_date)/365 service
        FROM    job_history
        WHERE   employee_id = p_empid;
    v_srvcyrs NUMBER(2) := 0;
    v_yrs NUMBER(2) := 0;
BEGIN
    FOR r_yrs IN emp_yrs_cur LOOP
        EXIT WHEN emp_yrs_cur%NOTFOUND;
        v_srvcyrs := v_srvcyrs + r_yrs.service;
    END LOOP;
    SELECT (SYSDATE - hire_date)
    INTO   v_yrs
    FROM   employees
    WHERE  employee_id = p_empid;
    v_srvcyrs := v_srvcyrs + v_yrs;
    RETURN v_srvcyrs;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20348, 'There is no employee with
        the specified ID');
END get_service_yrs;
/
```

- b. Invoke the function. You can use the following data:

```
EXECUTE DBMS_OUTPUT.PUT_LINE(get_service_yrs(999))
BEGIN DBMS_OUTPUT.PUT_LINE(get_service_yrs(999)); END;
*
ERROR at line 1:
ORA-20348: There is no employee with the specified ID
ORA-06512: at "SH9.GET_SERVICE_YRS", line 24
ORA-06512: at line 1

EXECUTE DBMS_OUTPUT.PUT_LINE ('Approximately .... ' ||
                                get_service_yrs(106) || ' years')

Approximately ... 3 years
PL/SQL procedure successfully completed.
```


Part A: Additional Practice 19 Solutions (continued)

- c. Query the JOB_HISTORY and EMPLOYEES tables for the specified employee to verify that the modifications are accurate.

```
SELECT employee_id, job_id, (end_date-start_date)/365 duration
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID	DURATION
102	IT_PROG	5.52876712
101	AC_ACCOUNT	4.10136986
101	AC_MGR	3.38082192
201	MK_REP	3.83835616
114	ST_CLERK	1.77260274
122	ST_CLERK	.997260274
200	AD_ASST	5.75342466
176	SA_REP	.77260274
176	SA_MAN	.997260274
200	AC_ACCOUNT	4.50410959
106	IT_PROG	3.24556171

11 rows selected.

```
SELECT job_id, (SYSDATE-hire_date)/365 duration
FROM   employees
WHERE  employee_id = 106;
```

JOB_ID	DURATION
SY_ANAL	.000092719

Part A: Additional Practice 20 Solutions

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

- a. Create a stored function called GET_JOB_COUNT to retrieve the total number of different jobs on which employee worked.

The function should accept one parameter to hold the employee ID. The function will return the number of different jobs that employee worked until now. This also includes the present job. Add exception handling to account for an invalid employee ID.

Hint: Verify distinct job IDs from the Job_history table. Verify whether the current job ID is one of the job IDs on which the employee worked.

```
CREATE OR REPLACE FUNCTION get_job_count
(p_empid IN employees.employee_id%TYPE)
RETURN NUMBER
IS
    v_currjob    employees.job_id%TYPE;
    v_numjobs    NUMBER := 0;
    n            NUMBER;
BEGIN
    SELECT COUNT(DISTINCT job_id)
        INTO v_numjobs
        FROM job_history
        WHERE employee_id = p_empid;
    SELECT COUNT(job_id)
        INTO n
        FROM employees
        WHERE employee_id = p_empid
        AND   job_id IN (SELECT DISTINCT job_id
                        FROM job_history
                        WHERE employee_id = p_empid);
    IF (n = 0) THEN    -- The current job is not one of the previous
        jobs
        v_numjobs := v_numjobs + 1;
    END IF;
    RETURN v_numjobs;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20348, 'This employee does not
                                         exist!');
END get_job_count;
/
```

Part A: Additional Practice 20 and 21 Solutions

- b. Invoke the function. You can use the following data:

```
EXECUTE DBMS_OUTPUT.PUT_LINE('Employee worked on ' ||  
                               get_job_count(176) || ' different jobs.')
```

```
Employee worked on 2 different jobs.  
PL/SQL procedure successfully completed.
```

21. Create a package specification and body called EMP_JOB_PKG that contains your ADD_JOBS, ADD_JOB_HIST, and UPD_SAL procedures as well as your GET_SERVICE_YRS function.

