



Oracle[®] PL/SQL

by Example

Fifth Edition

- ▶ Updated for Oracle 12c
- ▶ Hundreds of examples, questions, and answers
- ▶ Real-life labs
- ▶ No Oracle PL/SQL experience necessary
- ▶ Build PL/SQL Applications—NOW

BENJAMIN ROSENZWEIG • ELENA RAKHIMOV

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Oracle® PL/SQL by Example

Fifth Edition

Benjamin Rosenzweig
Elena Rakhimov



Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data
Rosenzweig, Benjamin.

Oracle PL/SQ® by example / Benjamin Rosenzweig, Elena Rakhimov.—Fifth edition.

pages cm

Includes index.

ISBN 978-0-13-379678-0 (pbk. : alk. paper)—ISBN 0-13-379678-7 (pbk. : alk. paper)

1. PL/SQL (Computer program language) 2. Oracle (Computer file) 3. Relational databases.

I. Rakhimov, Elena Silvestrova. II. Title.

QA76.73.P258R68 2015

005.75'6—

dc23

2014045792

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-379678-0

ISBN-10: 0-13-379678-7

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing, February 2015

*To my parents, Rosie and Sandy Rosenzweig,
for their love and support
—Benjamin Rosenzweig*

*To my family, for their excitement and encouragement
—Elena Rakhimov*

Contents

[Preface](#)

[Acknowledgments](#)

[About the Authors](#)

[Introduction to PL/SQL New Features in Oracle 12c](#)

[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 a DIRECTORY Object and with a 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](#)

[Chapter 1 PL/SQL Concepts](#)

[Lab 1.1: PL/SQL Architecture](#)

[PL/SQL Architecture](#)

[PL/SQL Block Structure](#)

[How PL/SQL Gets Executed](#)

[Lab 1.2: PL/SQL Development Environment](#)

[Getting Started with SQL Developer](#)

[Getting Started with SQL*Plus](#)

[Executing PL/SQL Scripts](#)

[Lab 1.3: PL/SQL: The Basics](#)

[DBMS_OUTPUT.PUT_LINE Statement](#)

[Substitution Variable Feature](#)

[Summary](#)

[Chapter 2 PL/SQL Language Fundamentals](#)

[Lab 2.1: PL/SQL Programming Fundamentals](#)

[PL/SQL Language Components](#)

[PL/SQL Variables](#)

[PL/SQL Reserved Words](#)

[Identifiers in PL/SQL](#)

[Anchored Data Types](#)

[Declare and Initialize Variables](#)

[Scope of a Block, Nested Blocks, and Labels](#)

[Summary](#)

[Chapter 3 SQL in PL/SQL](#)

[Lab 3.1: DML Statements in PL/SQL](#)

[Initialize Variables with SELECT INTO](#)

[Using the SELECT INTO Syntax for Variable Initialization](#)

[Using DML in a PL/SQL Block](#)

[Using a Sequence in a PL/SQL Block](#)

[Lab 3.2: Transaction Control in PL/SQL](#)

[Using COMMIT, ROLLBACK, and SAVEPOINT](#)

[Putting Together DML and Transaction Control](#)

[Summary](#)

[Chapter 4 Conditional Control: IF Statements](#)

[Lab 4.1: IF Statements](#)

[IF-THEN Statements](#)

[IF-THEN-ELSE Statement](#)

[Lab 4.2: ELSIF Statements](#)

[Lab 4.3: Nested IF Statements](#)

[Summary](#)

[Chapter 5 Conditional Control: CASE Statements](#)

[Lab 5.1: CASE Statements](#)

[CASE Statements](#)

[Searched CASE Statements](#)

[Lab 5.2: CASE Expressions](#)

[Lab 5.3: NULLIF and COALESCE Functions](#)

[NULLIF Function](#)

[COALESCE Function](#)

[Summary](#)

[Chapter 6 Iterative Control: Part I](#)

[Lab 6.1: Simple Loops](#)

[EXIT Statement](#)

[EXIT WHEN Statement](#)

[Lab 6.2: WHILE Loops](#)

[Using WHILE Loops](#)

[Premature Termination of the WHILE Loop](#)

[Lab 6.3: Numeric FOR Loops](#)

[Using the IN Option in the Loop](#)

[Using the REVERSE Option in the Loop](#)

[Premature Termination of the Numeric FOR Loop](#)

[Summary](#)

[Chapter 7 Iterative Control: Part II](#)

[Lab 7.1: CONTINUE Statement](#)

[Using CONTINUE Statement](#)

[CONTINUE WHEN Statement](#)

[Lab 7.2: Nested Loops](#)

[Using Nested Loops](#)

[Using Loop Labels](#)

[Summary](#)

[Chapter 8 Error Handling and Built-in Exceptions](#)

[Lab 8.1: Handling Errors](#)

[Lab 8.2: Built-in Exceptions](#)

[Summary](#)

[Chapter 9 Exceptions](#)

[Lab 9.1: Exception Scope](#)

[Lab 9.2: User-Defined Exceptions](#)

[Lab 9.3: Exception Propagation](#)

[Re-raising Exceptions](#)

[Summary](#)

[Chapter 10 Exceptions: Advanced Concepts](#)

[Lab 10.1: RAISE APPLICATION ERROR](#)

[Lab 10.2: EXCEPTION INIT Pragma](#)

[Lab 10.3: SQLCODE and SQLERRM](#)

[Summary](#)

[Chapter 11 Introduction to Cursors](#)

[Lab 11.1: Types of Cursors](#)

[Making Use of an Implicit Cursor](#)

[Making Use of an Explicit Cursor](#)

[Lab 11.2: Cursor Loop](#)

[Processing an Explicit Cursor](#)

[Making Use of a User-Defined Record](#)

[Making Use of Cursor Attributes](#)

[Lab 11.3: Cursor FOR LOOPS](#)

[Making Use of Cursor FOR LOOPS](#)

[Lab 11.4: Nested Cursors](#)

[Processing Nested Cursors](#)

[Summary](#)

[Chapter 12 Advanced Cursors](#)

[Lab 12.1: Parameterized Cursors](#)

[Cursors with Parameters](#)

[Lab 12.2: Complex Nested Cursors](#)

[Lab 12.3: FOR UPDATE and WHERE CURRENT Cursors](#)

[FOR UPDATE Cursor](#)

[FOR UPDATE OF in a Cursor](#)

[WHERE CURRENT OF](#) in a Cursor

[Summary](#)

[Chapter 13 Triggers](#)

[Lab 13.1: What Triggers Are](#)

[Database Trigger](#)

[BEFORE Triggers](#)

[AFTER Triggers](#)

[Autonomous Transaction](#)

[Lab 13.2: Types of Triggers](#)

[Row and Statement Triggers](#)

[INSTEAD OF Triggers](#)

[Summary](#)

[Chapter 14 Mutating Tables and Compound Triggers](#)

[Lab 14.1: Mutating Tables](#)

[What Is a Mutating Table?](#)

[Resolving Mutating Table Issues](#)

[Lab 14.2: Compound Triggers](#)

[What Is a Compound Trigger?](#)

[Resolving Mutating Table Issues with Compound Triggers](#)

[Summary](#)

[Chapter 15 Collections](#)

[Lab 15.1: PL/SQL Tables](#)

[Associative Arrays](#)

[Nested Tables](#)

[Collection Methods](#)

[Lab 15.2: Varrays](#)

[Lab 15.3: Multilevel Collections](#)

[Summary](#)

[Chapter 16 Records](#)

[Lab 16.1: Record Types](#)

[Table-Based and Cursor-Based Records](#)

[User-Defined Records](#)

[Record Compatibility](#)

[Lab 16.2: Nested Records](#)

[Lab 16.3: Collections of Records](#)

[Summary](#)

[Chapter 17 Native Dynamic SQL](#)

[Lab 17.1: EXECUTE IMMEDIATE Statements](#)

[Using the EXECUTE IMMEDIATE Statement](#)

[How to Avoid Common ORA Errors When Using EXECUTE IMMEDIATE](#)

[Lab 17.2: OPEN-FOR, FETCH, and CLOSE Statements](#)

[Opening Cursor](#)

[Fetching from a Cursor](#)

[Closing a Cursor](#)

[Summary](#)

[Chapter 18 Bulk SQL](#)

[Lab 18.1: FORALL Statements](#)

[Using FORALL Statements](#)

[SAVE EXCEPTIONS Option](#)

[INDICES OF Option](#)

[VALUES OF Option](#)

[Lab 18.2: The BULK COLLECT Clause](#)

[Lab 18.3: Binding Collections in SQL Statements](#)

[Binding Collections with EXECUTE IMMEDIATE Statements](#)

[Binding Collections with OPEN-FOR, FETCH, and CLOSE Statements](#)

[Summary](#)

[Chapter 19 Procedures](#)

[Benefits of Modular Code](#)

[Block Structure](#)

[Anonymous Blocks](#)

[Lab 19.1: Creating Procedures](#)

[Putting Procedure Creation Syntax into Practice](#)

[Querying the Data Dictionary for Information on Procedures](#)

[Lab 19.2: Passing Parameters IN and OUT of Procedures](#)

[Using IN and OUT Parameters with Procedures](#)

[Summary](#)

[Chapter 20 Functions](#)

[Lab 20.1: Creating Functions](#)

[Creating Stored Functions](#)

[Making Use of Functions](#)

[Lab 20.2: Using Functions in SQL Statements](#)

[Invoking Functions in SQL Statements](#)

[Writing Complex Functions](#)

[Lab 20.3: Optimizing Function Execution in SQL](#)

[Defining a Function Using the WITH Clause](#)

[Creating a Function with the UDF Pragma](#)

[Summary](#)

[Chapter 21 Packages](#)

[Lab 21.1: Creating Packages](#)

[Creating Package Specifications](#)

[Creating Package Bodies](#)

[Calling Stored Packages](#)

[Creating Private Objects](#)

[Lab 21.2: Cursor Variables](#)

[Lab 21.3: Extending the Package](#)

[Extending the Package with Additional Procedures](#)

[Lab 21.4: Package Instantiation and Initialization](#)

[Creating Package Variables During Initialization](#)

[Lab 21.5: SERIALLY REUSABLE Packages](#)

[Using the SERIALLY REUSABLE Pragma](#)

[Summary](#)

[Chapter 22 Stored Code](#)

[Lab 22.1: Gathering Information about Stored Code](#)

[Getting Stored Code Information from the Data Dictionary](#)

[Overloading Modules](#)

[Summary](#)

[**Chapter 23 Object Types in Oracle**](#)

[Lab 23.1: Object Types](#)

[Creating Object Types](#)

[Using Object Types with Collections](#)

[Lab 23.2: Object Type Methods](#)

[Constructor Methods](#)

[Member Methods](#)

[Static Methods](#)

[Comparing Objects](#)

[Summary](#)

[**Chapter 24 Oracle-Supplied Packages**](#)

[Lab 24.1: Extending Functionality with Oracle-Supplied Packages](#)

[Accessing Files within PL/SQL with UTL_FILE](#)

[Scheduling Jobs with DBMS_JOB](#)

[Generating an Explain Plan with DBMS_XPLAN](#)

[Generating Implicit Statement Results with DBMS_SQL](#)

[Lab 24.2: Error Reporting with Oracle-Supplied Packages](#)

[Using the DBMS.Utility Package for Error Reporting](#)

[Using the UTL_CALL_STACK Package for Error Reporting](#)

[Summary](#)

[**Chapter 25 Optimizing PL/SQL**](#)

[Lab 25.1: PL/SQL Tuning Tools](#)

[PL/SQL Profiler API](#)

[Trace API](#)

[PL/SQL Hierarchical Profiler](#)

[Lab 25.2: PL/SQL Optimization Levels](#)

[Lab 25.3: Subprogram Inlining](#)

[Summary](#)

[**Appendix A PL/SQL Formatting Guide**](#)

[Case](#)

[White Space](#)

[Naming Conventions](#)

[Comments](#)

[Other Suggestions](#)

[Appendix B Student Database Schema](#)

[Table and Column Descriptions](#)

[Index](#)

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 12c 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 11g or higher

- SQL Developer or SQL*Plus 11g 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 11g or Oracle 12c 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.

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 12c

Oracle 12c 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 12c, 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

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```
– 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*

[Click here to view code image](#)

```

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 12c, 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*

[Click here to view code image](#)

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

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

Prior to Oracle 12c, 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 12c, 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 12c, 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 12c 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*

[Click here to view code image](#)

```
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 12c, 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*

[Click here to view code image](#)

```
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 12c as well. To get the result set from the procedure test_return_result successfully, the client application must likewise be upgraded to Oracle 12c. Otherwise, the following error message is returned:

[Click here to view code image](#)

```
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 12c, a view could be created only as a definer rights unit. Starting with release 12c, 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*

[Click here to view code image](#)

```
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 12c, 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 12c, 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 12c. As stated previously, such behavior must be explicitly specified via INHERIT PRIVILEGES or INHERIT ANY PRIVILEGES privileges.

Invisible Columns

Starting with Oracle 12c, 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*

[Click here to view code image](#)

```
– 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 editable 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 12c, you are able to specify whether a schema object is editable

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 NONEDITABLE property.

PL/SQL Functions That Run Faster in SQL

Starting with Oracle 12c, 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*

[Click here to view code image](#)

```
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. Scrittore
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*

[Click here to view code image](#)

```
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. Scrittore
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 12c. 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 12c. 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*

[Click here to view code image](#)

```
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_DIRECTIVES 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 12c, 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.

1. PL/SQL Concepts

In this chapter, you will learn about

- [PL/SQL Architecture](#)
- [PL/SQL Development Environment](#)
- [PL/SQL: The Basics](#)

PL/SQL stands for “Procedural Language Extension to SQL.” Because of its tight integration with SQL, PL/SQL supports the great majority of the SQL features, such as SQL data manipulation, data types, operators, functions, and transaction control statements. As an extension to SQL, PL/SQL combines SQL with programming structures and subroutines available in any high-level language.

PL/SQL is used for both server-side and client-side development. For example, database triggers (code that is attached to tables—discussed in [Chapters 13](#) and [14](#)) on the server side and the logic behind an Oracle Form on the client side can be written using PL/SQL. In addition, PL/SQL can be used to develop web and mobile applications in both conventional and cloud environments when used in conjunction with a wide variety of Oracle development tools.

Lab 1.1: PL/SQL Architecture

After this lab, you will be able to

- [Describe PL/SQL Architecture](#)
- [Discuss PL/SQL Block Structure](#)
- [Understand How PL/SQL Gets Executed](#)

Many Oracle applications are built using multiple tiers, also known as N -tier architecture, where each tier represents a separate logical process. For example, a three-tier architecture would consist of three tiers: a data management tier, an application processing tier, and a presentation tier. In this architecture, the Oracle database resides in the data management tier, and the programs that make requests against this database reside in either the presentation tier or the application processing tier. Such programs can be written in many programming languages, including PL/SQL. An example of a three-tier architecture is shown in [Figure 1.1](#).

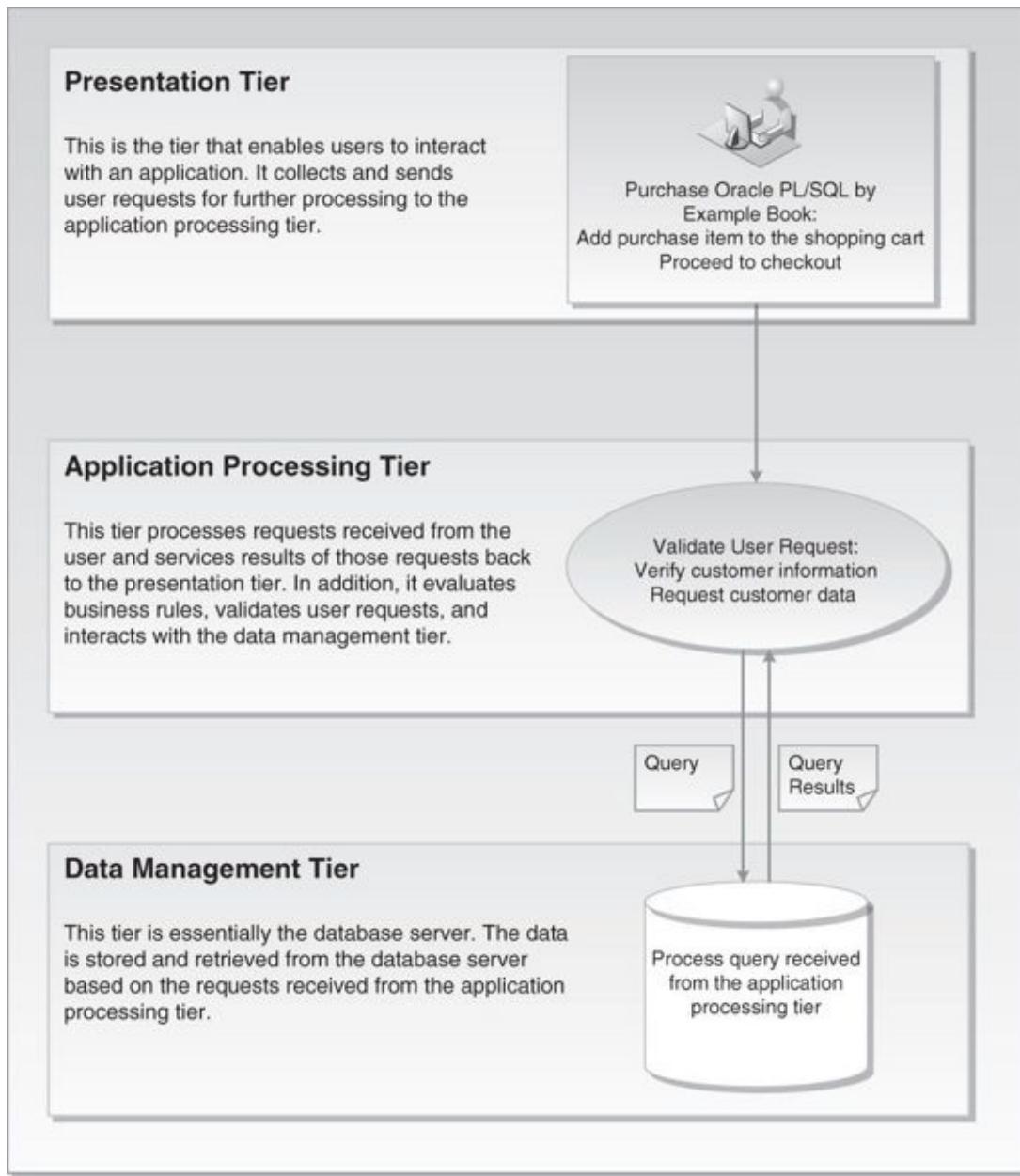


Figure 1.1 Three-Tier Architecture

PL/SQL Architecture

While PL/SQL is just like any other programming language, its main distinction is that it is not a stand-alone programming language. Rather, PL/SQL is a part of the Oracle RDBMS as well as various Oracle development tools such as Oracle Application Express (APEX) and Oracle Forms and Reports. As a result, PL/SQL may reside in any layer of the multitier architecture.

No matter which layer PL/SQL resides in, any PL/SQL block or subroutine is processed by the PL/SQL engine, which is a special component of various Oracle products. As a result, it is very easy to move PL/SQL modules between various tiers. The PL/SQL engine processes and executes any PL/SQL statements and sends any SQL statements to the SQL statement processor. The SQL statement processor is always located on the Oracle server. [Figure 1.2](#) illustrates the PL/SQL engine residing on the Oracle server.

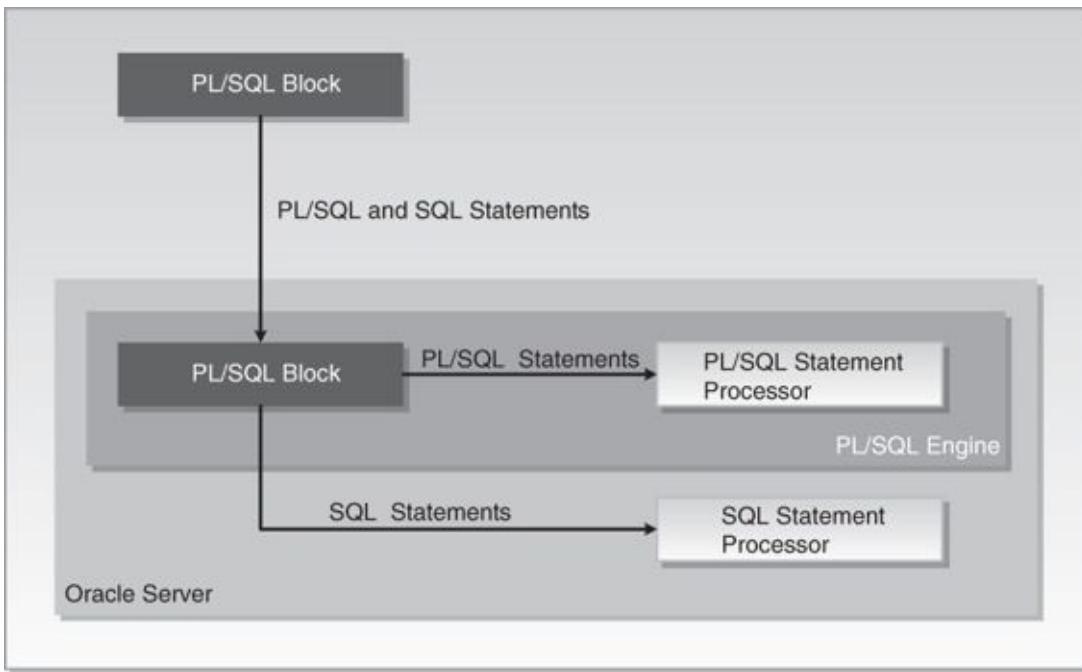


Figure 1.2 The PL/SQL Engine and Oracle Server

When the PL/SQL engine is located on the server, the whole PL/SQL block is passed to the PL/SQL engine on the Oracle server. The PL/SQL engine processes the block according to the scheme depicted in [Figure 1.2](#).

When the PL/SQL engine is located on the client, as it is in Oracle development tools, the PL/SQL processing is done on the client side. All SQL statements that are embedded within the PL/SQL block are sent to the Oracle server for further processing. When PL/SQL block contains no SQL statements, the entire block is executed on the client side.

Using PL/SQL has several advantages. For example, when you issue a **SELECT** statement in SQL*Plus or SQL Developer against the **STUDENT** table, it retrieves a list of students. The **SELECT** statement you issued at the client computer is sent to the database server to be executed. The results of this execution are then returned to the client. In turn, rows are displayed on your client machine.

Now, assume that you need to issue multiple **SELECT** statements. Each **SELECT** statement is a request against the database and is sent to the Oracle server. The results of each **SELECT** statement are sent back to the client. Each time a **SELECT** statement is executed, network traffic is generated. Hence, multiple **SELECT** statements will result in multiple round-trip transmissions, adding significantly to the network traffic.

When these **SELECT** statements are combined into a PL/SQL program, they are sent to the server as a single unit. The **SELECT** statements in this PL/SQL program are executed at the server. The server sends the results of these **SELECT** statements back to the client, also as a single unit. Therefore, a PL/SQL program encompassing multiple **SELECT** statements can be executed at the server and have all of the results returned to the client in the same round trip. This is obviously a more efficient process than having each **SELECT** statement execute independently. This model is illustrated in [Figure 1.3](#).

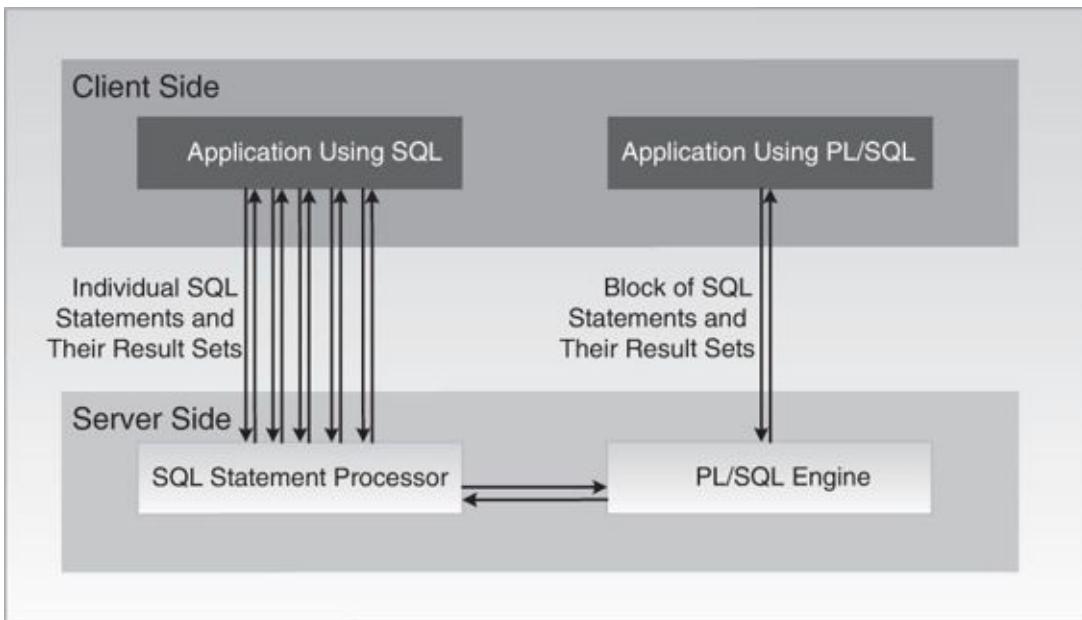


Figure 1.3 PL/SQL in Client–Server Architecture

[Figure 1.3](#) compares two applications. The first application uses four independent SQL statements that generate eight trips on the network. The second application combines SQL statements into a single PL/SQL block, which is then sent to the PL/SQL engine. The engine sends SQL statements to the SQL statement processor and checks the syntax of the PL/SQL statements. As you can see, only two trips are generated on the network with the second application.

In addition, applications written in PL/SQL are portable. They can run in any environment that Oracle products can run in. Because PL/SQL does not change from one environment to the next, different tools can use a PL/SQL script.

PL/SQL Block Structure

A block is the most basic unit in PL/SQL. All PL/SQL programs are combined into blocks. These blocks can also be nested within one another. Usually, PL/SQL blocks combine statements that represent a single logical task. Therefore, different tasks within a single program can be separated into blocks. With this structure, it is easier to understand and maintain the logic of the program.

PL/SQL blocks can be divided into two groups: named and anonymous. Named PL/SQL blocks are used when creating subroutines. These subroutines, which include procedures, functions, and packages, can be stored in the database and referenced by their names later. In addition, subroutines such as procedures and functions can be defined within the anonymous PL/SQL block. These subroutines exist as long as the block is executing and cannot be referenced outside the block. In other words, subroutines defined in one PL/SQL block cannot be called by another PL/SQL block or referenced by their names later. Subroutines are discussed in [Chapters 19](#) through [21](#). Anonymous PL/SQL blocks, as you have probably guessed, do not have names. As a result, they cannot be stored in the database or referenced later.

PL/SQL blocks contain three sections: a declaration section, an executable section, and an exception-handling section. The executable section is the only mandatory section of the

block; both the declaration and exception-handling sections are optional. As a result, a PL/SQL block has the structure illustrated in [Listing 1.1](#).

Listing 1.1 PL/SQL Block Structure

[Click here to view code image](#)

```
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

Declaration Section

The declaration section is the first section of the PL/SQL block. It contains definitions of PL/SQL identifiers such as variables, constants, cursors, and so on. PL/SQL identifiers are covered in detail throughout this book.

For Example

[Click here to view code image](#)

```
DECLARE
    v_first_name VARCHAR2(35);
    v_last_name   VARCHAR2(35);
```

This example shows the declaration section of an anonymous PL/SQL block. It begins with the keyword **DECLARE** and contains two variable declarations. The names of the variables, **v_first_name** and **v_last_name**, are followed by their data types and sizes. Notice that a semicolon terminates each declaration.

Executable Section

The executable section is the next section of the PL/SQL block. It contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

For Example

[Click here to view code image](#)

```
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
       WHERE student_id = 123;

    DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
END;
```

This example shows the executable section of the PL/SQL block. It begins with the keyword **BEGIN** and contains a **SELECT INTO** statement from the **STUDENT** table. The first and last names for student ID 123 are selected into two variables: **v_first_name** and **v_last_name**. [Chapter 3](#) contains a detailed explanation of the **SELECT INTO** statement. Next, the values of the variables, **v_first_name** and **v_last_name**, are

displayed on the screen with the help of the DBMS_OUTPUT.PUT_LINE statement. This statement will be covered later in this chapter in greater detail. The end of the executable section of this block is marked by the keyword END.

By the Way

The executable section of any PL/SQL block always begins with the keyword BEGIN and ends with the keyword END.

Exception-Handling Section

Two types of errors may occur when a PL/SQL block is executed: compilation or syntax errors and runtime errors. Compilation errors are detected by the PL/SQL compiler when there is a misspelled reserved word or a missing semicolon at the end of the statement.

For Example

[Click here to view code image](#)

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('This is a test')
END;
```

This example contains a syntax error: The DBMS_OUTPUT.PUT_LINE statement is not terminated by a semicolon.

Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler. These types of errors are detected or handled by the exception-handling section of the PL/SQL block. It contains a series of statements that are executed when a runtime error occurs within the block.

Once a runtime error occurs, control is passed to the exception-handling section of the block. The error is then evaluated, and a specific exception is raised or executed. This is best illustrated by the following example. All changes are shown in bold.

For Example

[Click here to view code image](#)

```
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
       WHERE student_id = 123;
    DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);

EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
END;
```

This example shows the exception-handling section of the PL/SQL block. It begins with the keyword EXCEPTION. The WHEN clause evaluates which exception must be raised. In this example, there is only one exception, called NO_DATA_FOUND, and it is raised when the SELECT statement does not return any rows. If there is no record for student ID 123 in

the STUDENT table, control is passed to the exception-handling section and the DBMS_OUTPUT.PUT_LINE statement is executed. [Chapters 8, 9, and 10](#) contain detailed explanations of the exception-handling section.

You have seen examples of the declaration section, executable section, and exception-handling section. These examples may be combined into a single PL/SQL block.

For Example *ch01_1a.sql*

[Click here to view code image](#)

```
DECLARE
    v_first_name VARCHAR2(35);
    v_last_name  VARCHAR2(35);
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
       WHERE student_id = 123;

    DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
END;
```

How PL/SQL Gets Executed

Every time an anonymous PL/SQL block is executed, the code is sent to the PL/SQL engine, where it is compiled. A named PL/SQL block is compiled only at the time of its creation, or if it has been changed. The compilation process includes syntax and semantic checking, as well as code generation.

Syntax checking involves checking PL/SQL code for syntax or compilation errors. As stated previously, a syntax error occurs when a statement does not exactly correspond to the syntax of the programming language. A misspelled keyword, a missing semicolon at the end of the statement, and an undeclared variable are all examples of syntax errors. Once syntax errors are corrected, the compiler can generate a parse tree.

By the Way

A parse tree is a tree-like structure that represents the language rules of a computer language.

Semantic checking involves further processing on the parse tree. It determines whether database objects such as table names and column names referenced in the SELECT statements are valid and whether you have privileges to access them. At the same time, the compiler can assign a storage address to program variables that are used to hold data. This process, which is called binding, allows Oracle software to reference storage addresses when the program is run.

Code generation creates code for the PL/SQL block in interpreted or native mode. Code created in interpreted mode is called p-code. P-code is a list of instructions to the PL/SQL

engine that are interpreted at run time. Code created in a native mode is a processor-dependent system code that is called native code. Because native code does not need to be interpreted at run time, it usually runs slightly faster.

The mode in which the PL/SQL engine generates code is determined by the `PLSQL_CODE_TYPE` database initialization parameter. By default, its value is set to `INTERPRETED`. This parameter is typically set by the database administrators.

For named blocks, both p-code and native code are stored in the database, and are used the next time the program is executed. Once the process of compilation has completed successfully, the status of a named PL/SQL block is set to `VALID`, and it is also stored in the database. If the compilation process was not successful, the status of the named PL/SQL block is set to `INVALID`.

Watch Out!

Successful compilation of the named PL/SQL block on one occasion does not guarantee successful execution of this block in the future. If, at the time of execution, any one of the stored objects referenced by the block is not present in the database or not accessible to the block, execution will fail. At such time, the status of the named PL/SQL block will be changed to `INVALID`.

Lab 1.2: PL/SQL Development Environment

After this lab, you will be able to

- [Get Started with SQL Developer](#)
 - [Get Started with SQL*Plus](#)
 - [Execute PL/SQL Scripts](#)
-

SQL Developer and SQL*Plus are two Oracle-provided tools that you can use to develop and run PL/SQL scripts. SQL*Plus is an old-style command-line utility tool that has been part of the Oracle platform since its infancy. It is included in the Oracle installation on every platform. SQL Developer is a free graphical tool used for database development and administration. It is a fairly recent addition to the Oracle tool set and is available either as a part of the Oracle installation or via download from Oracle's website.

Due to its graphical interface, SQL Developer is a much easier environment to use than SQL*Plus. It allows you to browse database objects, run SQL statements, and create, debug, and run PL/SQL statements. In addition, it supports syntax highlighting and formatting templates that become very useful when you are developing and debugging complex PL/SQL modules.

Even though SQL*Plus and SQL Developer are two very different tools, their underlying functionality and their interactions with the database are very similar. At run time, the SQL and PL/SQL statements are sent to the database. Once they are processed, the results are sent back from the database and displayed on the screen.

The examples used in this chapter are executed in both tools to illustrate some of the interface differences when appropriate. Note that the primary focus of this book is learning PL/SQL; thus these tools are covered only to the degree that is required to run PL/SQL examples provided by this book.

Getting Started with SQL Developer

If SQL Developer has been installed as part of the Oracle installation, you can launch it by accessing Start, All Programs, Oracle, Application Development, SQL Developer on Windows 7 and earlier versions. On Windows 8, SQL Developer is invoked by accessing Start, All Apps, Oracle, SQL Developer. Alternatively, you can download and install this tool as a separate module.

Once SQL Developer is installed, you need to create connection to the database server. This can be accomplished by clicking on the Plus icon located in the upper-left corner of the Connections tab. This activates the New>Select Database Connection dialog box, as shown in [Figure 1.4](#).

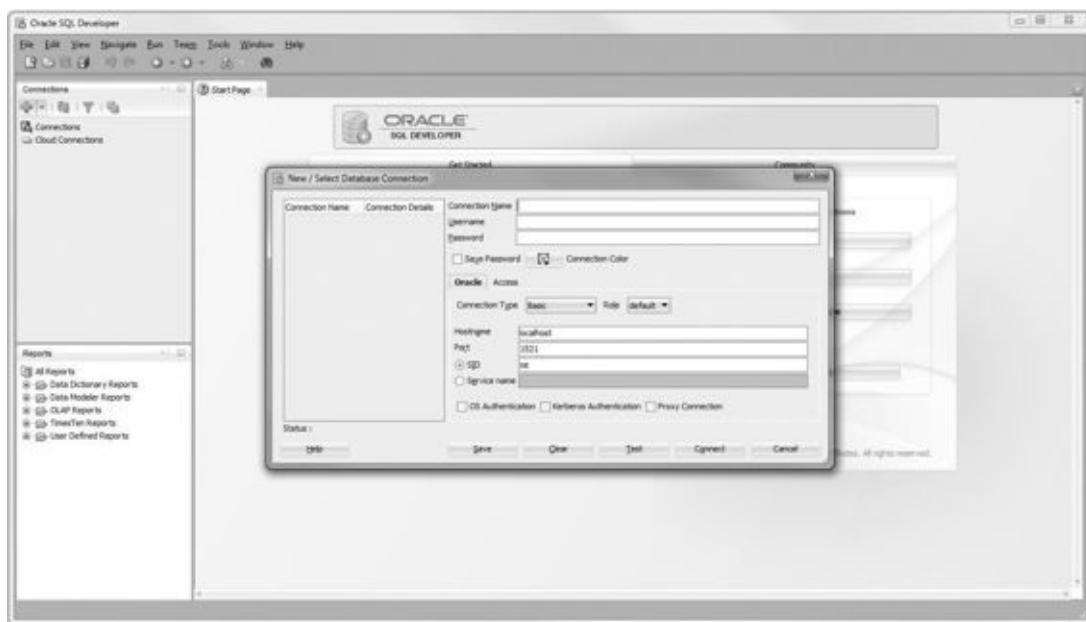


Figure 1.4 Creating a Database Connection in SQL Developer

In [Figure 1.4](#), you need to provide a connection name (StudentConnection), user name (student), and password (learn).

By the Way

Starting with Oracle 11g, the password is case sensitive.

In the same dialog box, you need to provide database connection information such as the hostname (typically the IP address of the machine or the machine name where the database server resides), the default port where that database listens for the connection requests (usually 1521), and the SID (system ID) or service name that identifies a particular database. Both the SID and service name would depend on the names you picked up for your installation of Oracle. The default SID is usually set to orcl.

Watch Out!

If you have not created the STUDENT schema yet, you will not be able to create this connection successfully. To create the STUDENT schema, refer to the installation instructions provided on the companion website.

Once the connection has been successfully created, you can connect to the database by double-clicking on the StudentConnection. By expanding the StudentConnection (clicking on the plus sign located to the left of it), you are able to browse various database objects available in the STUDENT schema. For example, [Figure 1.5](#) shows list of tables available in the STUDENT schema.

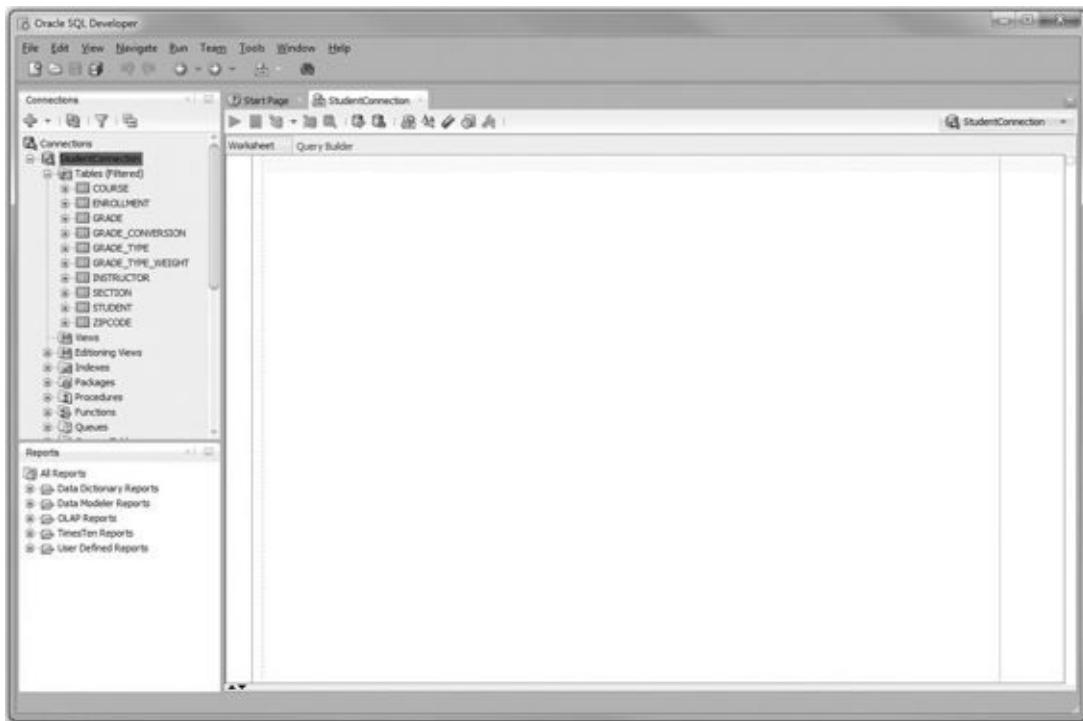


Figure 1.5 List of Tables in the STUDENT Schema

At this point you can start typing SQL or PL/SQL commands in the Worksheet window, shown in [Figure 1.5](#).

To disconnect from the STUDENT schema, you need to right-click on the StudentConnection and click on the Disconnect option. This is illustrated in [Figure 1.6](#).

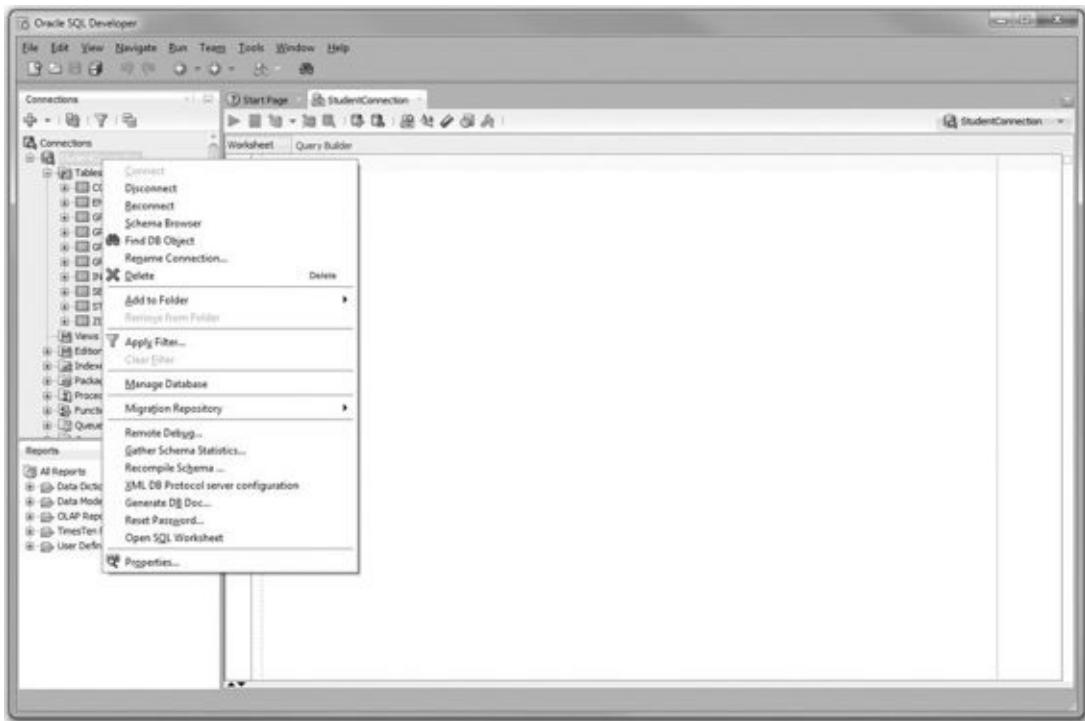


Figure 1.6 Disconnecting from a Database in SQL Developer

Getting Started with SQL*Plus

On Windows 7 and earlier versions, you can access SQL*Plus by choosing Start, All Programs, Oracle, Application Development, SQL*Plus under the Start button. On Windows 8, SQL*Plus is invoked by accessing Start, All Apps, Oracle, SQL*Plus.

When you open SQL*Plus, you are prompted to enter your user name and password (“student” and “learn,” respectively). In addition, you can invoke SQL*Plus by typing `sqlplus` in the command prompt window.

By the Way

In SQL*Plus, the password is not displayed on the screen, even as a masked text.

After successful login, you are able to enter your commands at the SQL> prompt. This is illustrated in [Figure 1.7](#).



Figure 1.7 Connecting to the Database in SQL*Plus

To terminate your connection to the database, type either EXIT or QUIT command and press Enter.

Did You Know?

Terminating the database connection in either SQL Developer or SQL*Plus terminates only your own client connection. In a multiuser environment, there may be potentially hundreds of client connections to the database server at any time. As these connections terminate and new ones are initiated, the database server continues to run and send various query results back to its clients.

Executing PL/SQL Scripts

As mentioned earlier, at run time SQL and PL/SQL statements are sent from the client machine to the database. Once they are processed, the results are sent back from the database to the client and are displayed on the screen. However, there are some differences between entering SQL and PL/SQL statements.

Consider the following example of a SQL statement.

For Example

