
Oracle9i: Develop PL/SQL Program Units

Student Guide • Volume 2

40056GC10
Production 1.0
July 2001
D33491

ORACLE®

Author

Nagavalli Pataballa

Technical Contributors and Reviewers

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

Publisher

Sheryl Domingue

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.

Contents

Preface

Curriculum Map

1 Overview of PL/SQL Subprograms

- Course Objectives 1-2
- Lesson Objectives 1-3
- Oracle Internet Platform 1-4
- PL/SQL Program Constructs 1-5
- Overview of Subprograms 1-6
- Block Structure for Anonymous PL/SQL Blocks 1-7
- Block Structure for PL/SQL Subprograms 1-8
- PL/SQL Subprograms 1-9
- Benefits of Subprograms 1-10
- Developing Subprograms by Using iSQL*Plus 1-11
- Invoking Stored Procedures and Functions 1-12
- Summary 1-13

2 Creating Procedures

- Objectives 2-2
- What Is a Procedure? 2-3
- Syntax for Creating Procedures 2-4
- Developing Procedures 2-5
- Formal Versus Actual Parameters 2-6
- Procedural Parameter Modes 2-7
- Creating Procedures with Parameters 2-8
- IN Parameters: Example 2-9
- OUT Parameters: Example 2-10
- Viewing OUT Parameters 2-12
- IN OUT Parameters 2-13
- Viewing IN OUT Parameters 2-14
- Methods for Passing Parameters 2-15
- DEFAULT Option for Parameters 2-16
- Examples of Passing Parameters 2-17
- Declaring Subprograms 2-18
- Invoking a Procedure from an Anonymous PL/SQL Block 2-19
- Invoking a Procedure from Another Procedure 2-20
- Handled Exceptions 2-21
- Unhandled Exceptions 2-23
- Removing Procedures 2-25
- Benefits of Subprograms 2-26
- Summary 2-27
- Practice 2 Overview 2-29

3 Creating Functions

- Objectives 3-2
- Overview of Stored Functions 3-3
- Syntax for Creating Functions 3-4
- Creating a Function 3-5
- Creating a Stored Function by Using iSQL*Plus 3-6
- Creating a Stored Function by Using iSQL*Plus: Example 3-7
- Executing Functions 3-8
- Executing Functions: Example 3-9
- Advantages of User-Defined Functions in SQL Expressions 3-10
- Invoking Functions in SQL Expressions: Example 3-11
- Locations to Call User-Defined Functions 3-12
- Restrictions on Calling Functions from SQL Expressions 3-13
- Restrictions on Calling from SQL 3-15
- Removing Functions 3-16
- Procedure or Function? 3-17
- Comparing Procedures and Functions 3-18
- Benefits of Stored Procedures and Functions 3-19
- Summary 3-20
- Practice 3 Overview 3-21

4 Managing Subprograms

- Objectives 4-2
- Required Privileges 4-3
- Granting Access to Data 4-4
- Using Invoker's-Rights 4-5
- Managing Stored PL/SQL Objects 4-6
- USER_OBJECTS 4-7
- List All Procedures and Functions 4-8
- USER_SOURCE Data Dictionary View 4-9
- List the Code of Procedures and Functions 4-10
- USER_ERRORS 4-11
- Detecting Compilation Errors: Example 4-12
- List Compilation Errors by Using USER_ERRORS 4-13
- List Compilation Errors by Using SHOW ERRORS 4-14
- DESCRIBE in iSQL*Plus 4-15
- Debugging PL/SQL Program Units 4-16
- Summary 4-17
- Practice 4 Overview 4-19

5 Creating Packages

- Objectives 5-2
- Overview of Packages 5-3
- Components of a Package 5-4
- Referencing Package Objects 5-5
- Developing a Package 5-6
- Creating the Package Specification 5-8
- Declaring Public Constructs 5-9
- Creating a Package Specification: Example 5-10
- Creating the Package Body 5-11
- Public and Private Constructs 5-12
- Creating a Package Body: Example 5-13
- Invoking Package Constructs 5-15
- Declaring a Bodiless Package 5-17
- Referencing a Public Variable from a Stand-Alone Procedure 5-18
- Removing Packages 5-19
- Guidelines for Developing Packages 5-20
- Advantages of Packages 5-21
- Summary 5-23
- Practice 5 Overview 5-26