- a. Make all the constructs public. Consider whether you still need the stand-alone procedures and functions you just packaged.

```
CREATE OR REPLACE PACKAGE emp_job_pkg  
IS  
    PROCEDURE add_jobs  
        (p_jobid    IN jobs.job_id%TYPE,  
         p_jobtitle  IN jobs.job_title%TYPE,  
         p_minsal    IN jobs.min_salary%TYPE  
        );  
    PROCEDURE add_job_hist  
        (p_empid    IN employees.employee_id%TYPE,  
         p_jobid     IN jobs.job_id%TYPE);  
    PROCEDURE upd_sal  
        (p_jobid     IN jobs.job_id%type,  
         p_minsal     IN jobs.min_salary%type,  
         p_maxsal     IN jobs.max_salary%type);  
    FUNCTION get_service_yrs  
        (p_empid IN employees.employee_id%TYPE)  
        RETURN NUMBER;  
END emp_job_pkg;  
/  
CREATE OR REPLACE PACKAGE BODY emp_job_pkg  
IS  
    PROCEDURE add_jobs  
        (p_jobid    IN jobs.job_id%TYPE,  
         p_jobtitle  IN jobs.job_title%TYPE,  
         p_minsal    IN jobs.min_salary%TYPE  
        )  
    IS  
        v_maxsal    jobs.max_salary%TYPE;  
    BEGIN  
        v_maxsal := 2 * p_minsal;  
        INSERT INTO jobs (job_id, job_title, min_salary, max_salary)  
            VALUES (p_jobid, p_jobtitle, p_minsal, v_maxsal);  
        DBMS_OUTPUT.PUT_LINE ('Added the following row into the JOBS  
table ...');  
        DBMS_OUTPUT.PUT_LINE (p_jobid||' '||p_jobtitle||'  
'||p_minsal||' '||v_maxsal);  
    END add_jobs;
```

Part A: Additional Practice 21 Solutions (continued)

```
PROCEDURE add_job_hist
(p_empid   IN employees.employee_id%TYPE,
 p_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_empid;
    UPDATE employees
    SET    hire_date = SYSDATE, job_id = p_jobid,
           salary = (SELECT min_salary+500 FROM jobs
                     WHERE job_id = p_jobid)
    WHERE employee_id = p_empid;
    DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_empid || ' details
                           to the JOB_HISTORY table');
    DBMS_OUTPUT.PUT_LINE('Updated current job of employee ' ||
                           p_empid || ' to ' || p_jobid);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR (-20001, 'Employee does not exist!');
END add_job_hist;

PROCEDURE upd_sal
(p_jobid   IN jobs.job_id%type,
 p_minsal  IN jobs.min_salary%type,
 p_maxsal  IN jobs.max_salary%type) IS
    v_dummy          VARCHAR2(1);
    e_resource_busy   EXCEPTION;
    sal_error         EXCEPTION;
    PRAGMA            EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
    IF (p_maxsal < p_minsal) THEN
        DBMS_OUTPUT.PUT_LINE('ERROR..MAX SAL SHOULD BE > MIN SAL');
        RAISE sal_error;
    END IF;
    SELECT '' INTO v_dummy FROM jobs WHERE job_id = p_jobid
    FOR UPDATE OF min_salary NOWAIT;
    UPDATE jobs
    SET    min_salary = p_minsal, max_salary = p_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 doesn't exist');
    WHEN sal_error THEN
        RAISE_APPLICATION_ERROR(-20001, 'Data error..Max salary
                                           should be more than min salary');
END upd_sal;
```

Part A: Additional Practice 21 Solutions (continued)

```
FUNCTION get_service_yrs
(p_empid IN employees.employee_id%TYPE)
RETURN number
IS
    CURSOR emp_yrs_cur IS
        SELECT (end_date - start_date)/365 service
        FROM   job_history
        WHERE  employee_id = p_empid;
    v_srvcyrs NUMBER(2) := 0;
    v_yrs NUMBER(2) := 0;
BEGIN
    FOR r_yrs IN emp_yrs_cur LOOP
        EXIT WHEN emp_yrs_cur%NOTFOUND;
        v_srvcyrs := v_srvcyrs + r_yrs.service;
    END LOOP;
    SELECT (SYSDATE - hire_date)
    INTO   v_yrs
    FROM   employees
    WHERE  employee_id = p_empid;
    v_srvcyrs := v_srvcyrs + v_yrs;
    RETURN v_srvcyrs;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20348, 'There is no employee with
the specified ID');
END get_service_yrs;

END emp_job_pkg;
/
```