```
SELECT first_name, last_name
  FROM student
 WHERE student_id = 102;
```

If this statement is executed in SQL Developer, the semicolon is optional. To execute this statement, you need to click on the triangle button in the StudentConnection SQL Worksheet or press the F9 key on your keyboard. The results of this query are then displayed in the Query Result window, as shown in [Figure 1.8](#).

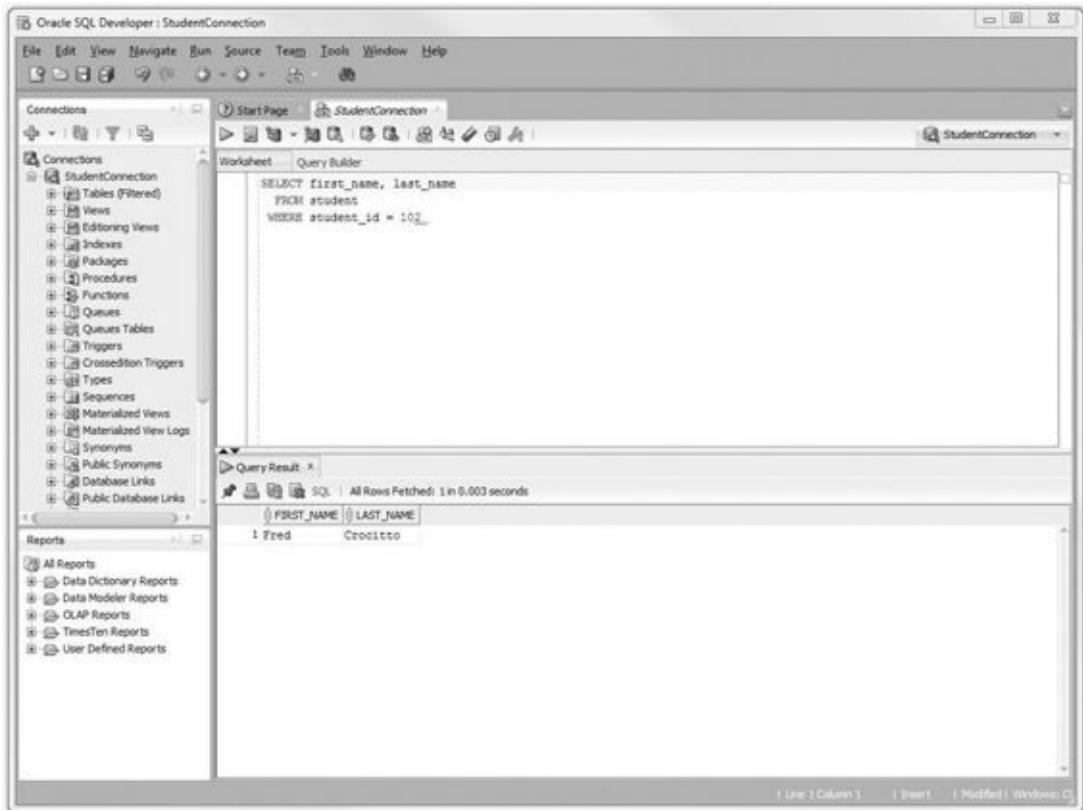


Figure 1.8 Executing a Query in SQL Developer

When the same SELECT statement is executed in SQL*Plus, the semicolon is required. It signals SQL*Plus that the statement is terminated. As soon as you press the Enter key, the query is sent to the database and the results are displayed on the screen, as shown in [Figure 1.9](#).

A screenshot of the SQL*Plus window. The title bar says "SQL Plus". The SQL prompt "SQL>" is followed by the query: "SELECT first_name, last_name FROM student WHERE student_id = 102;". The output shows the results of the query: "FIRST_NAME LAST_NAME" and "Fred Crocitto". The SQL prompt appears again at the bottom.

Figure 1.9 Executing a Query in SQL*Plus

Now, consider the example of the PL/SQL block used in the previous lab.

For Example *ch01_1a.sql*

[Click here to view code image](#)

```
DECLARE
    v_first_name VARCHAR2(35);
    v_last_name  VARCHAR2(35);
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
       WHERE student_id = 123;

    DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
END;
```

Note that each individual statement in this script is terminated by a semicolon. Each variable declaration, the `SELECT INTO` statement, both `DBMS_OUTPUT.PUT_LINE` statements, and the `END` keyword are all terminated by the semicolon. This syntax is necessary because in PL/SQL the semicolon marks termination of an individual statement within a block. In other words, the semicolon is not a block terminator.

Because SQL Developer is a graphical tool, it does not require a special block terminator. The preceding example can be executed in SQL Developer by clicking on the green triangle button in the StudentConnection SQL Worksheet or pressing the F9 key on your keyboard, as shown in [Figure 1.10](#).

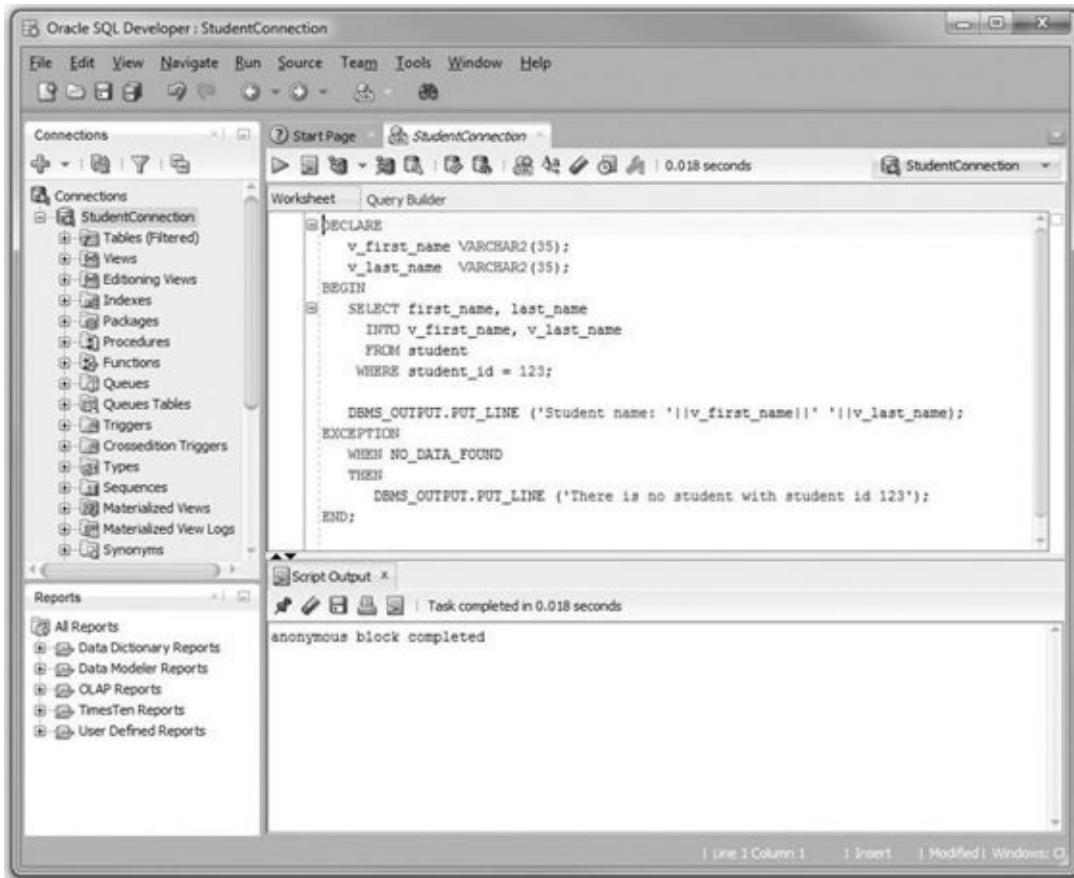
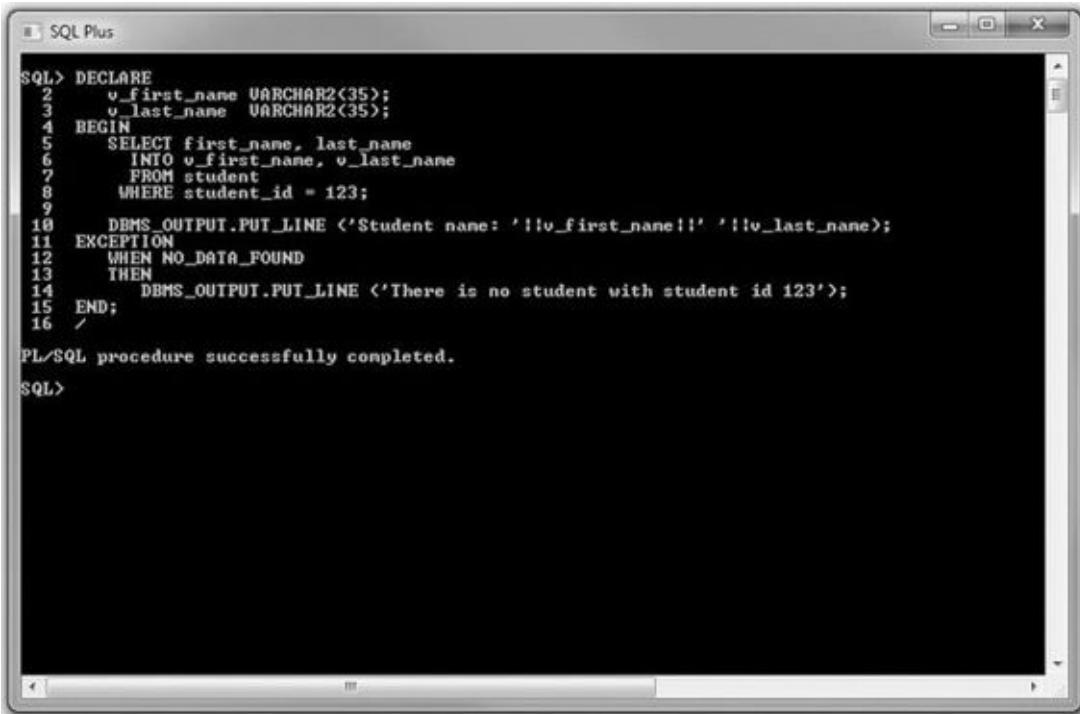


Figure 1.10 Executing a PL/SQL Block in SQL Developer

The block terminator becomes necessary when the same example is executed in

SQL*Plus. Because it is a command-line tool, SQL*Plus requires a textual way of knowing when the block has terminated and is ready for execution. The “/” is interpreted by SQL*Plus as a block terminator. Once you press the Enter key, the PL/SQL block is sent to the database and the results are displayed on the screen. This is shown in [Figure 1.11a](#).



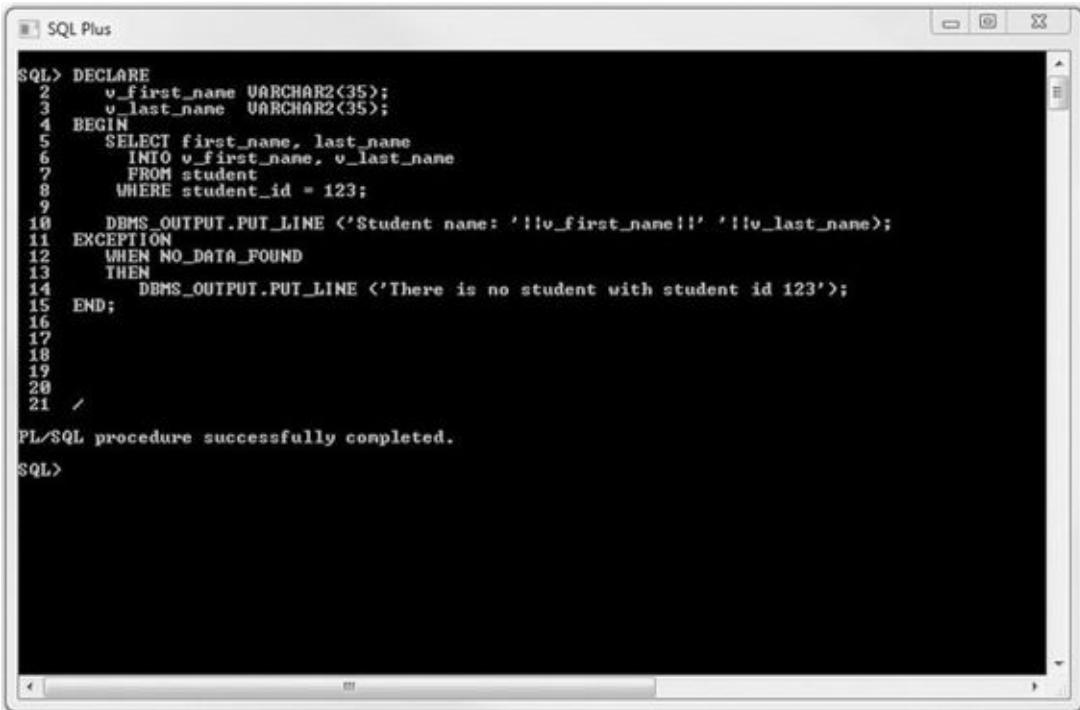
The screenshot shows a Windows-style window titled "SQL Plus". Inside, a PL/SQL block is being entered:

```
SQL> DECLARE
  2      v_first_name  VARCHAR2(35);
  3      v_last_name   VARCHAR2(35);
  4  BEGIN
  5      SELECT first_name, last_name
  6          INTO v_first_name, v_last_name
  7          FROM student
  8      WHERE student_id = 123;
  9
 10     DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name||);
 11  EXCEPTION
 12      WHEN NO_DATA_FOUND
 13      THEN
 14          DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
 15  END;
 16 /
```

After the final slash, the message "PL/SQL procedure successfully completed." is displayed.

Figure 1.11a Executing a PL/SQL Block in SQL*Plus with a Block Terminator

If you omit “/”, SQL*Plus will not execute the PL/SQL script. Instead, it will simply add a blank line to the script when you press the Enter key. This is shown in [Figure 1.11b](#).



The screenshot shows a Windows-style window titled "SQL Plus". Inside, the same PL/SQL block is entered, but it ends with a blank line instead of a slash:

```
SQL> DECLARE
  2      v_first_name  VARCHAR2(35);
  3      v_last_name   VARCHAR2(35);
  4  BEGIN
  5      SELECT first_name, last_name
  6          INTO v_first_name, v_last_name
  7          FROM student
  8      WHERE student_id = 123;
  9
 10     DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name||);
 11  EXCEPTION
 12      WHEN NO_DATA_FOUND
 13      THEN
 14          DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
 15  END;
 16
 17
 18
 19
 20
 21 /
```

The message "PL/SQL procedure successfully completed." is displayed at the bottom.

Figure 1.11b Executing a PL/SQL Block in SQL*Plus without a Block Terminator

Lab 1.3: PL/SQL: The Basics

After this lab, you will be able to

- [Use the DBMS_OUTPUT.PUT_LINE Statement](#)
- [Use the Substitution Variable Feature](#)

We noted earlier that PL/SQL is not a stand-alone programming language; rather, it exists only as a tool within the Oracle environment. As a result, it does not really have any capabilities to accept input from a user. This is accomplished with the special feature of the SQL Developer and SQL*Plus tools called a substitution variable.

Similarly, it is often helpful to provide the user with some pertinent information after the execution of a PL/SQL block, and this is accomplished with the help of the `DBMS_OUTPUT.PUT_LINE` statement. Note that unlike the substitution variable, this statement is part of the PL/SQL language.

DBMS_OUTPUT.PUT_LINE Statement

You already have seen some examples of how the `DBMS_OUTPUT.PUT_LINE` statement can be used—that is, to display information on the screen. The `DBMS_OUTPUT.PUT_LINE` is a call to the procedure `PUT_LINE`. This procedure is a part of the `DBMS_OUTPUT` package that is owned by the Oracle user `SYS`.

The `DBMS_OUTPUT.PUT_LINE` statement writes information to the buffer for storage. Once a program has completed, the information from the buffer is displayed on the screen. The size of the buffer can be set between 2000 and 1 million bytes.

To see the results of the `DBMS_OUTPUT.PUT_LINE` statement on the screen, you need to enable it. In SQL Developer, this is accomplished by selecting the View menu option and then choosing the `Dbms Output` option, as shown in [Figure 1.12a](#).

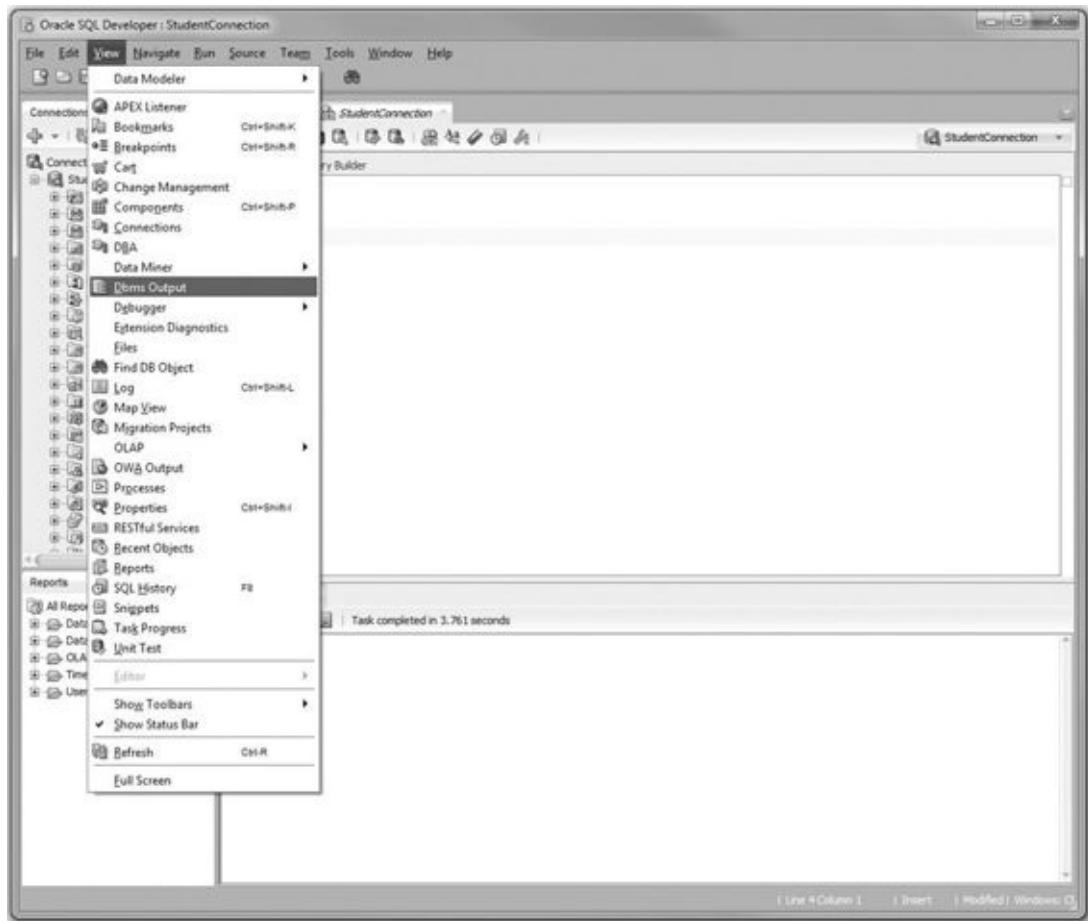


Figure 1.12a Enabling DBMS_OUTPUT in SQL Developer: Step 1

Once the Dbms Output window appears in SQL Developer, you need to click on the green plus button, as shown in [Figure 1.12b](#).

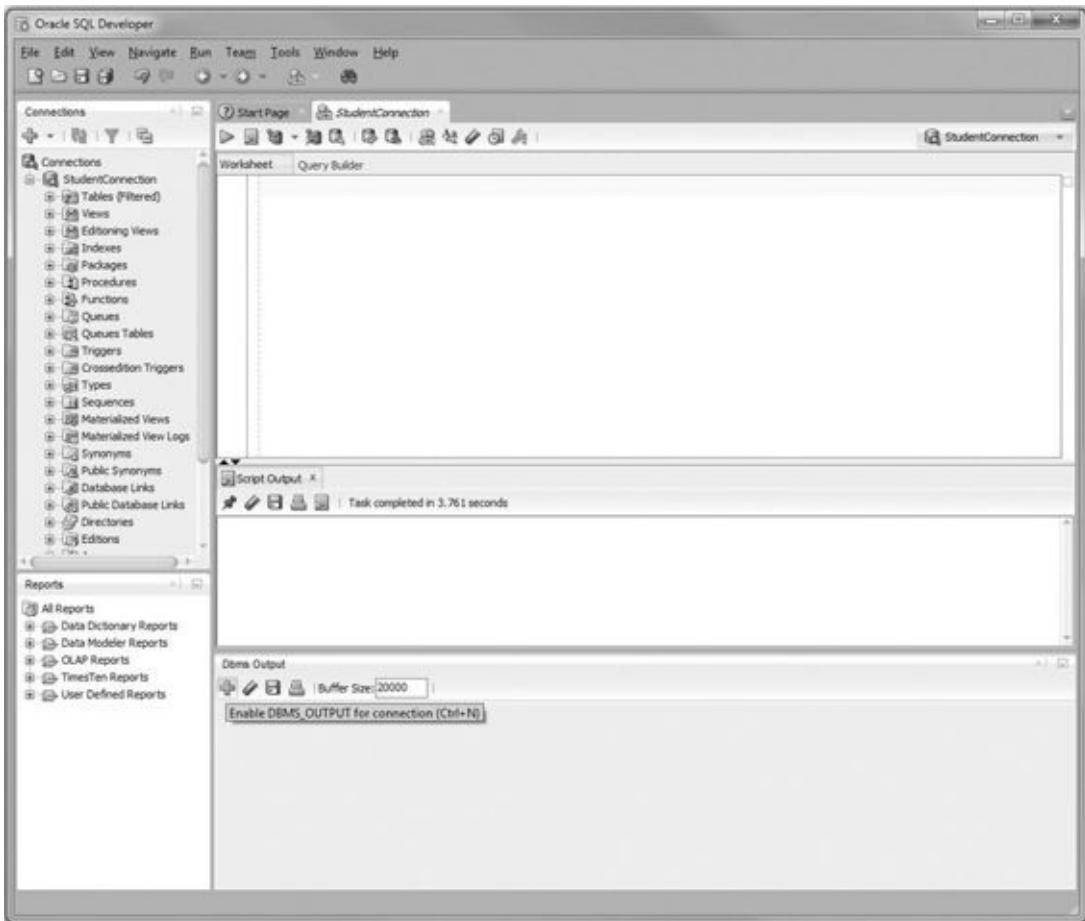


Figure 1.12b Enabling DBMS_OUTPUT in SQL Developer: Step 2

Once you click on the plus button, you will be prompted with the name of the connection for which you want to enable the statement. You need to select StudentConnection and click OK. The result of this operation is shown in [Figure 1.12c](#).

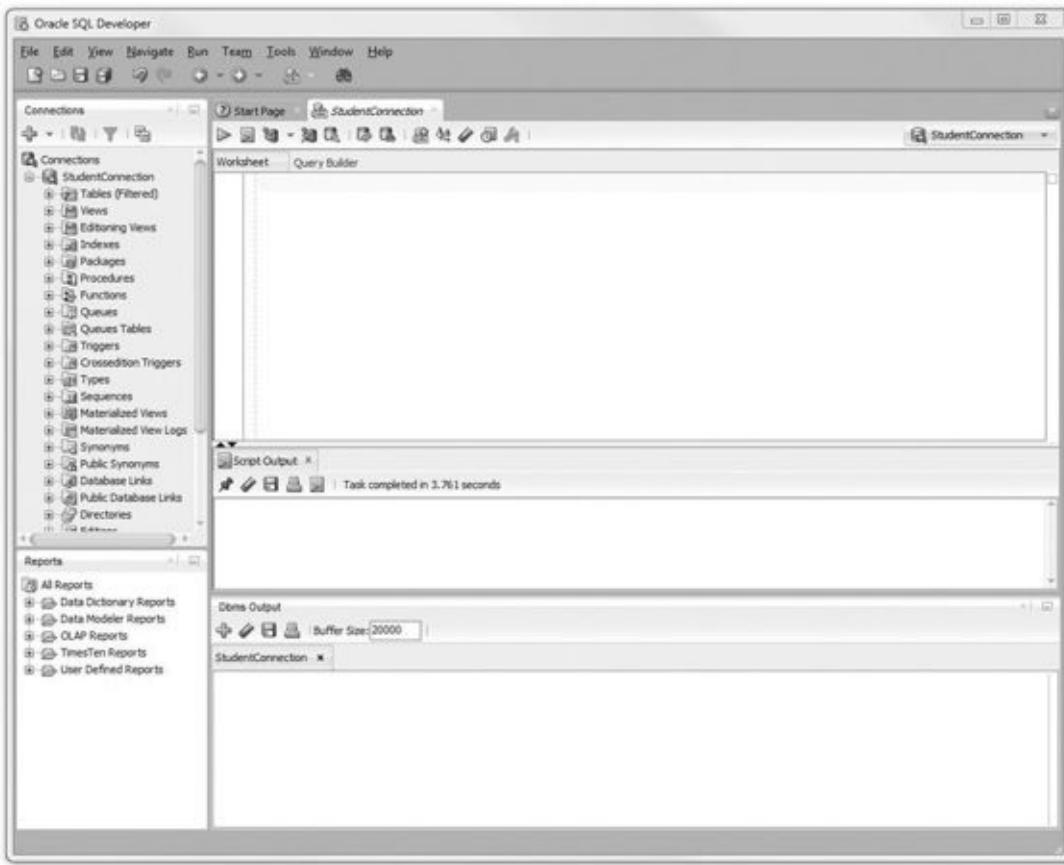


Figure 1.12c Enabling DBMS_OUTPUT in SQL Developer: Step 3

To enable the DBMS_OUTPUT statement in SQL*Plus, you enter one of the following statements before the PL/SQL block:

```
SET SERVEROUTPUT ON;
```

or

[Click here to view code image](#)

```
SET SERVEROUTPUT ON SIZE 5000;
```

The first SET statement enables the DBMS_OUTPUT.PUT_LINE statement, with the default value for the buffer size being used. The second SET statement not only enables the DBMS_OUTPUT.PUT_LINE statement, but also changes the buffer size from its default value to 5000 bytes.

Similarly, if you do not want information to be displayed on the screen by the DBMS_OUTPUT.PUT_LINE statement, you can issue the following SET command prior to the PL/SQL block:

```
SET SERVEROUTPUT OFF;
```

Substitution Variable Feature

Substitution variables are a special type of variables that enable PL/SQL to accept input from a user at a run time. They cannot be used to output values, however, because no memory is allocated for them. Substitution variables are replaced with the values provided by the user before the PL/SQL block is sent to the database. The variable names are usually prefixed by the ampersand (&) or double ampersand (&&) character.

Consider the following example.

For Example *ch01_1b.sql*

[Click here to view code image](#)

```
DECLARE
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(35);
    v_last_name  VARCHAR2(35);
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
       WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

When this example is executed, the user is asked to provide a value for the student ID. The student's name is then retrieved from the STUDENT table if there is a record with the given student ID. If there is no record with the given student ID, the message from the exception-handling section is displayed on the screen.

In SQL Developer, the substitution variable feature operates as shown in [Figure 1.13](#).

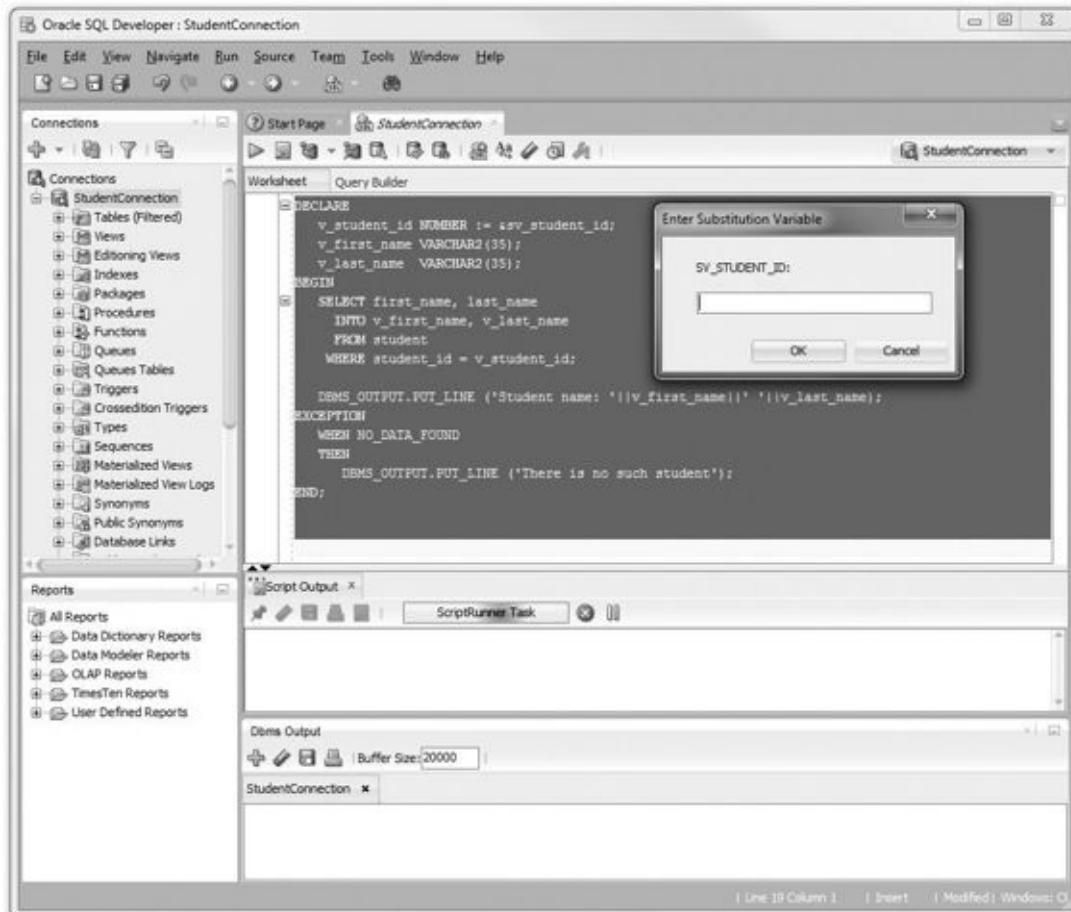


Figure 1.13 Using Substitution Variable in SQL Developer

Once the value for the substitution variable is provided, the results of the execution are displayed in the Script Output window, as shown in [Figure 1.14](#).

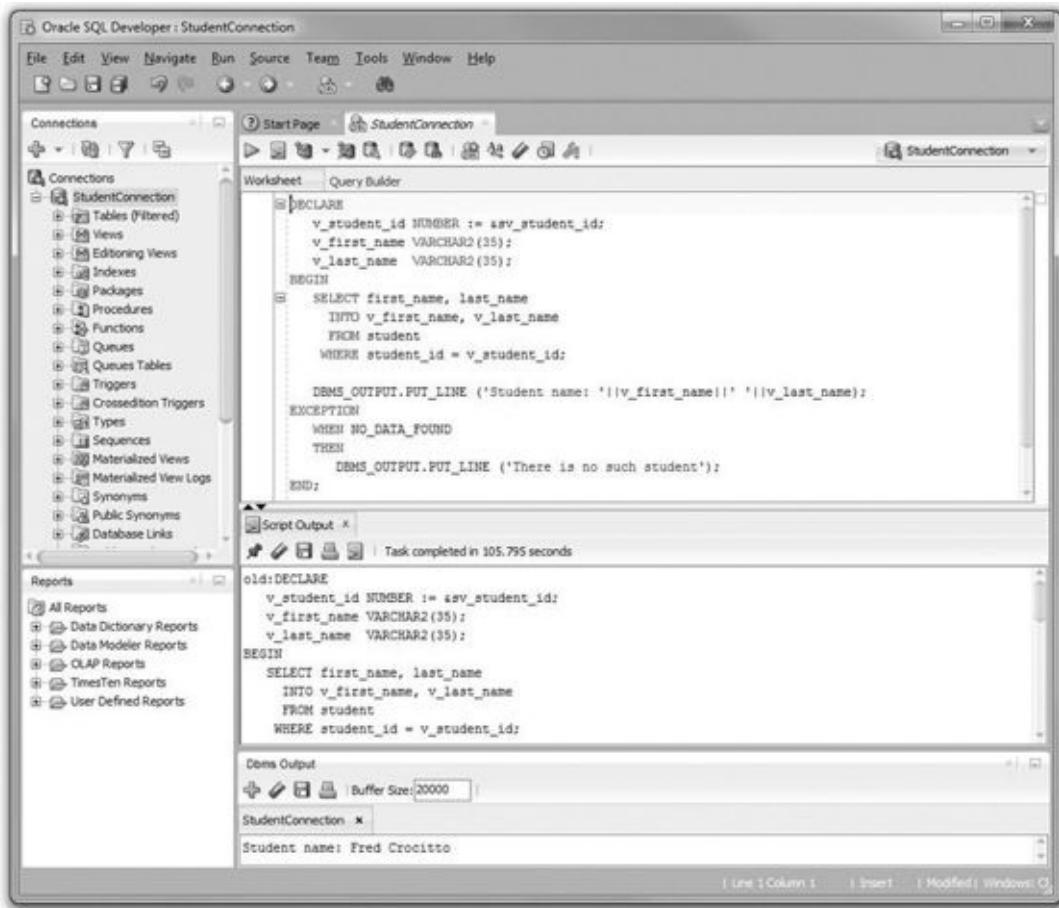
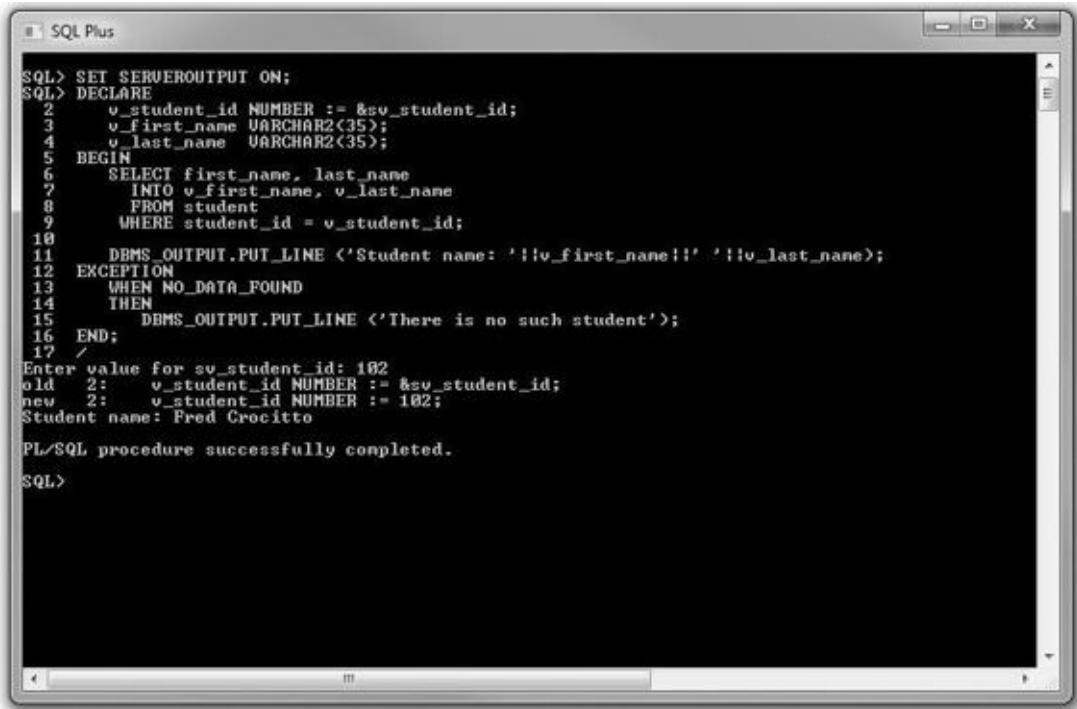


Figure 1.14 Using Substitution Variable in SQL Developer: Script Output Window

In [Figure 1.14](#), the substitution of the variable is shown in the Script Output window and the result of the execution is shown in the Dbms Output window.

In SQL*Plus, the substitution variable feature operates as shown in [Figure 1.15](#).



The screenshot shows the SQL*Plus interface with a PL/SQL block. The code uses substitution variables (&sv_student_id) and DBMS_OUTPUT.PUT_LINE to print student names. It includes an EXCEPTION block for handling NO_DATA_FOUND. The user is prompted to enter a value for &sv_student_id, which is then used in the output.

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
 2      v_student_id NUMBER := &sv_student_id;
 3      v_first_name VARCHAR2(35);
 4      v_last_name  VARCHAR2(35);
 5  BEGIN
 6      SELECT first_name, last_name
 7          INTO v_first_name, v_last_name
 8      FROM student
 9     WHERE student_id = v_student_id;
10
11     DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
12  EXCEPTION
13    WHEN NO_DATA_FOUND
14    THEN
15        DBMS_OUTPUT.PUT_LINE ('There is no such student');
16  END;
17 /
Enter value for sv_student_id: 102
old  2:  v_student_id NUMBER := &sv_student_id;
new  2:  v_student_id NUMBER := 102;
Student name: Fred Crocetto
PL/SQL procedure successfully completed.
SQL>
```

Figure 1.15 Using Substitution Variable in SQL*Plus

Note that SQL*Plus does not list the complete PL/SQL block in its results, but rather displays the substitution operation only.

The preceding example uses a single ampersand for the substitution variable. When a single ampersand is used throughout the PL/SQL block, the user is asked to provide a value for each occurrence of the substitution variable.

For Example ch01_2a.sql

[Click here to view code image](#)

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Today is'||'&sv_day');
  DBMS_OUTPUT.PUT_LINE ('Tomorrow will be'||'&sv_day');
END;
```

When executing this example in either SQL Developer or SQL*Plus, you are prompted twice to provide the value for the substitution variable. This example produces the following output:

```
Today is Monday
Tomorrow will be Tuesday
```

Did You Know?

When a substitution variable is used in the script, the output produced by the program contains the statements that show how the substitution was done.

If you do not want to see these lines displayed in the output produced by the script, use the **SET** command option before you run the script:

```
SET VERIFY OFF;
```

This command is supported by both SQL Developer and SQL*Plus.

As demonstrated earlier, when the same substitution variable is used with a single ampersand, the user is prompted to provide a value for each occurrence of this variable in the script. To avoid this task, you can prefix the first occurrence of the substitution variable by the double ampersand (**&&**) character, as highlighted in bold in the following example.

For Example *ch01_2b.sql*

[Click here to view code image](#)

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Today is'||'&&sv_day');
    DBMS_OUTPUT.PUT_LINE ('Tomorrow will be'||'&sv_day');
END;
```

In this example, the substitution variable **sv_day** is prefixed by a double ampersand in the first **DBMS_OUTPUT.PUT_LINE** statement. As a result, this version of the example produces different output:

```
Today is Monday
Tomorrow will be Monday
```

From the output shown, it is clear that the user is asked only once to provide the value for the substitution variable **sv_day**. In turn, both **DBMS_OUTPUT.PUT_LINE** statements use the value of Monday entered by the user.

When a substitution variable is assigned to the string (text) data type, it is a good practice to enclose it with single quotes. You cannot always guarantee that a user will provide text information in single quotes. This practice, which will make your program less error prone, is illustrated in the following code fragment.

For Example

[Click here to view code image](#)

```
DECLARE
    v_course_no VARCHAR2(5) := '&sv_course_no';
```

As mentioned earlier, substitution variables are usually prefixed by the ampersand (**&**) or double ampersand (**&&**) characters; these are the default characters that denote substitution variables. A special **SET** command option available in SQL Developer and SQL*Plus also allows you to change the default character to any other character or disable the substitution variable feature. This **SET** command has the following syntax:

```
SET DEFINE character
```

or

```
SET DEFINE ON
```

or

```
SET DEFINE OFF
```

The first **SET** command option changes the prefix of the substitution variable from an ampersand to another character. Note, however, that this character cannot be alphanumeric or white space. The second (**ON** option) and third (**OFF** option) control whether SQL*Plus will look for substitution variables. In addition, the **ON** option changes the value of the

character back to the ampersand.

Summary

In this chapter, you learned about PL/SQL architecture and how it may be used in a multitier environment. You also learned how PL/SQL is able to interact with users via substitution variables and the `DBMS_OUTPUT.PUT_LINE` statement. Finally, you learned about two PL/SQL development tools, SQL Developer and SQL*Plus. The examples shown in this chapter were executed in both tools to illustrate the differences between them. The main difference between the two is that SQL Developer has a graphical user interface and SQL*Plus has a command-line interface. The PL/SQL examples used throughout this book may be executed in either tool with the same results. Depending on your preference, you may choose one tool over the other. However, it is a good idea to become familiar with both, as these tools are part of almost every Oracle database installation.

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.

2. PL/SQL Language Fundamentals

In this chapter, you will learn about

- [PL/SQL Programming Fundamentals](#)

In the Introduction and [Chapter 1](#), you learned about the difference between machine language and a programming language. You have also learned how PL/SQL differs from SQL and how the PL/SQL basic block structure works. This is similar to learning the history behind a foreign language and the context in which it is used. To use the PL/SQL language, you must learn the keywords, including what they mean and when and how to use them. First, you will encounter the different types of keywords. You will then encounter their full syntax. Finally, in this chapter, you will expand on simple block structure with an exploration of scope and nesting blocks.

Lab 2.1: PL/SQL Programming Fundamentals

After this lab, you will be able to

- [Describe PL/SQL Language Components](#)
- [Explain the Use of PL/SQL Variables](#)
- [Identify PL/SQL Reserved Words](#)
- [Explain the Use of Identifiers in PL/SQL](#)
- [Describe Anchored Data Types](#)
- [Discuss the Scope of a Block, Nested Blocks, and Labels](#)

In most languages, you have only two sets of characters: numbers and letters. Some languages, such as Hebrew or Tibetan, have specific characters for vowels that are not placed in line with consonants. Other languages, such as Japanese, have three character sets: one for words originally taken from the Chinese language, another set for native Japanese words, and a third for other foreign words. To speak any foreign language, you must begin by learning these character sets. You then progress to learn how to make words from these character sets. Finally, you learn the parts of speech and can begin speaking the language.

You can think of PL/SQL as being a more complex language because it has many character types and, additionally, many types of words or lexical units that are made from these character sets. Once you learn these building blocks, you can progress to learn the structure of the PL/SQL language.

PL/SQL Language Components

Character Types

The PL/SQL engine accepts four types of characters: letters, digits, symbols (*, +, -, =, and so on), and white space. When elements from one or more of these character types are joined together, they create a lexical unit (lexical units can be a combination of character types). The lexical units are the words of the PL/SQL language. First you need to learn the PL/SQL vocabulary, and then you will move on to the syntax, or grammar. Soon you can start talking in PL/SQL.

Lexical Units

A language such as English contains different parts of speech. Each part of speech, such as a verb or a noun, behaves in a different way and must be used according to specific rules. Likewise, a programming language has lexical units that are the building blocks of the language. PL/SQL lexical units are classified into one of the following five groups:

1. **Identifiers.** Identifiers must begin with a letter and may be up to 30 characters long. A PL/SQL manual provides a more detailed list of restrictions. Generally, if you stick with characters, numbers, and “ ”, and you avoid reserved words, you will not run into problems.
2. **Reserved words.** Reserved words are words that PL/SQL saves for its own use (e.g., BEGIN, END, SELECT).
3. **Delimiters.** These are characters that have special meaning to PL/SQL, such as arithmetic operators and quotation marks.
4. **Literals.** A literal is any value (character, numeric, or Boolean [true/false]) that is not an identifier. Examples of literals include 123, “Declaration of Independence,” and FALSE.
5. **Comments.** These can be either single-line comments (i.e., —) or multiline comments (i.e., /* */).

See [Appendix A, “PL/SQL Formatting Guide,”](#) for details on formatting.

The PL/SQL engine recognizes different characters as having different meanings and, therefore, processes them differently. PL/SQL is neither a pure mathematical language nor a spoken language, yet it contains elements of both. Letters will form various lexical units such as identifiers or keywords, mathematic symbols will form lexical units known as delimiters that will perform an operation, and other symbols, such as /*, indicate comments that should not be processed.

PL/SQL Variables

Variables may be used to hold a temporary value.

[Click here to view code image](#)

Syntax : <variable-name> <data type> [optional default -assignment]

Variables may also be known as identifiers. There are some restrictions that you need to be familiar with. Specifically, variables must begin with a letter and may be up to 30 characters long. Consider the following example, which contains a list of valid identifiers:

For Example *ch02_1a.sql*

```
DECLARE  
v_student_id  
v_last_name  
V_LAST_NAME  
apt_#
```

Note that the identifiers `v_last_name` and `V_LAST_NAME` are considered identical, because PL/SQL is not case sensitive.

Next, consider an example of illegal identifiers:

For Example

```
X+Y  
1st_year  
student ID
```

The identifier `X+Y` is illegal because it contains the “`+`” sign. This sign is reserved by PL/SQL to denote an addition operation, and it is referred to as a mathematical symbol. The identifier `1st_year` is illegal because it starts with a number. Finally, the identifier `student ID` is illegal because it contains a space.

Now consider another example:

For Example

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;  
DECLARE  
    first&last_names VARCHAR2(30);  
BEGIN  
    first&last_names := 'TEST NAME';  
    DBMS_OUTPUT.PUT_LINE(first&last_names);  
END;
```

In this example, you declare a variable called `first&last_names`. Next, you assign a value to this variable and display this value on the screen.

