Contents

Preface

Oracle® PL/SQL by Example, *Fifth Edition,* presents the Oracle PL/SQL programming language in a unique and highly effective format. It challenges you to learn Oracle PL/SQL by using it rather than by simply reading about it.

Just as a grammar workbook would teach you about nouns and verbs by first showing you examples and then asking you to write sentences, *Oracle® PL/SQL by Example* teaches you about cursors, loops, procedures, triggers, and so on by first showing you examples and then asking you to create these objects yourself.

Who This Book Is For

This book is intended for anyone who needs a quick but detailed introduction to programming with Oracle's PL/SQL language. The ideal readers are those with some relational database experience, with some Oracle experience, specifically with SQL, SQL*Plus, and SQL Developer, but with little or no experience with PL/SQL or with most other programming languages.

The content of this book is based primarily on the material that was taught in an Introduction to PL/SQL class at Columbia University's Computer Technology and Applications (CTA) program in New York City. The student body was rather diverse, in that there were some students who had years of experience with information technology (IT) and programming, but no experience with Oracle PL/SQL, and then there were those with absolutely no experience in IT or programming. The content of the book, like the class, is balanced to meet the needs of both extremes. The additional exercises available through the companion website can be used as labs and homework assignments to accompany the lectures in such a PL/SQL course.

How This Book Is Organized

The intent of this workbook is to teach you about Oracle PL/SQL by explaining a programming concept or a particular PL/SQL feature and then illustrate it further by means of examples. Oftentimes, as the topic is discussed more in depth, these examples would be changed to illustrate newly covered material. In addition, most of the chapters of this book have Additional Exercises sections available through the companion website. These exercises allow you to test the depth of your understanding of the new material.

The basic structure of each chapter is as follows:

Objectives

Introduction

Lab

Lab . . .

Summary

The Objectives section lists topics covered in the chapter. Basically a single objective corresponds to a single Lab.

The Introduction offers a short overview of the concepts and features covered in the chapter.

Each Lab covers a single objective listed in the Objectives section of the chapter. In some instances the objective is divided even further into the smaller individual topics in the Lab. Then each such topic is explained and illustrated with the help of examples and corresponding outputs. Note that as much as possible, each example is provided in its entirety so that a complete code sample is readily available.

At the end of each chapter you will find a Summary section, which provides a brief conclusion of the material discussed in the chapter. In addition, the By the Way portion will state whether a particular chapter has an Additional Exercises section available on the companion website.

About the Companion Website

The companion Website is located at informit.com/title/0133796787. Here you will find three very important things:

- Files required to create and install the STUDENT schema.
- Files that contain example scripts used in the book chapters.

- Additional Exercises chapters, which have two parts:
  - A Questions and Answers part where you are asked about the material presented in a particular chapter along with suggested answers to these questions. Oftentimes, you are asked to modify a script based on some requirements and explain the difference in the output caused by these modifications. Note that this part is also organized into Labs similar to its corresponding chapter in the book.
  - A Try it Yourself part where you are asked to create scripts based on the requirements provided. This part is different from the Questions and Answers part in that there are no scripts supplied with the questions. Instead, you will need to create scripts in their entirety.

By the Way

You need to visit the companion website, download the student schema, and install it in your database prior to using this book if you would like the ability to execute the scripts provided in the chapters and on the site.

What You Will Need

There are software programs as well as knowledge requirements necessary to complete the Labs in this book. Note that some features covered throughout the book are applicable to Oracle 12*c* only. However, you will be able to run a great majority of the examples and complete Additional Exercises and Try it Yourself sections by using the following products:

- Oracle 11*g* or higher
- SQL Developer or SQL*Plus 11*g* or higher
- Access to the Internet

You can use either Oracle Personal Edition or Oracle Enterprise Edition to execute the examples in this book. If you use Oracle Enterprise Edition, it can be running on a remote server or locally on your own machine. It is recommended that you use Oracle 11*g* or Oracle 12*c* in order to perform all or a majority of the examples in this book. When a feature will only work in the latest version of Oracle database, the book will state so explicitly. Additionally, you should have access to and be familiar with SQL Developer or SQL*Plus.

You have a number of options for how to edit and run scripts in SQL Developer or from SQL*Plus. There are also many third-party programs to edit and debug PL/SQL code. Both, SQL Developer and SQL*Plus are used throughout this book, since these are two Oracle-provided tools and come as part of the Oracle installation.

By the Way

Chapter 1 has a Lab titled PL/SQL Development Environment that describes how to get started with SQL Developer and SQL*Plus. However, a great majority of the examples used in the book were executed in SQL Developer.

About the Sample Schema

The STUDENT schema contains tables and other objects meant to keep information about a registration and enrollment system for a fictitious university. There are ten tables in the system that store data about students, courses, instructors, and so on. In addition to storing contact information (addresses and telephone numbers) for students and instructors, and descriptive information about courses (costs and prerequisites), the schema also keeps track of the sections for particular courses, and the sections in which students have enrolled.

The SECTION table is one of the most important tables in the schema because it stores data about the individual sections that have been created for each course. Each section record also stores information about where and when the section will meet and which instructor will teach the section. The SECTION table is related to the COURSE and INSTRUCTOR tables.

The ENROLLMENT table is equally important because it keeps track of which students have enrolled in which sections. Each enrollment record also stores information about the student's grade and enrollment date. The enrollment table is related to the STUDENT and SECTION tables.

The STUDENT schema also has a number of other tables that manage grading for each student in each section.

The detailed structure of the STUDENT schema is described in Appendix B, Student Database Schema.

Acknowledgments

Ben Rosenzweig: I would like to thank my coauthor Elena Rakhimov for being a wonderful and knowledgeable colleague to work with. I would also like to thank Douglas Scherer for giving me the opportunity to work on this book as well as for providing constant support and assistance through the entire writing process. I am indebted to the team at Prentice Hall, which includes Greg Doench, Michelle Housley, and especially Songlin Qiu for her detailed edits. Finally, I would like to thank the many friends and family, especially Edward Clarin and Edward Knopping, for helping me through the long process of putting the whole book together, which included many late nights and weekends.