- b. Disable all the triggers before invoking the procedure and enable them after invoking the procedure, as suggested in question 16b.
- Invoke your ADD_JOBS procedure to create a new job with ID PR_MAN, job title Public Relations Manager, and salary of 6,250.
- Invoke your ADD_JOB_HIST procedure to modify the job of employee with employee ID 110 to job ID PR_MAN.

Hint: All of the above calls to the functions should be successful.

```
EXECUTE emp_job_pkg.add_jobs ('PR_MAN', 'Public Relations
                               Manager', 6250)
EXECUTE emp_job_pkg.add_job_hist(110, 'PR_MAN')
```

- c. Query the JOBS, JOB_HISTORY, and EMPLOYEES tables to verify the results.

```
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;
```

Part A: Additional Practice 22 Solutions

22. In this practice, use an Oracle-supplied package to schedule your GET_JOB_COUNT function to run semiannually.
- a. Create an anonymous block to call the DBMS_JOB Oracle-supplied package. Invoke the package function DBMS_JOB.SUBMIT and pass the following four parameters: a variable to hold the job number, the name of the subprogram you want to submit, SYSDATE as the date when the job will run, and an interval of ADDMONTHS(SYSDATE + 6) for semiannual submission.

```
DECLARE
    v_job    USER_JOBS.job%TYPE;
BEGIN
    DBMS_JOB.SUBMIT ( v_job, 'BEGIN DBMS_OUTPUT.PUT_LINE
                            (get_job_count(110)); END; ',
                    SYSDATE,
                    'ADD_MONTHS(SYSDATE, 6)');
    DBMS_JOB.RUN(v_job);
    DBMS_OUTPUT.PUT_LINE('JOB: ' || v_job ||
                        ' COMPLETED AT - ' || SYSDATE);
END;
/
```

Note: To force the job to run immediately, call DBMS_JOB.RUN(your_job_number) after calling DBMS_JOB.SUBMIT, . This executes the job waiting in the queue.

Execute the anonymous block.

- b. Check your results by querying the EMPLOYEES and JOB_HISTORY tables and querying the USER_JOBS dictionary view to see the status of your job submission.

```
SELECT job, what, schema_user, last_date, next_date, interval
FROM    USER_JOBS;
```

Part A: Additional Practice 23 Solutions

23. In this practice, create a trigger to ensure that the job ID of any new employee being hired to department 80 (the Sales department) is a sales manager or representative.

- a. Disable all the previously created triggers as discussed in question 16b.

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

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

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

- b. Create a trigger called CHK_SALES_JOB.

Fire the trigger before every row that is changed after insertions and updates to the JOB_ID column in the EMPLOYEES table. Check that the new employee has a job ID of SA_MAN or SA_REP in the EMPLOYEES table. Add exception handling and provide an appropriate message so that the update fails if the new job ID is not that of a sales manager or representative.

```
CREATE OR REPLACE TRIGGER chk_sales_job
BEFORE INSERT OR UPDATE OF job_id ON employees
FOR EACH ROW
DECLARE
    e_invalid_sales_job    EXCEPTION;
BEGIN
    IF :new.department_id = 80 THEN
        IF (:new.job_id NOT IN ( 'SA_MAN' , 'SA_REP')) THEN
            RAISE e_invalid_sales_job;
        END IF;
    END IF;
EXCEPTION
    WHEN e_invalid_sales_job THEN
        RAISE_APPLICATION_ERROR (-20444, 'This employee in department
            80 should be a Sales Manager or Sales Rep!');
END chk_sales_job;
/
```

Part A: Additional Practice 23 Solutions (continued)

- c. Test the trigger. You can use the following data:

```
UPDATE employees
  SET job_id = 'AD_VP'
  WHERE employee_id = 106;
```

```
UPDATE employees
  SET job_id = 'AD_VP'
  WHERE employee_id = 179;
```

```
UPDATE employees
  SET job_id = 'SA_MAN'
  WHERE employee_id = 179;
```