When run, this example produces the following output in SQL*Plus:

[Click here to view code image](#)

```
Enter value for last_names: Ben  
old 2:   first&last_names VARCHAR2(30);  
new 2:   firstBen VARCHAR2(30);  
Enter value for last_names: Ben  
old 4:   first&last_names := 'TEST NAME';  
new 4:   firstBen := 'TEST NAME';  
Enter value for last_names: Ben  
old 5:   DBMS_OUTPUT.PUT_LINE(first&last_names);  
new 5:   DBMS_OUTPUT.PUT_LINE(firstBen);  
TEST NAME  
PL/SQL procedure successfully completed.
```

When you run this example in SQL Developer, you will get a slightly different response. Instead of seeing the line “enter value for Last Name,” you will see a dialog box that says “Enter Substitution Values” with a box for `Last_Name`. This is how SQL Developer works with variables.

Consider the output produced. Because an ampersand (&) is part of the name of the variable `first&last_names`, the portion of the variable is considered to be a substitution variable (you learned about substitution variables in [Chapter 1](#)). In other words, the portion of the variable name after the ampersand (`last_names`) is treated by the PL/SQL compiler as a substitution variable. As a result, you are prompted to enter the value for the `last_names` variable every time the compiler encounters it.

While this example does not produce any syntax errors (it would if you don’t give the same response to each prompt), the variable `first&last_names` is still an invalid identifier because the ampersand character is reserved for substitution variables. To avoid this problem, change the name of the variable from `first&last_names` to `first_and_last_names`. In other words, you should use an ampersand sign in the name of a variable only when you use it as a substitution variable in your program. It is also important to consider which type of program you are developing and in which you are running your PL/SQL statements. This would be true if the program (or PL/SQL block) will be executed by SQL*Plus. Later, when you write stored code, you would not use the ampersand but you will make use of parameters.

By the Way

If you are using SQL Developer, you will need to go the menu view and click on “DBMS Output” prior to running this script. This will open a new window that shows only the output script. There will be three windows in the SQL Worksheet. The first window is where you put your PL/SQL or SQL statement. The second window is the script output; you will see both Oracle messages and the DBMS output here when you have `SET SERVEROUTPUT ON`. The third window, the DBMS output, shows only `DBMS_OUTPUT` that you have in the script. Click on the pencil with the eraser icon to clear the output windows.

For Example *ch02_1b.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    v_name VARCHAR2(30);
    v_dob DATE;
    v_us_citizen BOOLEAN;
BEGIN
    DBMS_OUTPUT.PUT_LINE(v_name || 'born on' || v_dob);
END;
```

When this example is run in SQL*Plus or SQL Developer, you would see the output `born on`. The reason is that the variables `v_name` and `v_dob` have no values.

Three variables are declared. When each one is declared, its initial value is null.

`v_name` is set as a `VARCHAR2` with a length of 30, `v_dob` is set as a character type date, and `v_us_citizen` is set to `BOOLEAN`. Once the executable section begins, the variables have no value and, therefore, when `DBMS_OUTPUT` is told to print their values, it prints nothing.

This case can be seen when the variables are replaced as follows: Instead of `v_name`, use `COALESCE(v_name, 'No Name')`; and instead of `v_dob`, use `COALESCE(v_dob, '01-Jan-1999')`. Then run the same block and you will get

```
No Name born on 01-Jan-1999
```

To make use of a variable, you must declare it in the declaration section of the PL/SQL block. You will have to give it a name and state its data type. You also have the option to give your variable an initial value. Note that if you do not assign an initial value to a variable, its value will be null. It is also possible to constrain the declaration to “not null,” in which case you must assign an initial value. Variables must first be declared; only then can they be referenced. PL/SQL does not allow forward references. You can set the variable to be a constant, which means it cannot change.

PL/SQL Reserved Words

Reserved words are ones that PL/SQL saves for its own use (e.g., `BEGIN`, `END`, and `SELECT`). You cannot use reserved words for names of variables, literals, or user-defined exceptions.

For Example

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
    exception VARCHAR2(15);
BEGIN
    exception := 'This is a test';
    DBMS_OUTPUT.PUT_LINE(exception);
END;
```

In this example, you declare a variable called `exception`. Next, you initialize this variable and display its value on the screen.

This example illustrates an invalid use of reserved words. To the PL/SQL compiler, “`exception`” is a reserved word that denotes the beginning of the exception-handling section. As a result, this word cannot be used to name a variable. This small piece of code will produce a long error message. The most important part of this error message is the following section:

[Click here to view code image](#)

```
exception VARCHAR2(15);

ORA-06550: line 2, column 4:
PLS-00103: Encountered the symbol "EXCEPTION" when expecting one of the
following:

begin function pragma procedure subtype type <an identifier>
<a double-quoted delimited-identifier> current cursor delete
```

```
exists prior.... /
```

Here is a question you should ask yourself: If you did not know that the word “exception” is a reserved word, do you think you would attempt to debug the preceding script after looking at this error message?

Identifiers in PL/SQL

Take a look at the use of identifiers in the following example:

For Example

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
    v_var1 VARCHAR2(20);
    v_var2 VARCHAR2(6);
    v_var3 NUMBER(5,3);
BEGIN
    v_var1 := 'string literal';
    v_var2 := '12.345';
    v_var3 := 12.345;
    DBMS_OUTPUT.PUT_LINE('v_var1: |||v_var1');
    DBMS_OUTPUT.PUT_LINE('v_var2: |||v_var2');
    DBMS_OUTPUT.PUT_LINE('v_var3: |||v_var3');
END;
```

In this example, you declare and initialize three variables. The values that you assign to them are literals. The first two values, 'string literal' and '12.345' are string literals because they are enclosed by single quotes. The third value, 12.345, is a numeric literal. When run, the example produces the following output:

[Click here to view code image](#)

```
v_var1: string literal
v_var2: 12.345
v_var3: 12.345
PL/SQL procedure successfully completed.
```

Consider another example that uses numeric literals.

For Example

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
    v_var1 NUMBER(2) := 123;
    v_var2 NUMBER(3) := 123;
    v_var3 NUMBER(5,3) := 123456.123;
BEGIN
    DBMS_OUTPUT.PUT_LINE('v_var1: |||v_var1');
    DBMS_OUTPUT.PUT_LINE('v_var2: |||v_var2');
    DBMS_OUTPUT.PUT_LINE('v_var3: |||v_var3');
END;
```

In this example, you declare and initialize three numeric variables. The first declaration and initialization (`v_var1 NUMBER(2) := 123`) causes an error because the value 123 exceeds the specified precision. The second variable declaration and initialization

(`v_var2 NUMBER(3) := 123`) does not cause any errors because the value 123 corresponds to the specified precision. The last declaration and initialization (`v_var3 NUMBER(5,3) := 123456.123`) causes an error because the value 123456.123 exceeds the specified precision. As a result, this example produces the following output:

[Click here to view code image](#)

```
ORA-06512: at line 2 ORA-06502: PL/SQL: numeric or value error: number
precision too large
ORA-06512: at line 2
```

Anchored Data Types

The data type that you assign to a variable can be based on a database object. This assignment is called an anchored declaration since the variable's data type depends on that of the underlying object. It is wise to make use of anchored data types when possible so that you do not have to update your PL/SQL when the data types of base objects change.

[Click here to view code image](#)

```
Syntax: <variable_name> <type attribute>%TYPE
```

The type is a direct reference to a database column.

For Example `ch02_2a.sql`

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    v_name student.first_name%TYPE;
    v_grade grade.numeric_grade%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE(NVL(v_name, 'No Name ') ||
        ' has grade of '||NVL(v_grade, 0));
END;
```

In the preceding example, the variable `v_name` was declared with the identical data type as the column `first_name` from the database table `STUDENT`: `varchar2(25)`. Additionally, the variable `v_grade` was declared with the identical data type as the column `grade_numeric` on the grade database table: `number NUMBER(3)`. Each has a value of null. The results of this example would be as follows:

[Click here to view code image](#)

```
No Name has grade of 0
PL/SQL procedure successfully completed.
```

Most Common Data Types

As a programmer, it is important to know the major data types that you can use in a programming language. This will determine the various options you have to solve a programmatic problem. Also, you need to keep in mind that some functions work only on certain types of data types. The following is a list of the major data types in the Oracle platform that you can use in PL/SQL.

VARCHAR2(maximum_length)

- Stores variable-length character data.
- Takes a required parameter that specifies a maximum length up to 32,767 bytes, with the Extended Data Types parameter enabled. Otherwise, the maximum length is 4000 bytes.
- Does not use a constant or variable to specify the maximum length; an integer literal must be used.

CHAR[(maximum_length)]

- Stores fixed-length (blank-padded if necessary) character data.
- Takes an optional parameter that specifies a maximum length up to 32,767 bytes.
- Does not use a constant or variable to specify the maximum length; an integer literal must be used. If maximum length is not specified, it defaults to 1.
- The maximum width of a CHAR database column is 2000 bytes; the default is 1 byte.

NUMBER[(precision, scale)]

- Stores fixed or floating-point numbers of virtually any size.
- The precision is the total number of digits.
- The scale determines where rounding occurs.
- It is possible to specify a precision and omit the scale, in which case the scale is 0 and only integers are allowed.
- Constants or variables cannot be used to specify a precision and scale; integer literals must be used.
- The maximum precision of a NUMBER value is 38 decimal digits.
- The scale can range from 0 to 127.
- For instance, a scale of 2 rounds to the nearest hundredth (3.456 becomes 3.46).
- The scale can be negative, which causes rounding to the left of the decimal point. For example, a scale of -3 rounds to the nearest thousandth (3456 becomes 3000). A scale of zero rounds to the nearest whole number. If you do not specify the scale, it defaults to zero.

DATE

- Stores fixed-length date values.
- Valid dates for DATE variables include January 1, 4712 BC, to December 31, AD 9999.
- When stored in a database column, date values include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight.

- Dates are actually stored in binary format and will be displayed according to the default format.

LONG

- Stores variable-length character strings.
- The LONG data type is like the VARCHAR2 data type, except that the maximum length of a LONG value is 2 gigabytes.
- You cannot select a value longer than 4000 bytes from a LONG column into a LONG variable.
- LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT statements, but not in expressions, SQL function calls, or certain SQL clauses, such as WHERE, GROUP BY, and CONNECT BY.

LONG RAW

- Stores raw binary data of variable length up to 2 gigabytes.

LOB (Large Object)

- There are four types of LOBs: BLOB, CLOB, NCLOB, and BFILE. These can store binary objects, such as image or video files, up to 4 gigabytes in length.
- A BFILE is a large binary file stored outside the database. The maximum size is 4 gigabytes.

RAW

- Data type for storing variable length binary data.
- Maximum size is 32,767 bytes, with the Extended Data Types parameter enabled. Otherwise, the maximum length is 2000 bytes.

Declare and Initialize Variables

In PL/SQL, variables must be declared before they can be referenced. This is done in the initial declarative section of a PL/SQL block. Recall that each declaration must be terminated with a semicolon. Variables can be assigned using the assignment operator “:=”. If you declare a variable to be a constant, it will retain the same value throughout the block; to do this, you must give it a value at declaration.

Type the following statements into a text file and run the script from a SQL*Plus or SQL Developer session.

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    v_cookies_amt NUMBER := 2;
    v_calories_per_cookie CONSTANT NUMBER := 300;
BEGIN
    DBMS_OUTPUT.PUT_LINE('I ate ' || v_cookies_amt ||
```

```

' cookies with ' || v_cookies_amt *
v_calories_per_cookie || ' calories.');
v_cookies_amt := 3;
DBMS_OUTPUT.PUT_LINE('I really ate ' ||
v_cookies_amt
|| ' cookies with ' || v_cookies_amt *
v_calories_per_cookie || ' calories.');
v_cookies_amt := v_cookies_amt + 5;
DBMS_OUTPUT.PUT_LINE('The truth is, I actually ate '
|| v_cookies_amt || ' cookies with ' ||
v_cookies_amt * v_calories_per_cookie
|| ' calories.');
END;

```

The output of running the preceding script will be as follows:

[Click here to view code image](#)

```

I ate 2 cookies with 600 calories.
I really ate 3 cookies with 900 calories.
The truth is, I actually ate 8 cookies with
2400 calories.
PL/SQL procedure successfully completed.

```

Initially the variable `v_cookies_amt` is declared to be a `NUMBER` with the value of 2, and the variable `v_calories_per_cookie` is declared to be a `CONSTANT NUMBER` with a value of 300 (since it is declared to be a `CONSTANT`, its value will not change). In the course of the procedure, the value of `v_cookies_amt` is set to be 3, and finally it is set to be its current value, 3 plus 5, thus becoming 8.

For Example `ch02_3a.sql`

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
  v_lname VARCHAR2(30);
  v_regdate DATE;
  v_pctincr CONSTANT NUMBER(4,2) := 1.50;
  v_counter NUMBER := 0;
  v_new_cost course.cost%TYPE;
  v_YorN BOOLEAN := TRUE;
BEGIN
  v_counter := ((v_counter + 5)*2) / 2;
  v_new_cost := (v_new_cost * v_counter)/4;
  --
  v_counter := COALESCE(v_counter, 0) + 1;
  v_new_cost := 800 * v_pctincr;
  --
  DBMS_OUTPUT.PUT_LINE(V_COUNTER);
  DBMS_OUTPUT.PUT_LINE(V_NEW_COST);
END;

```

PL/SQL variables are held together with expressions and operators. An expression is a sequence of variables and literals, separated by operators. These expressions are then used to manipulate data, perform calculations, and compare data.

Expressions are composed of a combination of operands and operators. An operand is an argument to the operator; it can be a variable, a constant, a function call. An operator is

what specifies the action (+, **, /, OR, and so on).

You can use parentheses to control the order in which Oracle evaluates an expression. Initially the variable `v_lname` is declared as a data type `VARCHAR2` with a length of 30 and a value of null. The variable `v_regdate` is declared as a date data type with a value of null. The variable `v_pctincr` is declared as a `CONSTANT NUMBER` with a length of 4, a precision of 2, and a value of 1.15. The variable `v_counter` is declared as a `NUMBER` with a value of 0. The variable `v_YorN` is declared as a variable of `BOOLEAN` data type and a value of `TRUE`.

Once the executable section is complete, the variable `v_counter` will be changed from null to 1. The value of `v_new_cost` will change from null to 1200 ($800 * 1.50$).

A common way to find out the value of a variable at different points in a block is to add `DBMS_OUTPUT.PUT_LINE(v_variable_name);` statements throughout the block.

The value of the variable `v_counter` will then change from 1 to 6, which is $((1 + 5) * 2)/2$, and the value of `new_cost` will go from 1200 to 1800, which is $(800 * 6)/4$. The output from running this procedure will be

[Click here to view code image](#)

```
6
1800
PL/SQL procedure successfully completed.
```

Operators (Delimiters): The Separators in an Expression

As a programmer, it is important to know the operators that you can use in a programming language. This will determine the various options you have to solve a programmatic problem. The following is a list of the operators you can use in PL/SQL.

[Click here to view code image](#)

```
Arithmetic ( ** , * , / , + , - )
Comparison( =, <>, !=, <, >, <=, >=, LIKE, IN, BETWEEN, IS NULL, IS
NOT NULL, NOT IN)
Logical (AND, OR, NOT)
String ( ||, LIKE )
Expressions
Operator Precedence
    ** , NOT
    +, - (arithmetic identity and negation), *, /, , - , ||, =, <>, !=
    ,
    <= , >= , < , > , LIKE, BETWEEN, IN, IS NULL
AND-logical conjunction
OR-logical inclusion
```

Scope of a Block, Nested Blocks, and Labels

When making use of variables in a PL/SQL block, it is important to understand their scope. This will allow you to understand how and when you can make use of variables. It will also help you debug the programs you write. The opening section of your PL/SQL block contains the declaration section—that is, the section where you declare the variables that the block will use.

Scope of a Variable

The scope, or existence, of structures defined in the declaration section is local to that block. The block also provides the scope for exceptions that are declared and raised. Exceptions are covered in more detail in [Chapters 8, 9, and 10](#).

The scope of a variable is the portion of the program in which the variable can be accessed, or where the variable is visible. It usually extends from the moment of declaration until the end of the block in which the variable was declared. The visibility of a variable is the part of the program where the variable can be accessed.

[Click here to view code image](#)

```
BEGIN -- outer block
    BEGIN -- inner block
        ...
    END; -- end of inner block
END; -- end of outer block
```

Labels and Nested Blocks

Labels can be added to a block to improve readability and to qualify the names of elements that exist under the same name in nested blocks. The name of the block must precede the first line of executable code (either BEGIN or DECLARE) as follows:

For Example *ch02_4a.sql*

[Click here to view code image](#)

```
set serveroutput on
<< find_stu_num >>
BEGIN
    DBMS_OUTPUT.PUT_LINE('The procedure
        find_stu_num has been executed.');
END find_stu_num;
```

The label optionally appears after END. For commenting purposes, you may alternatively use either -- or /* ... */. Blocks can be nested in the main section or in an exception handler. A nested block is a block that is placed fully within another block. Use of this type of nesting affects the scope and visibility of variables. The scope of a variable in a nested block begins when memory is allocated for the variable and extends from the moment of declaration until the END of the nested block from which it was declared. The visibility of a variable is the part of the program where the variable can be accessed.

For Example *ch02_4b.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
```

```

<< outer_block >>
DECLARE
    v_test NUMBER := 123;
BEGIN
    DBMS_OUTPUT.PUT_LINE
        ('Outer Block, v_test: '||v_test);
<< inner_block >>
DECLARE
    v_test NUMBER := 456;
BEGIN
    DBMS_OUTPUT.PUT_LINE
        ('Inner Block, v_test: '||v_test);
    DBMS_OUTPUT.PUT_LINE
        ('Inner Block, outer_block.v_test: ' ||
            Outer_block.v_test);
END inner_block;
END outer_block;

```

This example produces the following output:

[Click here to view code image](#)

```

Outer Block, v_test: 123
Inner Block, v_test: 456
Inner Block, outer_block.v_test: 123

```

For Example ch02_5a.sql

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
    e_show_exception_scope EXCEPTION;
    v_student_id      NUMBER := 123;
BEGIN
    DBMS_OUTPUT.PUT_LINE('outer student id is ' ||
        ||v_student_id);
    DECLARE
        v_student_id      VARCHAR2(8) := 125;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('inner student id is ' ||
            ||v_student_id);
        RAISE e_show_exception_scope;
    END;
EXCEPTION
    WHEN e_show_exception_scope
    THEN
        DBMS_OUTPUT.PUT_LINE('When am I displayed?');
        DBMS_OUTPUT.PUT_LINE('outer student id is ' ||
            ||v_student_id);
    END;

```

This example produces the following output:

```

outer student id is 123
inner student id is 125
When am I displayed?
outer student id is 123

```

The flow of logic in this block is as follows: The variable **e_Show_Exception_Scope** is declared as an exception type in the declaration section of the block. There is also a declaration of the variable called **v_student_id** of

data type NUMBER that is initialized to the number 123. This variable has a scope of the entire block, but it is visible only outside the inner block. Once the inner block begins, another variable, named v_student_id, is declared. This time it is of data type VARCHAR2(8) and is initialized to 125. This variable will have a scope and visibility only within the inner block. The use of DBMS_OUTPUT helps to show which variable is visible. The inner block raises the exception e_Show_Exception_Scope; as a consequence, the focus will move out of the execution section and into the exception section. The focus will look for an exception named e_Show_Exception_Scope. Since the inner block has no exception with this name, the focus will move to the outer block's exception section, where it finds the exception. The inner variable v_student_id is now out of scope and visibility. The outer variable v_student_id (which has always been in scope) now regains visibility. Because the exception has an IF/THEN construct, it will execute the DBMS_OUTPUT call.

This example illustrates a simple use of nested blocks. Later in the book you will see more complex examples. After you learn about exception handling in [Chapters 8, 9, and 10](#), you will see that there is greater opportunity to make use of nested blocks.

Summary

In this chapter, you learned the fundamentals of the PL/SQL language. The first section introduced the basic components of the PL/SQL language, which collectively allow you to construct simple PL/SQL code. Then you learned about variables, which allow you to store values that may change each time the program is run. You also learned about the PL/SQL keywords; these terms have specific meanings and cannot be used as names. Identifiers and anchored data types were covered to help you understand how data type values are used. The chapter ended by explaining the basics of a PL/SQL block, as well as how to nest blocks and make use of labels to organize your code.

By the Way

The companion website (informit.com/title/0133796787) 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.

3. SQL in PL/SQL

In this chapter, you will learn about

- [DML Statements in PL/SQL](#)
- [Transaction Control in PL/SQL](#)

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*

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```
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:

[Click here to view code image](#)

```
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*

[Click here to view code image](#)

```
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:

[Click here to view code image](#)

```
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*

[Click here to view code image](#)

```
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*

[Click here to view code image](#)

```
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*

[Click here to view code image](#)

```
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`

[Click here to view code image](#)

```
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*

[Click here to view code image](#)

```
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

[Click here to view code image](#)

```

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:

[Click here to view code image](#)

```
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*

[Click here to view code image](#)

```
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`

[Click here to view code image](#)

```
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 **SAVEPOINTS** are used. The final examples demonstrated how committed data could be reversed by using **ROLLBACKs** in conjunction with **SAVEPOINTS**.

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.

4. Conditional Control: IF Statements

In this chapter, you will learn about

- [IF Statements](#)
- [ELSIF Statements](#)
- [Nested IF Statements](#)

In almost every program that you write, you need to make decisions. For example, if it is the end of the fiscal year, bonuses must be distributed to the employees based on their salaries. To compute employee bonuses, a program needs to have a conditional control. In other words, it needs to employ a selection structure.

Conditional control allows you to control the flow of the execution of the program based on a condition. In programming terms, it means that the statements in the program are not executed sequentially. Rather, one group of statements or another will be executed depending on how the condition is evaluated.

In PL/SQL, there are three types of conditional control: IF, ELSIF, and CASE statements. In this chapter, you will explore two types of conditional control—IF and ELSIF—and learn how these types can be nested inside of each other. CASE statements are discussed in [Chapter 5](#).

Lab 4.1: IF Statements

After this lab, you will be able to

- [Use IF-THEN Statements](#)
- [Use IF-THEN-ELSE Statements](#)

An IF statement has two forms: IF-THEN and IF-THEN-ELSE. An IF-THEN statement allows you to specify only one group of actions to take. In other words, this group of actions is taken only when a condition evaluates to TRUE. An IF-THEN-ELSE statement allows you to specify two groups of actions, and the second group of actions is taken when a condition evaluates to FALSE or NULL.

IF - THEN Statements

An IF-THEN statement is the most basic kind of a conditional control and has the structure shown in [Listing 4.1](#).

Listing 4.1 IF-THEN Statement Structure

```
IF CONDITION  
THEN  
    STATEMENT 1;
```

```
...  
STATEMENT N;  
END IF;
```

The reserved word **IF** marks the beginning of the **IF** statement. Statements 1 through *N* are a sequence of executable statements that consist of one or more of the standard programming structures. The *CONDITION* between the keywords **IF** and **THEN** determines whether these statements are executed. **END IF** is a reserved phrase that indicates the end of the **IF-THEN** construct. This flow of the logic is illustrated in [Figure 4.1](#).

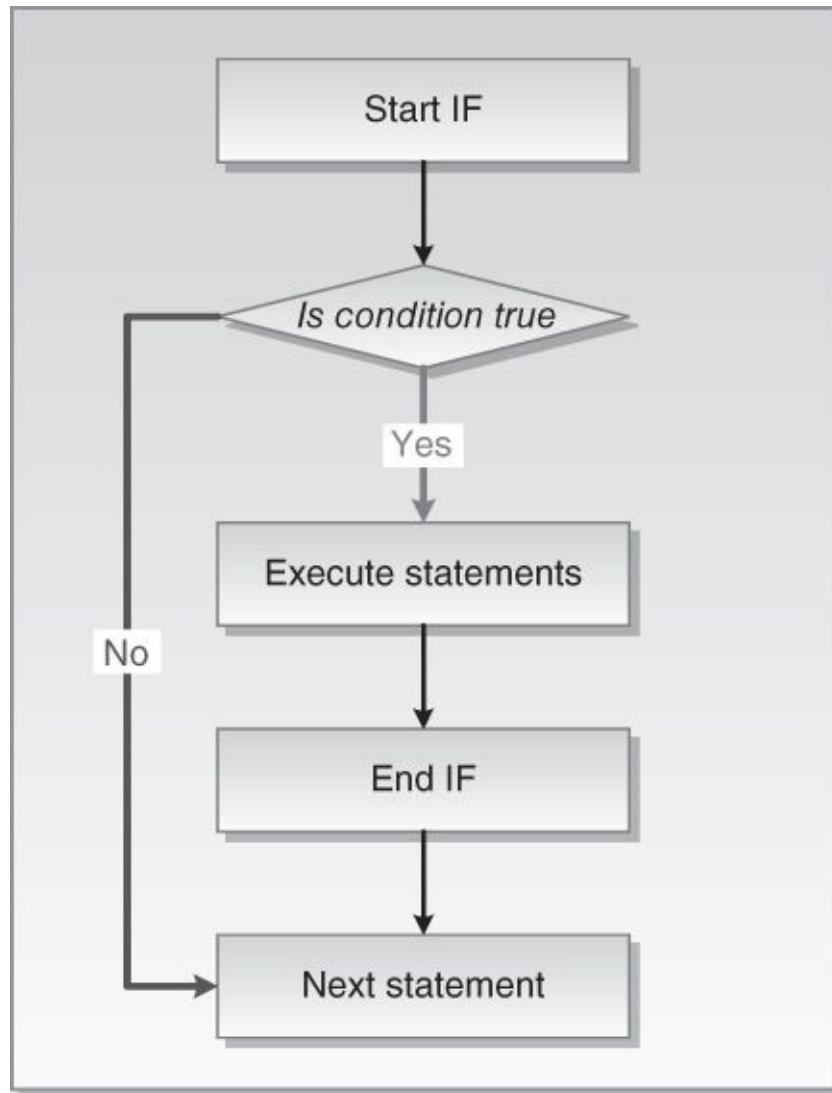


Figure 4.1 IF-THEN Statement

When an **IF-THEN** statement is executed, a condition is evaluated to either **TRUE** or **FALSE**. If the condition evaluates to **TRUE**, control passes to the first executable statement of the **IF-THEN** construct. If the condition evaluates to **FALSE**, control passes to the first executable statement after the **END IF** statement.

Consider the following example. You have two numeric values stored in the variables **v_num1** and **v_num2**. You need to arrange these values so that the smaller value is always stored in **v_num1**, and the larger value is always stored in the **v_num2**.

For Example *ch04_1a.sql*

[Click here to view code image](#)

```

DECLARE
    v_num1 NUMBER := 5;
    v_num2 NUMBER := 3;
    v_temp NUMBER;
BEGIN
    -- if v_num1 is greater than v_num2 rearrange their values
    IF v_num1 > v_num2
    THEN
        v_temp := v_num1;
        v_num1 := v_num2;
        v_num2 := v_temp;
    END IF;

    -- display the values of v_num1 and v_num2
    DBMS_OUTPUT.PUT_LINE ('v_num1 = ' || v_num1);
    DBMS_OUTPUT.PUT_LINE ('v_num2 = ' || v_num2);
END;

```

In this example, condition

```
v_num1 > v_num2
```

evaluates to TRUE because 5 is greater than 3. Next, the values are rearranged so that 3 is assigned to `v_num1` and 5 is assigned to `v_num2`. This step is done with the help of the third variable, `v_temp`, which is used for temporary storage.

This example produces the following output:

```
v_num1 = 3
v_num2 = 5
```

IF-THEN-ELSE Statement

An IF-THEN statement specifies the sequence of statements to execute only if the condition evaluates to TRUE. When this condition evaluates to FALSE or NULL, there is no special action to take except to proceed with execution of the program.

An IF-THEN-ELSE statement enables you to specify two groups of statements. One group of statements is executed when the condition evaluates to TRUE. Another group of statements is executed when the condition evaluates to FALSE or NULL. This structure is shown in [Listing 4.2](#).

Listing 4.2 IF-THEN-ELSE Statement Structure

```

IF CONDITION
THEN
    STATEMENT 1;
ELSE
    STATEMENT 2;
END IF;
STATEMENT 3;

```

When `CONDITION` evaluates to TRUE, control is passed to `STATEMENT 1`; when `CONDITION` evaluates to FALSE or NULL, control is passed to `STATEMENT 2`. After the IF-THEN-ELSE construct has completed, `STATEMENT 3` is executed. This flow of the logic is illustrated in [Figure 4.2](#).

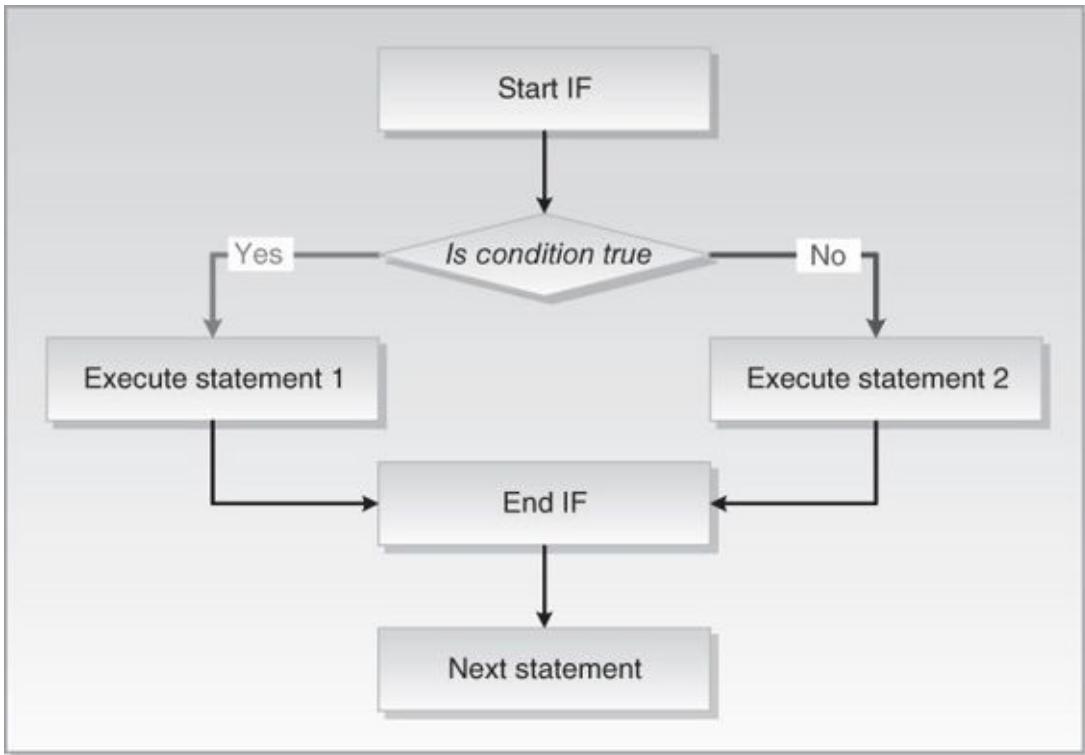


Figure 4.2 IF-THEN-ELSE Statement

Did You Know?

The IF-THEN-ELSE construct should be used when trying to choose between two mutually exclusive actions. Consider the following example:

[Click here to view code image](#)

```

DECLARE
    v_num NUMBER := &sv_user_num;
BEGIN
    -- test if the number provided by the user is even
    IF MOD(v_num,2) = 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
    ELSE
        DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END IF;
END;

```

For any given number of DBMS_OUTPUT.PUT_LINE statements, only one is executed. Hence, the IF-THEN-ELSE construct enables you to specify two and only two mutually exclusive actions.

When run, this example produces the following output:

```
24 is even number
```

Null Condition

In some cases, a condition used in an **IF** statement may evaluate to **NULL** instead of **TRUE** or **FALSE**. For the **IF-THEN** construct, the statements associated with the construct will not be executed if an associated condition evaluates to **NULL**. Instead, control of the execution will pass to the first executable statement after **END IF**. For the **IF-THEN-ELSE** construct, the statements specified after the keyword **ELSE** will be executed if an associated condition evaluates to **NULL**.

For Example *ch04_2a.sql*

[Click here to view code image](#)

```
DECLARE
    v_num1 NUMBER := 0;
    v_num2 NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
    IF v_num1 = v_num2
    THEN
        DBMS_OUTPUT.PUT_LINE ('v_num1 = v_num2');
    END IF;
    DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

This example produces the following output:

```
Before IF statement...
After IF statement...
```

The condition

```
v_num1 = v_num2
```

evaluates to **NULL** because variable **v_num2** is not assigned a value; therefore, it remains **NULL**. Notice that the **IF-THEN** construct behaves as if the condition evaluated to **FALSE**. In other words, the **DBMS_OUTPUT.PUT_LINE** statement associated with the **IF-THEN** construct does not execute.

Next, consider a similar example that employs the **IF-THEN-ELSE** construct (the newly added statements are shown in bold).

For Example *ch04_2b.sql*

[Click here to view code image](#)

```
DECLARE
    v_num1 NUMBER := 0;
    v_num2 NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
    IF v_num1 = v_num2
    THEN
        DBMS_OUTPUT.PUT_LINE ('v_num1 = v_num2');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('v_num1 != v_num2');
    END IF;
    DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

This example produces the following output:

```
Before IF statement...
v_num1 != v_num2
After IF statement...
```

Similarly, the condition

```
v_num1 = v_num2
```

evaluates to NULL, and the ELSE portion of the IF-THEN-ELSE construct is executed.

Lab 4.2: ELSIF Statements

After this lab, you will be able to

- Use the ELSIF Statement

An ELSIF statement has the structure shown in [Listing 4.3](#).

Listing 4.3 ELSIF Statement Structure

```
IF CONDITION 1
THEN
    STATEMENT 1;
ELSIF CONDITION 2
THEN
    STATEMENT 2;
ELSIF CONDITION 3
THEN
    STATEMENT 3;
...
ELSE
    STATEMENT N;
END IF;
```

The reserved word IF marks the beginning of an ELSIF construct. The words *CONDITION 1* through *CONDITION N* are a sequence of the conditions that evaluate to TRUE or FALSE. These conditions are mutually exclusive. In other words, if *CONDITION 1* evaluates to TRUE, *STATEMENT 1* is executed and control passes to the first executable statement after the reserved phrase END IF. The rest of the ELSIF construct is ignored. When *CONDITION 1* evaluates to FALSE, control passes to the ELSIF part and *CONDITION 2* is evaluated, and so forth. If none of the specified conditions evaluates as TRUE, control passes to the ELSE part of the ELSIF construct. An ELSIF statement can contain any number of ELSIF clauses. This flow of the logic is illustrated in [Figure 4.3](#).

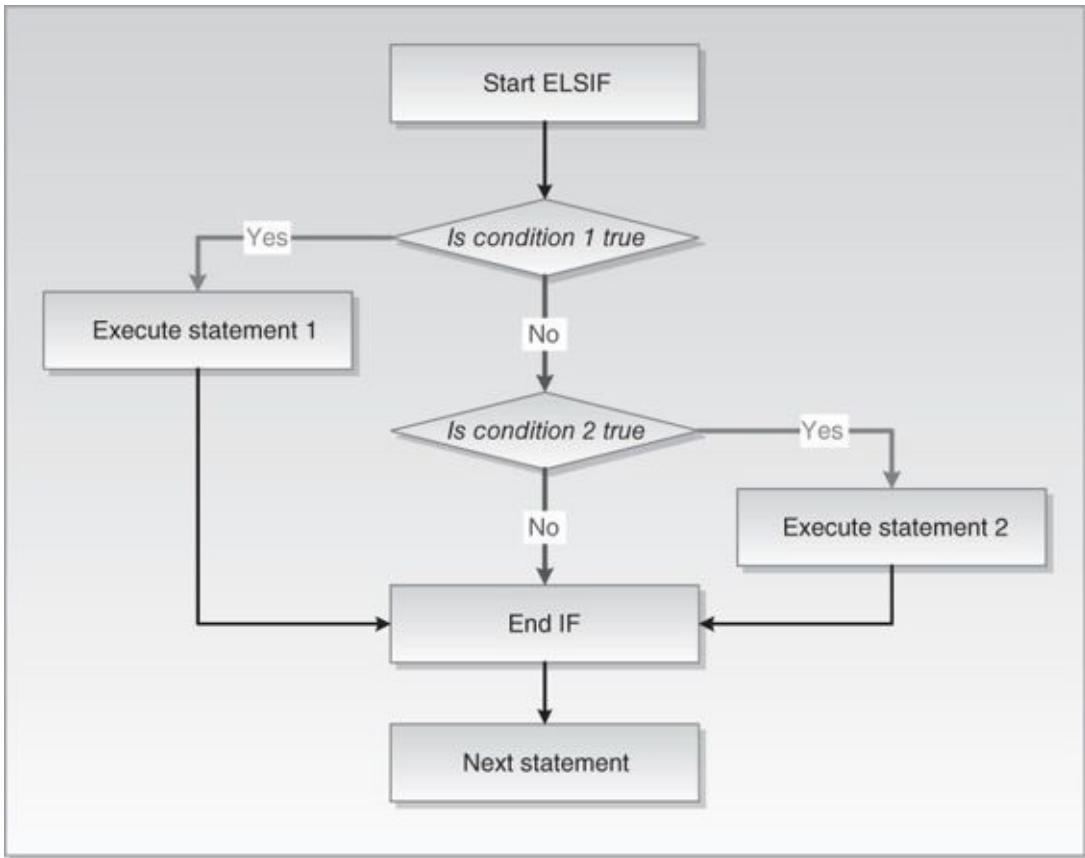


Figure 4.3 ELSIF Statement

[Figure 4.3](#) shows that if condition 1 evaluates to TRUE, statement 1 is executed and control passes to the first statement after END IF. If condition 1 evaluates to FALSE, control passes to condition 2. If condition 2 evaluates to TRUE, statement 2 is executed. Otherwise, control passes to the statement following END IF, and so forth. Consider the following example.

For Example ch04_3a.sql

[Click here to view code image](#)

```

DECLARE
    v_num NUMBER := &sv_num;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
    IF v_num < 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
    ELSIF v_num = 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num||' is equal to zero');
    ELSE
        DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
    END IF;
    DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;

```

The value of the variable `v_num` is provided at run time and evaluated with the help of the ELSIF statement. If the value of `v_num` is less than 0, the first `DBMS_OUTPUT.PUT_LINE` statement executes, and the ELSIF construct terminates. If the value of `v_num` is greater than 0, both conditions

```
v_num < 0
```

and

```
v_num = 0
```

evaluate to FALSE, and the ELSE part of the ELSIF construct executes.

Assume that the value of the variable v_num equals 5 at run time. This example produces the following output:

```
Before IF statement...
5 is a positive number
After IF statement...
```

Did You Know?

For an ELSIF statement:

- IF must always be matched with END IF.
- There must be a space between END and IF. When the space is omitted, the compiler produces the following error:

[Click here to view code image](#)

```
ORA-06550: line 13, column 4:  
PLS-00103: Encountered the symbol ";" when expecting one of the  
following: if
```

As you can see, this error message is not very clear, and it can take you some time to correct it, especially if you have not encountered it before.

- There is no second “E” in ELSIF.
- Conditions of an ELSIF statement must be mutually exclusive. These conditions are evaluated in sequential order, from the first to the last. Once a condition evaluates to TRUE, the remaining conditions of the ELSIF statement are not evaluated at all. Consider this example of an ELSIF construct:

[Click here to view code image](#)

```
IF v_num >= 0  
THEN  
    DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');  
ELSIF v_num <= 10  
THEN  
    DBMS_OUTPUT.PUT_LINE ('v_num is less than 10');  
ELSE  
    DBMS_OUTPUT.PUT_LINE ('v_num is less than ? or greater than ?');  
END IF;
```

Assume that the value of v_num is equal to 5. Both conditions of the ELSIF statement can evaluate to TRUE because 5 is greater than 0, and 5 is less than 10. However, once the first condition, v_num >= 0, evaluates to TRUE, the rest of the ELSIF construct is ignored.

For any value of v_num that is greater than or equal to 0 and less than or equal to 10, these conditions are not mutually exclusive. Therefore, the DBMS_OUTPUT.PUT_LINE statement associated with the ELSIF clause will not execute for any such value of v_num. For the second condition, v_num <= 10, to evaluate as TRUE, the value of v_num must be less than 0.

How would you rewrite this ELSIF construct to capture any value of v_num between 0 and 10 and display it on the screen with a single condition?

When using an ELSIF construct, it is not necessary to specify which action should be taken if none of the conditions evaluates to TRUE. In other words, an ELSE clause is not required in the ELSIF construct. Consider the following example:

For Example *ch04_3b.sql*

[Click here to view code image](#)

```
DECLARE
    v_num NUMBER := &sv_num;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
    IF v_num < 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
    ELSIF v_num > 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
    END IF;
    DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

As you can see, there is no action specified when **v_num** is equal to 0. If the value of **v_num** is equal to 0, both conditions will evaluate to FALSE, and the **ELSIF** statement will not execute at all. When a value of zero is specified for **v_num**, this example produces the following output:

```
Before IF statement...
After IF statement...
```

Did You Know?

You probably noticed that for all **IF** statement examples, the reserved words **IF**, **ELSIF**, **ELSE**, and **END IF** are entered on a separate line and aligned with the word **IF**. In addition, all executable statements in the **IF** construct are indented. The format of the **IF** construct makes no difference to the compiler, but the meaning of the formatted **IF** construct becomes obvious to us with this style.

The **IF-THEN-ELSE** statement

[Click here to view code image](#)

```
IF x = y THEN v_txt := 'YES'; ELSE v_txt := 'NO'; END IF;
```

is equivalent to

```
IF x = y
THEN
    v_txt := 'YES';
ELSE
    v_txt := 'NO';
END IF;
```

The formatted version of the **IF** construct is easier to read and understand.

Lab 4.3: Nested IF Statements

After this lab, you will be able to

- Use Nested IF Statements

You have encountered different types of conditional controls: **IF-THEN** statement, **IF-THEN-ELSE** statement, and **ELSIF** statement. These types of conditional controls can be nested inside of one another—for example, an **IF** statement can be nested inside an **ELSIF**, and vice versa. Consider the following example:

For Example ch04_4a.sql

[Click here to view code image](#)

```
DECLARE
    v_num1  NUMBER := &sv_num1;
    v_num2  NUMBER := &sv_num2;
    v_total NUMBER;
BEGIN
    IF v_num1 > v_num2
    THEN
        DBMS_OUTPUT.PUT_LINE ('IF part of the outer IF');
        v_total := v_num1 - v_num2;
    ELSE
        DBMS_OUTPUT.PUT_LINE ('ELSE part of the outer IF');
        v_total := v_num1 + v_num2;

        IF v_total < 0
        THEN
            DBMS_OUTPUT.PUT_LINE ('Inner IF');
            v_total := v_total * (-1);
        END IF;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('v_total = '||v_total);
END;
```

The **IF-THEN-ELSE** statement is called an **outer IF** statement because it encompasses the **IF-THEN** statement (shown in bold). The **IF-THEN** statement is called an **inner IF** statement because it is enclosed by the body of the **IF-THEN-ELSE** statement.

Assume that the values for **v_num1** and **v_num2** are **-4** and **3**, respectively. First, the condition

v_num1 > v_num2

of the outer **IF** statement is evaluated. Because **-4** is not greater than **3**, the **ELSE** part of the outer **IF** statement is executed. As a result, the message

ELSE part of the outer IF

is displayed, and the value of **v_total** is calculated. Next, the condition

v_total < 0

of the inner **IF** statement is evaluated. Because that value of **v_total** is equal **-1**, the

condition yields TRUE, and the message

Inner IF

is displayed. Next, the value of `v_total` is calculated again. This logic is demonstrated by the output produced by the example:

ELSE part of the outer IF
Inner IF
`v_total = 1`

Logical Operators

So far in this chapter, you have seen examples of different IF statements. All of these examples used test operators, such as `>`, `<`, and `=`, to evaluate a condition. Logical operators can be used to evaluate a condition as well. In addition, they allow a programmer to combine multiple conditions into a single condition if there is such a need.