Elena Rakhimov: My contribution to this book reflects the help and advice of many people. I am particularly indebted to my coauthor Ben Rosenzweig for making this project a rewarding and enjoyable experience. Many thanks to Greg Doench, Michelle Housley, and especially Songlin Qiu for her meticulous editing skills, and many others at Prentice Hall who diligently worked to bring this book to market. Thanks to Michael Rinomhota for his invaluable expertise in setting up the Oracle environment and Dan Hotka for his valuable comments and suggestions. Most importantly, to my family, whose excitement, enthusiasm, inspiration, and support encouraged me to work hard to the very end, and were exceeded only by their love.

This page intentionally left blank

About the Authors

Benjamin Rosenzweig is a Senior Project Manager at Misys Financial Software, where he has worked since 2002. Prior to that he was a principal consultant for more than three years at Oracle Corporation in the Custom Development Department. His computer experience ranges from creating an electronic Tibetan–English Dictionary in Kathmandu, Nepal, to supporting presentation centers at Goldman Sachs and managing a trading system at TIAA-CREF. Benjamin has been an instructor at the Columbia University Computer Technology and Application program in New York City since 1998. In 2002 he was awarded the "Outstanding Teaching Award" from the Chair and Director of the CTA program. He holds a B.A. from Reed College and a certificate in database development and design from Columbia University. His previous books with Prentice Hall are *Oracle Forms Developer: The Complete Video Course* (2000), and *Oracle Web Application Programming for PL/SQL Developers* (2003).

Elena Rakhimov has over 20 years of experience in database architecture and development in a wide spectrum of enterprise and business environments ranging from non-profit organizations to Wall Street to her current position with a prominent software company where she heads up the database team. Her determination to stay "hands-on" notwithstanding, Elena managed to excel in the academic arena having taught relational database programming at Columbia University's highly esteemed Computer Technology and Applications program. She was educated in database analysis and design at Columbia University and in applied mathematics at Baku State University in Azerbaijan. She currently resides in Vancouver, Canada.

Introduction to PL/SQL New Features in Oracle 12*c*

Oracle 12*c* has introduced a number of new features and improvements for PL/SQL. This introduction briefly describes features not covered in this book and points you to specific chapters for features that are within the scope of this book. The list of features described here is also available in the "Changes in This Release for Oracle Database PL/SQL Language Reference" section of the PL/SQL Language Reference manual offered as part of Oracle's online help.

The new PL/SQL features and enhancements are as follows:

- Invoker's rights functions can be result-cached
- More PL/SQL-only data types can cross the PL/SQL-to-SQL interface clause
- `ACCESSIBLE BY` clause
- `FETCH FIRST` clause
- Roles can be granted to PL/SQL packages and stand-alone subprograms
- More data types have the same maximum size in SQL and PL/SQL
- Database triggers on pluggable databases
- `LIBRARY` can be defined as `DIRECTORY` object and with `CREDENTIAL` clause
- Implicit statement results
- `BEQUEATH CURRENT_USER` views
- `INHERIT PRIVILEGES` and `INHERIT ANY PRIVILEGES` privileges
- Invisible columns
- Objects, not types, are editioned or noneditioned

- PL/SQL functions that run faster in SQL
- Predefined inquiry directives `$$PLSQL_UNIT_OWNER` and `$$PLSQL_UNIT_TYPE`
- Compilation parameter `PLSQL_DEBUG` is deprecated

Invoker's Rights Functions Can Be Result-Cached

When a stored subprogram is created in Oracle products, it may be created as either a *definer rights* (DR) unit or an *invoker rights* (IR) unit. A DR unit would execute with the permissions of its owner, whereas an IR unit would execute with the permissions of a user who invoked that particular unit. By default, a stored subprogram is created as a DR unit unless explicitly specified otherwise. Whether a particular unit is considered a DR or IR unit is controlled by the `AUTHID` property, which may be set to either `DEFINER` (default) or `CURRENT_USER`.

Prior to Oracle 12*c*, functions created with the invoker rights clause (`AUTHID CURRENT_USER`) could not be result-cached. To create a function as an IR unit, the `AUTHID` clause must be added to the function specification.

A result-cached function is a function whose parameter values and result are stored in the cache. As a consequence, when such a function is invoked with the same parameter values, its result is retrieved from the cache instead of being computed again. To enable a function for result-caching, the `RESULT_CACHE` clause must be added to the function specification. This is demonstrated by the following example (the invoker rights clause and result-caching are highlighted in bold).

For Example *Result-Caching Functions Created with Invoker's Rights*

```
CREATE OR REPLACE FUNCTION get_student_rec (p_student_id IN NUMBER)
RETURN STUDENT%ROWTYPE
AUTHID CURRENT_USER
RESULT_CACHE RELIES_ON (student)
IS
  v_student_rec STUDENT%ROWTYPE;
BEGIN
  SELECT *
    INTO v_student_rec
    FROM student
   WHERE student_id = p_student_id;

  RETURN v_student_rec;
EXCEPTION
  WHEN no_data_found
  THEN
    RETURN NULL;
END get_student_rec;
/

-- Execute newly created function
DECLARE
  v_student_rec STUDENT%ROWTYPE;
```

```
BEGIN
  v_student_rec := get_student_rec (p_student_id => 230);
END;
```

Note that if the student record for student ID 230 is in the result cache already, then the function will return the student record from the result cache. In the opposite case, the student record will be selected from the `STUDENT` table and added to the cache for future use. Because the result cache of the function relies on the `STUDENT` table, any changes applied and committed on the `STUDENT` table will invalidate all cached results for the `get_student_rec` function.

More PL/SQL-Only Data Types Can Cross the PL/SQL-to-SQL Interface Clause

In this release, Oracle has extended support of PL/SQL-only data types to dynamic SQL and client programs (OCI or JDBC). For example, you can bind collections variables when using the `EXECUTE IMMEDIATE` statement or the `OPEN FOR`, `FETCH`, and `CLOSE` statements. This topic is covered in greater detail in Lab 18.3, Binding Collections in SQL Statements, in Chapter 18.

`ACCESSIBLE BY` Clause

An optional `ACCESSIBLE BY` clause enables you to specify a list of PL/SQL units that may access the PL/SQL unit being created or modified. The `ACCESSIBLE BY` clause is typically added to the module header—for example, to the function or procedure header. Each unit listed in the `ACCESSIBLE BY` clause is called an *accessor*, and the clause itself is also called a *white list*. This is demonstrated in the following example (the `ACCESSIBLE BY` clause is shown in bold).