6 More Package Concepts

- Objectives 6-2
- Overloading 6-3
- Overloading: Example 6-4
- Using Forward Declarations 6-7
- Creating a One-Time-Only Procedure 6-9
- Restrictions on Package Functions Used in SQL 6-10
- User Defined Package: taxes_pack 6-11
- Invoking a User-Defined Package Function from a SQL Statement 6-12
- Persistent State of Package Variables: Example 6-13
- Persistent State of Package Variables 6-15
- Controlling the Persistent State of a Package Cursor 6-15
- Executing PACK_CUR 6-17
- PL/SQL Tables and Records in Packages 6-18
- Summary 6-19
- Practice 6 Overview 6-20

7 Oracle Supplied Packages

- Objectives 7-2
- Using Supplied Packages 7-3
- Using Native Dynamic SQL 7-4
- Execution Flow 7-5
- Using the DBMS_SQL Package 7-6
- Using DBMS_SQL 7-8
- Using the EXECUTE IMMEDIATE Statement 7-9
- Dynamic SQL Using EXECUTE IMMEDIATE 7-11
- Using the DBMS_DDL Package 7-12
- Using DBMS_JOB for Scheduling 7-13
- DBMS_JOB Subprograms 7-14
- Submitting Jobs 7-15
- Changing Job Characteristics 7-17
- Running, Removing, and Breaking Jobs 7-18
- Viewing Information on Submitted Jobs 7-19
- Using the DBMS_OUTPUT Package 7-20
- Interacting with Operating System Files 7-21
- What Is the UTL_FILE Package? 7-22
- File Processing Using the UTL_FILE Package 7-23
- UTL_FILE Procedures and Functions 7-24
- Exceptions Specific to the UTL_FILE Package 7-25
- The FOPEN and IS_OPEN Functions 7-26
- Using UTL_FILE 7-27
- The UTL_HTTP Package 7-29
- Using the UTL_HTTP Package 7-30
- Using the UTL_TCP Package 7-31
- Oracle-Supplied Packages 7-32
- Summary 7-37
- Practice 7 Overview 7-38

8 Manipulating Large Objects

- Objectives 8-2
- What Is a LOB? 8-3
- Contrasting LONG and LOB Data Types 8-4
- Anatomy of a LOB 8-5
- Internal LOBs 8-6
- Managing Internal LOBs 8-7
- What Are BFILES? 8-8
- Securing BFILES 8-9
- A New Database Object: DIRECTORY 8-10
- Guidelines for Creating DIRECTORY Objects 8-11

- Managing BFILES 8-12
- Preparing to Use BFILES 8-13
- The BFILENAME Function 8-14
- Loading BFILES 8-15
- Migrating from LONG to LOB 8-17
- The DBMS_LOB Package 8-19
- DBMS_LOB.READ and DBMS_LOB.WRITE 8-22
- Adding LOB Columns to a Table 8-23
- Populating LOB Columns 8-24
- Updating LOB by Using SQL 8-26
- Updating LOB by Using DBMS_LOB in PL/SQL 8-27
- Selecting CLOB Values by Using SQL 8-28
- Selecting CLOB Values by Using DBMS_LOB 8-29
- Selecting CLOB Values in PL/SQL 8-30
- Removing LOBs 8-31
- Temporary LOBs 8-32
- Creating a Temporary LOB 8-33
- Summary 8-34
- Practice 8 Overview 8-35