For Example `ch04_5a.sql`

[Click here to view code image](#)

```
DECLARE
    v_letter CHAR(1) := '&sv_letter';
BEGIN
    IF (v_letter >= 'A' AND v_letter <= 'Z') OR
        (v_letter >= 'a' AND v_letter <= 'z')
    THEN
        DBMS_OUTPUT.PUT_LINE ('This is a letter');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('This is not a letter');
        IF v_letter BETWEEN '0' and '9'
        THEN
            DBMS_OUTPUT.PUT_LINE ('This is a number');
        ELSE
            DBMS_OUTPUT.PUT_LINE ('This is not a number');
        END IF;
    END IF;
END;
```

In this example, the condition

[Click here to view code image](#)

```
(v_letter >= 'A' AND v_letter <= 'Z') OR
(v_letter >= 'a' AND v_letter <= 'z')
```

uses logical operators AND and OR. Two conditions

[Click here to view code image](#)

```
(v_letter >= 'A' AND v_letter <= 'Z')
```

and

[Click here to view code image](#)

```
(v_letter >= 'a' AND v_letter <= 'z')
```

are combined into one with the help of the OR operator. Notice the purposes of the parentheses. In this example, they are used to improve readability only, because the operator AND takes precedence over the operator OR.

When the symbol “?” is entered at run time, this example produces the following output:

```
This is not a letter  
This is not a number
```

Did You Know?

You can nest **IF** statements to any depth level up to maximum length of a PL/SQL block, and blocks themselves may be nested 255 levels deep.

Consider the following example, where **IF** statements are nested inside each other four levels deep:

[Click here to view code image](#)

```
DECLARE  
    v_var1 PLS_INTEGER := 100;  
    v_var2 PLS_INTEGER := 200;  
    v_var3 PLS_INTEGER := 300;  
    v_var4 PLS_INTEGER := 400;  
BEGIN  
    IF v_var1 >= 100  
    THEN  
        IF v_var2 >= 200  
        THEN  
            IF v_var3 >= 300  
            THEN  
                IF v_var4 >= 400  
                THEN  
                    DBMS_OUTPUT.PUT_LINE  
                    ('v_var1 = ''||v_var1||'', v_var2 = ''||v_var2||'  
                     ', v_var3 = ''||v_var3||'', v_var4 = ''||v_var4);  
                END IF;  
            END IF;  
        END IF;  
    END IF;  
END;
```

While this script is very simple and does not accomplish much, such deep nesting of **IF** statements is much more difficult to follow and may become very complex very quickly when implementing complex business solutions.

In this example, the four nested **IF** statements could be restructured as a single **IF** statement by combining these conditions with the **AND** operator:

[Click here to view code image](#)

```
IF v_var1 >= 100 AND v_var2 >= 200 AND v_var3 >= 300 AND v_var4 >= 400  
THEN  
    ...  
END IF;
```

Summary

In the chapter, you explored different types of IF statements and saw how they can be nested inside one another. You also learned how to employ logical operators when combining multiple distinct conditions into one unified condition for the purpose of evaluation. Conditional control structures are supported by almost every programming language; while the syntax may vary, the manner in which they are used remains unchanged.

In the next chapter, you will continue to learn about conditional control via CASE statements and CASE expressions. In addition, you will learn about the NULLIF and COALESCE functions that are supported by the SQL and PL/SQL languages.

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.

5. Conditional Control: CASE Statements

In this chapter, you will learn about

- [CASE Statements](#)
- [CASE Expressions](#)
- [NULLIF and COALESCE Functions](#)

In the previous chapter, you explored the concept of conditional control via **IF** and **ELSIF** statements. In this chapter, you will continue this exploration by examining different types of **CASE** statements and expressions. You will also learn how to use **NULIF** and **COALESCE** functions, which are considered extensions of **CASE**.

Lab 5.1: CASE Statements

After this lab, you will be able to

- [Use CASE Statements](#)
- [Use Searched CASE Statements](#)
- Use Nested CASE Statements

A **CASE** statement has two forms: **CASE** and **searched CASE**. A **CASE** statement allows you to specify a selector that determines which group of actions to take. A **searched CASE** statement does not have a selector; rather, it has search conditions that are evaluated to determine which group of actions to take.

CASE Statements

A **CASE** statement has structure shown in [Listing 5.1](#).

Listing 5.1 CASE Statement Structure

[Click here to view code image](#)

```
CASE SELECTOR
  WHEN EXPRESSION 1 THEN STATEMENT 1;
  WHEN EXPRESSION 2 THEN STATEMENT 2;
  ...
  WHEN EXPRESSION N THEN STATEMENT N;
  ELSE STATEMENT N+1;
END CASE;
```

The reserved word **CASE** marks the beginning of the **CASE** statement. A selector is a value that determines which **WHEN** clause should be executed. Each **WHEN** clause contains an **EXPRESSION** and one or more executable statements associated with it. The **ELSE** clause is optional and works similar to the **ELSE** clause used in the **IF-THEN-ELSE** statement. **END CASE** is a reserved phrase that indicates the end of the **CASE** statement.

This flow of the logic from the preceding structure of the CASE statement is illustrated in [Figure 5.1](#).

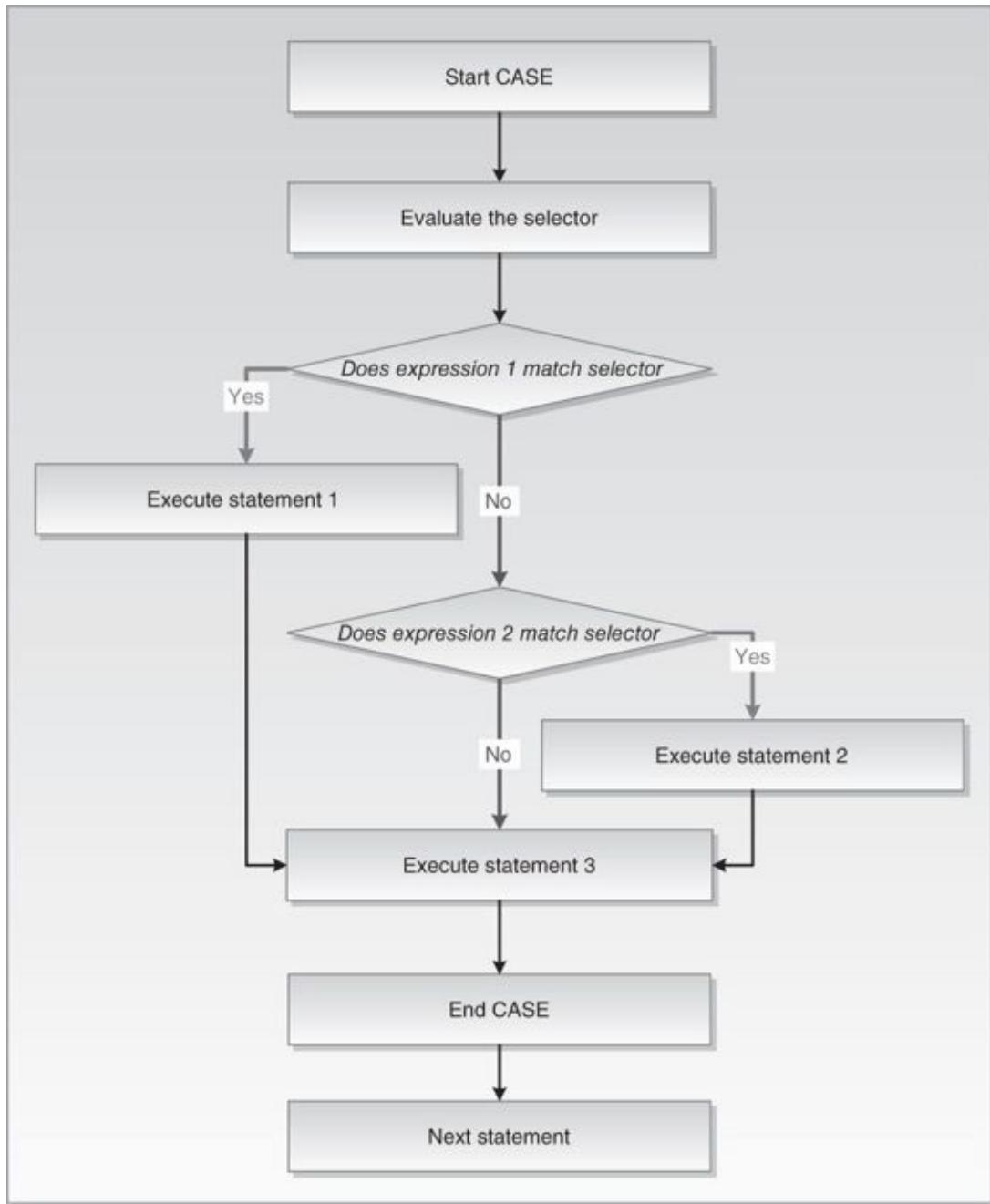


Figure 5.1 CASE Statement

Note that the selector is evaluated only once, and the WHEN clauses are evaluated sequentially. The value of an expression is compared to the value of the selector. If they are equal, the statement associated with a particular WHEN clause is executed, and any subsequent WHEN clauses are not evaluated. If no expression matches the value of the selector, the ELSE clause is executed.

Recall the example of the IF-THEN-ELSE statement from [Chapter 4](#), which is listed here for reference.

For Example *ch05_1a.sql*

[Click here to view code image](#)

```
DECLARE
  v_num NUMBER := &sv_user_num;
```

```

BEGIN
  -- test if the number provided by the user is even
  IF MOD(v_num,2) = 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END IF;
END;

```

Now consider a new version of the same example that uses the CASE statement instead of the IF-THEN-ELSE statement.

For Example ch05_1b.sql

[Click here to view code image](#)

```

DECLARE
  v_num      NUMBER := &sv_user_num;
  v_num_flag NUMBER;
BEGIN
  v_num_flag := MOD(v_num,2);

  -- test if the number provided by the user is even
  CASE v_num_flag
    WHEN 0
    THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
    ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END CASE;
END;

```

In this example, a new variable, `v_num_flag`, is used as a selector for the CASE statement. If the MOD function returns 0, then the number is even; otherwise, it is odd. If `v_num` is assigned the value of 7 at the run time, this example produces the following output:

```
7 is odd number
```

Searched CASE Statements

A searched CASE statement has search conditions that yield Boolean values: TRUE, FALSE, or NULL. When a particular search condition evaluates to TRUE, the group of statements associated with this condition is executed. This structure is shown in [Listing 5.2](#).

Listing 5.2 Searched CASE Statement Structure

[Click here to view code image](#)

```

CASE
  WHEN SEARCH CONDITION 1 THEN STATEMENT 1;
  WHEN SEARCH CONDITION 2 THEN STATEMENT 2;
  ...
  WHEN SEARCH CONDITION N THEN STATEMENT N;
  ELSE STATEMENT N+1;
END CASE;

```

When a search condition evaluates to TRUE, control passes to the statement associated with it. If no search condition evaluates to TRUE, then statements associated with the ELSE clause are executed. Note that the ELSE clause is optional. This flow of logic from the preceding structure of the searched CASE statement is illustrated in [Figure 5.2](#).

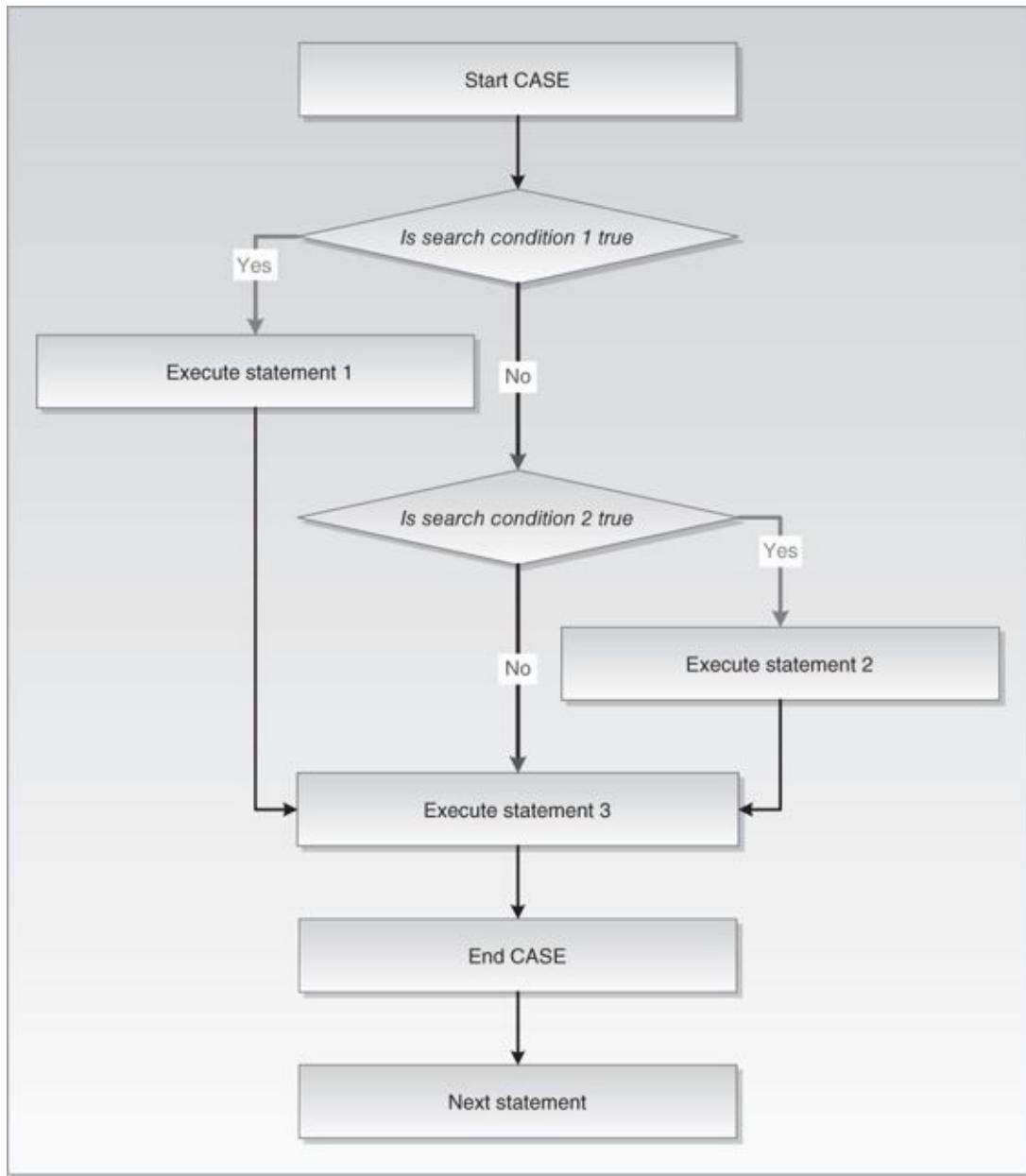


Figure 5.2 Searched CASE Statement

Consider the modified version of the `ch05_1b.sql` example, which you saw earlier in this lab. The changes are highlighted in bold.

For Example `ch05_1c.sql`

[Click here to view code image](#)

```

DECLARE
  v_num NUMBER := &sv_user_num;
BEGIN
  -- test if the number provided by the user is even
CASE
  WHEN MOD(v_num,2) = 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
END CASE;
  
```

```

    ELSE
        DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END CASE;
END;

```

In the previous example, the variable `v_num_flag` was used as a selector, and the result of the MOD function was assigned to it. The value of the selector was then compared to the value of the expression.

This example uses a searched CASE statement, so there is no selector present. The variable `v_num` is used as part of the search conditions, so there is no need to declare variable `v_num_flag`. This example produces the same output when the same value is provided for the `v_num`:

```
7 is odd number
```

Differences between CASE and Searched CASE Statements

It is important to note the differences between the CASE and searched CASE statements. You have seen that the searched CASE statement does not have a selector. In addition, its WHEN clauses contain search conditions that yield a Boolean value similar to the IF statement, not expressions that can yield a value of any type except a PL/SQL record, an index-by-table, a nested table, a vararray, BLOB, BFILE, or an object type. You will encounter some of these types in the future chapters.

Consider the two code fragments shown in [Table 5.1](#), which are based on the examples you saw earlier in this chapter.

CASE Statement	Searched CASE Statement
<pre> DECLARE v_num NUMBER := &sv_user_num; v_num_flag NUMBER; BEGIN v_num_flag := MOD(v_num, 2); -- test if the number provided by -- the user is even CASE v_num_flag WHEN 0 THEN ... END CASE; </pre>	<pre> DECLARE v_num NUMBER := &sv_user_num; BEGIN -- test if the number provided -- by the user is even CASE WHEN MOD(v_num, 2) = 0 THEN ... END CASE; </pre>

Table 5.1 CASE Statement versus Searched CASE Statement

In the code fragment on the left (CASE statement), `v_num_flag` is the selector. It is a PL/SQL variable that has been defined as a NUMBER. Because the value of the expression is compared to the value of the selector, the expression must return a similar data type. The expression ‘0’ contains a number, so its data type is also numeric.

In the code fragment on the right (searched CASE statement), there is no need for the selector, as it has been replaced by the searched expression

```
MOD(v_num, 2) = 0
```

This expression evaluates to TRUE or FALSE just like conditions of an IF statement.

Next, consider an example of the CASE statement that generates a syntax error because the data type returned by the expressions does not match the data type assigned to the selector.

For Example *ch05_2a.sql*

[Click here to view code image](#)

```
DECLARE
    v_num      NUMBER := &sv_num;
    v_num_flag NUMBER;
BEGIN
    CASE v_num_flag
        WHEN MOD(v_num, 2) = 0
        THEN
            DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
        ELSE
            DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END CASE;
END;
```

In this example, the variable **v_num_flag** has been defined as a NUMBER. However, the result of the expression evaluated by the WHEN clause yields a Boolean data type. As a result, this example produces the following syntax error:

[Click here to view code image](#)

```
ORA-06550: line 5, column 9:
PLS-00615: type mismatch found at 'V_NUM_FLAG' between CASE operand and
WHEN operands
ORA-06550: line 5, column 4:
PL/SQL: Statement ignored
```

Now consider a modified version of this example where **v_num_flag** variable has been declared as a Boolean (affected statements are shown in bold).

For Example *ch05_2b.sql*

[Click here to view code image](#)

```
DECLARE
    v_num      NUMBER := &sv_num;
    v_num_flag Boolean;
BEGIN
    CASE v_num_flag
        WHEN MOD(v_num, 2) = 0
        THEN
            DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
        ELSE
            DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END CASE;
END;
```

If **v_num** is assigned the value of 7 again, this example produces the following output:

```
7 is odd number
```

At first glance this seems to be the output that you would expect. However, consider the output produced by this example when the value of 4 is assigned to the variable **v_num**:

```
4 is odd number
```

Watch Out!

The second run of the example produced an incorrect output even though it did not generate any syntax errors. When the value of 4 is assigned to the variable `v_num`, the expression `MOD(v_num, 2) = 0` yields TRUE, and it is compared to the selector `v_num_flag`. However, the `v_num_flag` has not been initialized to any value, so it remains NULL. Because NULL does not equal TRUE, the statement associated with the ELSE clause is executed. To produce the correct output, the `v_num_flag` variable must be initialized to TRUE.

You learned earlier that the expressions in the CASE statements and searched conditions in the searched CASE statements are evaluated sequentially. Also, as soon as the expression or searched condition evaluates to the desired result, the rest of the expressions and searched conditions are ignored. Essentially, at this point, the executable statements associated with that expression or searched condition are executed. Once this execution completes, control passes to the first executable statement after the END CASE clause. This logic implies that the order in which you list the expressions and searched conditions affects the flow of the execution—a relationship illustrated by the following example.

For Example *ch05_3a.sql*

[Click here to view code image](#)

```
DECLARE
    v_final_grade  NUMBER := &sv_final_grade;
    v_letter_grade CHAR(1);
BEGIN
    CASE
        WHEN v_final_grade >= 60
        THEN
            v_letter_grade := 'D';
        WHEN v_final_grade >= 70
        THEN
            v_letter_grade := 'C';
        WHEN v_final_grade >= 80
        THEN
            v_letter_grade := 'B';
        WHEN v_final_grade >= 90
        THEN
            v_letter_grade := 'A';
        ELSE
            v_letter_grade := 'F';
    END CASE;
    -- control resumes here
    DBMS_OUTPUT.PUT_LINE ('Final grade is: '||v_final_grade);
    DBMS_OUTPUT.PUT_LINE ('Letter grade is: '||v_letter_grade);
END;
```

In this example of a searched CASE statement, the value of the letter grade is assigned based on the numeric grade provided by the user at run time. When a value of 67 is provided at run time for the variable `v_final_grade`, this example produces the following output:

```
Final grade is: 67
Letter grade is: D
```

At first sight, this is behavior that you expect. Next, consider the output produced by this example when 94 is assigned to the variable `v_final_grade`:

```
Final grade is: 94
Letter grade is: D
```

In this run, the example produced incorrect output. This error occurred because the first searched condition

```
v_final_grade >= 60
```

evaluated to TRUE and ‘D’ was assigned to the variable `v_letter_grade`. To correct this mistake, the order of the searched conditions could be changed as follows (all changes are highlighted in bold):

For Example `ch05_3b.sql`

[Click here to view code image](#)

```
DECLARE
    v_final_grade  NUMBER := &sv_final_grade;
    v_letter_grade CHAR(1);
BEGIN
    CASE
        WHEN v_final_grade >= 90
        THEN
            v_letter_grade := 'A';
        WHEN v_final_grade >= 80
        THEN
            v_letter_grade := 'B';
        WHEN v_final_grade >= 70
        THEN
            v_letter_grade := 'C';
        WHEN v_final_grade >= 60
        THEN
            v_letter_grade := 'D';
        ELSE
            v_letter_grade := 'F';
    END CASE;
    -- control resumes here
    DBMS_OUTPUT.PUT_LINE ('Final grade is: '||v_final_grade);
    DBMS_OUTPUT.PUT_LINE ('Letter grade is: '||v_letter_grade);
END;
```

In this version of the example, the order of the searched conditions has been reversed. As a result, it produces the correct output in both cases.

```
Final grade is: 67
Letter grade is: D
```

```
Final grade is: 94
Letter grade is: A
```

Lab 5.2: CASE Expressions

After this lab, you will be able to

- Use CASE Expressions

In [Chapter 2](#), you encountered various PL/SQL expressions. Recall that the result of an expression yields a single value that is assigned to a variable. In a similar manner, a CASE expression evaluates to a single value that may be assigned to a variable.

A CASE expression has a structure almost identical to a CASE statement. Thus, it also has two forms: CASE and searched CASE. Consider an example of a CASE statement used in the first lab in this chapter:

For Example *ch05_1d.sql*

[Click here to view code image](#)

```
DECLARE
    v_num      NUMBER := &sv_user_num;
    v_num_flag NUMBER;
BEGIN
    v_num_flag := MOD(v_num,2);

    -- test if the number provided by the user is even
    CASE v_num_flag
        WHEN 0
        THEN
            DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
        ELSE
            DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END CASE;
END;
```

Now consider the new version of the same example, which uses a CASE expression instead of a CASE statement. Changes are shown in bold.

For Example *ch05_1e.sql*

[Click here to view code image](#)

```
DECLARE
    v_num      NUMBER := &sv_user_num;
    v_num_flag NUMBER;
    v_result  VARCHAR2(30);
BEGIN
    v_num_flag := MOD(v_num,2);

    -- test if the number provided by the user is even
    v_result := CASE v_num_flag
        WHEN 0
        THEN
            v_num||' is even number'
        ELSE
            v_num||' is odd number'
        END;
    DBMS_OUTPUT.PUT_LINE (v_result);
END;
```

In this example, a new variable, `v_result`, is used to hold the value returned by the CASE expression. If the variable `v_num` is assigned the value of 8, this example produces the following output:

```
8 is even number
```

It is important to note some syntax differences between a CASE statement and a CASE expression. Consider the code fragments shown in [Table 5.2](#).

CASE Statement	CASE Expression
<pre>CASE v_num_flag WHEN 0 THEN DBMS_OUTPUT.PUT_LINE (v_num ' is even number'); ELSE DBMS_OUTPUT.PUT_LINE (v_num ' is odd number'); END CASE;</pre>	<pre>CASE v_num_flag WHEN 0 THEN v_num ' is even number' ELSE v_num ' is odd number' END;</pre>

Table 5.2 CASE Statement versus CASE Expression

In the CASE statement, the WHEN and ELSE clauses both contain a single executable statement. Each executable statement is terminated by a semicolon. In the CASE expression, the WHEN and ELSE clauses both contain an expression that is not terminated by a semicolon. One semicolon appears after the reserved word END, which terminates the CASE expression. Finally, the CASE statement is terminated by the reserved phrase END CASE.

Next, consider another version of the previous example, with the searched CASE expression (affected statements are shown in bold):

For Example *ch05_1f.sql*

[Click here to view code image](#)

```
DECLARE
  v_num      NUMBER := &sv_user_num;
  v_result  VARCHAR2(30);
BEGIN
  -- test if the number provided by the user is even
  v_result := CASE
    WHEN MOD(v_num,2) = 0
    THEN
      v_num||' is even number'
    ELSE
      v_num||' is odd number'
    END;
  DBMS_OUTPUT.PUT_LINE (v_result);
END;
```

In this example, there is no need to declare the variable `v_num_flag` because the searched CASE expression does not need a selector value, and the result of the MOD function is incorporated into the search condition. When this example is run, it produces output identical to the previous version:

8 is even number

You learned earlier that a CASE expression returns a single value that is then assigned to a variable. In the examples that you saw earlier, this assignment operation was accomplished via the assignment operator, `:=`. You might recall that there is another way to assign a value to a PL/SQL variable—that is, via a `SELECT INTO` statement. Consider an example of the CASE expression used in a `SELECT INTO` statement:

For Example *ch05_4a.sql*

[Click here to view code image](#)

```
DECLARE
    v_course_no    NUMBER;
    v_description  VARCHAR2(50);
    v_prereq       VARCHAR2(35);
BEGIN
    SELECT course_no
          ,description
          ,CASE
              WHEN prerequisite IS NULL
              THEN
                  'No prerequisite course required'
              ELSE
                  TO_CHAR(prerequisite)
              END prerequisite
        INTO v_course_no
          ,v_description
          ,v_prereq
     FROM course
    WHERE course_no = 20;

    DBMS_OUTPUT.PUT_LINE ('Course:      '||v_course_no);
    DBMS_OUTPUT.PUT_LINE ('Description: '||v_description);
    DBMS_OUTPUT.PUT_LINE ('Prerequisite: '||v_prereq);
END;
```

This script displays the course number, description, and number of a prerequisite course on the screen. Furthermore, if a given course does not have a prerequisite course, a message stating that fact is displayed on the screen. To achieve the desired results, a CASE expression is used as one of the columns in the `SELECT INTO` statement. Its value is assigned to the variable `v_prereq`. Notice that there is no semicolon after the reserved word `END` of the CASE expression.

This example produces the following output:

[Click here to view code image](#)

```
Course:      20
Description: Intro to Information Systems
Prerequisite: No prerequisite course required
```

Course 20 does not have a prerequisite course. As a result, the searched condition

```
WHEN prerequisite IS NULL
THEN
```

evaluates to TRUE, and the value “No prerequisite course required” is assigned to the variable `v_prereq`.

It is important to note why the function TO_CHAR is used in the ELSE clause of the CASE expression:

[Click here to view code image](#)

```
CASE
    WHEN prerequisite IS NULL
    THEN
        'No prerequisite course required'
    ELSE
        TO_CHAR(prerequisite)
END
```

A CASE expression returns a single value and, therefore, a single data type. For this reason, it is important to ensure that regardless of which part of a CASE expression executes, it always returns the same data type. In the preceding CASE expression, the WHEN clause returns the VARCHAR2 data type. The ELSE clause returns the value of the PREREQUISITE column of the COURSE table. This column has been defined as a NUMBER, so it is necessary to convert it to the string data type.

When the TO_CHAR function is not used, the CASE expression causes the following syntax error:

[Click here to view code image](#)

```
ORA-06550: line 13, column 17:
PL/SQL: ORA-00932: inconsistent datatypes: expected CHAR got NUMBER
ORA-06550: line 6, column 4:
PL/SQL: SQL Statement ignored
```

Lab 5.3: NULLIF and COALESCE Functions

After this lab, you will be able to

- [Use the NULLIF Function](#)
 - [Use the COALESCE Function](#)
-

The NULLIF and COALESCE functions are defined by the ANSI 1999 standard to be “CASE abbreviations.” Both functions can be used as variations on the CASE expression.

NULLIF Function

The NULLIF function compares two expressions. If they are equal, then the function returns NULL; otherwise, it returns the value of the first expression. The NULLIF function has the structure shown in [Listing 5.3](#).

Listing 5.3 NULLIF Function

[Click here to view code image](#)

```
NULLIF (EXPRESSION 1, EXPRESSION 2)
```

If EXPRESSION 1 is equal to EXPRESSION 2, the NULLIF function returns NULL. If EXPRESSION 1 does not equal EXPRESSION 2, the NULLIF function returns

EXPRESSION 1. Note that the **NULLIF** function does the opposite of the **NVL** function. If the first expression is **NULL**, then **NVL** function returns the second expression. If the first expression is not **NULL**, then the **NVL** function returns the first expression.

The **NULLIF** function is equivalent to the following **CASE** expression:

[Click here to view code image](#)

```
CASE
    WHEN EXPRESSION 1 = EXPRESSION 2 THEN NULL
    ELSE EXPRESSION 1
END
```

Consider the following example of a **NULLIF** function:

For Example *ch05_5a.sql*

[Click here to view code image](#)

```
DECLARE
    v_num          NUMBER := &sv_user_num;
    v_remainder   NUMBER;
BEGIN
    -- calculate the remainder and if it is zero return NULL
    v_remainder := NULLIF(MOD(v_num, 2),0);
    DBMS_OUTPUT.PUT_LINE ('v_remainder: '||v_remainder);
END;
```

This example is somewhat similar to an example that you saw earlier in this chapter. A value is assigned to the variable **v_num** at run time. Next, this value is divided by 2, and its remainder is compared to 0 via the **NULLIF** function. If the remainder equals 0, the **NULLIF** function returns **NULL**; otherwise, it returns the remainder. The value returned by the **NULLIF** function is stored in the variable **v_remainder** and displayed on the screen via the **DBMS_OUTPUT.PUT_LINE** statement.

Suppose that for the first run, 5 is assigned to the variable **v_num**. The example produces the following output:

```
v_remainder: 1
```

Now suppose that for the second run, 4 is assigned to the variable **v_num**. The example produces the following output:

```
v_remainder:
```

In the first run, 5 is not divisible by 2, and the **NULLIF** function returns the value of the remainder. In the second run, 4 is divisible by 2, and the **NULLIF** function returns **NULL** as the value of the remainder.

The **NULLIF** function has a restriction: *You cannot assign a literal **NULL** to EXPRESSION 1.* You learned about literals in [Chapter 2](#). Consider another run of the preceding example, in which the variable **v_num** is assigned a value of **NULL**, as shown in [Figure 5.3](#). In this instance, the example produces the following output:

```
v_remainder:
```

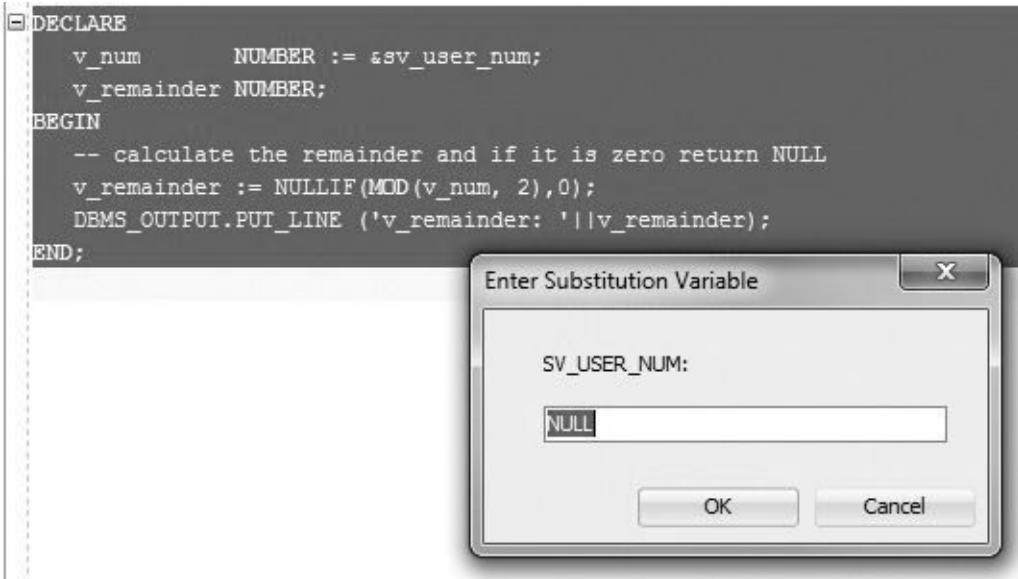


Figure 5.3 Assigning Literal NULL in SQL Developer

When NULL is assigned to the variable `v_num`, both the MOD and NULLIF functions return NULL. The preceding example does not produce any errors because the literal NULL is assigned to the variable `v_num`, and it is not used as the first expression of the NULLIF function.

Now consider a modified version of the preceding example (changes are shown in bold):

For Example *ch05_5b.sql*

[Click here to view code image](#)

```
DECLARE
    v_remainder NUMBER;
BEGIN
    -- calculate the remainder and if it is zero return NULL
    v_remainder := NULLIF(NULL,0);
    DBMS_OUTPUT.PUT_LINE ('v_remainder: '||v_remainder);
END;
```

In the previous version of this example, the MOD function was used as *EXPRESSION 1*. In this version, the literal NULL is used in place of the MOD function, so that the example produces the following syntax error:

[Click here to view code image](#)

```
v_remainder := NULLIF(NULL,0);
*
ERROR at line 5:
ORA-06550: line 5, column 26:
PLS-00619: the first operand in the NULLIF expression must not be NULL
ORA-06550: line 5, column 4:
PL/SQL: Statement ignored
```

COALESCE Function

The COALESCE function compares each expression to NULL from the list of expressions and returns the value of the first non-NULL expression. The COALESCE function has the structure shown in [Listing 5.4](#).

Listing 5.4 COALESCE Function

[Click here to view code image](#)

```
COALESCE (EXPRESSION 1, EXPRESSION 2, ..., EXPRESSION N)
```

If *EXPRESSION 1* evaluates to NULL, then *EXPRESSION 2* is evaluated. If *EXPRESSION 2* does not evaluate to NULL, then the function returns *EXPRESSION 2*. If *EXPRESSION 2* also evaluates to NULL, then the next expression is evaluated. If all expressions evaluate to NULL, the function returns NULL.

Note that the COALESCE function is like a nested NVL function:

[Click here to view code image](#)

```
NVL(EXPRESSION 1  
      ,NVL(EXPRESSION 2  
            ,NVL(EXPRESSION 3,...)  
              )  
        )
```

The COALESCE function can also be used as an alternative to a CASE expression. For example,

[Click here to view code image](#)

```
COALESCE (EXPRESSION 1, EXPRESSION 2)
```

is equivalent to

[Click here to view code image](#)

```
CASE  
  WHEN EXPRESSION 1 IS NOT NULL  
  THEN  
    EXPRESSION 1  
  ELSE  
    EXPRESSION 2  
END
```

If there are more than two expressions to evaluate, then

[Click here to view code image](#)

```
COALESCE (EXPRESSION 1, EXPRESSION 2, ..., EXPRESSION N)
```

is equivalent to

[Click here to view code image](#)

```
CASE  
  WHEN EXPRESSION 1 IS NOT NULL  
  THEN  
    EXPRESSION 1  
  ELSE  
    COALESCE (EXPRESSION 2, ..., EXPRESSION N)  
END
```

which in turn is equivalent to

[Click here to view code image](#)

```
CASE  
  WHEN EXPRESSION 1 IS NOT NULL  
  THEN  
    EXPRESSION 1
```

```

WHEN EXPRESSION 2 IS NOT NULL
THEN
    EXPRESSION 2
...
ELSE
    EXPRESSION N
END

```

Consider the following example of the COALESCE function:

For Example ch05_6a.sql

[Click here to view code image](#)

```

SELECT e.student_id
      ,e.section_id
      ,e.final_grade
      ,g.numeric_grade
      ,COALESCE(e.final_grade, g.numeric_grade, 0) grade
  FROM enrollment e
       ,grade g
 WHERE e.student_id = g.student_id
   AND e.section_id = g.section_id
   AND e.student_id = 102
   AND g.grade_type_code = 'FI';

```

This SELECT statement returns the following output:

[Click here to view code image](#)

STUDENT_ID	SECTION_ID	FINAL_GRADE	NUMERIC_GRADE
102	86	(null)	85
102	89	92	92

The value of GRADE equals the value of NUMERIC_GRADE in the first row. The COALESCE function compares the value of FINAL_GRADE to NULL. If it is NULL, then the value of NUMERIC_GRADE is compared to NULL. Because the value of NUMERIC_GRADE is not NULL, the COALESCE function returns the value of NUMERIC_GRADE. The value of GRADE equals the value of FINAL_GRADE in the second row. The COALESCE function returns the value of FINAL_GRADE because it is not NULL.

The COALESCE function shown in the previous example is equivalent to the following NVL statement and CASE expressions:

[Click here to view code image](#)

```

NVL(e.final_grade, NVL(g.numeric_grade, 0))

CASE
    WHEN e.final_grade IS NOT NULL
    THEN
        e.final_grade
    ELSE
        COALESCE(g.numeric_grade, 0)
END

```

and

[Click here to view code image](#)

```
CASE
  WHEN e.final_grade IS NOT NULL
  THEN
    e.final_grade
  WHEN g.numeric_grade IS NOT NULL
  THEN
    g.numeric_grade
  ELSE
    0
END
```

Summary

In [Chapter 4](#), you began exploring conditional control structures supported by Oracle’s PL/SQL language. In this chapter, you continued this exploration by learning about CASE statements and expressions and COALESCE and NULLIF functions. You have learned how to employ CASE structures in both SQL and PL/SQL languages.

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.

6. Iterative Control: Part I

In this chapter, you will learn about

- [Simple Loops](#)
- [WHILE Loops](#)
- [Numeric FOR Loops](#)

Generally, computer programs are written because certain tasks must be executed a number of times. For example, many companies need to process transactions on a monthly basis. A program allows the completion of this task by being executed at the end of each month.

Similarly, programs incorporate instructions that need to be executed repeatedly. For example, a program may need to write a number of records to a table. Through the use of a loop, the program can write the desired number of records to a table. In other words, loops are programming facilities that allow a set of instructions to be executed repeatedly.

In PL/SQL, there are four types of loops: simple loops, WHILE loops, numeric FOR loops, and cursor FOR loops. In this chapter, you will explore simple loops, WHILE loops, and numeric FOR loops. In [Chapter 7](#), you will see how these types of loops can be nested within one another. In addition, you will learn about the CONTINUE and CONTINUE WHEN statements, which were introduced in Oracle 11g. Cursor FOR loops are discussed in [Chapters 11](#) and [12](#).

Lab 6.1: Simple Loops

After this lab, you will be able to

- Use Simple Loops with EXIT Conditions
- [Use Simple Loops with EXIT WHEN Conditions](#)

A simple loop, as you can see from its name, is the most basic kind of loop and has the structure shown in [Listing 6.1](#).

Listing 6.1 Simple Loop Structure

```
LOOP
  STATEMENT 1;
  STATEMENT 2;
  ...
  STATEMENT N;
END LOOP;
```

The reserved word LOOP marks the beginning of the simple loop. Statements 1 through N are a sequence of statements that is executed repeatedly. These statements consist of one or more of the standard programming structures. END LOOP is a reserved phrase that

indicates the end of the loop construct. The flow of logic from this structure is illustrated in [Figure 6.1](#).

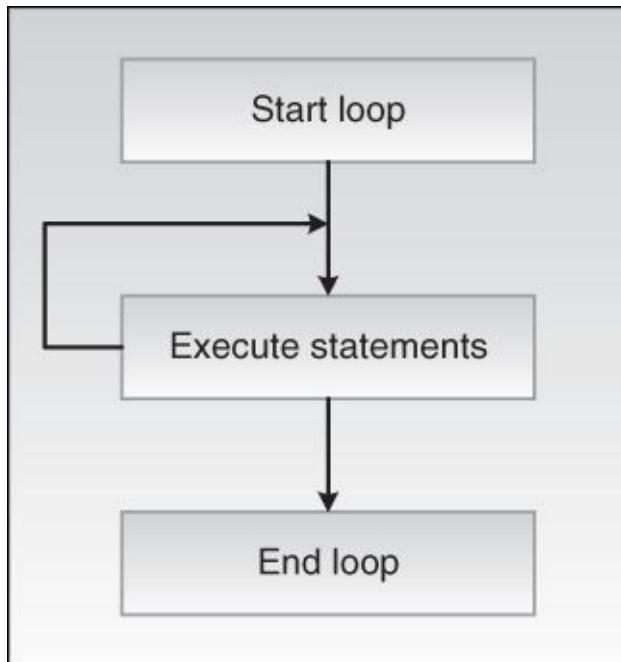


Figure 6.1 Simple Loop

Every time the simple loop is iterated, a sequence of statements is executed, and then control passes back to the top of the loop. The sequence of statements will execute an infinite number of times, because there is no statement specifying when the loop must terminate. Hence, a simple loop is called an infinite loop because there is no means to exit the loop. A properly constructed loop needs to have an exit condition that determines when the loop is complete. This exit condition has two forms: **EXIT** and **EXIT WHEN**.

EXIT Statement

The **EXIT** statement causes a loop to terminate when the exit condition evaluates to **TRUE**. The exit condition is evaluated with the help of an **IF** statement. When the exit condition evaluates to **TRUE**, control passes to the first executable statement after the **END LOOP** statement. The structure of this type of loop is shown in [Listing 6.2](#).

Listing 6.2 Simple Loop Structure with an EXIT Statement

```
LOOP
    STATEMENT 1;
    STATEMENT 2;
    IF EXIT CONDITION THEN
        EXIT;
    END IF;
END LOOP;
STATEMENT 3;
```

In this code listing, you can see that after the **EXIT CONDITION** evaluates to **TRUE**, control passes to **STATEMENT 3**, which is the first executable statement after the **END LOOP** statement. This flow of logic is illustrated in [Figure 6.2](#).

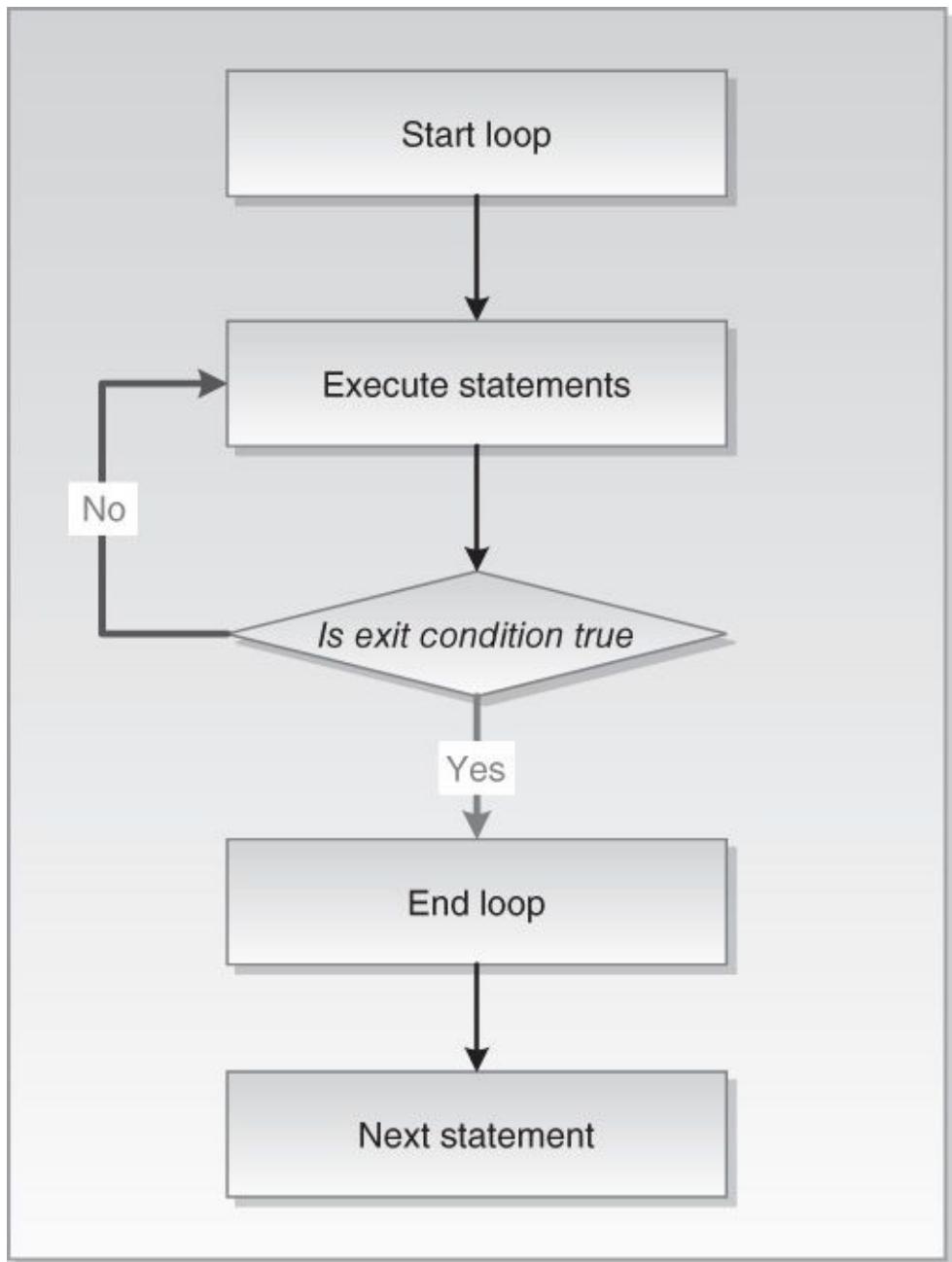


Figure 6.2 Simple Loop with the Exit Condition

As shown in [Figure 6.2](#), during each iteration, the loop executes a sequence of statements. Control then passes to the exit condition of the loop. If the exit condition evaluates to FALSE, control passes to the top of the loop. The sequence of statements will be executed repeatedly until the exit condition evaluates to TRUE. At that point, the loop is terminated via the EXIT statement, and control passes to the next executable statement following the loop.