For Example *Procedure Created with the `ACCESSIBLE BY` Clause*

```
CREATE OR REPLACE PROCEDURE test_proc1
ACCESSIBLE BY (TEST_PROC2)
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('TEST_PROC1');
END test_proc1;
/

CREATE OR REPLACE PROCEDURE test_proc2
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('TEST_PROC2');
  test_proc1;
END test_proc2;
/
```

```
-- Execute TEST_PROC2
BEGIN
  test_proc2;
END;
/

TEST_PROC2
TEST_PROC1

-- Execute TEST_PROC1 directly
BEGIN
  test_proc1;
END;
/

ORA-06550: line 2, column 4:
PLS-00904: insufficient privilege to access object TEST_PROC1
ORA-06550: line 2, column 4:
PL/SQL: Statement ignored
```

In this example, there are two procedures, `test_proc1` and `test_proc2`, and `test_proc1` is created with the `ACCESSIBLE BY` clause. As a consequence, `test_proc1` may be accessed by `test_proc2` only. This is demonstrated by two anonymous PL/SQL blocks. The first block executes `test_proc2` successfully. The second block attempts to execute `test_proc1` directly and, as a result, causes an error.

Note that both procedures were created within a single schema (`STUDENT`), and that both PL/SQL blocks were executed in the single session by the schema owner (`STUDENT`).

`FETCH FIRST` Clause

The `FETCH FIRST` clause is a new optional feature that is typically used with the "Top-N" queries as illustrated by the following example. The `ENROLLMENT` table used in this example contains student registration data. Each student is identified by a unique student ID and may be registered for multiple courses. The `FETCH FIRST` clause is shown in bold.

For Example *Using `FETCH FIRST` Clause with "Top-N" Query*

```
-- Sample student IDs from the ENROLLMENT table
SELECT student_id
  FROM enrollment;

STUDENT_ID
----------
       102
       102
       103
       104
       105
```

```
       106
       106
       107
       108
       109
       109
       110
       110
       …

-- "Top-N" query returns student IDs for the 5 students that registered for the most
-- courses
SELECT student_id, COUNT(*) courses
  FROM enrollment
GROUP BY student_id
ORDER BY courses desc
FETCH FIRST 5 ROWS ONLY;

STUDENT_ID       COURSES
----------       -------
       214             4
       124             4
       232             3
       215             3
       184             3
```

Note that FETCH FIRST clause may also be used in conjunction with the BULK COLLECT INTO clause as demonstrated here. The FETCH FIRST clause is shown in bold.

For Example *Using FETCH FIRST Clause with BULK COLLECT INTO Clause*

```
DECLARE
  TYPE student_name_tab IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;

  student_names student_name_tab;
BEGIN
  -- Fetching first 20 student names only
  SELECT first_name||' '||last_name
    BULK COLLECT INTO student_names
    FROM student
   FETCH FIRST 20 ROWS ONLY;

  DBMS_OUTPUT.PUT_LINE ('There are '||student_names.COUNT||' students');
END;
/
There are 20 students
```

Roles Can Be Granted to PL/SQL Packages and Stand-Alone Subprograms

Starting with Oracle 12*c*, you are able to grant roles to PL/SQL packages and stand-alone subprograms. Note that granting a role to a PL/SQL package or stand-alone subprogram does not alter its compilation. Instead, it affects how privileges required by the SQL statements that are issued by the PL/SQL unit at run time are checked.

Consider the following example where the READ role is granted to the function get_student_name.

For Example *Granting READ Role to the get_student_name Function*

```
GRANT READ TO FUNCTION get_student_name;
```

More Data Types Have the Same Maximum Size in SQL and PL/SQL

Prior to Oracle 12*c*, some data types had different maximum sizes in SQL and in PL/SQL. For example, in SQL the maximum size of NVARCHAR2 was 4000 bytes, whereas in PL/SQL it was 32,767 bytes. Starting with Oracle 12*c*, the maximum sizes of the VARCHAR2, NVARCHAR2, and RAW data types have been extended to 32,767 for both SQL and PL/SQL. To see these maximum sizes in SQL, the initialization parameter MAX_STRING_SIZE must be set to EXTENDED.

Database Triggers on Pluggable Databases

The pluggable database (PDB) is one of the components of Oracle's multitenant architecture. Typically it is a portable collection of schemas and other database objects. Starting with Oracle 12*c*, you are able to create event triggers on PDBs. Detailed information on triggers is provided in Chapters 13 and 14. Note that PDBs are outside the scope of this book, but detailed information on them may be found in Oracle's online Administration Guide.

LIBRARY Can Be Defined as a DIRECTORY Object and with a CREDENTIAL Clause

A LIBRARY is a schema object associated with a shared library of an operating system. It is created with the help of the CREATE OR REPLACE LIBRARY statement. A DIRECTORY is also an object that maps an alias to an actual directory on the server file system. The DIRECTORY object is covered very briefly in Chapter 25 as part of the install processes for the PL/SQL Profiler API and PL/SQL Hierarchical Profiler. In the Oracle 12*c* release, a LIBRARY object may be defined as a DIRECTORY object with an optional CREDENTIAL clause as shown here.

For Example *Creating `LIBRARY` as `DIRECTORY` Object*

```
CREATE OR REPLACE LIBRARY my_lib AS 'plsql_code' IN my_dir;
```

In this example, the `LIBRARY` object `my_lib` is created as a `DIRECTORY` object. The `'plsql_code'` is the name of the dynamic link library (DDL) in the `DIRECTORY` object `my_dir`. Note that for this library to be created successfully, the `DIRECTORY` object `my_dir` must be created beforehand. More information on `LIBRARY` and `DIRECTORY` objects can be found in Oracle's online Database PL/SQL Language Reference.

Implicit Statement Results

Prior to Oracle release 12*c*, result sets of SQL queries were returned explicitly from the stored PL/SQL subprograms via `REF CURSOR` out parameters. As a result, the invoker program had to bind to the `REF CURSOR` parameters and fetch the result sets explicitly as well.