9 Creating Database Triggers

- Objectives 9-2
- Types of Triggers 9-3
- Guidelines for Designing Triggers 9-4
- Database Trigger: Example 9-5
- Creating DML Triggers 9-6
- DML Trigger Components 9-7
- Firing Sequence 9-11
- Syntax for Creating DML Statement Triggers 9-13
- Creating DML Statement Triggers 9-14
- Testing SECURE_EMP 9-15
- Using Conditional Predicates 9-16
- Creating a DML Row Trigger 9-17
- Creating DML Row Triggers 9-18
- Using OLD and NEW Qualifiers 9-19
- Using OLD and NEW Qualifiers: Example Using Audit_Emp_Table 9-20
- Restricting a Row Trigger 9-21
- INSTEAD OF Triggers 9-22
- Creating an INSTEAD OF Trigger 9-23
- Creating an INSTEAD OF Trigger 9-26
- Differentiating Between Database Triggers and Stored Procedures 9-27
- Differentiating Between Database Triggers and Form Builder Triggers 9-28
- Managing Triggers 9-29
- DROP TRIGGER Syntax 9-30

Trigger Test Cases 9-31
Trigger Execution Model and Constraint Checking 9-32
Trigger Execution Model and Constraint Checking: Example 9-33
A Sample Demonstration for Triggers Using Package Constructs 9-34
After Row and After Statement Triggers 9-35
Demonstration: VAR_PACK Package Specification 9-36
Demonstration: Using the AUDIT_EMP Procedure 9-38
Summary 9-39
Practice 9 Overview 9-40

10 More Trigger Concepts

Objectives 10-2
Creating Database Triggers 10-3
Creating Triggers on DDL Statements 10-4
Creating Triggers on System Events 10-5
LOGON and LOGOFF Trigger Example 10-6
CALL Statements 10-7
Reading Data from a Mutating Table 10-8
Mutating Table: Example 10-9
Implementing Triggers 10-11
Controlling Security Within the Server 10-12
Controlling Security with a Database Trigger 10-13
Using the Server Facility to Audit Data Operations 10-14
Auditing by Using a Trigger 10-15
Enforcing Data Integrity Within the Server 10-16
Protecting Data Integrity with a Trigger 10-17
Enforcing Referential Integrity Within the Server 10-18
Protecting Referential Integrity with a Trigger 10-19
Replicating a Table Within the Server 10-20
Replicating a Table with a Trigger 10-21
Computing Derived Data within the Server 10-22
Computing Derived Values with a Trigger 10-23
Logging Events with a Trigger 10-24
Benefits of Database Triggers 10-26
Managing Triggers 10-27
Viewing Trigger Information 10-28
Using USER_TRIGGERS 10-29
Listing the Code of Triggers 10-30
Summary 10-31
Practice 10 Overview 10-32

11 Managing Dependencies

Objectives	11-2
Understanding Dependencies	11-3
Dependencies	11-4
Local Dependencies	11-5
A Scenario of Local Dependencies	11-6
Displaying Direct Dependencies by Using USER_DEPENDENCIES	11-7
Displaying Direct and Indirect Dependencies	11-8
Displaying Dependencies	11-9
Another Scenario of Local Dependencies	11-10
A Scenario of Local Naming Dependencies	11-11
Understanding Remote Dependencies	11-12
Concepts of Remote Dependencies	11-13
REMOTE_DEPENDENCIES_MODE Parameter	11-14
Remote Dependencies and Time Stamp Mode	11-15
Remote Procedure B Compiles at 8:00 a.m.	11-16
Local Procedure A Compiles at 9:00 a.m.	11-17
Execute Procedure A	11-18
Remote Procedure B Recompiled at 11:00 a.m.	11-19
Execute Procedure A	11-20
Signature Mode	11-21
Recompiling a PL/SQL Program Unit	11-22
Unsuccessful Recompile	11-23
Successful Recompile	11-24
Recompilation of Procedures	11-25
Packages and Dependencies	11-26
Summary	11-28
Practice 11 Overview	11-29

A PL/SQL Fundamentals Quiz

B PL/SQL Fundamentals Quiz Answers

C Practice Solutions

D Table Descriptions and Data

E Review of PL/SQL

F Creating Program Units by Using Procedure Builder

Index

Additional Practices

Additional Practice Solutions

Additional Practices: Table Descriptions and Data

Additional Practices

Additional Practices Overview

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

The additional practices comprise of two parts:

Part A provides supplemental practice to create stored procedures, functions, packages, and triggers, and to 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.