[Figure 6.2](#) also shows that the exit condition is included in the body of the loop. Therefore, the decision about loop termination is made inside the body of the loop, and the body of the loop, or a part of it, will always be executed at least once. However, the number of iterations of the loop depends on the evaluation of the exit condition and is not known until the loop completes.

This behavior is further illustrated by the following example:

For Example *ch06_1a.sql*

[Click here to view code image](#)

```
DECLARE
    v_counter BINARY_INTEGER := 0;
BEGIN
    LOOP
        -- increment loop counter by one
        v_counter := v_counter + 1;
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

        -- if exit condition yields TRUE exit the loop
        IF v_counter = 5
        THEN
            EXIT;
        END IF;
    END LOOP;
    -- control resumes here
    DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

In this example, the variable `v_counter` keeps count of the loop iterations and is often referred to as a loop counter. The statement

```
v_counter := v_counter + 1;
```

is used frequently when working with loops, as it increments the value of `v_counter` by 1. Once the value of the `v_counter` reaches 5, the exit condition

```
v_counter = 5
```

evaluates to TRUE and the loop terminates. As mentioned previously, as soon as the loop terminates, the control passes to the first executable statement after the `END LOOP` statement.

When executed, this example produces the following output:

```
v_counter = 1
v_counter = 2
v_counter = 3
v_counter = 4
v_counter = 5
Done...
```

Watch Out!

It is very important to initialize the variable `v_counter` for successful termination of the loop. If `v_counter` is not initialized, its value remains NULL and the statement

```
v_counter := v_counter + 1;
```

never increments the value of `v_counter` by 1 because NULL + 1 evaluates to NULL. As a result, the exit condition never evaluates to TRUE, and the loop becomes an infinite loop.

As mentioned previously, when working with the loops, the placement of the exit condition affects whether statements inside the body of the loop are executed during the last iteration of the loop. Consider a modified version of the previous example with the

exit condition placed immediately after the value of `v_counter` is incremented by 1. Affected statements are shown in bold.

For Example *ch06_1b.sql*

[Click here to view code image](#)

```
DECLARE
    v_counter BINARY_INTEGER := 0;
BEGIN
    LOOP
        -- increment loop counter by one
        v_counter := v_counter + 1;

        -- if exit condition yields TRUE exit the loop
        IF v_counter = 5
        THEN
            EXIT;
        END IF;

        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    END LOOP;
    -- control resumes here
    DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

This version of the example produces slightly different output:

```
v_counter = 1
v_counter = 2
v_counter = 3
v_counter = 4
Done...
```

In this version of the script, the portion of the loop before the exit condition executed 5 times. In other words, the variable `v_counter` was incremented by 1 five times.

However, on the fifth iteration of the loop, the exit condition evaluated to TRUE, so the

[Click here to view code image](#)

```
DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
```

statement was not executed. Instead, control passed to the first executable statement after the `END LOOP`. In essence, this placement of the exit condition caused partial execution of the loop body on the last iteration of the loop.

Did You Know?

- The EXIT statement is valid only when placed inside of a loop. When placed outside of a loop, it will cause a syntax error. To avoid this error, use the RETURN statement to terminate a PL/SQL block before its normal end is reached as follows:

[Click here to view code image](#)

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Line 1');
    RETURN;
    DBMS_OUTPUT.PUT_LINE ('Line 2');
END;
```

This example produces the following output:

```
Line 1
```

Because the RETURN statement terminates the PL/SQL block, the second DBMS_OUTPUT.PUT_LINE statement is never executed.

- If used without an exit condition, the EXIT statement will cause the simple loop to execute only once. Consider the following example:

[Click here to view code image](#)

```
DECLARE
    v_counter NUMBER := 0;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
        EXIT;
    END LOOP;
END;
```

This example produces the following output:

```
v_counter = 0
```

Because the EXIT statement is used without an exit condition, the loop terminates as soon as the EXIT statement executes.

EXIT WHEN Statement

The EXIT WHEN statement causes a loop to terminate only if the exit when condition evaluates to TRUE. Control then passes to the first executable statement after the END LOOP statement. The structure of a loop using an EXIT WHEN statement is shown in [Listing 6.3](#).

Listing 6.3 Simple Loop Structure with an EXIT WHEN Statement

```
LOOP
    STATEMENT 1;
    STATEMENT 2;
    EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;
```

[Figure 6.2](#) also illustrates the logic of the `EXIT WHEN` statement, as the flow of logic for the structure of `EXIT` and `EXIT WHEN` statements is the same even though two different forms of exit condition are used. In other words,

```
IF EXIT CONDITION THEN  
    EXIT;  
END IF;
```

is equivalent to

```
EXIT WHEN EXIT CONDITION;
```

This is further illustrated by the following modified version of the example given earlier in this lab (changes are highlighted in bold).

For Example *ch06_1c.sql*

[Click here to view code image](#)

```
DECLARE  
    v_counter BINARY_INTEGER := 0;  
BEGIN  
    LOOP  
        — increment loop counter by one  
        v_counter := v_counter + 1;  
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);  
  
        — if exit condition yields TRUE exit the loop  
        EXIT WHEN v_counter = 5;  
    END LOOP;  
    — control resumes here  
    DBMS_OUTPUT.PUT_LINE ('Done...');  
END;
```

In this version, the `IF` and `EXIT` statements have been replaced by the `EXIT WHEN` statement. As expected, this version produces the same output as the original example:

```
v_counter = 1  
v_counter = 2  
v_counter = 3  
v_counter = 4  
v_counter = 5  
Done...
```

Lab 6.2: WHILE Loops

After this lab, you will be able to

- [Use WHILE loops](#)
- Terminate WHILE loops prematurely

Using WHILE Loops

A WHILE loop has the structure as shown in [Listing 6.4](#).

Listing 6.4 WHILE Loop Structure

```

WHILE TEST CONDITION LOOP
  STATEMENT 1;
  STATEMENT 2;
  ...
  STATEMENT N;
END LOOP;

```

The reserved word **WHILE** marks the beginning of a loop construct. The **TEST CONDITION** is the test condition of the loop that evaluates to **TRUE** or **FALSE**. The result of this evaluation determines whether the loop is executed. Statements 1 through N are a sequence of statements that is executed repeatedly. **END LOOP** is a reserved phrase that indicates the end of the loop construct. This flow of the logic is illustrated in [Figure 6.3](#).

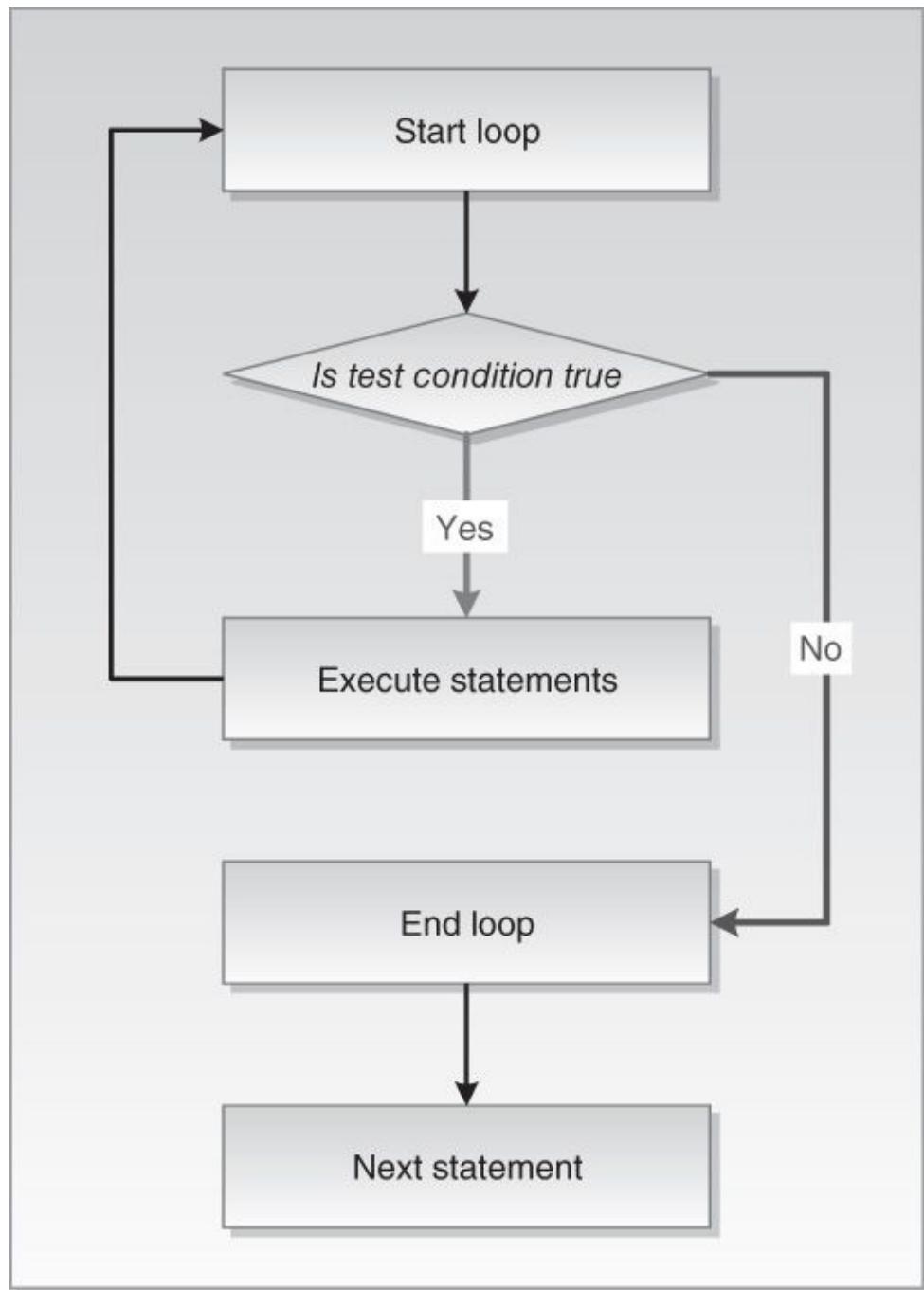


Figure 6.3 WHILE Loop

[Figure 6.3](#) shows that the test condition is evaluated prior to each iteration of the loop. If the **TEST CONDITION** evaluates to **TRUE**, the sequence of statements is executed, and control passes to the top of the loop for the next evaluation of the test condition. If the

TEST CONDITION evaluates to FALSE, the loop is terminated, and control passes to the next executable statement following the loop.

As mentioned earlier, before the body of the loop can be executed, the test condition must be evaluated. The decision as to whether to execute the statements in the body of the loop is made prior to entering the loop. As a result, the loop will not be executed at all if the test condition yields FALSE.

For Example ch06_2a.sql

[Click here to view code image](#)

```
DECLARE
    v_counter NUMBER := 5;
BEGIN
    WHILE v_counter < 5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

        -- decrement the value of v_counter by one
        v_counter := v_counter - 1;
    END LOOP;
END;
```

In this example, the body of the loop is not executed at all because the test condition

```
v_counter < 5
```

of the loop evaluates to FALSE as the variable **v_counter** is initialized to 5.

The test condition must evaluate to TRUE at least once for the statements in the loop to execute. However, it is also important to ensure that the test condition will eventually evaluate to FALSE. Otherwise, the WHILE loop will execute continually, as demonstrated by the following example (changes are shown in bold).

For Example ch06_2b.sql

[Click here to view code image](#)

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    WHILE v_counter < 5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

        -- decrement the value of v_counter by one
        v_counter := v_counter - 1;
    END LOOP;
END;
```

This is an example of an infinite WHILE loop. The test condition always evaluates to TRUE, because the value of **v_counter** is decremented by 1 and is always less than 5.

Now consider a modified version of this example, where the loop executes four times. In this example, the test condition eventually evaluates to FALSE because the value of **v_counter** is incremented by 1. Affected statements are shown in bold.

For Example ch06_2c.sql

[Click here to view code image](#)

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    WHILE v_counter < 5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

        -- increment the value of v_counter by one
        v_counter := v_counter + 1;
    END LOOP;
END;
```

This version of the example produces the following output:

```
v_counter = 1
v_counter = 2
v_counter = 3
v_counter = 4
```

Did You Know?

Boolean expressions can also be used to determine when the loop should terminate.

```
DECLARE
    v_test BOOLEAN := TRUE;
BEGIN
    WHILE v_test
    LOOP
        STATEMENTS;
        IF TEST CONDITION
        THEN
            v_test := FALSE;
        END IF;
    END LOOP;
END;
```

When using a Boolean expression as a test condition of a loop, you must ensure that a different value is eventually assigned to the Boolean variable to exit the loop. Otherwise, the loop will become infinite.

Premature Termination of the WHILE Loop

The EXIT and EXIT WHEN statements can be used inside the body of a WHILE loop. If the exit condition evaluates to TRUE before the test condition evaluates to FALSE, the loop is terminated prematurely. If the test condition evaluates to FALSE before the exit condition evaluates to TRUE, there is no premature termination of the loop. This structure is shown in [Listing 6.5](#).

Listing 6.5 Premature Termination of the WHILE Loop

```
WHILE TEST CONDITION LOOP
    STATEMENT 1;
    STATEMENT 2;
```

```

IF EXIT CONDITION
THEN
    EXIT;
END IF;
END LOOP;
STATEMENT 3;

```

Or

```

WHILE TEST CONDITION
LOOP
    STATEMENT 1;
    STATEMENT 2;
    EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;

```

Consider the following example:

For Example ch06_3a.sql

[Click here to view code image](#)

```

DECLARE
    v_counter NUMBER := 1;
BEGIN
    WHILE v_counter <= 5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

        IF v_counter = 2
        THEN
            EXIT;
        END IF;

        v_counter := v_counter + 1;
    END LOOP;
END;

```

Before the statements in the body of the WHILE loop are executed, the test condition

```
v_counter <= 5
```

must evaluate to TRUE. Then, the value of v_counter is displayed on the screen and incremented by 1. Next, the exit condition

```
v_counter = 2
```

is evaluated. As soon as the value of v_counter reaches 2, the loop is terminated.

According to the test condition, the loop should execute five times. However, the loop is executed only twice, because the exit condition is present inside the body of the loop. Therefore, the loop terminates prematurely.

Now try to reverse the test condition and the exit condition, as shown in the following example (all changes are shown in bold).

For Example ch06_3b.sql

[Click here to view code image](#)

```

DECLARE
    v_counter NUMBER := 1;

```

```

BEGIN
  WHILE v_counter <= 2
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    v_counter := v_counter + 1;

    IF v_counter = 5
    THEN
      EXIT;
    END IF;
  END LOOP;
END;

```

In this version of the example, the test condition is

```
v_counter <= 2
```

and the exit condition is

```
v_counter = 5
```

In this case, the loop is executed twice. However, it does not terminate prematurely, because the exit condition never evaluates to TRUE. As soon as the value of `v_counter` reaches 3, the test condition evaluates to FALSE, and the loop is terminated.

Both examples, when run, produce the following output:

```
v_counter = 1
v_counter = 2
```

These examples demonstrate not only the use of the `EXIT` statement inside the body of the `WHILE` loop, but also a bad programming practice. In the first example, the test condition can be changed so that there is no need to use an exit condition, because essentially both conditions are used to terminate the loop. In the second example, the exit condition is useless, because its terminal value is never reached. You should never include unnecessary code in your programs.

Lab 6.3: Numeric FOR Loops

After this lab, you will be able to

- Use numeric FOR loops with the `IN` option
- Use numeric FOR loops with the `REVERSE` option
- Terminate numeric FOR loops prematurely

A numeric FOR loop is called numeric because it requires an integer as its terminating value. The structure of such a loop is shown in [Listing 6.6](#).

Listing 6.6 Numeric FOR Loop Structure

[Click here to view code image](#)

```

FOR loop_counter IN [REVERSE] lower_limit..upper_limit
LOOP
  STATEMENT 1;
  STATEMENT 2;

```

```
...
STATEMENT N;
END LOOP;
```

The reserved word **FOR** marks the beginning of the **FOR** loop construct. The variable **loop_counter** is an implicitly defined index variable. There is no need to define the loop counter in the declaration section of the PL/SQL block; instead, this variable is defined by the loop construct. The **lower_limit** and **upper_limit** are integer numbers or expressions that evaluate to integer values at run time, and the double dot (. .) serves as the range operator. The **lower_limit** and **upper_limit** define the number of iterations for the loop, and their values are evaluated once, for the first iteration of the loop. At this point, it is determined how many times the loop will iterate. Statements 1 through N are a sequence of statements that is executed repeatedly. **END LOOP** is a reserved phrase that marks the end of the loop construct.

One of the reserved words **IN** or **IN REVERSE** must be present when defining the loop. When the **REVERSE** keyword is used, the loop counter will iterate from the upper limit to the lower limit. However, the syntax for the limit specification does not change. The lower limit is always referenced first. The flow of this logic is illustrated in [Figure 6.4](#).

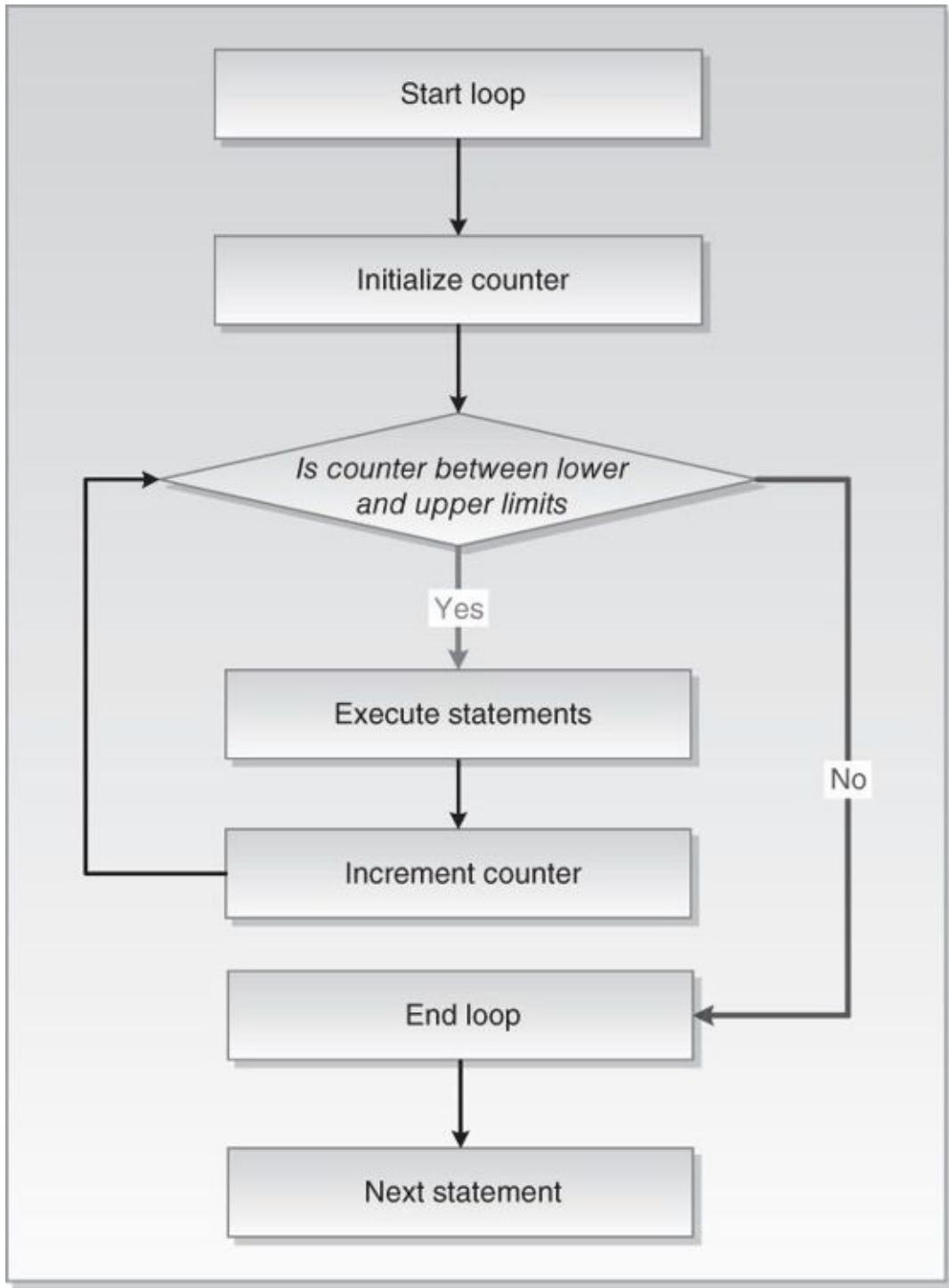


Figure 6.4 Numeric FOR Loop

[Figure 6.4](#) shows that the loop counter is initialized to the lower limit for the first iteration of the loop only. However, the value of the loop counter is tested for each iteration of the loop. As long as the value of `v_counter` ranges from the lower limit to the upper limit, the statements inside the body of the loop are executed. When the value of the loop counter falls outside the range specified by the lower limit and the upper limit, control passes to the first executable statement outside the loop.

Using the IN Option in the Loop

Consider the following example, which illustrates a numeric FOR loop that employs the IN option.

For Example *ch06_4a.sql*

[Click here to view code image](#)

```
BEGIN
  FOR v_counter IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
END;
```

In this example, there is no declaration section for the PL/SQL block because the only variable used, **v_counter**, is the loop counter. Numbers 1..5 specify the range of the integer numbers for which this loop is executed.

Notice that there is no statement

```
v_counter := v_counter + 1;
```

anywhere, either inside or outside the body of the loop. The value of **v_counter** is incremented implicitly by the **FOR** loop itself.

This example produces the following output when run:

```
v_counter = 1
v_counter = 2
v_counter = 3
v_counter = 4
v_counter = 5
```

If you include the statement

```
v_counter := v_counter + 1;
```

in the body of the loop, the PL/SQL script will report errors when you try to compile it. Consider the following example (newly added statement is shown in bold).

For Example *ch06_4b.sql*

[Click here to view code image](#)

```
BEGIN
  FOR v_counter IN 1..5
  LOOP
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE ('v_counter = '|| v_counter);
  END LOOP;
END;
```

When this example is run, it produces the following error message:

[Click here to view code image](#)

```
ORA-06550: line 4, column 7:
PLS-00363: expression 'V_COUNTER' cannot be used as an assignment target
ORA-06550: line 4, column 7:
PL/SQL: Statement ignored
```

Watch Out!

The loop counter is implicitly defined and incremented when a numeric FOR loop is used. As a result, it cannot be referenced outside the body of the FOR loop. Consider the following example:

[Click here to view code image](#)

```
BEGIN
    FOR v_counter IN 1..5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('Counter outside the loop is
        '||v_counter);
END;
```

When this example is run, it produces the following error message:

[Click here to view code image](#)

```
('Counter outside the loop is '||v_counter);
*
ORA-06550: line 6, column 58:
PLS-00201: identifier 'V_COUNTER' must be declared
ORA-06550: line 6, column 4:
PL/SQL: Statement ignored
```

Because the loop counter is declared implicitly by the loop, the variable `v_counter` cannot be referenced outside the loop. As soon as the loop completes, the loop counter ceases to exist.

Using the REVERSE Option in the Loop

Earlier in this lab, you encountered two options that are available when the value of the loop counter is evaluated, IN and IN REVERSE. You have already seen examples that demonstrate the usage of the IN option for the loop. The next example demonstrates the usage of the IN REVERSE option for the loop.

For Example `ch06_5a.sql`

[Click here to view code image](#)

```
BEGIN
    FOR v_counter IN REVERSE 1..5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    END LOOP;
END;
```

When this example is run, it produces the following output:

```
v_counter = 5
v_counter = 4
v_counter = 3
v_counter = 2
v_counter = 1
```

As mentioned earlier, even though the REVERSE keyword is present, the lower limit of

the loop is referenced first. However, the loop counter is evaluated from the upper limit to the lower limit. For the first iteration of the loop, `v_counter` (in our case, it is a loop counter) is initialized to 5 (the upper limit). Then its value is displayed on the screen. For the second iteration of the loop, the value of `v_counter` is decreased by 1, and displayed on the screen.

The number of times the body of the loop is executed is not affected by which option is used, `IN` or `IN REVERSE`. Only the values assigned to the lower limit and the upper limit determine how many times the body of the loop executes.

Premature Termination of the Numeric FOR Loop

The `EXIT` and `EXIT WHEN` statements covered in the previous labs can be used inside the body of a numeric `FOR` loop as well. If the exit condition evaluates to `TRUE` before the loop counter reaches its terminal value, the `FOR` loop is terminated prematurely. If the loop counter reaches its terminal value before the exit condition yields `TRUE`, there is no premature termination of the `FOR` loop. This structure is shown in [Listing 6.7](#).

Listing 6.7 Premature Termination of the Numeric FOR Loop

[Click here to view code image](#)

```
FOR loop_counter IN lower_limit..upper_limit
LOOP
    STATEMENT 1;
    STATEMENT 2;
    IF EXIT CONDITION THEN
        EXIT;
    END IF;
END LOOP;
STATEMENT 3;
```

Or

[Click here to view code image](#)

```
FOR loop_counter IN lower_limit..upper_limit
LOOP
    STATEMENT 1;
    STATEMENT 2;
    EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;
```

Consider the following example of a `FOR` loop that uses the `exit when` condition. This condition causes the loop to terminate prematurely.

For Example `ch06_6a.sql`

[Click here to view code image](#)

```
BEGIN
    FOR v_counter IN 1..5
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
        EXIT WHEN v_counter = 3;
    END LOOP;
END;
```

According to the range specified, the loop should execute five times. However, the loop executes only three times because the exit condition appears inside the body of the loop. Thus, the loop terminates prematurely, as indicated by the example's output:

```
v_counter = 1  
v_counter = 2  
v_counter = 3
```

Summary

In this chapter, you explored three types of loops supported in PL/SQL. You also learned how to employ exit conditions to prevent infinite loops and how to terminate loops prematurely. In the next chapter, you will continue to learn about loops and discover how they can be nested inside one another. Furthermore, you will learn about other loop features, `CONTINUE` and `CONTINUE WHEN`, that were introduced in Oracle 11g.

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.

7. Iterative Control: Part II

In this chapter, you will learn about

- [CONTINUE Statements](#)
- [Nested Loops](#)

In [Chapter 6](#), you explored three types of loops: simple loops, WHILE loops, and numeric FOR loops. You also learned that these types of loops can be terminated with an exit condition. In this chapter, you will learn about a new PL/SQL feature introduced in Oracle 11g called continue condition. Similar to the exit condition, the continue condition has two forms, CONTINUE and CONTINUE WHEN, and may be used inside the body of the loop only. You will also learn how to nest different types of loops inside one another.

Lab 7.1: CONTINUE Statement

After this lab, you will be able to

- [Use CONTINUE Statements](#)
- [Use CONTINUE WHEN Statements](#)

As mentioned previously, the continue condition has two forms: CONTINUE and CONTINUE WHEN.

Using CONTINUE Statement

The CONTINUE statement causes a loop to terminate its current iteration and pass control to the next iteration of the loop when the continue condition evaluates to TRUE. The continue condition is evaluated with the help of an IF statement. When the continue condition evaluates to TRUE, control passes to the first executable statement in the body of the loop. This structure is shown in [Listing 7.1](#).

Listing 7.1 Simple Loop Structure with a CONTINUE Statement

```
LOOP
  STATEMENT 1;
  STATEMENT 2;
  IF CONTINUE CONDITION THEN
    CONTINUE;
  END IF;
  STATEMENT 3;

  EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 4;
```

As soon as the *CONTINUE CONDITION* evaluates to TRUE, control passes back to *STATEMENT 1*, which is the first executable statement inside the body of the loop. In this

case, it causes partial execution of the loop, as the statements following after the continue condition inside the body of the loop are not executed. This flow of logic for the `CONTINUE` statement is illustrated in [Figure 7.1](#).

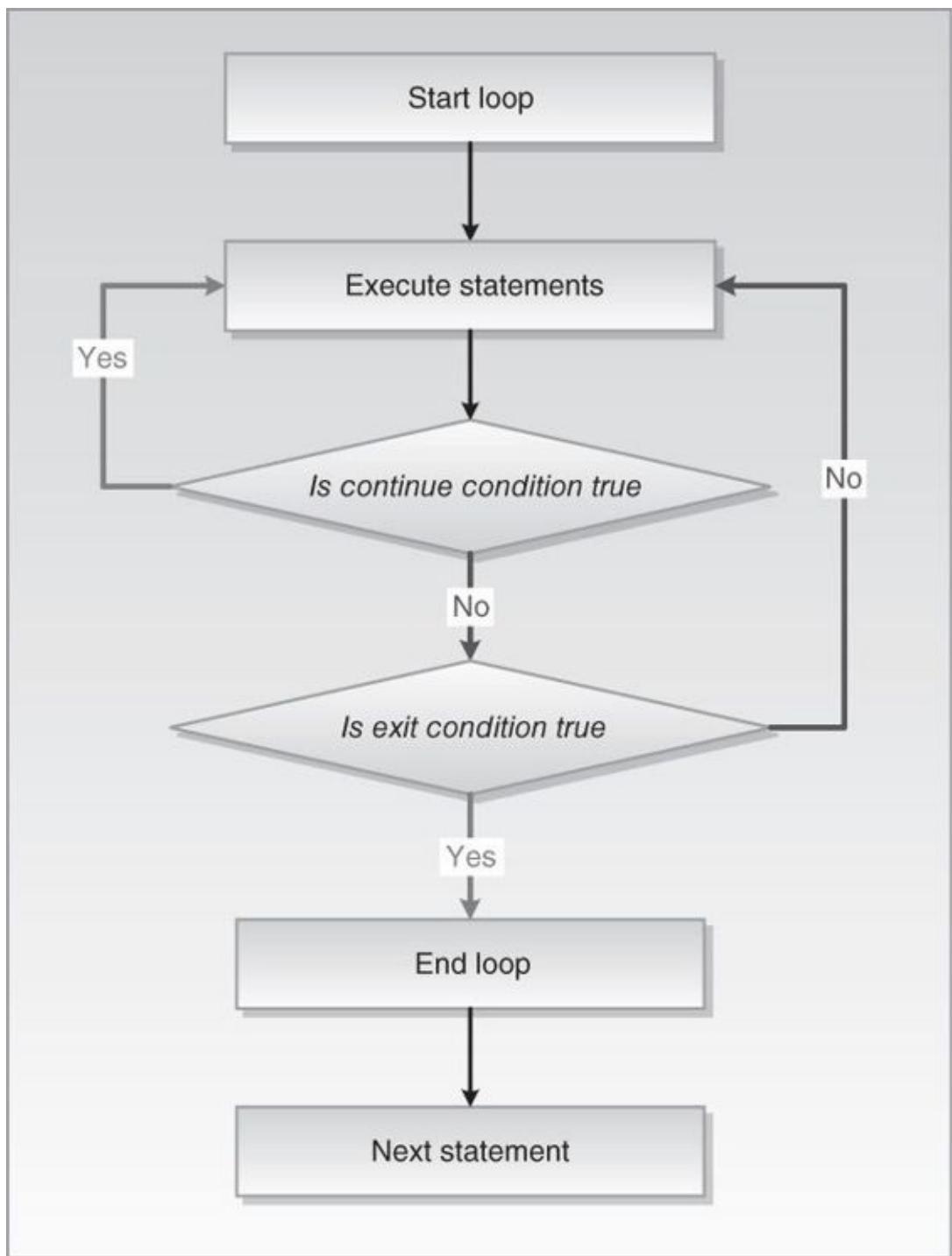


Figure 7.1 Simple Loop with the Continue Condition

As shown in [Figure 7.1](#), during each iteration, the loop executes a sequence of statements. Control then passes to the `CONTINUE CONDITION` of the loop. If the `CONTINUE CONDITION` evaluates to `TRUE`, control passes to the top of the loop. The sequence of statements will be executed repeatedly until the `CONTINUE CONDITION` evaluates to `FALSE`. When the `CONTINUE CONDITION` evaluates to `FALSE`, control passes to the next executable statement in the body of the loop, which in this case evaluates the `EXIT CONDITION`.

Did You Know?

- CONTINUE and CONTINUE WHEN statements can be used with all types of loops.
 - The difference between the exit and continue conditions is that the exit condition terminates the loop, whereas the continue condition terminates the current iteration of the loop.
-

Consider the following example, which illustrates how continue and exit conditions affect loop execution.

For Example *ch07_1a.sql*

[Click here to view code image](#)

```
DECLARE
    v_counter BINARY_INTEGER := 0;
BEGIN
    LOOP
        -- increment loop counter by one
        v_counter := v_counter + 1;
        DBMS_OUTPUT.PUT_LINE
            ('Before continue condition, v_counter = '||v_counter);

        -- if continue condition yields TRUE pass control to the first
        -- executable statement of the loop
        IF v_counter < 3
        THEN
            CONTINUE;
        END IF;
        DBMS_OUTPUT.PUT_LINE
            ('After continue condition, v_counter = '||v_counter);

        -- if exit condition yields TRUE exit the loop
        IF v_counter = 5
        THEN
            EXIT;
        END IF;

    END LOOP;
    -- control resumes here
    DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

When run, this script produces the following output:

[Click here to view code image](#)

```
Before continue condition, v_counter = 1
Before continue condition, v_counter = 2
Before continue condition, v_counter = 3
After continue condition, v_counter = 3
Before continue condition, v_counter = 4
After continue condition, v_counter = 4
Before continue condition, v_counter = 5
After continue condition, v_counter = 5
Done...
```

Next, let's take a closer look at what happens inside the body of the loop during its execution. For the first two iterations of the loop (the values of `v_counter` are 1 and 2, respectively), the continue condition

```
IF v_counter < 3
```

evaluates to TRUE, and control of the execution passes to the first statement inside the body of the loop. As a result, the value of `v_counter` is incremented by 1 and only the first `DBMS_OUTPUT.PUT_LINE` statement is executed:

[Click here to view code image](#)

```
Before continue condition, v_counter = 1
Before continue condition, v_counter = 2
```

In other words, for the first two iterations, only part of the loop prior to the `CONTINUE` statement is executed.

For the last three iterations of the loops (the values of `v_counter` are 3, 4, and 5, respectively) the continue condition evaluates to FALSE, and the second `DBMS_OUTPUT.PUT_LINE` is executed:

[Click here to view code image](#)

```
Before continue condition, v_counter = 3
After continue condition, v_counter = 3
Before continue condition, v_counter = 4
After continue condition, v_counter = 4
Before continue condition, v_counter = 5
After continue condition, v_counter = 5
```

In this case, all statements inside the body of the loop are executed.

Finally, when the value `v_counter` reaches 5, the exit condition

```
IF v_counter = 5
```

evaluates to TRUE and the loop terminates. The last `DBMS_OUTPUT.PUT_LINE` statement is executed as well.

Watch Out!

When the **CONTINUE** statement is used without a continue condition, the current iteration of the loop will terminate unconditionally and control of the execution will pass to the first executable statement in the body of the loop. Consider the following example:

[Click here to view code image](#)

```
DECLARE
    v_counter NUMBER := 0;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = ' || v_counter);
        CONTINUE;

        v_counter := v_counter + 1;
        EXIT WHEN v_counter = 5;
    END LOOP;
END;
```

Because the **CONTINUE** statement is used without a continue condition, this loop will never reach its **EXIT WHEN** condition and as a result will never terminate.

CONTINUE WHEN Statement

The **CONTINUE WHEN** statement causes a loop to terminate its current iteration and pass control to the next iteration of the loop only when the continue condition evaluates to **TRUE**. Control then passes to the first executable statement inside the body of the loop. The structure of a loop using a **CONTINUE WHEN** clause is shown in [Listing 7.2](#).

Listing 7.2 *Simple Loop Structure with a CONTINUE WHEN Statement*

[Click here to view code image](#)

```
LOOP
    STATEMENT 1;
    STATEMENT 2;
    CONTINUE WHEN CONTINUE CONDITION;

    EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;
```

Note that the flow of the logic illustrated in [Figure 7.1](#) applies to the **CONTINUE WHEN** statement as well. In other words,

```
IF CONDITION
THEN
    CONTINUE;
END IF;
```

is equivalent to

```
CONTINUE WHEN CONDITION;
```

This similarity is illustrated further by the modified version of the previous example. When executed, this version produces output much like that of the previous version (affected statements are shown in bold).

For Example *ch07_1b.sql*

[Click here to view code image](#)

```
DECLARE
    v_counter BINARY_INTEGER := 0;
BEGIN
    LOOP
        – increment loop counter by one
        v_counter := v_counter + 1;
        DBMS_OUTPUT.PUT_LINE
            ('Before continue condition, v_counter = '||v_counter);

        – if continue condition yields TRUE pass control to the first
        – executable statement of the loop
CONTINUE WHEN v_counter < 3;

        DBMS_OUTPUT.PUT_LINE
            ('After continue condition, v_counter = '||v_counter);

        – if exit condition yields TRUE exit the loop
        IF v_counter = 5
        THEN
            EXIT;
        END IF;

    END LOOP;
    – control resumes here
    DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

Watch Out!

The **CONTINUE** and **CONTINUE WHEN** statements are valid only when placed inside a loop. When placed outside a loop, they will cause a syntax error.

When you are working with the exit and continue conditions, the execution of a loop and the number of iterations are affected *by the placement of those conditions inside the body of the loop*. This is illustrated further by the following example (changes are highlighted in bold):

For Example *ch07_1c.sql*

[Click here to view code image](#)

```
DECLARE
    v_counter BINARY_INTEGER := 0;
BEGIN
    LOOP
        – increment loop counter by one
        v_counter := v_counter + 1;
        DBMS_OUTPUT.PUT_LINE
            ('Before continue condition, v_counter = '||v_counter);
```

```

– if continue condition yields TRUE pass control to the first
– executable statement of the loop
CONTINUE WHEN v_counter > 3;

DBMS_OUTPUT.PUT_LINE
    ('After continue condition, v_counter = '||v_counter);

– if exit condition yields TRUE exit the loop
IF v_counter = 5
THEN
    EXIT;
END IF;

END LOOP;
– control resumes here
DBMS_OUTPUT.PUT_LINE ('Done...');
END;

```

In this version of the script, the continue condition has been changed to

```
CONTINUE WHEN v_counter > 3;
```

This change leads to an infinite loop. As long as the value of **v_counter** is less than or equal to 3, the continue condition evaluates to FALSE. Therefore, for the first three iterations of the loop, all statements inside the body of the loop are executed along with the exit condition, which evaluates to FALSE.

Starting with the fourth iteration of the loop, the continue condition evaluates to TRUE, causing partial execution of the loop. Due to this partial execution, the exit condition cannot be reached, causing this loop to become infinite. To mitigate this situation, the placement of the continue and exit conditions should be changed as shown in the following example (changes are shown in bold):

For Example *ch07_1d.sql*

[Click here to view code image](#)

```

DECLARE
    v_counter BINARY_INTEGER := 0;
BEGIN
    LOOP
        – increment loop counter by one
        v_counter := v_counter + 1;

        – if exit condition yields TRUE exit the loop
IF v_counter = 5
THEN
    EXIT;
END IF;

DBMS_OUTPUT.PUT_LINE
    ('Before continue condition, v_counter = '||v_counter);

– if continue condition yields TRUE pass control to the first
– executable statement of the loop
CONTINUE WHEN v_counter > 3;

DBMS_OUTPUT.PUT_LINE

```

```

        ('After continue condition, v_counter = '||v_counter);
END LOOP;
-- control resumes here
DBMS_OUTPUT.PUT_LINE ('Done...');
END;

```

In this version of the script, the exit condition appears before the continue condition. Such placement of the exit condition guarantees eventual termination of the loop, as illustrated by the following output:

[Click here to view code image](#)

```

Before continue condition, v_counter = 1
After continue condition, v_counter = 1
Before continue condition, v_counter = 2
After continue condition, v_counter = 2
Before continue condition, v_counter = 3
After continue condition, v_counter = 3
Before continue condition, v_counter = 4
Done...

```

Here, on the fifth iteration of the loop, the value of `v_counter` is incremented by 1, and the exit condition evaluates to TRUE. As a result, none of the `DBMS_OUTPUT.PUT_LINE` statements inside the body of the loop are executed; instead, control of the execution passes to the first executable statement after the `END LOOP` and “Done...” is displayed on the screen.

Lab 7.2: Nested Loops

After this lab, you will be able to

- [Use Nested Loops](#)
- [Use Loop Labels](#)

Using Nested Loops

You have explored three types of loops: simple loops, `WHILE` loops, and numeric `FOR` loops. Any of these three types of loops can be nested inside one another. For example, a simple loop can be nested inside a `WHILE` loop, and vice versa. Consider the following example:

For Example `ch07_2a.sql`

[Click here to view code image](#)

```

DECLARE
    v_counter1 BINARY_INTEGER := 0;
    v_counter2 BINARY_INTEGER;
BEGIN
    WHILE v_counter1 < 3
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter1: '||v_counter1);
        v_counter2 := 0;
        LOOP
            DBMS_OUTPUT.PUT_LINE ('  v_counter2: '||v_counter2);

```

```

    v_counter2 := v_counter2 + 1;
    EXIT WHEN v_counter2 >= 2;
END LOOP;
v_counter1 := v_counter1 + 1;
END LOOP;
END;

```

In this example, the **WHILE** loop is called an outer loop because it encompasses the simple loop. The simple loop (highlighted in bold) is called an inner loop because it is enclosed by the body of the **WHILE** loop.

The outer loop is controlled by the loop counter, **v_counter1**, and it will execute providing the value of **v_counter1** is less than 3. With each iteration of the loop, the value of **v_counter1** is displayed on the screen. Next, the value of **v_counter2** is initialized to 0. Note that **v_counter2** is not initialized at the time of the declaration. The simple loop is placed inside the body of the **WHILE** loop, so the value of **v_counter2** must be initialized every time before control passes to the simple loop.

Once control passes to the inner loop, the value of **v_counter2** is displayed on the screen and incremented by 1. Next, the exit when condition is evaluated. If this condition evaluates to **FALSE**, control passes back to the top of the simple loop. If it evaluates to **TRUE**, control passes to the first executable statement outside the loop. In our case, control passes back to the outer loop, the value of **v_counter1** is incremented by 1, and the test condition of the **WHILE** loop is evaluated again.

This logic is demonstrated by the output produced by the example:

```

v_counter1: 0
  v_counter2: 0
  v_counter2: 1
v_counter1: 1
  v_counter2: 0
  v_counter2: 1
v_counter1: 2
  v_counter2: 0
  v_counter2: 1

```

Notice that for each value of **v_counter1**, two values of **v_counter2** are displayed. For the first iteration of the outer loop, the value of **v_counter1** is equal to 0. Once control passes to the inner loop, the value of **v_counter2** is displayed on the screen twice, and so forth.

Using Loop Labels

Earlier in the book, you learned about labeling of PL/SQL blocks. Loops can be labeled in a similar manner, as illustrated in [Listing 7.3](#).

Listing 7.3 Loop Labels

[Click here to view code image](#)

```

<<label_name>>
FOR loop_counter IN lower_limit..upper_limit
LOOP
  STATEMENT 1;

```