Starting with this release, the `REF CURSOR` out parameters can be replaced by two procedures of the `DBMS_SQL` package, `RETURN_RESULT` and `GET_NEXT RESULT`. These procedures enable stored PL/SQL subprograms to return result sets of SQL queries implicitly, as illustrated in the following example (the reference to the `RETURN_RESULT` procedure is highlighted in bold):

For Example *Using `DBMS_SQL.RETURN_RESULT` Procedure*

```
CREATE OR REPLACE PROCEDURE test_return_result
AS
  v_cur  SYS_REFCURSOR;
BEGIN
  OPEN v_cur
   FOR
    SELECT first_name, last_name
      FROM instructor
     FETCH FIRST ROW ONLY;

  DBMS_SQL.RETURN_RESULT (v_cur);
END test_return_result;
/

BEGIN
  test_return_result;
END;
/
```

In this example, the `test_return_result` procedure returns the instructor's first and last names to the client application implicitly. Note that the cursor `SELECT` statement employs a `FETCH FIRST ROW ONLY` clause, which was introduced in Oracle 12*c* as well. To get the result set from the procedure `test_return_result` successfully, the client application must likewise be upgraded to Oracle 12*c*. Otherwise, the following error message is returned:

```
ORA-29481: Implicit results cannot be returned to client.
ORA-06512: at "SYS.DBMS_SQL", line 2785
ORA-06512: at "SYS.DBMS_SQL", line 2779
ORA-06512: at "STUDENT.TEST_RETURN_RESULT", line 10
ORA-06512: at line 2
```

`BEQUEATH CURRENT_USER` Views

Prior to Oracle 12*c*, a view could be created only as a definer rights unit. Starting with release 12*c*, a view may be created as an invoker's rights unit as well (this is similar to the `AUTHID` property of a stored subprogram). For views, however, this behavior is achieved by specifying a `BEQUEATH DEFINER` (default) or `BEQUEATH CURRENT_USER` clause at the time of its creation as illustrated by the following example (the `BEQUEATH CURRENT_USER` clause is shown in bold):

For Example *Creating View with `BEQUEATH CURRENT_USER` Clause*

```
CREATE OR REPLACE VIEW my_view
BEQUEATH CURRENT_USER
AS
  SELECT table_name, status, partitioned
    FROM user_tables;
```

In this example, `my_view` is created as an IR unit. Note that adding this property to the view does not affect its primary usage. Rather, similarly to the `AUTHID` property, it determines which set of permissions will be applied at the time when the data is selected from this view.

`INHERIT PRIVILEGES` and `INHERIT ANY PRIVILEGES` Privileges

Starting with Oracle 12*c*, an invoker's rights unit will execute with the invoker's permissions only if the owner of the unit has `INHERIT PRIVILEGES` or `INHERIT ANY PRIVILEGES` privileges. For example, before Oracle 12*c*, suppose user1 created a function F1 as an invoker's rights unit and granted execute privilege on it to user2, who happened to have more privileges than user1. Then when user2 ran function

F1, the function would run with the permissions of user2, potentially performing operations for which user1 might not have had permissions. This is no longer the case with Oracle 12*c*. As stated previously, such behavior must be explicitly specified via `INHERIT PRIVILEGES` or `INHERIT ANY PRIVILEGES` privileges.

Invisible Columns

Starting with Oracle 12*c*, it is possible to define and manipulate invisible columns. In PL/SQL, records defined as `%ROWTYPE` are aware of such columns, as illustrated by the following example (references to the invisible columns are shown in bold):

For Example *`%ROWTYPE` Records and Invisible Columns*

```
-- Make NUMERIC_GRADE column invisible
ALTER TABLE grade MODIFY (numeric_grade INVISIBLE);
/
table GRADE altered

DECLARE
  v_grade_rec grade%ROWTYPE;
BEGIN
  SELECT *
    INTO v_grade_rec
    FROM grade
   FETCH FIRST ROW ONLY;

  DBMS_OUTPUT.PUT_LINE ('student ID: '||v_grade_rec.student_id);
  DBMS_OUTPUT.PUT_LINE ('section ID: '||v_grade_rec.section_id);
  -- Referencing invisible column causes an error
  DBMS_OUTPUT.PUT_LINE ('grade:      '||v_grade_rec.numeric_grade);
END;
/
ORA-06550: line 12, column 54:
PLS-00302: component 'NUMERIC_GRADE' must be declared
ORA-06550: line 12, column 4:
PL/SQL: Statement ignored

-- Make NUMERIC_GRADE column visible
ALTER TABLE grade MODIFY (numeric_grade VISIBLE);
/
table GRADE altered

DECLARE
  v_grade_rec grade%ROWTYPE;
BEGIN
  SELECT *
    INTO v_grade_rec
    FROM grade
   FETCH FIRST ROW ONLY;

  DBMS_OUTPUT.PUT_LINE ('student ID: '||v_grade_rec.student_id);
  DBMS_OUTPUT.PUT_LINE ('section ID: '||v_grade_rec.section_id);
  -- This time the script executes successfully
  DBMS_OUTPUT.PUT_LINE ('grade:      '||v_grade_rec.numeric_grade);
END;
/
```

```
student ID: 123
section ID: 87
grade:      99
```

As you can gather from this example, the first run of the anonymous PL/SQL block did not complete due to the reference to the invisible column. Once the NUMERIC_GRADE column has been set to visible again, the script is able to complete successfully.

Objects, Not Types, Are Editioned or Noneditioned

An edition is a component of the edition-based redefinition feature that allows you to make a copy of an object—for example, a PL/SQL package—and make changes to it without affecting or invalidating other objects that may be dependent on it. With introduction of this feature, objects created in the database may be defined as editioned or noneditioned. For an object to be editioned, its object type must be editionable and it must have the EDITIONABLE property. Similarly, for an object to be noneditioned, its object type must be noneditioned or it must have the NONEDITIONABLE property.

Starting with Oracle 12*c*, you are able to specify whether a schema object is editionable or noneditionable in the CREATE OR REPLACE and ALTER statements. In this new release, a user (schema) that has been enabled for editions is able to own a noneditioned object even if its type is editionable in the database but noneditionable in the schema itself or if this object has NONEDITIONABLE property.