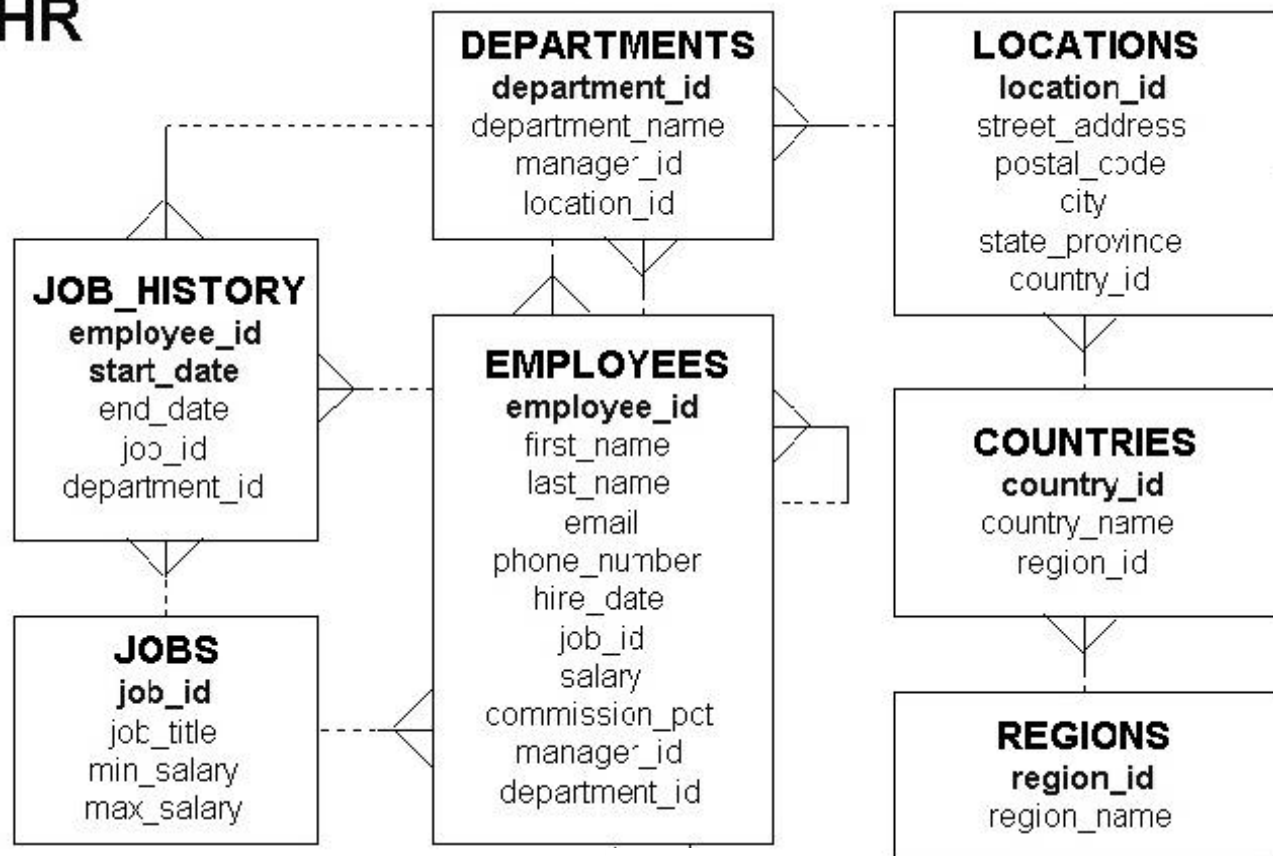
Part B is a case study which can be completed at the end of the course. This part supplements the practices for creating and managing program units. The tables used in the case study are based on a video database and contain the TITLE, TITLE_COPY, RENTAL, RESERVATION, and MEMBER tables.

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 create procedures.

1. 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

2. 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 1b.