Hint: The middle statement should produce the error message specified in your trigger.

```
1 row updated.
UPDATE employees
*
ERROR at line 1:
ORA-20444: This employee in department 80 should be a Sales Manager or Sales Rep!
ORA-06512: at "SH9.CHK_SALES_JOB", line 11
ORA-04088: error during execution of trigger 'SH9.CHK_SALES_JOB'
1 row updated.
```

- d. Query the EMPLOYEES table to view the changes. Commit the changes.

```
SELECT job_id, department_id, salary
FROM   employees
WHERE  employee_id = 179;
```

- e. Enable all the triggers previously that you disabled, as discussed in question 16b.

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


Part A: Additional Practice 24 Solutions

24. In this practice, 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.

- a. Create a trigger called CHECK_SAL_RANGE.

Fire the trigger before every row that is changed when data is updated in the MIN_SALARY and MAX_SALARY columns in the JOBS table. For any minimum or maximum salary value that is changed, check that 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. Include exception handling to cover a salary range change that affects the record of any existing employee.

```
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, 'There are employees whose
        salary is out of the specified range. Can not update with
        the specified salary range.');
```

```
END check_sal_range;
/
```

- b. Test the trigger. You can use the following data:

```
SELECT * FROM jobs WHERE job_id = 'SY_ANAL';
SELECT employee_id, job_id, salary
  FROM employees
  WHERE job_id = 'SY_ANAL';
UPDATE jobs
  SET min_salary = 5000, max_salary = 7000
  WHERE job_id = 'SY_ANAL';
UPDATE jobs
  SET min_salary = 7000, max_salary = 18000
  WHERE job_id = 'SY_ANAL';
```

Part B: Additional Practice 1 Solutions

1. Run the script `buildvid1.sql` to create all of the required tables and sequences needed for this exercise.

Run the script `buildvid2.sql` to populate all the tables created through by the script `buildvid1.sql`

Part B: Additional Practice 2 Solutions

2. Create a package named VIDEO 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 of 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 out 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.

Part B: Additional Practice 2 Solutions

```
CREATE OR REPLACE PACKAGE video
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_member_id  IN rental.member_id%TYPE,
     p_title_id   IN rental.title_id%TYPE)
    RETURN DATE;

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

  PROCEDURE return_movie
    (p_title_id   IN rental.title_id%TYPE,
     p_copy_id    IN rental.copy_id%TYPE,
     p_status     IN title_copy.status%TYPE);
END video;
/
```

Part B: Additional Practice 2 Solutions (continued)

```
CREATE OR REPLACE PACKAGE BODY video
IS
    /* PRIVATE PROGRAMS */
    PROCEDURE exception_handler
        (p_code      IN  NUMBER,
         p_context    IN  VARCHAR2)
    IS
    BEGIN
        IF p_code = -1 THEN
            RAISE_APPLICATION_ERROR(-20001, 'The number is
                assigned to this member is already in use, try again.');
```

```
ELSIF p_code = -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_member_id IN reservation.member_id%TYPE,
     p_title_id   IN reservation.title_id%TYPE)
```

```
IS
```

```
    CURSOR rented_cur IS
```

```
        SELECT exp_ret_date
          FROM rental
         WHERE title_id = p_title_id
         AND act_ret_date IS NULL;
```

```
BEGIN
```

```
    INSERT INTO reservation (res_date, member_id, title_id)
      VALUES(SYSDATE, p_member_id, p_title_id);
```

```
    COMMIT;
```

```
    FOR rented_rec IN rented_cur LOOP
```

```
        DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on: '
            || rented_rec.exp_ret_date);
```

```
        EXIT WHEN rented_cur%found;
```

```
    END LOOP;
```

```
EXCEPTION
```

```
    WHEN OTHERS THEN
```

```
        exception_handler(SQLCODE, 'RESERVE_MOVIE');
```

```
END reserve_movie;
```