PL/SQL Functions That Run Faster in SQL

Starting with Oracle 12*c*, you can create user-defined functions that may run faster when they are invoked in the SQL statements. This may be accomplished as follows:

- User-defined function declared in the WITH clause of a SELECT statement
- User-defined function created with the UDF pragma

Consider the following example, where the format_name function is created in the WITH clause of the SELECT statement. This newly created function returns the formatted student name.

For Example *Creating a User-Defined Function in the WITH Clause*

```
WITH
  FUNCTION format_name (p_salutation IN VARCHAR2
                       ,p_first_name IN VARCHAR2
                       ,p_last_name  IN VARCHAR2)
```

```
  RETURN VARCHAR2
  IS
  BEGIN
    IF p_salutation IS NULL
    THEN
      RETURN p_first_name||' '||p_last_name;
    ELSE
      RETURN p_salutation||' '||p_first_name||' '||p_last_name;
    END IF;
  END;
SELECT format_name (salutation, first_name, last_name) student_name
  FROM student
 FETCH FIRST 10 ROWS ONLY;

STUDENT_NAME
-----------------
Mr. George Kocka
Ms. Janet Jung
Ms. Kathleen Mulroy
Mr. Joel Brendler
Mr. Michael Carcia
Mr. Gerry Tripp
Mr. Rommel Frost
Mr. Roger Snow
Ms. Z.A. Scrittorale
Mr. Joseph Yourish
```

Next, consider another example where the `format_name` function is created with the `UDF` pragma.

For Example *Creating a User-Defined Function in the `UDF` Pragma*

```
CREATE OR REPLACE FUNCTION format_name (p_salutation IN VARCHAR2
                                       ,p_first_name IN VARCHAR2
                                       ,p_last_name  IN VARCHAR2)
RETURN VARCHAR2
AS
  PRAGMA UDF;
BEGIN
  IF p_salutation IS NULL
  THEN
    RETURN p_first_name||' '||p_last_name;
  ELSE
    RETURN p_salutation||' '||p_first_name||' '||p_last_name;
  END IF;
END;
/
SELECT format_name (salutation, first_name, last_name) student_name
  FROM student
 FETCH FIRST 10 ROWS ONLY;

STUDENT_NAME
-----------------
Mr. George Kocka
Ms. Janet Jung
Ms. Kathleen Mulroy
Mr. Joel Brendler
Mr. Michael Carcia
Mr. Gerry Tripp
Mr. Rommel Frost
Mr. Roger Snow
Ms. Z.A. Scrittorale
Mr. Joseph Yourish
```

Predefined Inquiry Directives `$$PLSQL_UNIT_OWNER` and `$$PLSQL_UNIT_TYPE`

In PL/SQL, there are a number of predefined inquiry directives, as described in the following table (`$$PLSQL_UNIT_OWNER` and `$$PLSQL_UNIT_TYPE` are highlighted in bold):

Name	Description
`$$PLSQL_LINE`	The number of the code line where it appears in the PL/SQL subroutine.
`$$PLSQL_UNIT`	The name of the PL/SQL subroutine. For the anonymous PL/SQL blocks, it is set to `NULL`.
`$$PLSQL_UNIT_OWNER`	A new directive added in release 12*c*. This is the name of the owner (schema) of the PL/SQL subroutine. For anonymous PL/SQL blocks, it is set to `NULL`.
`$$PLSQL_UNIT_TYPE`	A new directive added in release 12*c*. This is the type of the PL/SQL subroutine—for example, `FUNCTION`, `PROCEDURE`, or `PACKAGE BODY`.
`$$plsql_compilation_parameter`	A set of PL/SQL compilation parameters, some of which are `PLSQL_CODE_TYPE`, which specifies the compilation mode for PL/SQL subroutines, and others of which are `PLSQL_OPTIMIZE_LEVEL` (covered in Chapter 25).

The following example demonstrates how directives may be used.

For Example *Using Predefined Inquiry Directives*

```
CREATE OR REPLACE PROCEDURE test_directives
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Procedure test_directives');
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_UNIT_OWNER: '||$$PLSQL_UNIT_OWNER);
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_UNIT_TYPE:  '||$$PLSQL_UNIT_TYPE);
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_UNIT:       '||$$PLSQL_UNIT);
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_LINE:       '||$$PLSQL_LINE);
END;
/

BEGIN
  -- Execute TEST_DERECTIVES procedure
  test_directives;
  DBMS_OUTPUT.PUT_LINE ('Anonymous PL/SQL block');
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_UNIT_OWNER: '||$$PLSQL_UNIT_OWNER);
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_UNIT_TYPE:  '||$$PLSQL_UNIT_TYPE);
```

```
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_UNIT:       '||$$PLSQL_UNIT);
  DBMS_OUTPUT.PUT_LINE ('$$PLSQL_LINE:       '||$$PLSQL_LINE);
END;
/

Procedure test_directives
$$PLSQL_UNIT_OWNER: STUDENT
$$PLSQL_UNIT_TYPE:  PROCEDURE
$$PLSQL_UNIT:       TEST_DIRECTIVES
$$PLSQL_LINE:       8
Anonymous PL/SQL block
$$PLSQL_UNIT_OWNER:
$$PLSQL_UNIT_TYPE:  ANONYMOUS BLOCK
$$PLSQL_UNIT:
$$PLSQL_LINE:       8
```

Compilation Parameter `PLSQL_DEBUG` Is Deprecated

Starting with Oracle release 12*c*, the `PLSQL_DEBUG` parameter is deprecated. To compile PL/SQL subroutines for debugging, the `PLSQL_OPTIMIZE_LEVEL` parameter should be set to 1. Chapter 25 covers the `PLSQL_OPTIMIZE_LEVEL` parameter and various optimization levels supported by the PL/SQL performance optimizer in greater detail.

This page intentionally left blank

3

SQL in PL/SQL

In this chapter, you will learn about

This chapter is a collection of some fundamental elements of using SQL statements in PL/SQL blocks. In the previous chapter, you initialized variables with the ":=" syntax; in this chapter, we will introduce the method of using a SQL select statement to update the value of a variable. These variables can then be used in DML statements (insert, delete, or update). Additionally, we will demonstrate how you can use a sequence in your DML statements within a PL/SQL block much as you would in a stand-alone SQL statement.