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 1b) 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 (Refer to 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

3. 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)
EXECUTE upd_sal ('SY_ANAL', 7000, 14000)
```

```
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
```

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.

Part A

4. 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 or her 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.

5. 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

6. 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 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.

7. 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 2b.

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.

8. 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.

9. 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 2b.
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 2b.

10. 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';
```

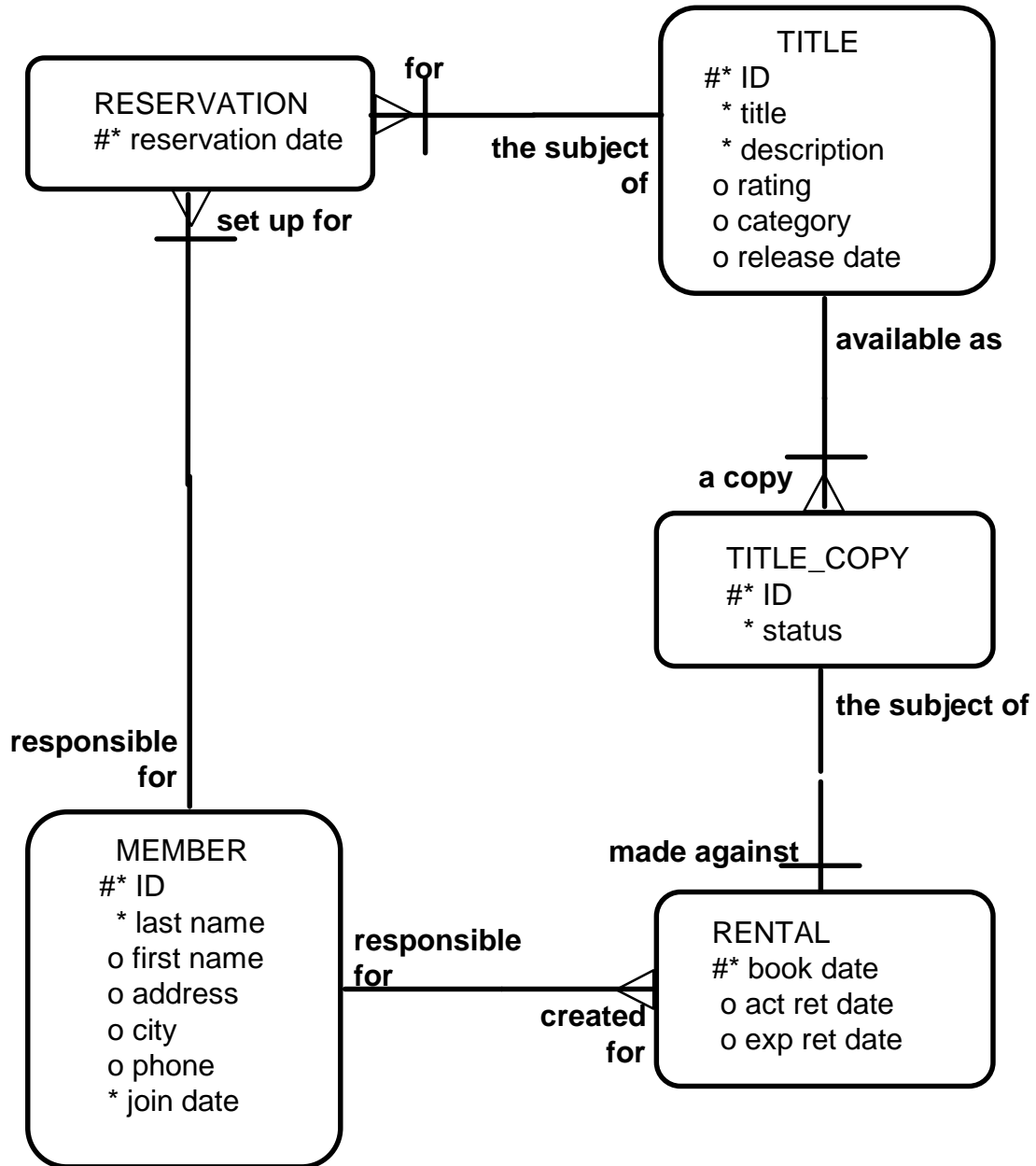
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

```
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 previous 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 Solutions

1. 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 2 Solutions

2. 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 1b.

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 1b) 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 2 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 3 Solutions

3. 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;
/
```


```