Part B: Additional Practice 2 Solutions (continued)

```
/* PUBLIC PROGRAMS */

PROCEDURE return_movie
(p_title_id    IN rental.title_id%TYPE,
 p_copy_id     IN rental.copy_id%TYPE,
 p_status      IN title_copy.status%TYPE)
IS
  v_dummy VARCHAR2(1);
  CURSOR res_cur IS
    SELECT *
      FROM reservation
     WHERE title_id = p_title_id;
BEGIN
  SELECT ''
    INTO v_dummy
  FROM title
 WHERE title_id = p_title_id;
  UPDATE rental
    SET act_ret_date = SYSDATE
  WHERE title_id = p_title_id
        AND copy_id = p_copy_id
        AND act_ret_date IS NULL;
  UPDATE title_copy
    SET status = UPPER(p_status)
  WHERE title_id = p_title_id
        AND copy_id = p_copy_id;
  FOR res_rec IN res_cur LOOP
    IF res_cur%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;
```

Part B: Additional Practice 2 Solutions (continued)

```
/* PUBLIC PROGRAMS */

FUNCTION new_rental
  (p_member_id IN rental.member_id%TYPE,
   p_title_id  IN rental.title_id%TYPE)
  RETURN DATE
IS
  CURSOR copy_cur IS
    SELECT *
      FROM title_copy
     WHERE title_id = p_title_id
     FOR UPDATE;
  v_flag  BOOLEAN := FALSE;
BEGIN
  FOR copy_rec IN copy_cur LOOP
    IF copy_rec.status = 'AVAILABLE' THEN
      UPDATE title_copy
        SET status = 'RENTED'
        WHERE CURRENT OF copy_cur;
      INSERT INTO rental(book_date, copy_id, member_id,
                        title_id, exp_ret_date)
        VALUES(SYSDATE, copy_rec.copy_id, p_member_id,
                p_title_id, SYSDATE + 3);

      v_flag := TRUE;
      EXIT;
    END IF;
  END LOOP;
  COMMIT;
  IF v_flag THEN
    RETURN (SYSDATE + 3);
  ELSE
    reserve_movie(p_member_id, p_title_id);
    RETURN NULL;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
END new_rental;
```

Part B: Additional Practice 2 Solutions (continued)

```
/* PUBLIC PROGRAMS */
FUNCTION new_rental
  (p_member_name IN member.last_name%TYPE,
   p_title_id    IN rental.title_id%TYPE)
  RETURN DATE
IS
  CURSOR copy_cur IS
    SELECT *
      FROM title_copy
     WHERE title_id = p_title_id
    FOR UPDATE;
  v_flag BOOLEAN := FALSE;
  p_member_id member.member_id%TYPE;
  CURSOR member_cur IS
    SELECT member_id, last_name, first_name
      FROM member
     WHERE LOWER(last_name) = LOWER(p_member_name)
     ORDER BY last_name, first_name;
BEGIN
  SELECT member_id
    INTO p_member_id
  FROM member
  WHERE lower(last_name) = lower(p_member_name);
  FOR copy_rec IN copy_cur LOOP
    IF copy_rec.status = 'AVAILABLE' THEN
      UPDATE title_copy
        SET status = 'RENTED'
      WHERE CURRENT OF copy_cur;
      INSERT INTO rental (book_date, copy_id, member_id,
                        title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, p_member_id,
                p_title_id, SYSDATE + 3);

      v_flag := TRUE;
      EXIT;
    END IF;
  END LOOP;
  COMMIT;
  IF v_flag THEN
    RETURN(SYSDATE + 3);
  ELSE
    reserve_movie(p_member_id, p_title_id);
    RETURN NULL;
  END IF;
```


Part B: Additional Practice 2 Solutions (continued)

```
/* NEW RENTAL CONTINUED FROM PRIOR PAGE */
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE(
      'Warning! More than one member by this name. ');
  FOR member_rec IN member_cur 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');
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;
/
```

Part B: Additional Practice 3 Solutions

3. The business hours for the video store are 8:00 a.m. to 10:00 p.m., Sunday through Friday, and 8:00 a.m. to 12:00 a.m. on Saturday. To ensure that the tables can only be modified 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 trigger.