A transaction in Oracle is a series of SQL statements that have been grouped together into a logical unit by the programmer. A programmer chooses to do this to maintain data integrity. Each application (SQL*Plus, SQL Developer, and various third-party PL/SQL tools) maintains a single database session for each instance of a user login. The changes to the database that have been executed by a single application session are not actually "saved" into the database until a commit occurs. Work within a transaction up to and just prior to the commit can be rolled back; once a commit has been issued, however, work within that transaction cannot be rolled back. Note that those SQL statements should be either committed or rejected as a group.

To exert transaction control, a SAVEPOINT statement can be used to break down large PL/SQL statements into individual units that are easier to manage. In this chapter, we will cover the basic elements of transaction control so you will know how to manage your PL/SQL code through use of the COMMIT, ROLLBACK, and (principally) SAVEPOINT statement.

Lab 3.1: DML Statements in PL/SQL

After this lab, you will be able to

- Initialize Variables with SELECT INTO
- Use the SELECT INTO Syntax for Variable Initialization
- Use DML in a PL/SQL Block
- Make Use of a Sequence in a PL/SQL Block

Initialize Variables with SELECT INTO

In PL/SQL, there are two main methods of giving values to variables in a PL/SQL block. The first one, which you learned in Chapter 1, is initialization with the ":=" syntax. In this lab we will learn how to initialize a variable with a select statement by making use of the SELECT INTO syntax.

A variable that has been declared in the declaration section of the PL/SQL block can later be given a value with a select statement. The correct syntax is as follows:

```
SELECT item_name
INTO   variable_name
FROM   table_name;
```

Note that any single row function can be performed on the item to give the variable a calculated value.

For Example *ch03_1a.sql*

```
SET SERVEROUTPUT ON
DECLARE
  v_average_cost VARCHAR2(10);
```

```
BEGIN
  SELECT TO_CHAR(AVG(cost), '$9,999.99')
    INTO v_average_cost
    FROM course;
  DBMS_OUTPUT.PUT_LINE('The average cost of a '||
    'course in the CTA program is '||
    v_average_cost);
END;
```

In this example, a variable is given the value of the average cost of a course in the `course` table. First, the variable must be declared in the declaration section of the PL/SQL block. In this example, the variable is given the data type of `VARCHAR2(10)` because of the functions used on the data. The select statement that would produce this outcome in SQL*Plus would be

```
SELECT TO_CHAR(AVG(cost), '$9,999.99')
FROM   course;
```

The `TO_CHAR` function is used to format the cost; in doing this, the number data type is converted to a character data type. Once the variable has a value, it can be displayed to the screen using the `PUT_LINE` procedure of the `DBMS_OUTPUT` package. The output of this PL/SQL block would be:

```
The average cost of a course in the CTA program
is $1,198.33
PL/SQL procedure successfully completed.
```

In the declaration section of the PL/SQL block, the variable `v_average_cost` is declared as a `varchar2`. In the executable section of the block, this variable is given the value of the average cost from the `course` table by means of the `SELECT INTO` syntax. The SQL function `TO_CHAR` is issued to format the number. The `DBMS_OUTPUT` package is then used to show the result to the screen.

Using the `SELECT INTO` Syntax for Variable Initialization

The previous PL/SQL block may be rearranged so the `DBMS_OUTPUT` section is placed before the `SELECT INTO` statement.

For Example *ch03_1a.sql*

```
SET SERVEROUTPUT ON
DECLARE
  v_average_cost VARCHAR2(10);
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('The average cost of a '||
    'course in the CTA program is '||
    v_average_cost);
  SELECT TO_CHAR(AVG(cost), '$9,999.99')
    INTO v_average_cost
    FROM course;
END;
```

You will then see the following result:

```
The average cost of a course in the CTA program is
PL/SQL procedure successfully completed.
```

The variable `v_average_cost` will be set to `NULL` when it is first declared. Because the `DBMS_OUTPUT` section precedes the point at which the variable is given a value, the output for the variable will be `NULL`. After the `SELECT INTO` statement, the variable will be given the same value as in the original block, but it will not be displayed because there is not another `DBMS_OUTPUT` line in the PL/SQL block.

Data Definition Language (DDL) statements are not valid in a simple PL/SQL block (more advanced techniques such as procedures in the `DBMS_SQL` package will enable you to make use of DDL), yet data manipulation (using Data Manipulation Language [DML]) is easily achieved either by using variables or by simply putting a DML statement into a PL/SQL block. Here is an example of a PL/SQL block that updates an existing entry in the `zipcode` table.

For Example *ch03_2a.sql*

```
SET SERVEROUTPUT ON
DECLARE
  v_city zipcode.city%TYPE;
BEGIN
  SELECT 'COLUMBUS'
    INTO v_city
    FROM dual;
  UPDATE zipcode
     SET city = v_city
   WHERE ZIP = 43224;
END;
```

It is also possible to insert data into a database table in a PL/SQL block, as shown in the following example.

For Example *ch03_3a.sql*

```
DECLARE
  v_zip zipcode.zip%TYPE;
  v_user zipcode.created_by%TYPE;
  v_date zipcode.created_date%TYPE;
```

```
BEGIN
  SELECT 43438, USER, SYSDATE
    INTO v_zip, v_user, v_date
    FROM dual;
  INSERT INTO zipcode
    (ZIP, CREATED_BY ,CREATED_DATE, MODIFIED_BY,
     MODIFIED_DATE
    )
     VALUES(v_zip, v_user, v_date, v_user, v_date);
END;
```

By the Way

SELECT statements in PL/SQL that return no rows or too many rows will cause an error to occur that can be trapped by using an exception. You will learn more about handling exceptions in Chapters 8, 9, and 10.

Using DML in a PL/SQL Block

This section demonstrates how DML is used in PL/SQL. The following PL/SQL block inserts a new student into the `student` table.

For Example *ch03_4a.sql*

```
BEGIN
  SELECT MAX(student_id)
    INTO v_max_id
    FROM student;
  INSERT into student
    (student_id, last_name, zip,
     created_by, created_date,
     modified_by, modified_date,
     registration_date
    )
   VALUES (v_max_id + 1, 'Rosenzweig',
           11238, 'BROSENZ ', '01-JAN-2014',
           'BROSENZ', '10-JAN-2014', '15-FEB-2014'
          );
END;
```

To generate a unique ID, the maximum `student_id` is selected into a variable and then incremented by 1. In this example, there is a foreign key on the `zip` item in the student table, which means that the ZIP code you choose to enter must be in the `zipcode` table.