```
...
STATEMENT N;
END LOOP label_name;
```

The label must appear immediately before the beginning of the loop. The preceding syntax shows that the label can be optionally used at the end of the loop statement. It is very helpful to label nested loops—such labels improve the script's readability. Consider the following example:

For Example *ch07_3a.sql*

[Click here to view code image](#)

```
BEGIN
<<outer_loop>>
FOR i IN 1..3
LOOP
    DBMS_OUTPUT.PUT_LINE ('i = '||i);
    <<inner_loop>>
    FOR j IN 1..2
    LOOP
        DBMS_OUTPUT.PUT_LINE ('j = '||j);
    END LOOP inner_loop;
END LOOP outer_loop;
END;
```

For both outer and inner loops, the statement `END LOOP` must be used. If the loop label is added to each `END LOOP` statement, it becomes easier to understand which loop is being terminated.

Loop labels can also be used when referencing loop counters, as shown in the following example:

For Example *ch07_4a.sql*

[Click here to view code image](#)

```
BEGIN
<<outer>>
FOR v_counter IN 1..3
LOOP
    <<inner>>
    FOR v_counter IN 1..2
    LOOP
        DBMS_OUTPUT.PUT_LINE ('outer.v_counter '||outer.v_counter);
        DBMS_OUTPUT.PUT_LINE ('inner.v_counter '||inner.v_counter);
    END LOOP inner;
END LOOP outer;
END;
```

In this example, both the inner and outer loops use the same loop counter, `v_counter`. To reference both the outer and inner values of `v_counter`, loop labels are used. This example produces the following output:

```
outer.v_counter 1
inner.v_counter 1
outer.v_counter 1
inner.v_counter 2
outer.v_counter 2
inner.v_counter 1
```

```
outer.v_counter 2
inner.v_counter 2
outer.v_counter 3
inner.v_counter 1
outer.v_counter 3
inner.v_counter 2
```

Note that the script is able to differentiate between two variables having the same name because loop labels are used when the variables are referenced. If no loop labels are used when **v_counter** is referenced, the output produced by this script will change significantly. Basically, once control passes to the inner loop, the value of **v_counter** from the outer loop is unavailable. When control passes back to the outer loop, the value of **v_counter** becomes available again, as shown in the following example (affected statements are shown in bold):

For Example *ch07_4b.sql*

[Click here to view code image](#)

```
BEGIN
  <<outer>>
  FOR v_counter IN 1..3
  LOOP
    DBMS_OUTPUT.PUT_LINE ('outer.v_counter '|| v_counter);
    <<inner>>
    FOR v_counter IN 1..2
    LOOP
      DBMS_OUTPUT.PUT_LINE ('  outer.v_counter '||v_counter);
      DBMS_OUTPUT.PUT_LINE ('    inner.v_counter '||v_counter);
    END LOOP inner;
  END LOOP outer;
END;
```

To highlight the loop behavior, a new **DBMS_OUTPUT.PUT_LINE** statement has been added to the outer loop, and the loop labels have been removed when referencing the variable **v_counter**. When executed, this version of the script produces the following output:

```
outer.v_counter 1
  outer.v_counter 1
  inner.v_counter 1
  outer.v_counter 2
  inner.v_counter 2
outer.v_counter 2
  outer.v_counter 1
  inner.v_counter 1
  outer.v_counter 2
  inner.v_counter 2
outer.v_counter 3
  outer.v_counter 1
  inner.v_counter 1
  outer.v_counter 2
  inner.v_counter 2
```

As you can see, inside the inner loop, the value of the **v_counter** from the outer loop is not available when it is referenced without the loop label. In this example, the same name for two different loop counters is used to demonstrate another use of loop labels.

However, *it is not considered a good programming practice to use the same name for different variables.*

Summary

In [Chapter 6](#), you began exploring the various types of loops supported in PL/SQL. In this chapter, you continued this exploration by learning about additional loop features introduced in Oracle 11g. You also discovered about how various loop types may be nested inside one another. Finally, you learned how loop labels may be used to improve code readability and maintainability when working with nested loops.

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.

8. Error Handling and Built-in Exceptions

In this chapter, you will learn about

- [Handling Errors](#)
- [Built-in Exceptions](#)

In [Chapter 1](#), you encountered two types of errors that can be found in a program: compilation errors and runtime errors. You also learned that a special section in a PL/SQL block handles runtime errors. In this so-called exception-handling section, runtime errors are referred to as exceptions. The exception-handling section allows programmers to specify which actions should be taken when a specific exception occurs.

In PL/SQL, there are two types of exceptions: built-in exceptions and user-defined exceptions. In this chapter, you will learn how to handle certain kinds of runtime errors with the help of built-in exceptions. User-defined exceptions are discussed in [Chapters 9](#) and [10](#).

Lab 8.1: Handling Errors

After this lab, you will be able to

- Understand the Importance of Error Handling

The following example illustrates some of the differences between compilation and runtime errors:

For Example *ch08_1a.sql*

[Click here to view code image](#)

```
DECLARE
  v_num1    INTEGER := &sv_num1;
  v_num2    INTEGER := &sv_num2;
  v_result NUMBER;
BEGIN
  v_result = v_num1 / v_num2;
  DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
END;
```

This example is a very simple program in which there are two variables, `v_num1` and `v_num2`. A user supplies values for these variables. Next, `v_num1` is divided by `v_num2`, and the result of this division is stored in the third variable, `v_result`. Finally, the value of the variable `v_result` is displayed on the screen.

Now, assume that a user supplies values of 3 and 5 for the variables `v_num1` and `v_num2`, respectively. As a result, the example produces the following output:

[Click here to view code image](#)

```
ORA-06550: line 6, column 13:
PLS-00103: Encountered the symbol "=" when expecting one of the following:
```

```
:= . ( @ % ;
The symbol “:= was inserted before “=” to continue.
```

You probably noticed that the script did not execute successfully. A syntax error was encountered at line 6. Close inspection of the example shows that the statement

```
v_result = v_num1 / v_num2;
```

contains an equal sign operator where an assignment operator should be used. The statement should be rewritten as follows:

```
v_result := v_num1 / v_num2;
```

Once the corrected example is run again, the following output is produced:

```
v_result: .6
```

The example now executes successfully because the syntax error has been corrected.

Next, if you change the values of the variables `v_num1` and `v_num2` to 4 and 0, respectively, the following output is produced:

[Click here to view code image](#)

```
ORA-01476: divisor is equal to zero
ORA-06512: at line 6
01476. 00000 - "divisor is equal to zero"
```

Even though this example does not contain syntax errors, the script terminated prematurely because the value entered for `v_num2`, the divisor, was 0. Division by 0 is undefined, so this operation leads to an error.

This example illustrates a runtime error that cannot be detected by the compiler. In other words, for some of the values entered for the variables `v_num1` and `v_num2`, this example executes successfully. When other values are entered for `v_num1` and `v_num2`, this example cannot execute. As a result, a runtime error occurs. Recall that the compiler cannot detect runtime errors. In this case, a runtime error occurs because the compiler does not know the result of the division of `v_num1` by `v_num2`. This result can be determined only at run time—hence, this error is referred to as a runtime error.

To handle this type of error in the program, an exception handler must be added. The exception-handling section has the structure shown in [Listing 8.1](#).

Listing 8.1 Exception-Handling Section

[Click here to view code image](#)

```
EXCEPTION
  WHEN EXCEPTION_NAME
  THEN
    ERROR PROCESSING STATEMENTS;
```

Note that the exception-handling section appears after the executable section of the block. Therefore, the preceding example can be rewritten in the following manner (newly added statements are shown in bold):

For Example *ch08_1b.sql*

[Click here to view code image](#)

```

DECLARE
    v_num1    INTEGER := &sv_num1;
    v_num2    INTEGER := &sv_num2;
    v_result NUMBER;
BEGIN
    v_result := v_num1 / v_num2;
    DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
EXCEPTION
    WHEN ZERO_DIVIDE
    THEN
        DBMS_OUTPUT.PUT_LINE ('A number cannot be divided by zero.');
END;

```

The section of the example in bold shows the exception-handling section of the block. When this version of the example is executed with values of 4 and 0 for variables `v_num1` and `v_num2`, respectively, the following output is produced:

[Click here to view code image](#)

A number cannot be divided by zero.

This output shows that once an attempt to divide `v_num1` by `v_num2` was made, the exception-handling section of the block was executed. Therefore, the error message specified by the exception-handling section was displayed on the screen.

This version of the output illustrates several of the advantages that arise from use of an exception-handling section. You probably noticed that the output looks cleaner compared to the previous version. Even though the error message is still displayed on the screen, the output is more informative. In short, it is oriented more toward a user than a programmer.

Watch Out!

On many occasions, a user does not have access to the code. Therefore, references to line numbers and keywords in a program are not significant to most users.

An exception-handling section allows a program to execute to completion, instead of terminating prematurely. It also provides for isolation of error-handling routines. In other words, all error-processing code for a specific block can be placed within a single section. As a result, the logic of the program becomes easier to follow and understand. Finally, adding an exception-handling section enables event-driven processing of errors. As in the example shown earlier, when a specific exception event occurs, such as division by 0, the exception-handling section executes, and the error message specified by the `DBMS_OUTPUT.PUT_LINE` statement is displayed on the screen.

Lab 8.2: Built-in Exceptions

After this lab, you will be able to

- Use Built-in Exceptions

As mentioned earlier, a PL/SQL block has the structure shown in [Listing 8.2](#).

Listing 8.2 PL/SQL Block Structure

[Click here to view code image](#)

```
DECLARE
  ...
BEGIN
  EXECUTABLE STATEMENTS;
EXCEPTION
  WHEN EXCEPTION_NAME
  THEN
    ERROR PROCESSING STATEMENTS;
END;
```

When an error occurs that raises a built-in exception, the exception is said to be raised implicitly. In other words, if a program breaks an Oracle rule, control passes to the exception-handling section of the block. At this point, the error-processing statements are executed. After the exception-handling section of the block has executed, the block terminates; that is, control does not return to the executable section of the block. The following example illustrates this point:

For Example ch08_2a.sql

[Click here to view code image](#)

```
DECLARE
  v_student_name VARCHAR2(50);
BEGIN
  SELECT first_name || ' ' || last_name
    INTO v_student_name
   FROM student
  WHERE student_id = 101;

  DBMS_OUTPUT.PUT_LINE ('Student name is ' || v_student_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

This example produces the following output:

```
There is no such student
```

Because there is no record in the STUDENT table with student ID 101, the SELECT INTO statement does not return any rows. As a result, control passes to the exception-handling section of the block, and the error message “There is no such student” is displayed on the screen. Even though a DBMS_OUTPUT.PUT_LINE statement appears right after the SELECT INTO statement, it will not be executed because control has been transferred to the exception-handling section. Control will never return to the executable section of this block, which contains the first DBMS_OUTPUT.PUT_LINE statement.

While every Oracle runtime error has a number associated with it, it must be handled by its name in the exception-handling section. One of the outputs from the example used in the previous lab of this chapter included the following error message:

[Click here to view code image](#)

```
ORA-01476: divisor is equal to zero
```

where ORA-01476 stands for the error number. This error number refers to the error named ZERO_DIVIDE. Some common Oracle runtime errors are predefined in PL/SQL as exceptions. The following list identifies some of these predefined exceptions and explains how they are raised:

- **NO_DATA_FOUND**: This exception is raised when a `SELECT INTO` statement that makes no calls to group functions, such as `SUM` or `COUNT`, does not return any rows. For example, suppose you issue a `SELECT INTO` statement against the `STUDENT` table where the student ID equals 101. If no record in the `STUDENT` table meets this criterion (student ID equals 101), the `NO_DATA_FOUND` exception is raised.

When a `SELECT INTO` statement calls a group function, such as `COUNT`, the result set is never empty. When used in a `SELECT INTO` statement against the `STUDENT` table, function `COUNT` will return 0 for the value of student ID 123. Hence, a `SELECT INTO` statement that calls a group function will never raise the `NO_DATA_FOUND` exception.

- **TOO_MANY_ROWS**: This exception is raised when a `SELECT INTO` statement returns more than one row. By definition, a `SELECT INTO` can return only a single row. If a `SELECT INTO` statement returns more than one row, the definition of the `SELECT INTO` statement is violated. This causes the `TOO_MANY_ROWS` exception to be raised.

For example, you issue a `SELECT INTO` statement against the `STUDENT` table for a specific ZIP code. It is highly likely that this `SELECT INTO` statement will return more than one row, because many students may live in the same ZIP code area.

- **ZERO_DIVIDE**: This exception is raised when a division operation is performed in the program and a divisor is equal to zero. An example in the previous lab of this chapter illustrates how this exception is raised.
- **LOGIN_DENIED**: This exception is raised when a user is trying to log in to Oracle with an invalid username or password.
- **PROGRAM_ERROR**: This exception is raised when a PL/SQL program has an internal problem.
- **VALUE_ERROR**: This exception is raised when a conversion or size mismatch error occurs. For example, suppose you select a student's last name into a variable that has been defined as `VARCHAR2(5)`. If the student's last name contains more than five characters, the `VALUE_ERROR` exception is raised.
- **DUP_VALUE_ON_INDEX**: This exception is raised when a program tries to store a duplicate value in a column or columns that have a unique index defined on them. For example, suppose you are trying to insert a record into the `SECTION` table for the course number 25, section 1. If a record for the given course and section number already exists in the `SECTION` table, the `DUP_VAL_ON_INDEX` exception is raised because these columns have a unique index defined on them.

So far, you have seen examples of programs that are able to handle a single exception

only. For example, a PL/SQL block contains an exception handler with a single exception `ZERO_DIVIDE`. However, many times you need to handle different exceptions in the PL/SQL block. Moreover, often you need to specify different actions that must be taken when a particular exception is raised, as the following example illustrates:

For Example *ch08_3a.sql*

[Click here to view code image](#)

```
DECLARE
    v_student_id NUMBER      := &sv_student_id;
    v_enrolled    VARCHAR2(3) := 'NO';
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Check if the student is enrolled');
    SELECT 'YES'
        INTO v_enrolled
        FROM enrollment
        WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('The student is enrolled into one course');
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('The student is not enrolled');

    WHEN TOO_MANY_ROWS
    THEN
        DBMS_OUTPUT.PUT_LINE ('The student is enrolled in multiple courses');
END;
```

This example contains two exceptions in a single exception-handling section. The first exception, `NO_DATA_FOUND`, will be raised if there are no records in the `ENROLLMENT` table for a particular student. The second exception, `TOO_MANY_ROWS`, will be raised if a particular student is enrolled in more than one course.

Consider what happens if you run this example for three different values of student ID: 102, 103, and 319. In the first run, when the student ID is 102, the example produces the following output:

[Click here to view code image](#)

```
Check if the student is enrolled
The student is enrolled in multiple courses
```

In this case, the first `DBMS_OUTPUT.PUT_LINE` statement is executed, and the message “Check if the ...” is displayed on the screen. Then the `SELECT INTO` statement is executed. You have probably noticed that the `DBMS_OUTPUT.PUT_LINE` statement following the `SELECT INTO` statement was not executed. When the `SELECT INTO` statement is executed for student ID 102, multiple rows are returned. Because the `SELECT INTO` statement can return only a single row, control is passed to the exception-handling section of the block. Next, the PL/SQL block raises the proper exception. As a result, the message “The student is enrolled in multiple courses” is displayed on the screen; this message is specified by the exception `TOO_MANY_ROWS`.

Did You Know?

That built-in exceptions are raised implicitly. Therefore, you need to specify only which action must be taken in the case of a particular exception.

In the second run, when the student ID is 103, the example produces different output:

[Click here to view code image](#)

```
Check if the student is enrolled
The student is enrolled into one course
```

For this run, the first DBMS_OUTPUT.PUT_LINE statement is executed, and the message “Check if the ...” is displayed on the screen. Then the SELECT INTO statement is executed. When the SELECT INTO statement is executed for student ID 103, a single row is returned. Next, the DBMS_OUTPUT.PUT_LINE statement following the SELECT INTO statement is executed. As a result, the message “The student is enrolled into one course” is displayed on the screen. Notice that for this value of the variable v_student_id, no exception has been raised.

In the third run, when the student ID is 319, the example produces the following output:

[Click here to view code image](#)

```
Check if the student is enrolled
The student is not enrolled
```

Just as in the previous runs, the first DBMS_OUTPUT.PUT_LINE statement is executed, and the message “Check if the ...” is displayed on the screen. Then the SELECT INTO statement is executed. When the SELECT INTO statement is executed for student ID 319, no rows are returned. As a result, control passes to the exception-handling section of the PL/SQL block, and the proper exception is raised. In this case, the NO_DATA_FOUND exception is raised because the SELECT INTO statement failed to return a single row. Thus, the message “The student is not enrolled” is displayed on the screen.

So far, you have seen examples of exception-handling sections that have particular exceptions, such as NO_DATA_FOUND and ZERO_DIVIDE. However, you cannot always predict beforehand which exception might be raised by a PL/SQL block. In cases like this, a special exception handler called OTHERS is used. All predefined Oracle errors (exceptions) can be handled with the use of the OTHERS handler.

Consider the following example:

For Example *ch08_4a.sql*

[Click here to view code image](#)

```
DECLARE
    v_instructor_id    NUMBER := &sv_instructor_id;
    v_instructor_name  VARCHAR2(50);
BEGIN
    SELECT first_name||' '||last_name
        INTO v_instructor_name
        FROM instructor
       WHERE instructor_id = v_instructor_id;
```

```
DBMS_OUTPUT.PUT_LINE ('Instructor name is '||v_instructor_name);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

When a value of 100 is provided at run time for the variable `v_instructor_id`, this example produces the following output:

```
An error has occurred
```

This example demonstrates not only the use of the `OTHERS` exception handler, but also a bad programming practice. The exception `OTHERS` has been raised because there is no record in the `INSTRUCTOR` table for instructor ID 100.

This is a simple example, where it is possible to guess which exception handlers should be used. In many instances, however, you may find a number of programs that have been written with a single exception handler, `OTHERS`. This is a bad programming practice, because such use of this exception handler does not give you or your user detailed feedback. You do not really know which error has occurred, and your user does not know whether he or she entered some information incorrectly. Other special error-reporting functions, `SQLCODE` and `SQLERRM`, are very useful when used with the `OTHERS` handler that provide more details. You will learn about them in [Chapter 10](#).

Summary

In this chapter, you began exploring the concepts of error handling and built-in exceptions supported in PL/SQL. In the next two chapters, you will continue learning about exceptions, their scope and propagation, and ways to define your own exceptions. Finally, in [Chapter 24](#), you will discover how to produce meaningful error reporting within your code with the help of Oracle's built-in packages `DBMS.Utility` and `UTL_CallStack`. You will also see why the `UTL_CallStack` package introduced in Oracle 12c is a better alternative when it comes to error reporting.

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.

9. Exceptions

In this chapter, you will learn about

- [Exception Scope](#)
- [User-Defined Exceptions](#)
- [Exception Propagation](#)

In [Chapter 8](#), you explored the concept of error handling and built-in exceptions. In this chapter you will continue that exploration by examining whether an exception can catch a runtime error occurring in the declaration, executable, or exception-handling section of a PL/SQL block. You will also learn how to define your own exceptions and how to re-raise an exception.

Lab 9.1: Exception Scope

After this lab, you will be able to

- Understand the Scope of an Exception

You are already familiar with the term scope—for example, the scope of a variable. Even though variables and exceptions serve different purposes, the same scope rules apply to them. These rules are best illustrated by means of an example.

For Example *ch09_1a.sql*

[Click here to view code image](#)

```
DECLARE
    v_student_id NUMBER := &sv_student_id;
    v_name        VARCHAR2(30);
BEGIN
    SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
        INTO v_name
        FROM student
       WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('Student name is ' || v_name);
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

In this example, student's name is displayed on the screen for a given value of student ID provided at run time. If there is no record in the STUDENT table corresponding to the value of `v_student_id`, the exception `NO_DATA_FOUND` is raised. Therefore, you can say that the exception `NO_DATA_FOUND` covers this block, or that this block is the scope of this exception. In other words, *the scope of an exception is the portion of the block that is covered by this exception*.

Now, you can expand on that understanding (newly added statements are shown in bold):

For Example *ch09_1b.sql*

[Click here to view code image](#)

```
<<outer_block>>
DECLARE
    v_student_id NUMBER := &sv_student_id;
    v_name        VARCHAR2(30);
    v_total       NUMBER(1);

BEGIN
    SELECT RTRIM(first_name)||' '||RTRIM(last_name)
        INTO v_name
        FROM student
       WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('Student name is '||v_name);

<<inner_block>>
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM enrollment
       WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('Student is registered for '||v_total||'
course(s)');
EXCEPTION
    WHEN VALUE_ERROR OR INVALID_NUMBER
    THEN
        DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

The new version of the example includes an inner block. This block has a structure similar to the outer block; that is, it has a `SELECT INTO` statement and an exception section to handle errors. When a `VALUE_ERROR` or `INVALID_NUMBER` error occurs in the inner block, the exception is raised.

Notice that the exceptions `VALUE_ERROR` and `INVALID_NUMBER` have been defined for the inner block only. Therefore, they can be handled only if they are raised in the inner block. If one of these errors occurs in the outer block, the program will be unable to terminate successfully.

In contrast, the exception `NO_DATA_FOUND` has been defined in the outer block; therefore, it is global to the inner block. However, this version of the example never raises the exception `NO_DATA_FOUND` in the inner block. Why do you think this is the case?

Did You Know?

If you define an exception in a block, it is local to that block. However, it is global to any blocks enclosed by that block. In other words, in the case of nested blocks, any exception defined in the outer block becomes global to its inner blocks.

Note what happens when the example is changed so that the exception **NO_DATA_FOUND** can be raised by the inner block (all changes are shown in bold).

For Example *ch09_1c.sql*

[Click here to view code image](#)

```
<<outer_block>>
DECLARE
    v_student_id NUMBER := &sv_student_id;
    v_name        VARCHAR2(30);
    v_registered CHAR;

BEGIN
    SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
        INTO v_name
        FROM student
       WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('Student name is ' || v_name);

<<inner_block>>
BEGIN
    SELECT 'Y'
        INTO v_registered
        FROM enrollment
       WHERE student_id = v_student_id;
    DBMS_OUTPUT.PUT_LINE ('Student is registered');
EXCEPTION
    WHEN VALUE_ERROR OR INVALID_NUMBER
    THEN
        DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;

EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

The new version of the example has a different **SELECT INTO** statement. To answer the question posed earlier, the exception **NO_DATA_FOUND** can be raised by the inner block because the **SELECT INTO** statement does not contain a group function, **COUNT()**. This function always returns a result, so when no rows are returned by the **SELECT INTO** statement, the value returned by the **COUNT(*)** equals zero.

Now, consider the output produced by this example when a value of 284 is provided for the student ID:

[Click here to view code image](#)

```
Student name is Salewa Lindeman
There is no such student
```

You have probably noticed that this example produces only a partial output. Even though you are able to see the student's name, the error message is displayed, indicating that this student does not exist. This error message is displayed because the exception `NO_DATA_FOUND` is raised in the inner block.

The `SELECT INTO` statement of the outer block returns the student's name, which is then displayed on the screen by the first `DBMS_OUTPUT.PUT_LINE` statement. Next, control passes to the inner block. The `SELECT INTO` statement of the inner block does not return any rows. As a result, an error occurs and the `NO_DATA_FOUND` exception is raised.

Next, PL/SQL tries to find a handler for the `NO_DATA_FOUND` exception in the inner block. Because there is no such handler in the inner block, control is transferred to the exception section of the outer block. The exception section of the outer block contains the handler for the exception `NO_DATA_FOUND`. Consequently, this handler executes, and the message "There is no such student" is displayed on the screen. The process, which is called exception propagation, is discussed in detail in [Lab 9.3](#).

Be aware that this example has been provided for illustrative purposes only. In its current version, it is not very useful. The `SELECT INTO` statement of the inner block is prone to another exception, `TOO_MANY_ROWS`, which is not handled by this example. In addition, the error message "There is no such student" is not very descriptive when the exception `NO_DATA_FOUND` is raised by the inner block.

Lab 9.2: User-Defined Exceptions

After this lab, you will be able to

- Use User-Defined Exceptions

Often in your programs, you may need to handle problems that are specific to the program you write. For example, suppose your program asks a user to enter a value for the student ID. This value is then assigned to the variable `v_student_id`, which is used later in the program. Generally, you want a positive number for an ID. By mistake, however, the user enters a negative number. However, no error has occurred because the variable `v_student_id` has been defined as a number, and the user has supplied a legitimate numeric value. Therefore, you may want to implement your own exception to handle this situation.

This type of an exception is called a user-defined exception because it is defined by the programmer. As a result, before such an exception can be used, it must be declared. A user-defined exception is declared in the declarative part of a PL/SQL block, as shown in [Listing 9.1](#).

Listing 9.1 User-Defined Exception Declaration

```
DECLARE
    exception_name EXCEPTION;
```

Notice that this declaration looks similar to a variable declaration. That is, you specify an exception name followed by the keyword EXCEPTION.

Consider the following code fragment:

For Example

```
DECLARE  
  e_invalid_id EXCEPTION;
```

In this code fragment, the name of the exception is prefixed by the letter “e.” This is not a required syntax; rather, it allows you to differentiate between variable names and exception names.

Once an exception has been declared, the executable statements associated with that exception are specified in the exception-handling section of the block. The format of the exception-handling section is the same as for built-in exceptions. Consider the following code fragment:

For Example

[Click here to view code image](#)

```
DECLARE  
  e_invalid_id EXCEPTION;  
BEGIN  
  ...  
EXCEPTION  
  WHEN e_invalid_id  
  THEN  
    DBMS_OUTPUT.PUT_LINE ('An ID cannot be negative');  
END;
```

You already know that built-in exceptions are raised implicitly. In other words, when a certain error occurs, a built-in exception associated with this error is raised. Of course, you are assuming that you have included this exception in the exception-handling section of your program. For example, a TOO_MANY_ROWS exception is raised when a SELECT INTO statement returns multiple rows.

A user-defined exception must be raised explicitly. In other words, you need to specify in your program under which circumstances an exception must be raised, as shown in [Listing 9.2](#).

[Listing 9.2 Raising a User-Defined Exception](#)

[Click here to view code image](#)

```
DECLARE  
  exception_name EXCEPTION;  
BEGIN  
  ...  
  IF CONDITION  
  THEN  
    RAISE exception_name;  
  END IF;  
  ...  
EXCEPTION  
  WHEN exception_name  
  THEN
```

```
ERROR PROCESSING STATEMENTS;
END;
```

In this structure, the circumstances under which a user-defined exception must be raised are determined with the help of the **IF** statement. If *CONDITION* evaluates to **TRUE**, a user-defined exception is raised with the help of the **RAISE** statement. If *CONDITION* evaluates to **FALSE**, the program proceeds with its normal execution. In other words, the statements following the **IF** statement are executed. Note that any form of the **IF** statement can be used to check when a user-defined exception must be raised.

In the next example, which is based on the code fragments provided earlier in this lab, you will see that the exception **e_invalid_id** is raised when the user enters a negative number for the variable **v_student_id**.

For Example *ch09_2a.sql*

[Click here to view code image](#)

```
DECLARE
    v_student_id      STUDENT.STUDENT_ID%TYPE := &sv_student_id;
    v_total_courses   NUMBER;
    e_invalid_id      EXCEPTION;
BEGIN
    IF v_student_id < 0
    THEN
        RAISE e_invalid_id;
    END IF;

    SELECT COUNT(*)
        INTO v_total_courses
        FROM enrollment
        WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('The student is registered for
    '||v_total_courses||' courses');
    DBMS_OUTPUT.PUT_LINE ('No exception has been raised');

EXCEPTION
    WHEN e_invalid_id
    THEN
        DBMS_OUTPUT.PUT_LINE ('An ID cannot be negative');
END;
```

In this example, the exception **e_invalid_id** is raised with the help of the **IF** statement. Once a value is supplied for the variable **v_student_id**, the sign of this numeric value is checked. If the value is less than zero, the **IF** statement evaluates to **TRUE** and the exception **e_invalid_id** is raised. In turn, control passes to the exception-handling section of the block. Next, statements associated with this exception are executed. In this case, the message “An ID cannot be negative” is displayed on the screen. If the value entered for the **v_student_id** is positive, the **IF** statement yields **FALSE** and the rest of the statements in the body of the block are executed.

Consider executing this example for two values of **v_student_id**, 102 and -102. The first run of the example (the student ID is 102) produces the following output:

[Click here to view code image](#)

```
The student is registered for 2 courses
```

No exception has been raised

For this run, the user provides a positive value for the variable `v_student_id`. As a result, the `IF` statement evaluates to `FALSE`, and the `SELECT INTO` statement determines how many records are in the `ENROLLMENT` table for the given student ID. Next, the messages “The student is registered for 2 courses” and “No exception has been raised” are displayed on the screen. At this point, the body of the PL/SQL block has executed to completion.

A second run of the example (the student ID is `-102`) produces the following output:

An ID cannot be negative

For this run, the user entered a negative value for the variable `v_student_id`. The `IF` statement evaluates to `TRUE` and the exception `e_invalid_id` is raised. As a result, control of the execution passes to the exception-handling section of the block, and the message “An ID cannot be negative” is displayed on the screen.

Watch Out!

The `RAISE` statement should be used in conjunction with an `IF` statement. Otherwise, control of the execution will be transferred to the exception-handling section of the block for every single execution. Consider the following example:

[Click here to view code image](#)

```
DECLARE
  e_test_exception EXCEPTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Exception has not been raised');
  RAISE e_test_exception;
  DBMS_OUTPUT.PUT_LINE ('Exception has been raised');
EXCEPTION
  WHEN e_test_exception
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

Every time this example is run, the following output is produced:

[Click here to view code image](#)

```
Exception has not been raised
An error has occurred
```

Even though no error has occurred, control is transferred to the exception-handling section. It is important for you to check whether the error has occurred before raising the exception associated with that error.

The same scope rules apply to user-defined exceptions as apply to built-in exceptions. An exception declared in the inner block must be raised in the inner block and defined in the exception-handling section of the inner block. Consider the following example:

For Example `ch09_3a.sql`

[Click here to view code image](#)

```

<<outer_block>>
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Outer block');
    <<inner_block>>
    DECLARE
        e_my_exception EXCEPTION;
    BEGIN
        DBMS_OUTPUT.PUT_LINE ('Inner block');
    EXCEPTION
        WHEN e_my_exception
        THEN
            DBMS_OUTPUT.PUT_LINE ('An error has occurred');
    END;

    IF 10 > &sv_number
    THEN
        RAISE e_my_exception;
    END IF;
END;

```

In this example, the exception, `e_my_exception`, has been declared in the inner block. However, you are trying to raise this exception in the outer block. This example causes a syntax error because the exception declared in the inner block ceases to exist once the inner block terminates. As a result, this example produces the following output when a value of 11 is provided at run time:

[Click here to view code image](#)

```

ORA-06550: line 19, column 13:
PLS-00201: identifier 'E_MY_EXCEPTION' must be declared
ORA-06550: line 19, column 7:
PL/SQL: Statement ignored

```

Notice that the error message

[Click here to view code image](#)

```
PLS-00201: identifier 'E_MY_EXCEPTION' must be declared
```

is the same error message you get when you try to use a variable that has not been declared.

Lab 9.3: Exception Propagation

After this lab, you will be able to

- Understand How Exceptions Propagate
- [Re-raise Exceptions](#)

You already have seen how different types of exceptions are raised when a runtime error occurs in the executable portion of the PL/SQL block. However, a runtime error may also occur in the declaration section of the block or in the exception-handling section of the block. The rules that govern how exceptions are raised in these situations are referred to as exception propagation.

Consider the first case, in which a runtime error occurs in the executable section of the

PL/SQL block. This case should be treated as a review because the examples given earlier in this chapter show how an exception is raised when an error occurs in the executable section of the block.

If a specific exception is associated with a particular error, control passes to the exception-handling section of the block. Once the statements associated with the exception are executed, control passes to the host environment or to the enclosing block. If there is no exception handler for this error, the exception is propagated to the enclosing block (outer block). The steps just described are then repeated. If no exception handler is found, the execution of the program halts, and control is transferred to the host environment.

Next, consider the second case, in which a runtime error occurs in the declaration section of the block. If there is no outer block, the execution of the program halts, and control passes to the host environment. Consider the following example:

For Example ch09_4a.sql

[Click here to view code image](#)

```
DECLARE
    v_test_var CHAR(3):= 'ABCDE';
BEGIN
    DBMS_OUTPUT.PUT_LINE ('This is a test');
EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
        DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

When executed, this example produces the following output:

[Click here to view code image](#)

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too
small
ORA-06512: at line 2
```

In this example, the assignment statement in the declaration section of the block causes an error. Even though an exception handler for this error exists, the block is not able to execute successfully. Based on this example, you can conclude that *when a runtime error occurs in the declaration section of the PL/SQL block, the exception-handling section of this block is not able to catch the error.*

Next, consider a modified version of the same example that employs nested PL/SQL blocks (changes are shown in bold).

For Example ch09_4b.sql

[Click here to view code image](#)

```
<<outer_block>>
BEGIN
    <<inner_block>>
    DECLARE
        v_test_var CHAR(3):= 'ABCDE';
    BEGIN
        DBMS_OUTPUT.PUT_LINE ('This is a test');
    EXCEPTION
```

```

    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
        DBMS_OUTPUT.PUT_LINE ('An error has occurred in the inner block');
    END;
EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
        DBMS_OUTPUT.PUT_LINE ('An error has occurred in the program');
END;

```

When executed, this example produces the following output:

[Click here to view code image](#)

An error has occurred in the program

In this version of the example, the PL/SQL block is enclosed by another block, and the program is able to complete. In this case, the exception defined in the outer block is raised when the error occurs in the declaration section of the inner block. Therefore, you can conclude that *when a runtime error occurs in the declaration section of the inner block, the exception immediately propagates to the enclosing (outer) block.*

Finally, consider a third case, in which a runtime error occurs in the exception-handling section of the block. Just as in the previous case, if there is no outer block, the execution of the program halts and control passes to the host environment. Consider the following example:

For Example *ch09_5a.sql*

[Click here to view code image](#)

```

DECLARE
    v_test_var CHAR(3) := 'ABC';
BEGIN
    v_test_var := '1234';
    DBMS_OUTPUT.PUT_LINE ('v_test_var: '||v_test_var);
EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
        v_test_var := 'ABCD';
        DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;

```

When executed, this example produces the following output:

[Click here to view code image](#)

```

ORA-06502: PL/SQL: numeric or value error: character string buffer too
small
ORA-06512: at line 9
ORA-06502: PL/SQL: numeric or value error: character string buffer too
small

```

As you can see, the assignment statement in the executable section of the block causes an error. In turn, control is transferred to the exception-handling section of the block. However, the assignment statement in the exception-handling section of the block raises the same error. As a result, the output of this example displays the same error message twice. The first message is generated by the assignment statement in the executable section of the block, and the second message is generated by the assignment statement of

the exception-handling section of this block. Based on this example, you can conclude that *when a runtime error occurs in the exception-handling section of the PL/SQL block, the exception-handling section of this block is not able to prevent the error.*

Next, consider a modified version of the same example with nested PL/SQL blocks (affected statements are shown in bold):

For Example ch09_5b.sql

[Click here to view code image](#)

```
<<outer_block>>
BEGIN
  <<inner_block>>
  DECLARE
    v_test_var CHAR(3) := 'ABC';
  BEGIN
    v_test_var := '1234';
    DBMS_OUTPUT.PUT_LINE ('v_test_var: '||v_test_var);
  EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
      v_test_var := 'ABCD';
      DBMS_OUTPUT.PUT_LINE ('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the program');
END;
```

When executed, this version produces the following output:

[Click here to view code image](#)

```
An error has occurred in the program
```

In this version of the example, the PL/SQL block is enclosed by another block, and the program is able to complete. In this case, the exception defined in the outer block is raised when the error occurs in the exception-handling section of the inner block. Therefore, you can conclude that *when a runtime error occurs in the exception-handling section of the inner block, the exception immediately propagates to the enclosing block.*

In the previous two examples, an exception was raised implicitly by a runtime error in the exception-handling section of the block. However, an exception can also be raised explicitly in the exception-handling section of the block by the RAISE statement. Consider the following example:

For Example ch09_6a.sql

[Click here to view code image](#)

```
<<outer_block>>
DECLARE
  e_exception1 EXCEPTION;
  e_exception2 EXCEPTION;
BEGIN
  <<inner_block>>
  BEGIN
    RAISE e_exception1;
```

```

EXCEPTION
  WHEN e_exception1
  THEN
    RAISE e_exception2;
  WHEN e_exception2
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the inner block');
END;
EXCEPTION
  WHEN e_exception2
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the program');
END;

```

This example produces the following output:

[Click here to view code image](#)

An error has occurred in the program

The declaration portion of the block contains declarations of two exceptions, `e_exception1` and `e_exception2`. The exception `e_exception1` is raised in the inner block via the `RAISE` statement. In the exception-handling section of the inner block, the exception `e_exception1` tries to raise `e_exception2`. Even though an exception handler for `e_exception2` exists in the inner block, control is still transferred to the outer block. This happens because only one exception can be raised in the exception-handling section of the block. Only after one exception has been handled can another be raised, but two or more exceptions cannot be raised simultaneously. This flow of execution is illustrated in [Figure 9.1](#).

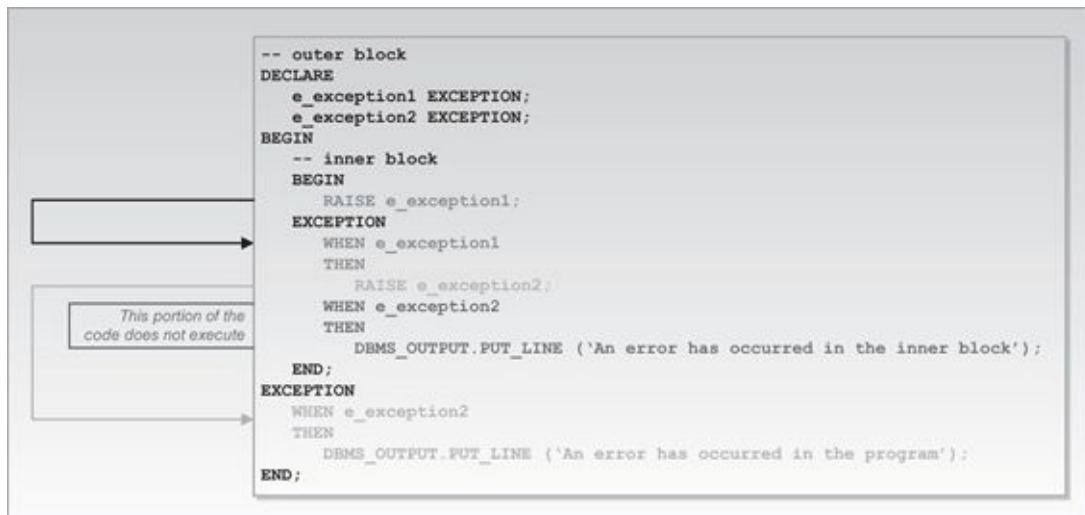


Figure 9.1 Flow of the Execution for Example ch09_6a.sql

Essentially, when the exception `e_exception2` is raised in the exception-handling section of the inner block, it cannot be handled in the same exception-handling section. Thus, the portion of the code surrounded by rectangular brackets never executes. Instead, control passes to the exception-handling section of the outer block and the message “An error has occurred in the program” is displayed on the screen.

Watch Out!

When an exception is raised in a PL/SQL block that does not have an appropriate exception-handling mechanism and is not enclosed by another block, control is transferred to the host environment, and the program is not able to complete successfully. The following code fragment illustrates this case:

[Click here to view code image](#)

```
DECLARE
  e_exception1 EXCEPTION;
BEGIN
  RAISE e_exception1;
END;

ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 4
```

Note that this behavior applies to built-in exceptions and was also seen in [Chapter 8](#).

Re-raising Exceptions

On some occasions you may want to be able to stop your program if a certain type of error occurs. In other words, you may want to handle an exception in the inner block and then pass it to the outer block. This process is called re-raising an exception. The following example helps to illustrate this point:

For Example *ch09_7a.sql*

[Click here to view code image](#)

```
<<outer_block>>
DECLARE
  e_exception EXCEPTION;
BEGIN
  <<inner_block>>
  BEGIN
    RAISE e_exception;
  EXCEPTION
    WHEN e_exception
    THEN
      RAISE;
  END;
EXCEPTION
  WHEN e_exception
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

In this example, the exception `e_exception` is first declared in the outer block, then raised in the inner block. As a result, control is transferred to the exception-handling section of the inner block. The `RAISE` statement in the exception-handling section of the block causes the exception to propagate to the exception-handling section of the outer block. Notice that when the `RAISE` statement is used in the exception-handling section of

the inner block, it is not followed by the exception name.

When run, this example produces the following output:

An error has occurred

Watch Out!

When an exception is re-raised in the block that is not enclosed by any other block, the program is unable to complete successfully. Consider the following example:

```
DECLARE
    e_exception EXCEPTION;
BEGIN
    RAISE e_exception;
EXCEPTION
    WHEN e_exception
    THEN
        RAISE;
END;
```

When run, this example produces the following output:

[Click here to view code image](#)

```
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 7
```

Summary

In this chapter, you learned about exception scope and propagation, and saw how to define and raise your own exceptions. In addition, you learned how to re-raise an exception. In the next chapter, you will discover how to produce meaningful error reporting within your code with the help of Oracle's built-in functions, SQLCODE and SQLERRM.

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.

10. Exceptions: Advanced Concepts

In this chapter, you will learn about

- [RAISE_APPLICATION_ERROR](#)
- [EXCEPTION_INIT Pragma](#)
- [SQLCODE and SQLERRM](#)

In [Chapters 8](#) and [9](#), you encountered the concept of error handling as well as built-in and user-defined exceptions. You also learned about the rules that govern exception scope, propagation, and ways to re-raise an exception.

In this chapter you will conclude your exploration of error handling and exceptions with a study of advanced topics. After working through this chapter, you will be able to associate an error number with an error message, and will be able to trap a runtime error when you have an Oracle error number but no name by which the error can be referenced.

Lab 10.1: RAISE_APPLICATION_ERROR

After this Lab, you will be able to

- Use RAISE_APPLICATION_ERROR

RAISE_APPLICATION_ERROR is a special built-in procedure provided by Oracle. It allows programmers to create meaningful error messages for a specific application. The RAISE_APPLICATION_ERROR procedure works with user-defined exceptions, and its syntax is shown in [Listing 10.1](#).

Listing 10.1 Two forms of the RAISE_APPLICATION_ERROR Procedure

[Click here to view code image](#)

```
RAISE_APPLICATION_ERROR (error_number, error_message);
```

Or

[Click here to view code image](#)

```
RAISE_APPLICATION_ERROR (error_number, error_message, keep_errors);
```

As you can see, there are two forms of the RAISE_APPLICATION_ERROR procedure. The first form contains only two parameters: `error_number` and `error_message`. The `error_number` is the number of the error that a programmer associates with a specific error message; it can range from -20,999 to -20,000. The `error_message` is the text of the error, and it can contain up to 2048 characters.

The second form of RAISE_APPLICATION_ERROR contains one additional parameter: `keep_errors`. It is an optional Boolean parameter. If `keep_errors` is set to TRUE, the new error will be added to the list of errors that have already been raised. This list of errors is called the error stack. If `keep_errors` is set to FALSE, the new

error replaces the error stack. The default value for the parameter `keep_errors` is `FALSE`.

The `RAISE_APPLICATION_ERROR` procedure works with unnamed user-defined exceptions. That is, it associates the number of the error with the text of the error. In turn, the user-defined exception does not have a name associated with it.

Consider the following example used in [Chapter 9](#), `ch09_2a.sql`. This example illustrates the use of the named user-defined exception and the `RAISE` statement. Within the example, compare the modified version using the unnamed user-defined exception and the `RAISE_APPLICATION_ERROR` procedure. The named user-defined exception and the `RAISE` statement are shown in bold.