Note: In order for your trigger to fail, you 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 p.m. to 8:00 a.m.

```
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;
/
```

Part B: Additional Practice 3 Solutions (continued)

```
CREATE OR REPLACE TRIGGER member_trig
  BEFORE INSERT OR UPDATE OR DELETE ON member
BEGIN
  time_check;
END;
/
CREATE OR REPLACE TRIGGER rental_trig
  BEFORE INSERT OR UPDATE OR DELETE ON rental
BEGIN
  time_check;
END;
/
CREATE OR REPLACE TRIGGER title_copy_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title_copy
BEGIN
  time_check;
END;
/
CREATE OR REPLACE TRIGGER title_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title
BEGIN
  time_check;
END;
/
CREATE OR REPLACE TRIGGER reservation_trig
  BEFORE INSERT OR UPDATE OR DELETE ON reservation
BEGIN
  time_check;
END;
/
```

Additional Practices: Table Descriptions and Data

Part A

The tables and data used in part A are the same as those in the appendix B, "Table Descriptions and Data."

Part B: Tables Used

```
SELECT * FROM tab;
```

TNAME	TABTYPE	CLUSTERID
MEMBER	TABLE	
RENTAL	TABLE	
RESERVATION	TABLE	
TITLE	TABLE	
TITLE_COPY	TABLE	

Part B: MEMBER Table

DESCRIBE member

Name	Null?	Type
MEMBER_ID	NOT NULL	NUMBER(10)
LAST_NAME	NOT NULL	VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
ADDRESS		VARCHAR2(100)
CITY		VARCHAR2(30)
PHONE		VARCHAR2(25)
JOIN_DATE	NOT NULL	DATE

SELECT * FROM member;

MEMBER_ID	LAST_NAME	FIRST_NAME	ADDRESS	CITY	PHONE	JOIN_DATE
101	Velasquez	Carmen	283 King Street	Seattle	587-99-6666	03-MAR-90
102	Ngao	LaDoris	5 Modrany	Bratislava	586-355-8882	08-MAR-90
103	Nagayama	Midori	68 Via Centrale	Sao Paolo	254-852-5764	17-JUN-91
104	Quick-To-See	Mark	6921 King Way	Lagos	63-559-777	07-APR-90
105	Ropeburn	Audry	86 Chu Street	Hong Kong	41-559-87	04-MAR-90
106	Urguhart	Molly	3035 Laurier Blvd.	Quebec	418-542-9988	18-JAN-91
107	Menchu	Roberta	Boulevard de Waterloo 41	Brussels	322-504-2228	14-MAY-90
108	Biri	Ben	398 High St.	Columbus	614-455-9863	07-APR-90
109	Catchpole	Antoinette	88 Alfred St.	Brisbane	616-399-1411	09-FEB-92
110	Haas	James	Chestnut Street	Boston	617-123-4567	06-MAR-01
111	Biri	Allan	Hiawatha Drive	New York	516-123-4567	06-MAR-01
112	Velasquez	Carmen	283 King Street	Seattle	587-99-6666	03-MAR-90
113	Ngao	LaDoris	5 Modrany	Bratislava	586-355-8882	08-MAR-90
114	Nagayama	Midori	68 Via Centrale	Sao Paolo	254-852-5764	17-JUN-91
MEMBER_ID	LAST_NAME	FIRST_NAME	ADDRESS	CITY	PHONE	JOIN_DATE
115	Quick-To-See	Mark	6921 King Way	Lagos	63-559-777	07-APR-90
116	Ropeburn	Audry	86 Chu Street	Hong Kong	41-559-87	04-MAR-90
117	Urguhart	Molly	3035 Laurier Blvd.	Quebec	418-542-9988	18-JAN-91
118	Menchu	Roberta	Boulevard de Waterloo 41	Brussels	322-504-2228	14-MAY-90
119	Biri	Ben	398 High St.	Columbus	614-455-9863	07-APR-90
120	Catchpole	Antoinette	88 Alfred St.	Brisbane	616-399-1411	09-FEB-92
121	Haas	James	Chestnut Street	Boston	617-123-4567	06-MAR-01
122	Biri	Allan	Hiawatha Drive	New York	516-123-4567	06-MAR-01

22 rows selected.