Using an Oracle Sequence

An Oracle sequence is an Oracle database object that can be used to generate unique numbers. You can use sequences to generate primary key values automatically.

Accessing and Incrementing Sequence Values

Once a sequence is created, you can access its values in SQL statements with these pseudocolumns:

- `CURRVAL`: Returns the current value of the sequence.
- `NEXTVAL`: Increments the sequence and returns the new value.

The following example creates the sequence `eseq`.

For Example

```
CREATE SEQUENCE eseq
  INCREMENT BY 10
```

The first reference to `ESEQ.NEXTVAL` returns 1. The second returns 11. Each subsequent reference will return a value 10 greater than the one previous.

(Even though you will be guaranteed unique numbers, you are not guaranteed contiguous numbers. In some systems this may be a problem—for example, when generating invoice numbers.)

Drawing Numbers from a Sequence

A sequence value can be inserted directly into a table without first selecting it. (In very old versions of Oracle prior to Oracle 7.3, it was necessary to use the `SELECT INTO` syntax and put the new sequence number into a variable; you could then insert the variable.)

For this example, a table called `test01` will be used. The table `test01` is first created, followed by the sequence `test_seq`. Then the sequence is used to populate the table.

For Example *ch03_5a.sql*

```
CREATE TABLE test01 (col1 number);
CREATE SEQUENCE test_seq
  INCREMENT BY 5;
BEGIN
  INSERT INTO test01
    VALUES (test_seq.NEXTVAL);
END;
/
Select * FROM test01;
```

Using a Sequence in a PL/SQL Block

In this example, a PL/SQL block is used to insert a new student in the `student` table. The PL/SQL code makes use of two variables, `USER` and `SYSDATE`, that are used in the select statement. The existing `student_id_seq` sequence is used to generate a unique ID for the new student.

For Example *ch03_6a.sql*

```
DECLARE
  v_user student.created_by%TYPE;
  v_date student.created_date%TYPE;
BEGIN
  SELECT USER, sysdate
    INTO v_user, v_date
    FROM dual;
 INSERT INTO student
   (student_id, last_name, zip,
    created_by, created_date, modified_by,
    modified_date, registration_date
   )
   VALUES (student_id_seq.nextval, 'Smith',
           11238, v_user, v_date, v_user, v_date,
           v_date
          );
END;
```

In the declaration section of the PL/SQL block, two variables are declared. They are both set to be data types within the `student` table using the `%TYPE` method of declaration. This ensures the data types match the columns of the tables into which they will be inserted. The two variables `v_user` and `v_date` are given values from the system by means of `SELECT INTO` statements. The value of the `student_id` is generated by using the next value of the `student_id_seq` sequence.

Lab 3.2: Transaction Control in PL/SQL

After this lab, you will be able to

- Use the `COMMIT`, `ROLLBACK`, and `SAVEPOINT` Statements
- Put Together DML and Transaction Control

Using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`

Transactions are a means to break programming code into manageable units. Grouping transactions into smaller elements is a standard practice that ensures an application will save only correct data. Initially, any application will have to connect to the database to access the data. When a user is issuing DML statements in an application, however, these changes are not visible to other users until a `COMMIT` or `ROLLBACK` has been issued. The Oracle platform guarantees a read-consistent view of the data. Until that point, all data that have been inserted or updated will be held in memory and will be available only to the current user. The rows that have been changed will be locked by the current user and will not be available for updating to other users until the locks have been released. A COMMIT or ROLLBACK statement will release these locks. Transactions can be controlled more readily by marking points of the transaction with the SAVEPOINT command.

- COMMIT: Makes events within a transaction permanent.
- ROLLBACK: Erases events within a transaction.

Additionally, you can use a SAVEPOINT to control transactions. Transactions are defined in the PL/SQL block from one SAVEPOINT to another. The use of the SAVEPOINT command allows you to break your SQL statements into units so that in a given PL/SQL block, some units can be committed (saved to the database), others can be rolled back (undone), and so forth.

By the Way

The Oracle platform makes a distinction between a transaction and a PL/SQL block. The start and end of a PL/SQL block do not necessarily mean the start and end of a transaction.

To demonstrate the need for transaction control, we will examine a two-step data manipulation process. Suppose that the fees for all courses in the CTA database that have a prerequisite course need to be increased by 10 percent; at the same time, all courses that do not have a prerequisite need to be decreased by 10 percent. This is a two-step process. If the first step is successful but the second step is not, then the data concerning course cost would be inconsistent in the database. Because this adjustment is based on a change in percentage, there would be no way to track which part of this course adjustment was successful and which part was not.

In the following example, one PL/SQL block performs two updates on the cost item in the course table. In the first step (this code is commented for the purpose of emphasizing each update), the cost is updated with a cost that is 10 percent less whenever the course does not have a prerequisite. In the second step, the cost is increased by 10 percent whenever the course has a prerequisite.

For Example *ch03_7a.sql*

```
BEGIN
-- STEP 1
   UPDATE course
      SET cost = cost - (cost * 0.10)
     WHERE prerequisite IS NULL;
```

```
-- STEP 2
   UPDATE course
      SET cost = cost + (cost * 0.10)
    WHERE prerequisite IS NOT NULL;
END;
```

Let's assume that the first update statement succeeds, but the second update statement fails because the network went down. The data in the course table is now inconsistent because courses with no prerequisite have had their cost reduced but courses with prerequisites have not been adjusted. To prevent this sort of situation, statements must be combined into a transaction. Thus either both statements will succeed or both statements will fail.

A transaction usually combines SQL statements that represent a logical unit of work. The transaction begins with the first SQL statement issued after the previous transaction, or with the first SQL statement issued after connecting to the database. The transaction ends with the `COMMIT` or `ROLLBACK` statement.

COMMIT

When a `COMMIT` statement is issued to the database, the transaction has ended, and the following results are true:

- All work done by the transaction becomes permanent.
- Other users can see changes in data made by the transaction.
- Any locks acquired by the transaction are released.

A `COMMIT` statement has the following syntax:

```
COMMIT [WORK];
```

The word `WORK` is optional and is used to improve readability. Until a transaction is committed, only the user executing that transaction can see changes in the data made by his or her session.