For Example `ch10_1a.sql` ([Chapter 9](#), example `ch09_2a.sql`)

[Click here to view code image](#)

```
DECLARE
    v_student_id      STUDENT.STUDENT_ID%TYPE := &sv_student_id;
    v_total_courses  NUMBER;
    e_invalid_id     EXCEPTION;
BEGIN
    IF v_student_id < 0
    THEN
        RAISE e_invalid_id;
    END IF;
    SELECT COUNT(*)
        INTO v_total_courses
        FROM enrollment
        WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('The student is registered for
'||v_total_courses||' courses');
    DBMS_OUTPUT.PUT_LINE ('No exception has been raised');
EXCEPTION
    WHEN e_invalid_id
    THEN
        DBMS_OUTPUT.PUT_LINE ('An ID cannot be negative');
END;
```

Now, examine the modified example (affected statements are shown in bold):

For Example `ch10_1b.sql`

[Click here to view code image](#)

```
DECLARE
    v_student_id      STUDENT.STUDENT_ID%TYPE := &sv_student_id;
    v_total_courses  NUMBER;
BEGIN
    IF v_student_id < 0
    THEN
        RAISE_APPLICATION_ERROR (-20000, 'An ID cannot be negative');
    END IF;

    SELECT COUNT(*)
        INTO v_total_courses
        FROM enrollment
        WHERE student_id = v_student_id;
```

```
DBMS_OUTPUT.PUT_LINE ('The student is registered for '||  
v_total_courses||' courses');  
END;
```

The second version of the example does not include the name of the exception, the RAISE statement, or the error-handling section of the PL/SQL block. Instead, it has a single RAISE_APPLICATION_ERROR statement.

Did You Know?

Even though the RAISE_APPLICATION_ERROR is a built-in procedure, it is referenced as a statement when used in the PL/SQL block.

Both versions of the example achieve the same result: The processing stops if a negative number is provided for the variable v_student_id. However, the second version of this example produces output that has the look and feel of an error message.

Now, run both versions of the example with the value of -4 for the variable v_student_id. The first version of the example produces the following output:

```
An ID cannot be negative
```

The second version of the example produces the following output:

[Click here to view code image](#)

```
ORA-20000: An ID cannot be negative  
ORA-06512: at line 7
```

The output produced by the first version of the example contains the error message “An ID cannot be negative.” The same error message generated by the second version of the example resembles an error message generated by the system, because the error number ORA-20000 precedes the error message. Also, note that when you run these examples in SQL Developer, the error message produced by the first version of the example appears in the Dbms Output window, whereas the same error message produced by the second version of the example appears in the Script Output window.

The RAISE_APPLICATION_ERROR procedure can work with built-in exceptions as well. Consider the following example:

For Example ch10_2a.sql

[Click here to view code image](#)

```
DECLARE  
    v_student_id STUDENT.STUDENT_ID%TYPE := &sv_student_id;  
    v_name        VARCHAR2(50);  
BEGIN  
    SELECT first_name||' '||last_name  
      INTO v_name  
     FROM student  
    WHERE student_id = v_student_id;  
    DBMS_OUTPUT.PUT_LINE (v_name);  
EXCEPTION  
    WHEN NO_DATA_FOUND  
    THEN  
        RAISE_APPLICATION_ERROR (-20001, 'This ID is invalid');
```

```
END;
```

When the user enters a value of 100 for the student ID, the example produces the following output:

[Click here to view code image](#)

```
ORA-20001: This ID is invalid  
ORA-06512: at line 13
```

The built-in exception NO_DATA_FOUND is raised because there is no record in the STUDENT table corresponding to this value of the student ID. However, the number of the error message does not refer to the exception NO_DATA_FOUND; rather, the error message “This ID is invalid” is displayed.

The RAISE_APPLICATION_ERROR procedure allows programmers to return error messages in a manner that is consistent with Oracle errors. However, it is up to the programmer to maintain the relationship between the error numbers and the error messages. For example, you have designed an application to maintain the enrollment information on students. In this application you have associated the error text “This ID is invalid” with the error number ORA-20001. This error message can be used by your application for any invalid ID. Once you have associated the error number (ORA-20001) with a specific error message (“This ID is invalid”), you should not assign this error number to another error message. If you do not maintain the relationship between error numbers and error messages, the error-handling interface of your application might become very confusing to the users and to yourself.

Lab 10.2: EXCEPTION_INIT Pragma

After this lab, you will be able to

- Use the EXCEPTION_INIT Pragma

Often your programs need to handle an Oracle error that has a particular number associated with it, but lacks a name by which it can be referenced. In this situation, you are unable to write a handler to trap this error, but you can use a construct called pragma to handle the exception. A pragma is a special instruction to the PL/SQL compiler that is processed at the time of the compilation. The EXCEPTION_INIT pragma allows you to associate an Oracle error number with a name for a user-defined error. Once you associate an error name with an Oracle error number, you can reference the error and write a handler for it.

The EXCEPTION_INIT pragma appears in the declaration section of a block as shown in [Listing 10.2](#).

Listing 10.2 *Associating the EXCEPTION_INIT Pragma with a User-Defined Exception*

[Click here to view code image](#)

```
DECLARE  
  exception_name EXCEPTION;  
  PRAGMA EXCEPTION_INIT (exception_name, error_code);
```

Notice that the declaration of the user-defined exception appears before the `EXCEPTION_INIT` pragma where it is used. The `EXCEPTION_INIT` pragma has two parameters: `exception_name` and `error_code`. The `exception_name` is the name of your exception, and the `error_code` is the number of the Oracle error you want to associate with your exception. Consider the following example:

For Example *ch10_3a.sql*

[Click here to view code image](#)

```
DECLARE
  v_zip ZIPCODE.ZIP%TYPE := '&sv_zip';
BEGIN
  DELETE FROM zipcode
  WHERE zip = v_zip;
  DBMS_OUTPUT.PUT_LINE ('Zip'||v_zip||' has been deleted');
  COMMIT;
END;
```

In this example, the record corresponding to the value of the ZIP code provided by a user is deleted from the `ZIPCODE` table. Next, the message that a specific ZIP code has been deleted is displayed on the screen.

Take a look at the results produced by this example when the user enters 06870 for the value of `v_zip`:

[Click here to view code image](#)

```
ORA-02292: integrity constraint (STUDENT.STU_ZIP_FK)violated - child
record found
ORA-06512: at line 4
```

The error message generated by this example occurs because you are trying to delete a record from the `ZIPCODE` table while its child records exist in the `STUDENT` table, thereby violating the referential integrity constraint `STU_ZIP_FK`. In other words, there is a record with a foreign key (`STU_ZIP_FK`) in the `STUDENT` table (child table) that references a record in the `ZIPCODE` table (parent table).

This error has Oracle error number ORA-02292 assigned to it, but it does not have a name. To handle this error in the script, you need to associate the error number with a user-defined exception.

Suppose you modify the example as follows (all changes are shown in bold):

For Example *ch10_3b.sql*

[Click here to view code image](#)

```
DECLARE
  v_zip          ZIPCODE.ZIP%TYPE := '&sv_zip';
  e_child_exists EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_child_exists, -2292);
BEGIN
  DELETE FROM zipcode
  WHERE zip = v_zip;

  DBMS_OUTPUT.PUT_LINE ('Zip'||v_zip||' has been deleted');
  COMMIT;
EXCEPTION
```

```
WHEN e_child_exists
THEN
  DBMS_OUTPUT.PUT_LINE ('Delete students for this ZIP code first');
END;
```

In this version of the script, you declare the exception `e_child_exists`. You then associate this exception with the error number –2292. Note that you do not use ORA-02292 in the `EXCEPTION_INIT` pragma. Next, you add the exception-handling section to the PL/SQL block, thereby trapping this error.

Did you notice that even though the exception `e_child_exists` is a user-defined exception, you did not use the `RAISE` statement, as you did in [Chapter 9](#)? This is because you have associated the user-defined exception with the specific Oracle error number. Recall that even though an Oracle exception has a number associated with it, it must be referenced by its name in the exception-handling section. Since Oracle error number –2292 does not have a name associated with it, you performed that association explicitly via the `EXCEPTION_INIT` pragma.

When you run this version of the example using the same value of ZIP code, it produces the following output:

[Click here to view code image](#)

```
Delete students for this zipcode first
```

This output contains a new error message displayed by the `DBMS_OUTPUT.PUT_LINE` statement. It is more descriptive than the previous version of the output. Remember that the user of the program probably does not know about the referential integrity constraints existing in the database. Therefore, the `EXCEPTION_INIT` pragma improves the readability of your error-handling interface. If the need arises, you can use multiple `EXCEPTION_INIT` pragmas in your program.

Lab 10.3: SQLCODE and SQLERRM

After this lab, you will be able to

- Use `SQLCODE` and `SQLERRM`

In [Chapter 8](#), you learned about the Oracle exception `OTHERS`. All Oracle errors can be trapped with the help of the `OTHERS` exception handler, as illustrated in the following example:

For Example `ch10_4a.sql`

[Click here to view code image](#)

```
DECLARE
  v_zip    VARCHAR2(5) := '&sv_zip';
  v_city   VARCHAR2(15);
  v_state  CHAR(2);
BEGIN
  SELECT city, state
    INTO v_city, v_state
    FROM zipcode
   WHERE zip = v_zip;
```

```

DBMS_OUTPUT.PUT_LINE (v_city||', '||v_state);

EXCEPTION
  WHEN OTHERS
    THEN
      DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;

```

When the user enters 07458 for the value of the ZIP code, this example produces the following output:

An error has occurred

This output informs you that an error occurred at runtime, but you do not know what the error is and what caused it. Maybe there is no record in the ZIPCODE table corresponding to the value provided at runtime, maybe there is a data type mismatch caused by the `SELECT INTO` statement, or maybe the `SELECT INTO` statement returned more than one row. As you can see, even though this is a simple example, a number of runtime errors might potentially occur.

Of course, you cannot always identify every possible runtime error that might occur when a program is running. Therefore, it is a good practice to include the `OTHERS` exception handler in your script. To improve the error-handling interface of your program, the Oracle platform provides two built-in functions, `SQLCODE` and `SQLERRM`, that can be used with the `OTHERS` exception handler. The `SQLCODE` function returns the Oracle error number, and the `SQLERRM` function returns the error message. The maximum length of a message returned by the `SQLERRM` function is 512 bytes, which is the maximum length of an Oracle database error message.

Consider what happens if you modify the preceding example by adding the `SQLCODE` and `SQLERRM` functions as follows (modifications are highlighted in bold):

For Example *ch10_4b.sql*

[Click here to view code image](#)

```

DECLARE
  v_zip      VARCHAR2(5) := '&sv_zip';
  v_city     VARCHAR2(15);
  v_state    CHAR(2);
  v_err_code NUMBER;
  v_err_msg  VARCHAR2(200);
BEGIN
  SELECT city, state
    INTO v_city, v_state
    FROM zipcode
   WHERE zip = v_zip;

  DBMS_OUTPUT.PUT_LINE (v_city||', '||v_state);

EXCEPTION
  WHEN OTHERS
    THEN
      v_err_code := SQLCODE;
      v_err_msg  := SUBSTR(SQLERRM, 1, 200);
      DBMS_OUTPUT.PUT_LINE ('Error code: '||v_err_code);

```

```
DBMS_OUTPUT.PUT_LINE ('Error message: '||v_err_msg);
END;
```

When executed, this version of the example produces the following output:

[Click here to view code image](#)

```
Error code: -6502
Error message: ORA-06502: PL/SQL: numeric or value error: character string
buffer too small
```

This version of the script includes two new variables: `v_err_code` and `v_err_msg`. In the exception-handling section of the block, the value returned by the `SQLCODE` function is assigned to the variable `v_err_code`, and the value returned by the `SQLERRM` function is assigned to the variable `v_err_msg`. Next, the error number and the error message are displayed on the screen via the `DBMS_OUTPUT.PUT_LINE` statements.

Notice that this output is more informative than the output produced by the previous version of the example because it displays the error message. Once you know which runtime error has occurred in your program, you can take steps to prevent its recurrence.

Generally, the `SQLCODE` function returns a negative number for an error number. However, there are a few exceptions:

- When the `SQLCODE` function is referenced outside the exception section, it returns 0 for the error code. The value of 0 means successful completion.
- When the `SQLCODE` function is used with the user-defined exception, it returns +1 for the error code.
- The `SQLCODE` function returns a value of 100 when the `NO_DATA_FOUND` exception is raised.

The `SQLERRM` function accepts an error number as a parameter, and it returns an error message corresponding to the error number. Usually, it works with the value returned by the `SQLCODE` function. However, you can provide the error number yourself if such a need arises. Consider the following example:

For Example *ch10_5a.sql*

[Click here to view code image](#)

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Error code: '||SQLCODE);
  DBMS_OUTPUT.PUT_LINE ('Error message1: '||SQLERRM(SQLCODE));
  DBMS_OUTPUT.PUT_LINE ('Error message2: '||SQLERRM(100));
  DBMS_OUTPUT.PUT_LINE ('Error message3: '||SQLERRM(200));
  DBMS_OUTPUT.PUT_LINE ('Error message4: '||SQLERRM(-20000));
END;
```

In this example, the `SQLCODE` and `SQLERRM` functions are used in the executable section of the PL/SQL block. The `SQLERRM` function accepts the value provided by `SQLCODE` in the second `DBMS_OUTPUT.PUT_LINE` statement. In the following `DBMS_OUTPUT.PUT_LINE` statements, the `SQLERRM` function accepts the values of 100, 200, and -20,000 respectively. When executed, this example produces the following output:

[Click here to view code image](#)

```
Error code: 0
Error message1: ORA-0000: normal, successful completion
Error message2: ORA-01403: no data found
Error message3: -200: non-ORACLE exception
Error message4: ORA-20000:
```

The first `DBMS_OUTPUT.PUT_LINE` statement displays the value of the `SQLCODE` function. Because no exception has been raised, it returns 0. Next, the value returned by the `SQLCODE` function is accepted as a parameter by the `SQLERRM` function. This function returns the message “ORA-0000: normal, successful completion.” Next, the `SQLERRM` function accepts 100 as its parameter and returns “ORA-01402: no data....” Notice that when the `SQLERRM` function accepts 200 as its parameter, it is not able to find an Oracle exception that corresponds to the error number 200. Finally, when the `SQLERRM` function accepts –20,000 as its parameter, no error message is returned. Recall that –20,000 is an error number that can be associated with a named user-defined exception.

Summary

In this chapter, you have concluded your exploration of error handling and exceptions. You learned how to use the `RAISE_APPLICATION_ERROR` procedure and the `SQLCODE` and `SQLERRM` functions to create meaningful error messages in your code. In addition, you learned about the `EXCEPTION_INIT` pragma, which enables you to associate a user-defined exception with the Oracle error number.

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.

11. Introduction to Cursors

In this chapter, you will learn about

- [Types of Cursors](#)
- [Cursor Loop](#)
- [Cursor FOR LOOPS](#)
- [Nested Cursors](#)

Cursors are memory areas where the Oracle platform executes SQL statements. In database programming, cursors are internal data structures that allow processing of SQL query results. For example, you use a cursor to operate on all the rows of the STUDENT table for those students taking a particular course (having associated entries in the ENROLLMENT table). In this chapter, you will learn to declare an explicit cursor that enables a user to process many rows returned by a query and to write code that will process each row one at a time.

Lab 11.1: Types of Cursors

After this lab, you will be able to

- [Make Use of an Implicit Cursor](#)
- [Make Use of an Explicit Cursor](#)

For the Oracle platform to process an SQL statement, it needs to create an area of memory known as the context area; this will contain the information necessary to process the statement. This information includes the number of rows processed by the statement and a pointer to the parsed representation of the statement (parsing an SQL statement is the process whereby information is transferred to the server, at which point the SQL statement is evaluated as being valid). In a query, the active set refers to the rows that will be returned.

A cursor is a handle, or pointer, to the context area. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. Two important features about the cursor are as follows:

1. Cursors allow you to fetch and process rows returned by a SELECT statement, one row at a time.
2. A cursor is named so that it can be referenced.

There are two types of cursors:

1. An *implicit* cursor is automatically declared by Oracle every time an SQL statement is executed. The user will not be aware of this happening and will not be able to control or process the information in an implicit cursor.

2. An *explicit* cursor is defined by the program for any query that returns more than one row of data. That means the programmer has declared the cursor within the PL/SQL code block. This declaration allows the application to sequentially process each row of data as it is returned by the cursor.

Making Use of an Implicit Cursor

To better understand the capabilities of an explicit cursor, you first need to understand the process followed for an implicit cursor.

Process of an Implicit Cursor

- Any given PL/SQL block issues an implicit cursor whenever an SQL statement is executed, as long as an explicit cursor does not exist for that SQL statement.
- A cursor is automatically associated with every DML (Data Manipulation Language) statement (**UPDATE**, **DELETE**, **INSERT**).
- All **UPDATE** and **DELETE** statements have cursors that identify the set of rows that will be affected by the operation.
- An **INSERT** statement needs a place to receive the data that is to be inserted in the database; the implicit cursor fulfills this need.
- The most recently opened cursor is called the “SQL” cursor.

The implicit cursor is used to process **INSERT**, **UPDATE**, **DELETE**, and **SELECT INTO** statements. During the processing of an implicit cursor, the Oracle platform automatically performs the **OPEN**, **FETCH**, and **CLOSE** operations.

Did You Know?

An implicit cursor can tell you how many rows were affected by an update. Cursors have attributes such as **ROWCOUNT**. **SQL%ROWCOUNT**, for example, returns the numbers of rows updated. It can be used as follows:

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
BEGIN
    UPDATE student
        SET first_name = 'B'
        WHERE first_name LIKE 'B%';
    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
END;
```

Consider the following example of an implicit cursor:

For Example *ch11_1a.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
    v_first_name VARCHAR2(35);
    v_last_name VARCHAR2(35);
```

```

BEGIN
  SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM student
   WHERE student_id = 123;
  DBMS_OUTPUT.PUT_LINE ('Student name: ' ||
    v_first_name||' '||v_last_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
    ('There is no student with student ID 123');
END;

```

The Oracle platform automatically associates an implicit cursor with the `SELECT INTO` statement and fetches the values for the variables, `v_first_name` and `v_last_name`. Once the `SELECT INTO` statement completes, Oracle closes the implicit cursor.

Unlike an implicit cursor, an explicit cursor is defined by the program for any query that returns more than one row of data. Thus you need to process an explicit cursor as follows: First you declare a cursor. Second, you open an earlier declared cursor. Third, you fetch the earlier declared and opened cursor. Finally, you close the cursor.

Making Use of an Explicit Cursor

The only means of generating an explicit cursor is for the cursor to be named in the declaration section of the PL/SQL block.

The advantages of declaring an explicit cursor over using an indirect implicit cursor are that the explicit cursor gives more programmatic control to the programmer. Implicit cursors are less efficient than explicit cursors, which makes it harder to trap data errors.

The process of working with an explicit cursor consists of the following steps:

- 1. Declaring** the cursor. This initializes the cursor into memory.
- 2. Opening** the cursor. The previously declared cursor can now be opened; memory is allocated.
- 3. Fetching** the cursor. The previously declared and opened cursor can now retrieve data; this is the process of fetching the cursor.
- 4. Closing** the cursor. The previously declared, opened, and fetched cursor must now be closed to release memory allocation.

Declaring a Cursor

Declaring a cursor defines the name of the cursor and associates it with a `SELECT` statement. The first step is to declare the cursor with the following syntax:

[Click here to view code image](#)

```
CURSOR c_cursor_name IS select statement
```

Did You Know?

The naming conventions that are used in this book advise you to always name a cursor as `C_CURSORNAME`. When you include `C_` at the beginning of the name, it will always be clear to you that the name is referencing a cursor.

It is not possible to make use of a cursor unless the complete cycle of (1) declaring, (2) opening, (3) fetching, and (4) closing has been performed. To explain these four steps, the following examples show code fragments for each step. After we go over the process step by step, you will then be shown the complete process.

The following example is a PL/SQL fragment that demonstrates the first step of declaring a cursor. A cursor named `C_MyCursor` is declared as a `SELECT` statement of all the rows in the `zipcode` table that have the item state equal to "NY."

For Example *ch11_1b.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR C_MyCursor IS
    SELECT *
      FROM zipcode
     WHERE state = 'NY';
...
--<code would continue here with opening, fetching,
and closing of the cursor>
```

Did You Know?

Cursor names follow the same rules of scope and visibility that apply to the PL/SQL identifiers. Because the name of the cursor is a PL/SQL identifier, it must be declared before it is referenced. Any valid select statement can be used to define a cursor, including joins and statements with the `UNION` or `MINUS` clause.

Record Types

A record is a composite data structure, which means that it is composed of one or more elements. Records are very much like a row of a database table, but each element of the record does not stand on its own. PL/SQL supports three kinds of records: (1) table based, (2) cursor based, and (3) programmer defined.

A table-based record is one whose structure is drawn from the list of columns in the table. A cursor-based record is one whose structure matches the elements of a predefined cursor. To create a table-based or cursor-based record, use the `%ROWTYPE` attribute.

[Click here to view code image](#)

```
<record_name> <table_name or cursor_name>%ROWTYPE
```

For Example *ch11_1c.sql*

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
    vr_student student%ROWTYPE;
BEGIN
    SELECT *
        INTO vr_student
        FROM student
        WHERE student_id = 156;
    DBMS_OUTPUT.PUT_LINE (vr_student.first_name||' '
        ||vr_student.last_name||' has an ID of 156');
EXCEPTION
    WHEN no_data_found
    THEN
        RAISE_APPLICATION_ERROR(-2001,'The Student ' ||
            'is not in the database');
END;

```

The variable `vr_student` is a record type of the existing database table `student`. That is, it has the same components as a row in the `student` table. A cursor-based record is much the same, except that it is drawn from the select list of an explicitly declared cursor. When referencing elements of the record, you use the same syntax that you use with tables:

`record_name.item_name`

To define a variable that is based on a cursor record, you must first declare the cursor. In the following lab, you will start by declaring a cursor and then proceed with the process of opening the cursor, fetching from the cursor, and finally closing the cursor.

A table-based record is drawn from a particular table structure. Consider the following code fragment:

For Example

[Click here to view code image](#)

```

DECLARE
    vr_zip ZIPCODE%ROWTYPE;
    vr_instructor INSTRUCTOR%ROWTYPE;

```

The record `vr_zip` has a structure similar to a row of the `ZIPCODE` table. Its elements are `CITY`, `STATE`, and `ZIP`. Note that if the `CITY` column of the `ZIPCODE` table has been defined as `VARCHAR2(15)`, the attribute `CITY` of the `vr_zip` record will have the same data type structure. Similarly, the record `vr_instructor` is based on the row of the `INSTRUCTOR` table.

Making Use of Record Types

Here is an example of a record type in an anonymous PL/SQL block.

For Example

[Click here to view code image](#)

```

SET SERVEROUTPUT ON;
DECLARE
    vr_zip ZIPCODE%ROWTYPE;
BEGIN
    SELECT *

```

```

    INTO vr_zip
    FROM zipcode
    WHERE rownum < 2;
    DBMS_OUTPUT.PUT_LINE('City: '||vr_zip.city);
    DBMS_OUTPUT.PUT_LINE('State: '||vr_zip.state);
    DBMS_OUTPUT.PUT_LINE('Zip: '||vr_zip.zip);
END;

```

In this example, you select a single row for the ZIPCODE table into the vr_zip record. Next, you display each element of the record on the screen. Notice that to reference each attribute of the record, dot notation is used. When run, the example produces the following output:

[Click here to view code image](#)

```

City: Santurce
State: PR
Zip: 00914
PL/SQL procedure successfully completed.

```

A cursor-based record is based on the list of elements of a predefined cursor. The record vr_student_name has a structure similar to a row returned by the SELECT statement defined in the cursor. It contains two attributes, the student's first and last names. A cursor-based record can be declared only after its corresponding cursor has been declared; otherwise, a compilation error will occur.

For Example

[Click here to view code image](#)

```

DECLARE
  CURSOR c_student_name IS
    SELECT first_name, last_name
      FROM student;
  vr_student_name c_student_name%ROWTYPE;

```

Lab 11.2: Cursor Loop

After this lab, you will be able to

- [Process an Explicit Cursor](#)
- [Make Use of User-Defined Records](#)
- [Make Use of Cursor Attributes](#)

To process a cursor, you will have to loop through it. In this section, we explain the details of each step of the loop by going through a code example.

Processing an Explicit Cursor

The following example shows the declaration section of a PL/SQL block that defines a cursor named c_student, based on the student table with the last_name and the first_name concatenated into one item called name and leaving out the created_by and modified_by columns. The block then declares a record based on this cursor.

For Example

[Click here to view code image](#)

```
DECLARE
  CURSOR c_student is
    SELECT first_name||' '||Last_name name
      FROM student;
  vr_student c_student%ROWTYPE;
```

Opening a Cursor

The next step in controlling an explicit cursor is to open it. When the open cursor statement is processed, four actions will take place automatically:

1. The variables (including bind variables) in the WHERE clause are examined.
2. Based on the values of the variables, the active set is determined and the PL/SQL engine executes the query for that cursor. Variables are examined at cursor open time only.
3. The PL/SQL engine identifies the active set of data—the rows from all involved tables that meet the WHERE clause criteria.
4. The active set pointer is set to the first row.

The syntax for opening a cursor is

```
OPEN cursor_name;
```

Did You Know?

A pointer into the active set is also established at the cursor open time. The pointer determines which row is the next to be fetched by the cursor. More than one cursor can be opened at a time.

The following example shows how the cursor `c_student` would be opened by continuing the previous example.

For Example

[Click here to view code image](#)

```
DECLARE
  CURSOR c_student is
    SELECT first_name||' '||Last_name name
      FROM student;
  vr_student c_student%ROWTYPE;
BEGIN
  OPEN c_student;
```

Fetching Rows in a Cursor

After the cursor has been declared and opened, you can retrieve data from the cursor. The process of getting the data from the cursor is referred to as fetching the cursor. There are two methods of fetching a cursor, which use the following commands:

[Click here to view code image](#)

```
FETCH cursor_name INTO PL/SQL variables;
```

or

[Click here to view code image](#)

```
FETCH cursor_name INTO PL/SQL record;
```

When the cursor is fetched, the following occurs:

1. The fetch command is used to retrieve one row at a time from the active set. This is generally done inside a loop. The values of each row in the active set can then be stored into the corresponding variables or PL/SQL record one at a time, performing operations on each one successively.
2. After each **FETCH** command, the active set pointer is moved forward to the next row. Thus, each fetch will return successive rows of the active set, until the entire set is returned. The last **FETCH** command will not assign values to the output variables; thus they will still contain their prior values.

Closing a Cursor

Once all of the rows in the cursor have been processed (retrieved), the cursor should be closed. This tells the PL/SQL engine that the program is finished with the cursor, and the resources associated with it can be freed. The syntax for closing the cursor is

```
CLOSE cursor_name;
```

Did You Know?

Once a cursor is closed, it is no longer valid to fetch from it. Likewise, it is not possible to close an already closed cursor. Either attempt will result in an Oracle error.

For Example *ch11_2a.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
    CURSOR c_student_name IS
        SELECT first_name, last_name
        FROM student
        WHERE rownum <= 5;
    vr_student_name c_student_name%ROWTYPE;
BEGIN
    OPEN c_student_name;
    LOOP
        FETCH c_student_name INTO vr_student_name;
        EXIT WHEN c_student_name%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Student name: ' ||
            vr_student_name.first_name
            ||' '||vr_student_name.last_name);
    END LOOP;
    CLOSE c_student_name;
END;
```

In this example, a cursor is declared that returns five student names. Next, a cursor-

based record is declared. In the body of the program, an explicit cursor is processed via the cursor loop. In the body of the loop, each record is returned by the cursor to the cursor-based record, `vr_student_name`. Next the content of the cursor is displayed on the screen. When run, the example produces the following output:

[Click here to view code image](#)

```
Student name: Austin V. Cadet
Student name: Frank M. Orent
Student name: Yvonne Winnicki
Student name: Mike Madej
Student name: Paula Valentine
PL/SQL procedure successfully completed.
```

Consider the same example with single modification. Notice that the `DBMS_OUTPUT.PUT_LINE` statement has been moved outside the loop.

For Example *ch11_2b.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR c_student_name IS
    SELECT first_name, last_name
      FROM student
     WHERE rownum <= 5;
  vr_student_name c_student_name%ROWTYPE;
BEGIN
  OPEN c_student_name;
  LOOP
    FETCH c_student_name INTO vr_student_name;
    EXIT WHEN c_student_name%NOTFOUND;
  END LOOP;
  CLOSE c_student_name;
  DBMS_OUTPUT.PUT_LINE('Student name: ' ||
    vr_student_name.first_name || ' ' ||
    vr_student_name.last_name);
END;
```

The `DBMS_OUTPUT.PUT_LINE` has been moved outside the loop. First the loop will process the five student records. The values for each record will be placed in the record `vr_student_name`, but each time the loop iterates it will replace the value in the record with a new value. When the five iterations of the loop are finished, it will exit because of the `EXIT WHEN` condition, leaving the `vr_student_name` record with the last value that was in the cursor. This is the only value that will be displayed via the `DBMS_OUTPUT.PUT_LINE`, which comes after the loop is closed.

Making Use of a User-Defined Record

A user-defined record is based on the record type defined by a programmer. To make use of such a record, first you declare a record type, and then you declare a record variable based on the record type.

For Example

[Click here to view code image](#)

```

type type_name IS RECORD
  (field_name 1 DATATYPE 1,
   field_name 2 DATATYPE 2,
   ...
   field_name N DATATYPE N);
record_name TYPE_NAME%ROWTYPE;

```

Consider the following code fragment:

For Example

[Click here to view code image](#)

```

SET SERVEROUTPUT ON;
DECLARE
  -- declare user-defined type
  TYPE instructor_info IS RECORD
    (instructor_id instructor.instructor_id%TYPE,
     first_name instructor.first_name%TYPE,
     last_name instructor.last_name%TYPE,
     sections NUMBER(1));
  -- declare a record based on the type defined above
  rv_instructor instructor_info;

```

In this code fragment, a type `instructor_info` is defined. This type contains four attributes: the instructor's ID, first name, and last name, and the number of sections taught by this instructor. Next, a record based on the type just described is declared. As a result, this record has structure similar to the type, `instructor_info`.

Consider the following example:

For Example *ch11_2c.sql*

[Click here to view code image](#)

```

SET SERVEROUTPUT ON;
DECLARE
  TYPE instructor_info IS RECORD
    (first_name instructor.first_name%TYPE,
     last_name instructor.last_name%TYPE,
     sections NUMBER);
  rv_instructor instructor_info;
BEGIN
  SELECT RTRIM(i.first_name),
         RTRIM(i.last_name), COUNT(*)
    INTO rv_instructor
   FROM instructor i, section s
  WHERE i.instructor_id = s.instructor_id
    AND i.instructor_id = 102
 GROUP BY i.first_name, i.last_name;
 DBMS_OUTPUT.PUT_LINE('Instructor, ' ||
  rv_instructor.first_name ||
  ' ||rv_instructor.last_name ||
  ', teaches '||rv_instructor.sections ||
  ' section(s)');
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
      ('There is no such instructor');
END;

```

In this example, a record called `vr_instructor` is declared. This record is based on the type defined previously. In the body of the PL/SQL block, this record is initialized with the help of the `SELECT INTO` statement, and then the values in the cursor are displayed on the screen. Note that the columns of the `SELECT INTO` statement are listed in the same order that the attributes are defined in `instructor_info` type. There is no need to use dot notation for this record initialization.

When run, this example produces the following output:

[Click here to view code image](#)

```
Instructor, Tom Wojick, teaches 9 section(s)
PL/SQL procedure successfully completed.
```

Making Use of Cursor Attributes

[Table 11.1](#) lists the attributes of a cursor, which are used to determine the result of a cursor operation when *fetched* or *opened*.

Cursor Attribute	Syntax	Explanation
<code>%NOTFOUND</code>	<code>cursor_name%NOTFOUND</code>	A Boolean attribute that returns TRUE if the previous <code>FETCH</code> did not return a row, and FALSE if it did
<code>%FOUND</code>	<code>cursor_name%FOUND</code>	A Boolean attribute that returns TRUE if the previous <code>FETCH</code> returned a row, and FALSE if it did not
<code>%ROWCOUNT</code>	<code>cursor_name%ROWCOUNT</code>	The number of records fetched from a cursor at that point in time
<code>%ISOPEN</code>	<code>Cursor_name%ISOPEN</code>	A Boolean attribute that returns TRUE if the cursor is open, FALSE if it is not

Table 11.1 Explicit Cursor Attributes

You can make use of the attribute `%NOTFOUND` to close a loop. It would also be a wise idea to add an exception clause to the end of the block to close the cursor if it is still open. If you add the following statements to the end of the PL/SQL block, it will be complete:

[Click here to view code image](#)

```
EXIT WHEN c_student%NOTFOUND;
END LOOP;
CLOSE c_student;
EXCEPTION
WHEN OTHERS
THEN
IF c_student%ISOPEN
THEN
CLOSE c_student;
END IF;
END;
```

Cursor attributes can be used with implicit cursors by using the prefix “SQL”—for example, `SQL%ROWCOUNT`.

If you use the `SELECT INTO` syntax in your PL/SQL block, you will be creating an implicit cursor. You can then use these attributes on the implicit cursor.

For Example *ch11_3a.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON;
DECLARE
    v_city zipcode.city%type;
BEGIN
    SELECT city
        Into v_city
        from zipcode
        Where zip = 07002;
    IF SQL%ROWCOUNT = 1
    THEN
        DBMS_OUTPUT.PUT_LINE(v_city ||' has a ' ||
            'ZIP code of 07002');
    ELSIF SQL%ROWCOUNT = 0
    THEN
        DBMS_OUTPUT.PUT_LINE('The ZIP code 07002 is ' ||
            ' not in the database');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Stop harassing me');
    END IF;
END;
```

The PL/SQL block ch11_3a would display the following output:

[Click here to view code image](#)

```
Bayonne has a ZIP code of 07002
PL/SQL procedure successfully completed.
```

The declaration section declares a variable, **v_city**, anchored to the data type of the **city** item in the **zipcode** table. The **SELECT** statement causes an implicit cursor to be opened, fetched, and then closed. The **IF** clause makes use of the attribute **%ROWCOUNT** to determine whether the implicit cursor has a row count of 1. If it does have a row count of 1, then the first **DBMS_OUTPUT** line will be displayed. Notice that this example does not handle a situation where the row count is greater than 1. Since the **zipcode** table's primary key is the ZIP code, this could happen.

If you rerun this block after changing 07002 to 99999, you will get the following result:

```
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 4
```

A **SELECT** statement in a PL/SQL block that does not return any rows will raise a “no data found” exception. Because there is no exception handler, the preceding error would be displayed.

You may have expected the second and third conditions of the **IF** statement to capture the instance of a **%ROWCOUNT** equal to 0. Now that you understand a **SELECT** statement that returns no rows will raise a **WHEN NO_DATA_FOUND** exception, it would be a good idea to handle this situation by adding a **WHEN NO_DATA_FOUND** exception to the existing block. You can add a **%ROWCOUNT** in the exception, either to display the row count in a **DBMS_OUTPUT** statement or to create an **IF** statement that handles the various possibilities.

Putting It All Together

Here is an example of the complete cycle of declaring, opening, fetching, and closing a cursor, including use of cursor attributes.

For Example *ch11_4.sql*

[Click here to view code image](#)

```
1> DECLARE
2>   v_sid      student.student_id%TYPE;
3>   CURSOR c_student IS
4>     SELECT student_id
5>       FROM student
6>      WHERE student_id < 110;
7> BEGIN
8>   OPEN c_student;
9>   LOOP
10>    FETCH c_student INTO v_sid;
11>    EXIT WHEN c_student%NOTFOUND;
12>    DBMS_OUTPUT.PUT_LINE('STUDENT ID : '||v_sid);
13> END LOOP;
14> CLOSE c_student;
15> EXCEPTION
16> WHEN OTHERS
17> THEN
18>   IF c_student%ISOPEN
19>     THEN
20>       CLOSE c_student;
21>     END IF;
22> END;
```

This example illustrates a complete cursor fetch loop, in which multiple rows of data are returned from the query. The cursor is declared in the declaration section of the block (lines 1–6) just like other identifiers. In the executable section of the block (lines 7–15), a cursor is opened using the **OPEN** (line 8) statement. Because the cursor returns multiple rows, a loop is used to assign returned data to the variables with a **FETCH** statement (line 10). Because the loop statement has no other means of termination, an exit condition must be specified. In this case, one of the attributes for the cursor is **%NOTFOUND** (line 11). The cursor is then closed to free the memory allocation (line 14). Additionally, if the exception handler is called, a check is made to see if the cursor is open (line 18) or closed (line 20).

This example is now modified to make use of the cursor attributes **%FOUND** and **%ROWCOUNT**.

For Example *ch11_5.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
  v_sid      student.student_id%TYPE;
  CURSOR c_student IS
    SELECT student_id
      FROM student
     WHERE student_id < 110;
BEGIN
  OPEN c_student;
  LOOP
```

```

FETCH c_student INTO v_sid;
IF c_student%FOUND THEN
DBMS_OUTPUT.PUT_LINE
  ('Just FETCHED row '
   || TO_CHAR(c_student%ROWCOUNT) ||
   ' Student ID: '||v_sid);
ELSE
  EXIT;
END IF;
END LOOP;
CLOSE c_student;
EXCEPTION
WHEN OTHERS
THEN
  IF c_student%ISOPEN
  THEN
    CLOSE c_student;
  END IF;
END;

```

In this script, there has been a modification to the loop structure. Instead of relying on an exit condition, an **IF** statement is used. The **IF** statement makes use of the cursor attribute **%FOUND**, which returns **TRUE** when a row is “found” in the cursor and **FALSE** when it is not found. The attribute **%ROWCOUNT** then returns a number, which is the current row number of the cursor.

The next example demonstrates how to fetch a cursor that has taken data from the **student** table into the cursor variable **%ROWTYPE**. The cursor selects only those students with a **student_id** less than 110. The columns are the **STUDENT_ID**, **LAST_NAME**, **FIRST_NAME**, and a count of the number of classes the student is enrolled in. The cursor is fetched with a loop and then all of the columns are output.

For Example *ch11_6.sql*

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
  CURSOR c_student_enroll IS
    SELECT s.student_id, first_name, last_name,
           COUNT(*) enroll,
           (CASE
             WHEN count(*) = 1 Then ' class.'
             WHEN count(*) is null then
                 ' no classes.'
             ELSE ' classes.'
           END) class
    FROM student s, enrollment e
   WHERE s.student_id = e.student_id
     AND s.student_id <110
   GROUP BY s.student_id, first_name, last_name;
  r_student_enroll  c_student_enroll%ROWTYPE;
BEGIN
  OPEN c_student_enroll;
  LOOP
    FETCH c_student_enroll INTO r_student_enroll;
    EXIT WHEN c_student_enroll%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Student INFO: ID ' ||

```

```

    r_student_enroll.student_id||' is '' ||
    r_student_enroll.first_name|| ' ' ||
    r_student_enroll.last_name||
    ' is enrolled in '||r_student_enroll.enroll||
    r_student_enroll.class);
END LOOP;
CLOSE c_student_enroll;
EXCEPTION
  WHEN OTHERS
  THEN
    IF c_student_enroll %ISOPEN
      THEN
        CLOSE c_student_enroll;
    END IF;
  END;

```

In the declaration section, a cursor `c_student_enroll` is defined as well as a record, which is the type of a row of the cursor. The cursor loop structure makes use of an exit condition with the `%NOTFOUND` cursor attribute. When there are no more rows, the `%NOTFOUND` attribute will be `TRUE` and will cause the loop to exit. While the cursor is open and the loop is processing, it will fetch a row of the cursor in a record, one at a time. The DBMS output will cause each row to be displayed to the screen. Finally, the cursor is closed. An exception clause will also close the cursor if any error is raised.

Assorted Tips on Cursors

Cursor SELECT LIST

Match the select list with PL/SQL variables or PL/SQL record components.

The number of variables must be equal to the number of columns or expressions in the select list. The number of the components of a record must match the columns or expressions in the select list.

Cursor Scope

The scope of a cursor declared in the main block (or an enclosing block) extends to the sub-blocks.

Expressions in a Cursor SELECT List

PL/SQL variables, expressions, and even functions can be included in the cursor select list.

Column Aliases in Cursors

An alternative name you provide to a column or expression in the select list. In an explicit cursor column, aliases are required for calculated columns when

- You `FETCH` into a record declared with a `%ROWTYPE` declaration against that cursor.
 - You want to reference the calculated column in the program.
-

Lab 11.3: Cursor FOR LOOPS

After this lab, you will be able to

- [Make Use of Cursor FOR LOOPS](#)

Making Use of Cursor FOR LOOPS

An alternative method of handling cursors is called the cursor FOR LOOP because of the simplified syntax that is used. When using the cursor FOR LOOP, the process of opening, fetching, and closing is handled implicitly. This makes the blocks much simpler to code and easier to maintain.

The cursor FOR LOOP specifies a sequence of statements to be repeated once for each row returned by the cursor. You can use a cursor FOR LOOP when you need to fetch and process each and every record from a cursor until you want to stop processing and exit the loop.

To make use of this column, you need to create a new table called `table_log` with the following script:

[Click here to view code image](#)

```
create table table_log  
    (description VARCHAR2(250));
```

Then run this script:

For Example `ch11_7.sql`

[Click here to view code image](#)

```
DECLARE  
    CURSOR c_student IS  
        SELECT student_id, last_name, first_name  
            FROM student  
            WHERE student_id < 110;  
BEGIN  
    FOR r_student IN c_student  
    LOOP  
        INSERT INTO table_log  
            VALUES(r_student.last_name);  
    END LOOP;  
END;  
SELECT * from table_log;
```

The following PL/SQL block reduces by 5 percent the cost of all courses having an enrollment of eight or more students. It makes use of a cursor FOR LOOP that updates the course table with the discounted cost.

For Example `ch11_8.sql`

[Click here to view code image](#)

```
DECLARE  
    CURSOR c_group_discount IS  
        SELECT DISTINCT s.course_no
```

```

    FROM section s, enrollment e
    WHERE s.section_id = e.section_id
      GROUP BY s.course_no, e.section_id, s.section_id
      HAVING COUNT(*)>=8;
BEGIN
  FOR r_group_discount IN c_group_discount  LOOP
    UPDATE course
      SET cost = cost * .95
      WHERE course_no = r_group_discount.course_no;
  END LOOP;
  COMMIT;
END;

```

The cursor `c_group_discount` is declared in the declaration section. The proper SQL is used to generate the `SELECT` statement to answer the question given. The cursor is processed in a `FOR LOOP` in each iteration of the loop and the SQL update statement is executed. As a consequence, the cursor does not have to be opened, fetched, and closed. Also, a cursor attribute does not have to be used to create an exit condition for the loop that is processing the cursor.

Lab 11.4: Nested Cursors

After this lab, you will be able to

- [Process Nested Cursors](#)

Cursors can be nested inside each other. Although this may sound complex, it is really just a loop inside a loop, much like nested loops, which were covered in previous chapters. If you have one parent cursor and two child cursors, then each time the parent cursor makes a single loop, it will loop through each child cursor once and then begin a second round. In the next two examples, you will encounter a nested cursor with a single child cursor.

Processing Nested Cursors

In the following example, line numbers were added so that individual lines could be referenced in the explanation that follows. You will have to remove the line numbers when you run the code.