Part B: RENTAL Table

```
DESCRIBE rental
```

Name	Null?	Type
BOOK_DATE	NOT NULL	DATE
COPY_ID	NOT NULL	NUMBER(10)
MEMBER_ID	NOT NULL	NUMBER(10)
TITLE_ID	NOT NULL	NUMBER(10)
ACT_RET_DATE		DATE
EXP_RET_DATE		DATE

```
SELECT * FROM rental;
```

BOOK_DATE	COPY_ID	MEMBER_ID	TITLE_ID	ACT_RET_D	EXP_RET_D
05-MAR-01	2	101	93		07-MAR-01
04-MAR-01	3	102	95		06-MAR-01
03-MAR-01	1	101	98		05-MAR-01
02-MAR-01	1	106	97	04-MAR-01	04-MAR-01
03-MAR-01	1	101	92	04-MAR-01	05-MAR-01
06-MAR-01	2	110	98		09-MAR-01
05-MAR-01	2	101	93		07-MAR-01
04-MAR-01	3	102	95		06-MAR-01
03-MAR-01	1	101	98		05-MAR-01
02-MAR-01	1	106	97	04-MAR-01	04-MAR-01
03-MAR-01	1	101	92	04-MAR-01	05-MAR-01

11 rows selected.

Part B: RESERVATION Table

```
DESCRIBE reservation
```

Name	Null?	Type
RES_DATE	NOT NULL	DATE
MEMBER_ID	NOT NULL	NUMBER(10)
TITLE_ID	NOT NULL	NUMBER(10)

```
SELECT * FROM reservation;
```

RES_DATE	MEMBER_ID	TITLE_ID
05-MAR-01	101	93
04-MAR-01	106	102
06-MAR-01	110	98
05-MAR-01	101	93
04-MAR-01	106	102
06-MAR-01	110	98

6 rows selected.

Part B: TITLE Table

DESCRIBE title

Name	Null?	Type
TITLE_ID	NOT NULL	NUMBER(10)
TITLE	NOT NULL	VARCHAR2(60)
DESCRIPTION	NOT NULL	VARCHAR2(400)
RATING		VARCHAR2(4)
CATEGORY		VARCHAR2(20)
RELEASE_DATE		DATE

SELECT * FROM title;

TITLE_ID	TITLE	DESCRIPTION	RATI	CATEGORY	RELEASE_D
92	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
93	Alien Again	Another installment of science fiction history. Can the heroine save the planet from the alien life form?	R	SCIFI	19-MAY-95
94	The Glob	A meteor crashes near a small American town and unleashes carnivorous goo in this classic.	NR	SCIFI	12-AUG-95
95	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
96	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
97	Soda Gang	After discovering a cached of drugs, a young couple find themselves pitted against a vicious gang.	NR	ACTION	01-JUN-95
98	Interstellar Wars	Futuristic interstellar action movie. Can the rebels save the humans from the evil Empire?	PG	SCIFI	07-JUL-77
99	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
100	Alien Again	Another installment of science fiction history. Can the heroine save the planet from the alien life form?	R	SCIFI	19-MAY-95
101	The Glob	A meteor crashes near a small American town and unleashes carnivorous goo in this classic.	NR	SCIFI	12-AUG-95
102	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
103	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
104	Soda Gang	After discovering a cached of drugs, a young couple find themselves pitted against a vicious gang.	NR	ACTION	01-JUN-95
105	Interstellar Wars	Futuristic interstellar action movie. Can the rebels save the humans from the evil Empire?	PG	SCIFI	07-JUL-77

14 rows selected.

Part B: TITLE_COPY Table

```
DESCRIBE title_copy
```

Name	Null?	Type
COPY_ID	NOT NULL	NUMBER(10)
TITLE_ID	NOT NULL	NUMBER(10)
STATUS	NOT NULL	VARCHAR2(15)

```
SELECT * FROM title_copy;
```

COPY_ID	TITLE_ID	STATUS
1	92	AVAILABLE
1	93	AVAILABLE
2	93	RENTED
1	94	AVAILABLE
1	95	AVAILABLE
2	95	AVAILABLE
3	95	RENTED
1	96	AVAILABLE
1	97	AVAILABLE
1	98	RENTED
2	98	RENTED

11 rows selected.