Suppose User A issues the following command on a `student` table that exists in another schema but has a public synonym of `student`:

For Example *ch03_8a.sql*

```
BEGIN
INSERT INTO student
  (student_id, last_name, zip, registration_date,
   created_by, created_date, modified_by,
   modified_date
  )
```

```
  VALUES (student_id_seq.nextval, 'Tashi', 10015,
          '01-JAN-99', 'STUDENTA', '01-JAN-99',
          'STUDENTA', '01-JAN-99'
         );
END;
```

Then User B enters the following command to query the table known by its public synonym `student`, while logged on to his session.

```
SELECT *
FROM student
WHERE last_name = 'Tashi';
```

Then User A issues the following command:

```
COMMIT;
```

Now if User B enters the same query again, he will not see the same results.

In this example, there are two sessions: User A and User B. User A inserts a record into the `student` table. User B queries the `student` table, but does not get the record that was inserted by User A. User B cannot see the information because User A has not committed the work. When User A commits the transaction, User B, upon resubmitting the query, sees the records inserted by User A.

ROLLBACK

When a `ROLLBACK` statement is issued to the database, the transaction has ended, and the following results are true:

- All work done by the transaction is undone, as if it hadn't been issued.
- Any locks acquired by the transaction are released.

A `ROLLBACK` statement has the following syntax:

```
ROLLBACK [WORK];
```

The `WORK` keyword is optional and provides for increased readability.

SAVEPOINT

The `ROLLBACK` statement undoes all work done by the user in a specific transaction. With the `SAVEPOINT` command, however, only part of the transaction can be undone. A `SAVEPOINT` command has the following syntax:

```
SAVEPOINT name;
```

The word name is the SAVEPOINT statement's name. Once a SAVEPOINT is defined, the program can roll back to that SAVEPOINT. A ROLLBACK statement, then, has the following syntax:

```
ROLLBACK [WORK] to SAVEPOINT name;
```

When a ROLLBACK to SAVEPOINT statement is issued to the database, the following results are true:

- Any work done since the SAVEPOINT is undone. The SAVEPOINT remains active, however, until a full COMMIT or ROLLBACK is issued. It can be rolled back again, if desired.
- Any locks and resources acquired by the SQL statements since the SAVEPOINT will be released.
- The transaction is not finished, because SQL statements are still pending.

Putting Together DML and Transaction Control

This section combines all the elements of transaction control that have been covered in this chapter. The following piece of code is an example of a PL/SQL block with three SAVEPOINTs.

For Example *ch03_9a.sql*

```
BEGIN
  INSERT INTO student
    ( student_id, Last_name, zip, registration_date,
      created_by, created_date, modified_by,
      modified_date
    )
    VALUES ( student_id_seq.nextval, 'Tashi', 10015,
             '01-JAN-99', 'STUDENTA', '01-JAN-99',
             'STUDENTA','01-JAN-99'
           );
  SAVEPOINT A;
  INSERT INTO student
    ( student_id, Last_name, zip, registration_date,
      created_by, created_date, modified_by,
      modified_date
    )
    VALUES (student_id_seq.nextval, 'Sonam', 10015,
            '01-JAN-99', 'STUDENTB','01-JAN-99',
            'STUDENTB', '01-JAN-99'
           );
  SAVEPOINT B;
  INSERT INTO student
   ( student_id, Last_name, zip, registration_date,
     created_by, created_date, modified_by,
     modified_date
   )
```

```
    VALUES (student_id_seq.nextval, 'Norbu', 10015,
            '01-JAN-99', 'STUDENTB', '01-JAN-99',
            'STUDENTB', '01-JAN-99'
           );
  SAVEPOINT C;
  ROLLBACK TO B;
END;
```

If you were to run the following SELECT statement immediately after running the preceding example, you would not be able to see any data because the ROLLBACK to (SAVEPOINT) B has undone the last insert statement where the student Norbu was inserted.

```
SELECT *
FROM student
WHERE last_name = 'Norbu';
```

The result would be "no rows selected."

Three students were inserted in this PL/SQL block: first Tashi in SAVEPOINT A, then Sonam in SAVEPOINT B, and finally Norbu in SAVEPOINT C. When the command to roll back to B was issued, the insert of Norbu was undone.

If the following command was entered after the script ch03_9a.sql, then the insert in SAVEPOINT B would be undone—that is, the insert of Sonam:

```
ROLLBACK to SAVEPOINT A;
```

Tashi was the only student that was successfully entered into the database. The ROLLBACK to SAVEPOINT A undid the insert statements for Norbu and Sonam.

By the Way

SAVEPOINT is often used before a complicated section of the transaction. If this part of the transaction fails, it can be rolled back, allowing the earlier part to continue.

Did You Know?

It is important to note the distinction between transactions and PL/SQL blocks. When a block starts, it does not mean that the transaction starts. Likewise, the start of the transaction need not coincide with the start of a block.

Here is an example of a single PL/SQL block with multiple transactions.

For Example *ch03_10a.sql*

```
DECLARE
   v_Counter NUMBER;
 BEGIN
   v_counter := 0;
   FOR i IN 1..100
   LOOP
     v_counter := v_counter + 1;
     IF v_counter = 10
     THEN
       COMMIT;
       v_counter := 0;
     END IF;
   END LOOP;
 END;
```

In this example, as soon as the value of `v_counter` becomes equal to 10, the work is committed. Thus there will be a total of 10 transactions contained in this one PL/SQL block.

Summary

In this chapter, you learned how to make use of variables and the various ways to populate variables. Use of DML (Data Manipulation Language) within a PL/SQL block was illustrated in examples with insert statements. These examples also made use of sequences to generate unique numbers.

The last section of the chapter covered transactional control in PL/SQL by explaining what it means to commit data as well as how `SAVEPOINT`s are used. The final examples demonstrated how committed data could be reversed by using `ROLLBACK`s in conjunction with `SAVEPOINT`s.

By the Way

The companion website provides additional exercises and suggested answers for this chapter, with discussion related to how those answers resulted. The main purpose of these exercises is to help you test the depth of your understanding by utilizing all of the skills that you have acquired throughout this chapter.

This page intentionally left blank

Index