For Example `ch11_9.sql`

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
1  DECLARE
2    v_zip zipcode.zip%TYPE;
3    v_student_flag CHAR;
4    CURSOR c_zip IS
5      SELECT zip, city, state
6        FROM zipcode
7       WHERE state = 'CT';
8    CURSOR c_student IS
9      SELECT first_name, last_name
10         FROM student
11        WHERE zip = v_zip;

```

```

12   BEGIN
13     FOR r_zip IN c_zip
14       LOOP
15       v_student_flag := 'N';
16       v_zip := r_zip.zip;
17       DBMS_OUTPUT.PUT_LINE(CHR(10));
18       DBMS_OUTPUT.PUT_LINE('Students living in ' ||
19         r_zip.city);
20     FOR r_student IN c_student
21       LOOP
22       DBMS_OUTPUT.PUT_LINE(
23         r_student.first_name ||
24         ' ' || r_student.last_name);
25       v_student_flag := 'Y';
26     END LOOP;
27     IF v_student_flag = 'N'
28     THEN
29       DBMS_OUTPUT.PUT_LINE
          ('No students for this ZIP code');
30     END IF;
31   END LOOP;
32 END;

```

There are two cursors in the example: a cursor of the ZIP codes and a cursor of the list of students. The variable `v_zip` is initialized in line 16 to be the ZIP code of the current record of the `c_zip` cursor. The `c_student` cursor ties in the `c_zip` cursor by means of this variable. Thus, when the cursor is processed in lines 20–26, it retrieves students who have the ZIP code of the current record for the parent cursor. The parent cursor is processed in lines 13–31. Each iteration of the parent cursor executes the `DBMS_OUTPUT` statement in lines 16 and 17 only once. The `DBMS_OUTPUT` statement in line 22 is executed once for each iteration of the child loop, producing a line of output for each student. The `DBMS_OUTPUT.PUT_LINE` statement in line 29 executes only if the inner loop did not execute; this is accomplished by setting a variable `v_student_flag`. The variable is set to `N` in the beginning of the parent loop. If the child loop executes at least once, the variable will be set to `Y`. After the child loop is closed, a check is made with an `IF` statement to determine the value of the variable. If it is still `N`, then it can be safely concluded that the inner loop did not process. This will then allow the last `DBMS_OUTPUT.PUT_LINE` statement to execute. Nested cursors are more often parameterized. Parameters in cursors are explained in depth in [Lab 12.2, “Complex Nested Cursors.”](#)

The next example is a PL/SQL block with two cursor `FOR LOOP`. The parent cursor retrieves the `student_id`, `first_name`, and `last_name` from the `student` table for students with a `student_id` less than 110 and outputs one line with this information. For each student, the child cursor loops through all the courses in which the student is enrolled, outputting the `course_no` and the description.

For Example *ch11_10.sql*

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
  v_sid student.student_id%TYPE;

```

```

CURSOR c_student IS
    SELECT student_id, first_name, last_name
        FROM student
        WHERE student_id < 110;
CURSOR c_course IS
    SELECT c.course_no, c.description
        FROM course c, section s, enrollment e
        WHERE c.course_no = s.course_no
        AND s.section_id = e.section_id
        AND e.student_id = v_sid;
BEGIN
    FOR r_student IN c_student
    LOOP
        v_sid := r_student.student_id;
        DBMS_OUTPUT.PUT_LINE(chr(10));
        DBMS_OUTPUT.PUT_LINE(' The Student ' ||
            r_student.student_id || ' ||
            r_student.first_name || ' ||
            r_student.last_name);
        DBMS_OUTPUT.PUT_LINE(' is enrolled in the ' ||
            'following courses: ');
        FOR r_course IN c_course
        LOOP
            DBMS_OUTPUT.PUT_LINE(r_course.course_no || ' ' || r_course.description);
        END LOOP;
    END LOOP;
END;

```

The **SELECT** statements for the two cursors are defined in the declaration section of the PL/SQL block. A variable to store the **student_id** from the parent cursor is also declared. The course cursor is the child cursor. Because it makes use of the variable **v_sid**, the variable must be declared first. Both cursors are processed with a **FOR LOOP**, which eliminates the need for **OPEN**, **FETCH**, and **CLOSE** statements. When the parent student loop is processed, the first step is to initialize the variable **v_sid**, and the value is then used when the child loop is processed. A **DBMS_OUTPUT** statement is used so that the display is generated for each cursor loop. The parent cursor will display the student name once, and the child cursor will display the name of each course in which the student is enrolled.

The following example shows a nested cursor:

For Example ch11_11.sql

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
    v_amount course.cost%TYPE;
    v_instructor_id instructor.instructor_id%TYPE;
    CURSOR c_inst IS
        SELECT first_name, last_name, instructor_id
            FROM instructor;
    CURSOR c_cost IS
        SELECT c.cost
            FROM course c, section s, enrollment e
            WHERE s.instructor_id = v_instructor_id
            AND c.course_no = s.course_no

```

```

        AND s.section_id = e.section_id;
BEGIN
  FOR r_inst IN c_inst
  LOOP
    v_instructor_id := r_inst.instructor_id;
    v_amount := 0;
    DBMS_OUTPUT.PUT_LINE(
      'Amount generated by instructor ' ||
      r_inst.first_name || ' ' || r_inst.last_name
      || ' is');
    FOR r_cost IN c_cost
    LOOP
      v_amount := v_amount + NVL(r_cost.cost, 0);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE
      (' ' || TO_CHAR(v_amount, '$999,999'));
  END LOOP;
END;

```

The declaration section contains a declaration for two variables. The first is `v_amount` of the data type matching that of the cost in the `course` table; the second is the `v_instructor_id` of the data type matching the `instructor_id` in the `instructor` table. There are also two declarations for two cursors. The first is for `c_inst`, which consists of the `first_name`, `last_name`, and `instructor_id` for an instructor from the `instructor` table. The second cursor, `c_cost`, will produce a result set of the cost of the course taken for each student enrolled in a course given by the instructor that matches the variable `v_instructor_id`. These two cursors will be run in nested fashion.

First, the cursor `c_inst` is opened in a `FOR LOOP`. The value of the variable `v_instructor_id` is initialized to match the `instructor_id` of the current row of the `c_inst` cursor. The variable `v_amount` is initialized to 0.

The second cursor is opened within the loop for the first cursor. Consequently, for each iteration of the cursor `c_inst`, the second cursor will be opened, fetched, and closed. The second cursor will loop through all the costs generated by each student enrolled in a course for the instructor, which is the current value of the `c_inst` cursor. Each time the nest loop iterates, it will increase the variable `v_amount` by adding the current cost in the `c_cost` loop.

Prior to opening the `c_cost` loop, a `DBMS_OUTPUT` statement displays the instructor name. After the `c_cost` cursor loop is closed, a `DBMS_OUTPUT` statement displays the total amount generated by all the enrollments of the current instructor.

The result set would be as follows:

[Click here to view code image](#)

```

Amount generated by instructor Fernand Hanks is
$49,110
Amount generated by instructor Tom Wojick is
$24,582
Amount generated by instructor Nina Schorin is
$43,319
Amount generated by instructor Gary Pertz is

```

```
$29,317
Amount generated by instructor Anita Morris is
$18,662
Amount generated by instructor Todd Smythe is
$21,092
Amount generated by instructor Marilyn Frantzen is
$34,311
Amount generated by instructor Charles Lowry is
$37,512
Amount generated by instructor Rick Chow is
$0
Amount generated by instructor Irene Willig is
$0
```

In this example, the nested cursor is tied to the current row of the outer cursor by means of the variable `v_instructor_id`. A more common way of handling this task is to pass a parameter to a cursor. You will learn more about how to achieve this in [Chapter 12](#).

Summary

In this chapter you learned how to make use of various types of cursors. First you learned how the Oracle platform processes an implicit cursor, and then you learned all of the steps required to use an explicit cursor. Additionally, you learned about the various record types and saw how to use them in the context of a cursor. Finally, you learned about three types of cursor loops: a regular loop, a `FOR LOOP`, and nested cursors.

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.

12. Advanced Cursors

In this chapter, you will learn about

- [Parameterized Cursors](#)
- [Complex Nested Cursors](#)
- [FOR UPDATE and WHERE CURRENT Cursors](#)

In [Chapter 11](#), you mastered the basic concepts of cursors. In this chapter, you will learn how to dynamically alter the WHERE clause of a cursor by passing parameters when you call the cursor. In [Chapter 21](#), you will take cursors to another level—that is, in the context of a package, you will learn to implement cursor variables.

Lab 12.1: Parameterized Cursors

After this lab, you will be able to

- [Use Parameters in a Cursor](#)

Cursors with Parameters

A cursor can be declared with parameters. This approach enables a cursor to generate a specific result set that is simultaneously narrow and reusable. A cursor of all the data from the ZIPCODE table may be very useful, for instance, but it would be even more useful for certain data processing if it held information for only one state. At this point, you know how to create such a cursor. But wouldn't it be more useful if you could create a cursor that could accept a parameter of a state and then run through only the city and ZIP code for that state?

For Example

[Click here to view code image](#)

```
CURSOR c_zip (p_state IN zipcode.state%TYPE) IS
  SELECT zip, city, state
    FROM zipcode
   WHERE state = p_state;
```

The main points to keep in mind for parameters in cursors are as follows:

- Cursor parameters make the cursor more reusable.
- Cursor parameters can be assigned default values.
- The scope of the cursor parameters is local to the cursor.
- The mode of the parameters can only be IN.

When a cursor has been declared as taking a parameter, it must be called with a value for that parameter. The c_zip cursor that was just declared is called as follows:

```
OPEN c_zip (parameter_value)
```

The same cursor could be opened with a cursor **FOR** loop as follows:

```
FOR r_zip IN c_zip('NY')
LOOP ...
```

The cursor from the previous example is expanded into a parameterized cursor in the next example. This example includes a **DBMS_OUTPUT** line that displays the ZIP code, city, and state. This is identical to the process we used earlier in cursor **FOR** loops, except that now when the cursor is opened, a parameter is passed.

For Example *ch12_1.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR c_zip (p_state IN zipcode.state%TYPE) IS
    SELECT zip, city, state
      FROM zipcode
     WHERE state = p_state
BEGIN
  FOR r_zip IN c_zip('NJ')
  LOOP...
    DBMS_OUTPUT.PUT_LINE(r_zip.city ||
      ' '||r_zip.zip);
  END LOOP;
END;
```

To complete the block, the cursor declaration is surrounded by **DECLARE** and **BEGIN**. The cursor is opened by passing the parameter “NJ,” and then, for each iteration of the cursor loop, the ZIP code and the city are displayed by using the built-in package **DBMS_OUTPUT**.

Lab 12.2: Complex Nested Cursors

After this lab, you will be able to

- Use Complex Nested Cursors

Nesting cursors allows for looping through data at various stages. For example, one cursor might loop through ZIP codes. When it hits one ZIP code, a second cursor might be nested that loops through students who live in that ZIP code. Working through a specific example will help explain this approach in more detail.

The following PL/SQL code is complex. It involves all of the topics covered so far in this chapter. There is a nested cursor with three levels, meaning a grandparent cursor, a parent cursor, and a child cursor.

For Example *ch12_2.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
1  DECLARE
2    CURSOR c_student IS
3      SELECT first_name, last_name, student_id
```

```

4      FROM student
5      WHERE last_name LIKE 'J%';
6  CURSOR c_course
7      (i_student_id IN student.student_id%TYPE)
8  IS
9      SELECT c.description, s.section_id sec_id
10     FROM course c, section s, enrollment e
11    WHERE e.student_id = i_student_id
12    AND c.course_no = s.course_no
13    AND s.section_id = e.section_id;
14  CURSOR c_grade(i_section_id IN section.section_id%TYPE,
15                  i_student_id IN student.student_id%TYPE)
16  IS
17      SELECT gt.description grd_desc,
18          TO_CHAR
19              (AVG(g.numeric_grade), '999.99') num_grd
20      FROM enrollment e,
21          grade g, grade_type gt
22     WHERE e.section_id = i_section_id
23     AND e.student_id = g.student_id
24     AND e.student_id = i_student_id
25     AND e.section_id = g.section_id
26     AND g.grade_type_code = gt.grade_type_code
27      GROUP BY gt.description ;
28 BEGIN
29     FOR r_student IN c_student
30     LOOP
31         DBMS_OUTPUT.PUT_LINE(CHR(10));
32         DBMS_OUTPUT.PUT_LINE(r_student.first_name ||
33             ' '||r_student.last_name);
34         FOR r_course IN c_course(r_student.student_id)
35         LOOP
36             DBMS_OUTPUT.PUT_LINE ('Grades for course :'|||
37                 r_course.description);
38             FOR r_grade IN c_grade(r_course.sec_id,
39                                 r_student.student_id)
40             LOOP
41                 DBMS_OUTPUT.PUT_LINE(r_grade.num_grd|||
42                     ' '|||r_grade.grd_desc);
43             END LOOP;
44         END LOOP;
45     END LOOP;
46 END;

```

The grandparent cursor, `c_student`, is declared in lines 2–5. It takes no parameters and is a collection of students with a last name beginning with “J”. The parent cursor is declared in lines 6–13. This cursor, `c_course`, takes the parameter of the `student_ID` to generate a list of courses taken by that student. The child cursor, `c_grade`, is declared in lines 14–27. It takes two parameters, the `section_id` and the `student_id`. In this way it can generate an average of the different grade types for that student for that course.

The grandparent cursor loop begins on line 29, and only the student name is displayed with `DBMS_OUTPUT`. The parent cursor loop begins on line 35. It takes the parameter of the `student_id` from the grandparent cursor. Only the description of the course is displayed. The child cursor loop begins on line 40. It takes the parameter of the `section_id` from the parent cursor and the `student_id` from the grandparent cursor.

The grades are then displayed. The grandparent cursor loop ends on line 45, the parent cursor on line 44, and, finally, the child cursor on line 43.

The output of this script is a student name, followed by the courses that student is taking and the average grade he or she has earned for each grade type.

Lab 12.3: FOR UPDATE and WHERE CURRENT Cursors

After this lab, you will be able to

- [Use a FOR UPDATE Cursor](#)
- [Use WHERE CURRENT in a Cursor](#)

FOR UPDATE Cursor

The cursor **FOR UPDATE** clause is used with a cursor only when you want to update tables in the database. Generally, when you execute a **SELECT** statement, you are not locking any rows. The purpose of using the **FOR UPDATE** clause is to lock the rows of the tables that you want to update, so that another user cannot perform an update until you complete your update and release the lock. The next **COMMIT** or **ROLLBACK** statement releases the lock.

The **FOR UPDATE** clause will change the manner in which the cursor operates in only a few respects. When you open a cursor, all rows that meet the restriction criteria are identified as part of the active set. Using the **FOR UPDATE** clause will lock these rows that have been identified in the active set. If the **FOR UPDATE** clause is used, then rows may not be fetched from the cursor until a **COMMIT** has been issued. It is important for you to consider where to place the **COMMIT** statement. Be careful to consider the issues discussed in relation to transaction management in [Chapter 3](#).

The syntax is simply to add **FOR UPDATE** to the end of the cursor definition. If multiple items are being selected but you want to lock only one of them, then end the cursor definition with the following syntax:

```
FOR UPDATE OF <item_name>
```

For Example *ch12_3.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR c_course IS
    SELECT course_no, cost
      FROM course FOR UPDATE;
BEGIN
  FOR r_course IN c_course
  LOOP
    IF r_course.cost < 2500
    THEN
      UPDATE course
        SET cost = r_course.cost + 10
        WHERE course_no = r_course.course_no;
```

```
    END IF;
  END LOOP;
END;
```

This example shows how to update the cost of all courses with a cost of less than 2500. It will increment each of these costs by 10.

Several issues must be taken into account with **FOR UPDATE** cursors in terms of where to place a **COMMIT** statement. The following example demonstrates one way of handling this issue.

For Example ch12_4.sql

[Click here to view code image](#)

```
DECLARE
  CURSOR c_grade(
    i_student_id IN enrollment.student_id%TYPE,
    i_section_id IN enrollment.section_id%TYPE)
  IS
    SELECT final_grade
      FROM enrollment
     WHERE student_id = i_student_id
       AND section_id = i_section_id
      FOR UPDATE;
  CURSOR c_enrollment IS
    SELECT e.student_id, e.section_id
      FROM enrollment e, section s
     WHERE s.course_no = 135
       AND e.section_id = s.section_id;
BEGIN
  FOR r_enroll IN c_enrollment
  LOOP
    FOR r_grade IN c_grade(r_enroll.student_id,
                           r_enroll.section_id)
    LOOP
      UPDATE enrollment
        SET final_grade = 90
       WHERE student_id = r_enroll.student_id
         AND section_id = r_enroll.section_id;
    END LOOP;
  END LOOP;
END;
```

Placing a **COMMIT** statement after each update can be costly. If there are a lot of updates and the **COMMIT** comes after the block loop, however, there is a risk of a rollback segment not being large enough. Normally, the **COMMIT** statement would go after the loop, except when the transaction count is high; in such a situation, you might want to code something that does a **COMMIT** for each 10,000 records. If this script were part of a large procedure, you might want to put a **SAVEPOINT** after the loop. Then, if you need to roll back this update at a later point, it would be an easy task.

If this example were run, the **final_grade** for all students enrolled in course 135 would be updated to 90. There are two cursors here. One cursor captures the students who are enrolled in course 135 and places them into the active set. The other cursor takes the **student_id** and the **section_id** from this active set, selects the corresponding **final_grade** from the **enrollment** table, and locks the entire **enrollment** table. The

enrollment cursor loop is begun first, and it passes the `student_id` and the `section_id` as `IN` parameters to the second cursor loop of the `c_grade` cursor, which performs the update. A `COMMIT` statement should be added immediately after the update to ensure that each update is committed to the database.

FOR UPDATE OF in a Cursor

`FOR UPDATE OF` can be used when creating a cursor for an update operation that is based on multiple tables. `FOR UPDATE OF` locks the rows of a table that both contain one of the specified columns and are members of the active set. In other words, it is the means of specifying which table you want to lock. If the `FOR UPDATE OF` clause is used, then rows may not be fetched from the cursor until a `COMMIT` has been issued.

For Example *ch12_5.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR c_stud_zip IS
    SELECT s.student_id, z.city
      FROM student s, zipcode z
     WHERE z.city = 'Brooklyn'
       AND s.zip = z.zip
    FOR UPDATE OF phone;
BEGIN
  FOR r_stud_zip IN c_stud_zip
  LOOP
    UPDATE student
      SET phone = '718'||SUBSTR(phone,4)
     WHERE student_id = r_stud_zip.student_id;
  END LOOP;
END;
```

This example updates the phone numbers of students living in Brooklyn by changing the area code to 718. The cursor declaration locks the phone column of the student table. This lock is never released, however, because there is no `COMMIT` or `ROLLBACK` statement.

WHERE CURRENT OF in a Cursor

Use `WHERE CURRENT OF` when you want to update the most recently fetched row. This clause can only be used with a `FOR UPDATE OF` cursor. The advantage of the `WHERE CURRENT OF` clause is that it enables you to eliminate the `WHERE` clause in the `UPDATE` statement.

For Example *ch12_6.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR c_stud_zip IS
    SELECT s.student_id, z.city
      FROM student s, zipcode z
     WHERE z.city = 'Brooklyn'
       AND s.zip = z.zip
```

```
FOR UPDATE OF phone;
BEGIN
  FOR r_stud_zip IN c_stud_zip
  LOOP
    DBMS_OUTPUT.PUT_LINE(r_stud_zip.student_id);
    UPDATE student
      SET phone = '718'||SUBSTR(phone, 4)
      WHERE CURRENT OF c_stud_zip;
  END LOOP;
END;
```

These last two examples perform the same update. The `WHERE CURRENT OF` clause allows you to eliminate a match in the `UPDATE` statement, because the update is being performed for the current record of the cursor only.

Did You Know?

The `FOR UPDATE` and `WHERE CURRENT OF` syntax can be used with cursors that are performing a delete as well as an update.

Summary

The chapter explored various advanced topics involving cursors. First, you learned how to pass parameters to cursors to restrict the result set of a cursor. Then, you learned how to nest cursors. Finally, you saw the syntax for creating cursors that make database updates.

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.

13. Triggers

In this chapter, you will learn about

- [What Triggers Are](#)
- [Types of Triggers](#)

In [Chapter 1](#), you encountered the concept of named PL/SQL blocks such as procedures, functions, and packages that can be stored in the database. In this chapter, you will learn about another type of named PL/SQL block called a database trigger. You will also learn about different characteristics of triggers and their usage in the database.

Lab 13.1: What Triggers Are

After this lab, you will be able to

- [Define a Database Trigger](#)
- [Use BEFORE and AFTER Triggers](#)
- [Employ Autonomous Transactions](#)

Database Trigger

A database trigger is a named PL/SQL block that is stored in a database and executed implicitly when a *triggering event* occurs. The act of executing a trigger is referred to as firing the trigger. A triggering event can be any of the following:

- A DML (for example, `INSERT`, `UPDATE`, or `DELETE`) statement executed against a database table. Such trigger can fire before or after a triggering event. For example, if you have defined a trigger to fire before an `INSERT` statement on the `STUDENT` table, this trigger fires each time before you insert a row in the `STUDENT` table.
- A DDL (for example, `CREATE` or `ALTER`) statement executed either by a particular user against a schema or by any user. Such triggers are often used for auditing purposes and are specifically helpful to Oracle database administrators. They can record various schema changes, including when those changes were made and by which user.
- A system event such as startup or shutdown of the database.
- A user event such as login and logoff. For example, you can define a trigger that fires after a login on a database and that records the username and time of login.

The general syntax for creating a trigger is shown in [Listing 13.1](#) (the reserved words and phrases surrounded by brackets are optional).

Listing 13.1 General Syntax for Creating a Trigger

[Click here to view code image](#)

```
CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] TRIGGER trigger_name
{BEFORE|AFTER} triggering_event ON table_name
[FOR EACH ROW]
[FOLLOWS|PRECEDES another_trigger]
[ENABLE/DISABLE]
[WHEN condition]
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

The reserved word CREATE specifies that you are creating a new trigger. The reserved word REPLACE specifies that you are modifying an existing trigger. REPLACE is optional. Note, however, that both CREATE and REPLACE are included in most cases.

Suppose you create a trigger as shown in [Listing 13.2](#).

Listing 13.2 Creating Trigger

```
CREATE TRIGGER trigger_name
...
```

In a few days, you decide to modify this trigger. If you do not include the reserved word REPLACE in the CREATE clause of the trigger, an error message will be generated when you compile the trigger. The error message states that the name of your trigger is already used by another object. Once REPLACE is included in the CREATE clause of the trigger, there is less chance of an error occurring because if it is a new trigger, it is created, and if it is an old trigger, it is replaced.

However, you should be mindful when using the reserved word REPLACE for a number of reasons. First, if you happen to use REPLACE with the name of an existing stored function, procedure, or package, you will end up with different database objects that have the same name. This occurs because triggers have a separate naming space in the database. While sharing of the same name by a trigger and a procedure, function, or package does not cause errors, potentially it might become confusing; thus it is not considered a good programming practice. Second, when you use the reserved word REPLACE and decide to associate a different table with your trigger, an error message is generated. For example, assume you created a trigger STUDENT_BI on the STUDENT table. Next, you decide to modify this trigger and associate it with the ENROLLMENT table. As a result, the following error message is generated:

[Click here to view code image](#)

```
ORA-04095: trigger 'STUDENT_BI' already exists on another table, cannot
replace it
```

The optional reserved words EDITIONABLE and NONEDITIONABLE specify whether a trigger is an editioned or noneditioned object. Note that this designation applies only if editioning has been enabled for object type TRIGGER.

Did You Know?

Oracle introduced a very important feature called edition-based redefinition in version 11g, release 2. This feature enables you to apply changes to various database objects without invalidating the whole system, thereby allowing for near-zero downtime. For example, previously making structural changes to a table would invalidate numerous functions, procedures, and packages dependent on that table. As a result, you would need to check and recompile all invalidated database objects, potentially requiring downtime for the database. With edition-based redefinition, you can implement all these changes seamlessly and migrate users from the old version of the system to the new version without incurring any downtime.

Edition-based redefinition is outside the scope of this book. Detailed information on this feature can be found in Oracle's online help (www.oracle.com).

The `trigger_name` references the name of the trigger. `BEFORE` or `AFTER` specifies when the trigger fires (before or after the triggering event). The `triggering_event` references a DML statement issued against the table. The `table_name` is the name of the table associated with the trigger. The clause `FOR EACH ROW` specifies that a trigger is a row-level trigger and fires once for each row either inserted, updated, or deleted. You will encounter row- and statement-level triggers in [Lab 13.2](#). A `WHEN` clause specifies a condition that must evaluate to `TRUE` for the trigger to fire. For example, this condition may specify a certain restriction on the column of a table.

The next two options, `FOLLOWS/PRECEDES` and `ENABLE/DISABLE`, were added to the `CREATE OR REPLACE TRIGGER` clause in Oracle 11g. Prior to Oracle 11g, you needed to issue the `ALTER TRIGGER` command to enable or disable a trigger once it had been created. The `ENABLE/DISABLE` option specifies whether a trigger is created in the enabled or disabled state. When a trigger is enabled, it fires when a triggering event occurs. Conversely, when a trigger is disabled, it does not fire when a triggering event occurs. Note that when a trigger is first created without `ENABLE/DISABLE` option, it is enabled by default. To disable the trigger, you need to issue the `ALTER TRIGGER` command, as shown in [Listing 13.3](#).

Listing 13.3 Disabling Trigger

[Click here to view code image](#)

```
ALTER TRIGGER trigger_name DISABLE;
```

Similarly, to enable a trigger that was disabled previously, you issue the `ALTER TRIGGER` command, as shown in [Listing 13.4](#).

Listing 13.4 Enabling Trigger

[Click here to view code image](#)

```
ALTER TRIGGER trigger_name ENABLE;
```

The `FOLLOWS/PRECEDES` option allows you to specify the order in which triggers

should fire. It applies to triggers that are defined on the same table and fire at the same timing point. For example, if you defined two triggers on the STUDENT table that fire before the insert operation is carried out, Oracle does not guarantee the order in which these triggers will fire unless you explicitly specify it with the FOLLOWSPRECEDES clause. Note that the trigger referenced in the FOLLOWSPRECEDES clause must already exist and have been successfully compiled.

The portion of the trigger described to this point is often referred to as the trigger header. Next, we define the trigger body. The body of a trigger has the same structure as an anonymous PL/SQL block. Similar to the case for a PL/SQL block, the declaration and exception sections are optional.

Triggers are used for different purposes, such as the following:

- Enforcing complex business rules that cannot be defined by using integrity constraints
- Maintaining complex security rules
- Automatically generating values for derived columns
- Collecting statistical information on table accesses
- Preventing invalid transactions
- Providing value auditing

The body of a trigger is a PL/SQL block. However, several restrictions apply when you decide to create a trigger:

- A trigger may not issue a transactional control statement such as COMMIT, SAVEPOINT, or ROLLBACK. When the trigger fires, all operations performed by the trigger become part of a transaction. When a transaction is committed or rolled back, the operations performed by the trigger are committed or rolled back as well. An exception to this rule is a trigger that contains an autonomous transaction. Autonomous transactions are discussed in detail later in this lab.
- Any function or procedure called by a trigger may not issue a transactional control statement unless it contains an autonomous transaction.
- It is not permissible to declare LONG or LONG RAW variables in the body of a trigger.

Did You Know?

If you drop a table, the table's database triggers are dropped as well.

BEFORE Triggers

Consider the following example of a trigger on the STUDENT table mentioned earlier in this chapter. This trigger fires before the INSERT statement on the STUDENT table and populates the STUDENT_ID, CREATED_DATE, MODIFIED_DATE, CREATED_BY, and MODIFIED_BY columns. The column STUDENT_ID is populated with the number generated by the STUDENT_ID_SEQ sequence, and the columns CREATED_DATE, MODIFIED_DATE, CREATED_USER, and MODIFIED_USER are populated with the current date and the current user name information, respectively.

For Example ch13_1a.sql

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON STUDENT
FOR EACH ROW
BEGIN
  :NEW.student_id    := STUDENT_ID_SEQ.NEXTVAL;
  :NEW.created_by   := USER;
  :NEW.created_date := SYSDATE;
  :NEW.modified_by  := USER;
  :NEW.modified_date:= SYSDATE;
END;
```

This trigger fires for each row before the INSERT statement on the STUDENT table. Notice that the name of the trigger is STUDENT_BI, where “STUDENT” references the name of the table on which the trigger is defined and the letters “BI” mean “before insert.” There is no specific requirement for naming triggers; however, this approach to naming a trigger is descriptive because the name of the trigger contains the name of the table affected by the triggering event, the time of the triggering event (before or after), and the triggering event itself.

In the body of the trigger, there is a *pseudorecord*, :NEW, which allows for accessing a row that is currently being processed. In other words, a row is inserted into the STUDENT table. The :NEW pseudorecord is of a type TRIGGERING_TABLE%TYPE, so, in this case, it is of the STUDENT%TYPE type. To access individual members of the pseudorecord :NEW, dot notation is used. In other words, :NEW.CREATED_BY refers to the member CREATED_BY of the :NEW pseudorecord, and the name of the record is separated by the dot from the name of its member.

Did You Know?

In addition to the :NEW pseudorecord, an :OLD pseudorecord exists. It allows you to access the current information of the record that is being updated or deleted. Thus the :OLD pseudorecord is undefined for the INSERT statements and the :NEW pseudorecord is undefined for the DELETE statements. However, the PL/SQL compiler does not generate syntax errors when :OLD or :NEW pseudorecords are used in triggers where the triggering event is an INSERT or DELETE operation, respectively. In this case, the member values are set to NULL for the :OLD and :NEW pseudorecords.

Take a closer look at the statement that assigns a sequence value to the STUDENT_ID column. The ability to access a sequence via PL/SQL expressions is a new feature added in Oracle 11g. Prior to Oracle 11g, sequences could be accessed only via queries, as shown in the next version of the example.

For Example *Code Fragment Based on ch13_1a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER student_bi
...
DECLARE
  v_student_id STUDENT.STUDENT_ID%TYPE;
BEGIN
  SELECT STUDENT_ID_SEQ.NEXTVAL
    INTO v_student_id
   FROM dual;
...
END;
```

To create this trigger on the STUDENT table in SQL Developer, you may choose from the two options. First, the trigger can be created by executing the script in the Worksheet window, just as you would with any other PL/SQL block. At the time of trigger compilation, you are prompted to enter the value for bind variables because of the references to the :NEW and :OLD pseudorecords in the body of the trigger, as shown in [Figure 13.1](#). Note that check box next to NULL. If it is checked, simply click the Apply button and the trigger will be created. If this check box is not checked, then check it and click the Apply button.

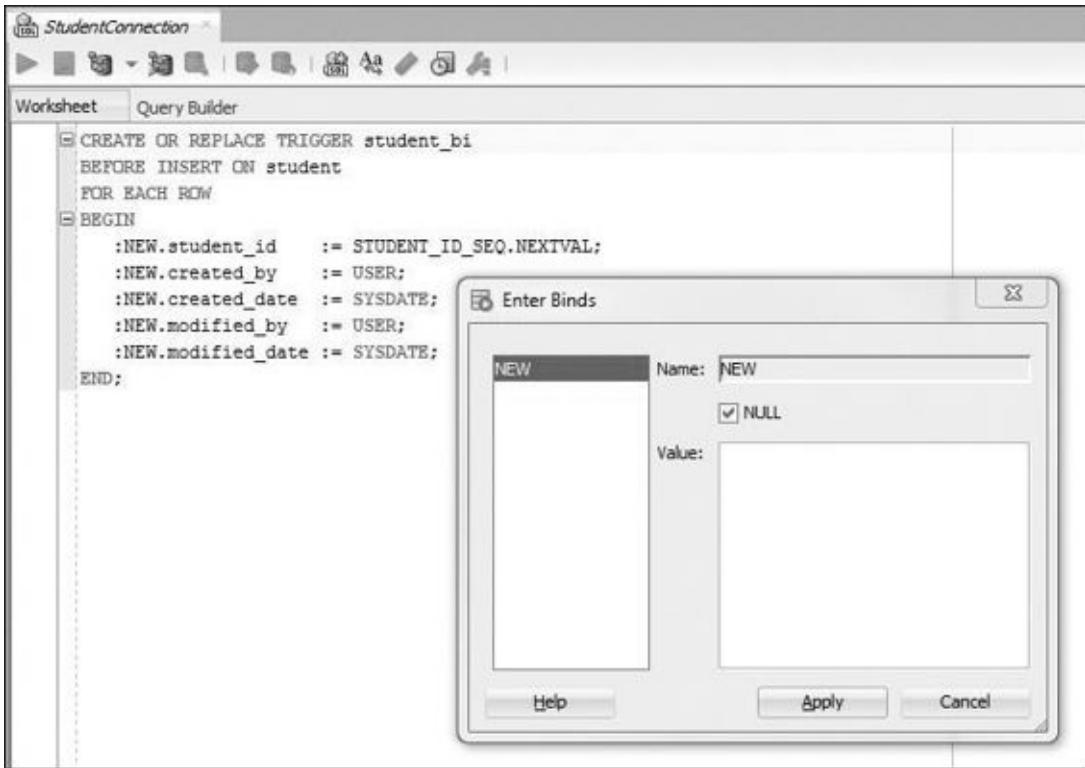


Figure 13.1 Creating a Database Trigger in the Worksheet Window

The second option for creating a trigger is to right-click on Triggers and choose the New Trigger option, as shown in [Figure 13.2](#). This activates the Create Trigger window, as shown in [Figure 13.3](#). In this window, you provide schema name, trigger name, table name, the timing of the triggering event, and the event on which the trigger should fire.

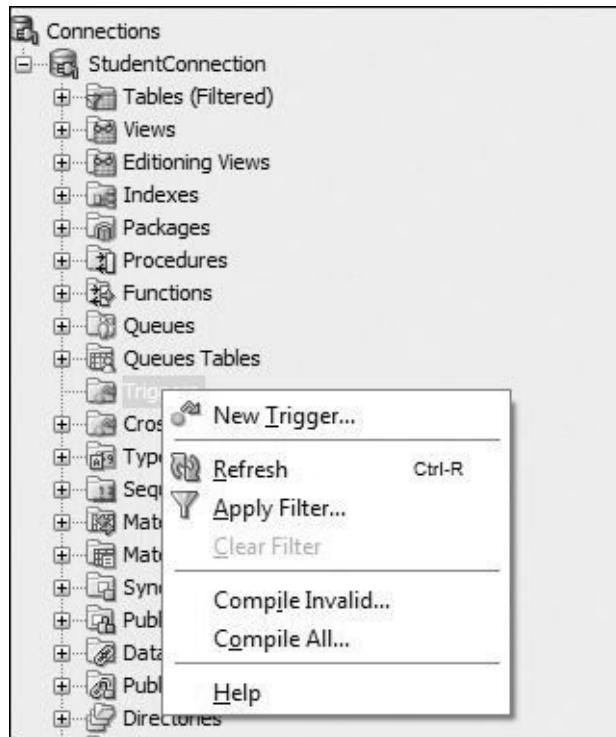


Figure 13.2 Creating a Database Trigger via New Trigger Option

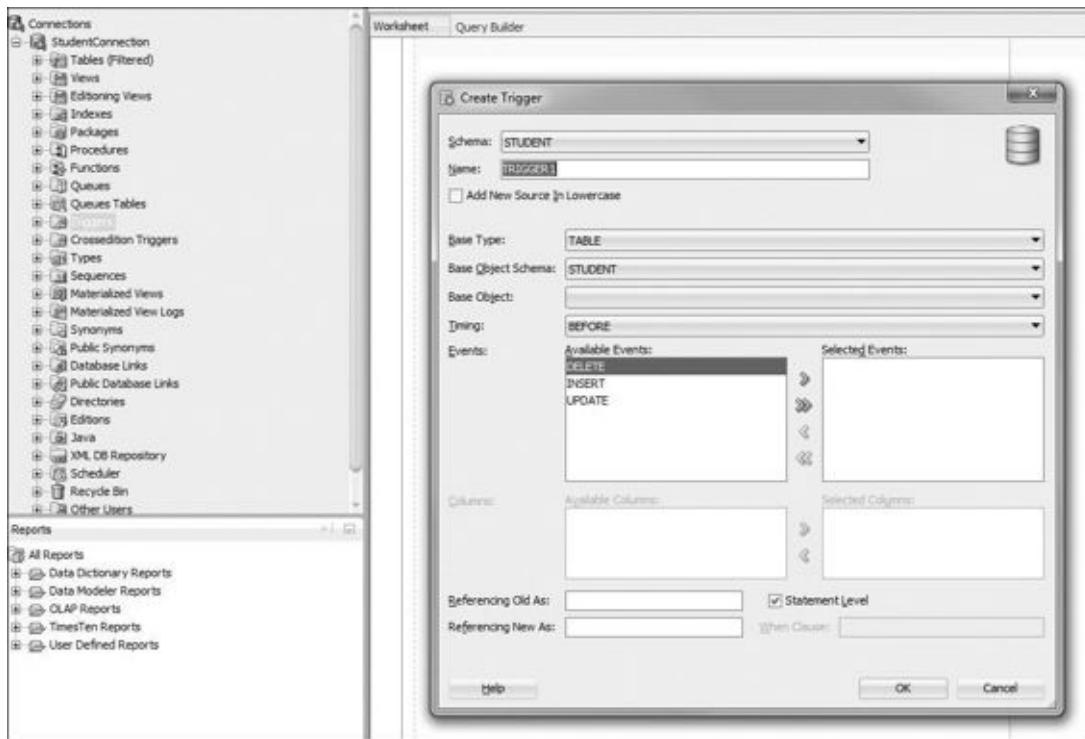


Figure 13.3 Create Trigger Window

Note that the schema name has already been set to STUDENT, and a default name for the trigger has been provided, TRIGGER1, that should be changed to STUDENT_BI. In addition, the Base Type has been set to a TABLE and Base Object Schema has been set to STUDENT. This implies that a trigger is being created on a table in the STUDENT schema. Next, the Base Object must be selected from the drop-down menu—in this case, it is STUDENT table. Under the Events option, the INSERT option is moved from the Available Events to Selected Events. By default, the Statement Level check box is enabled. Because you are creating a row-level trigger, this option should be unchecked. Finally, there is an option to provide different names for the :NEW and :OLD pseudorecords and one or more conditions for the WHEN clause. After you fill in the Create Trigger window for the STUDENT_BI trigger, it should contain the information shown in [Figure 13.4](#).

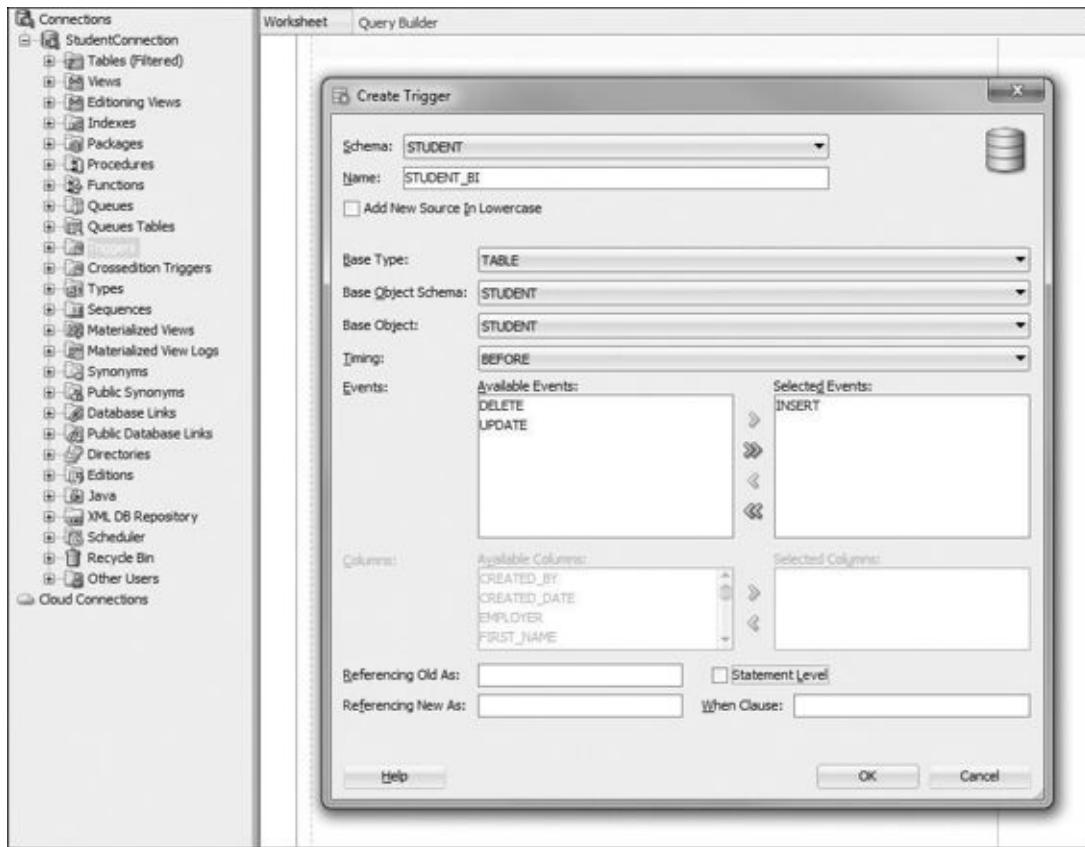


Figure 13.4 Creating a STUDENT_BI Trigger in the Trigger Window

Once all of the required information has been provided in the Create Trigger window, the trigger is created, as shown in [Figure 13.5](#). Notice that information provided in the Create Trigger window was used to create the trigger header. The trigger body contains a single statement, NULL.

```

CREATE OR REPLACE TRIGGER STUDENT_BI
BEFORE INSERT ON STUDENT
FOR EACH ROW
BEGIN
    NULL;
END;

```

Figure 13.5 Newly Created STUDENT_BI Trigger

Next, you need to provide the executable statements for the body of the trigger and compile the trigger. To do so, you click the Compile button, as shown in [Figure 13.6](#).

The screenshot shows the Oracle SQL Developer interface. In the top left, there's a tab labeled 'StudentConnection'. To its right is another tab labeled 'STUDENT_BI'. Below the tabs, a menu bar has items: 'Code', 'Details', 'Errors', 'Dependencies', 'Profiles', 'Grants', and 'References'. A toolbar below the menu contains icons for 'Find', 'Search', 'Run', 'Stop', 'Refresh', and 'Edit'. On the far right of the toolbar is a dropdown menu. The main area displays the PL/SQL code for the 'STUDENT_BI' trigger:

```
CREATE OR REPLACE TRIGGER STUDENT_BI
BEFORE INSERT ON STUDENT
FOR EACH ROW
BEGIN
:NEW.student_id    := STUDENT_ID_SEQ.NEXTVAL;
:NEW.created_by   := USER;
:NEW.created_date := SYSDATE;
:NEW.modified_by  := USER;
:NEW.modified_date:= SYSDATE;
END;
```

In the top right corner of the code editor, there is a button labeled 'Compile (compile with or without debug info)'.

Figure 13.6 Compiling the STUDENT_BI Trigger

Now the trigger STUDENT_BI has been created on the STUDENT table. Please note that going forward all triggers in this chapter and [Chapter 14](#) are created by using the Worksheet window rather than the Create Trigger window.

Now that you have created a trigger on the STUDENT table, consider the following INSERT statement.

For Example *INSERT Statement on the STUDENT Table*

[Click here to view code image](#)

```
INSERT INTO STUDENT
(student_id, first_name, last_name, zip, registration_date,
created_by, created_date, modified_by, modified_date)
VALUES
(STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '00914', SYSDATE,
USER, SYSDATE, USER, SYSDATE);
```

This INSERT statement contains values for the columns STUDENT_ID, CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE. Note that for every row you insert into the STUDENT table, you must provide the values for these columns; they are always derived in the same fashion. Now that you have created the trigger, however, there is no need to provide values for these columns in the INSERT statement because the trigger automatically populates these columns in a consistent manner every time an INSERT statement is executed against the STUDENT table. Therefore, the INSERT statement can be modified as follows:

For Example *Modified INSERT Statement on the STUDENT Table*

[Click here to view code image](#)

```
INSERT INTO STUDENT
(first_name, last_name, zip, registration_date)
VALUES
('John', 'Smith', '00914', SYSDATE);
```

This version of the INSERT statement is significantly shorter than the previous version. Specifically, instead of providing values for nine columns, you need to provide values for only four columns. The columns STUDENT_ID, CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE are no longer present.

You should use BEFORE triggers in the following situations:

- When a trigger provides values for derived columns before an INSERT or UPDATE statement is completed. For example, the trigger can provide audit-oriented columns such as CREATED_DATE and MODIFIED_DATE.
- When a trigger determines whether an INSERT, UPDATE, or DELETE statement should be allowed to complete. For example, when you insert a record into the INSTRUCTOR table, a trigger can verify whether the value provided for the column ZIP is valid—in other words, whether there is a record in the ZIPCODE table corresponding to the value of zip that you provided.

AFTER Triggers

Assume there is a table called AUDIT_TRAIL having the structure shown in [Figure 13.7](#). This table is used to collect user access information on different tables in the STUDENT schema. For example, you can record who deleted records from the INSTRUCTOR table and when they were deleted.

COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1 TABLE_NAME	VARCHAR2(30 BYTE)	Yes	(null)	1 (null)	
2 TRANSACTION_NAME	VARCHAR2(10 BYTE)	Yes	(null)	2 (null)	
3 TRANSACTION_USER	VARCHAR2(30 BYTE)	Yes	(null)	3 (null)	
4 TRANSACTION_DATE	DATE	Yes	(null)	4 (null)	

Figure 13.7 AUDIT_TRAIL Table Structure

To accomplish this, you would need to create a trigger on the INSTRUCTOR table, as shown in the following example.

For Example ch13_2a.sql

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER instructor_aud
AFTER UPDATE OR DELETE ON INSTRUCTOR
DECLARE
    v_trans_type VARCHAR2(10);
BEGIN
    v_trans_type := CASE
        WHEN UPDATING THEN 'UPDATE'
        WHEN DELETING THEN 'DELETE'
    END;
    INSERT INTO audit_trail
        (TABLE_NAME, TRANSACTION_NAME, TRANSACTION_USER, TRANSACTION_DATE)
    VALUES
        ('INSTRUCTOR', v_trans_type, USER, SYSDATE);
END;
```

This trigger fires after an UPDATE or DELETE statement is issued on the INSTRUCTOR table. The body of the trigger contains two Boolean functions, UPDATING

and **DELETING**. The function **UPDATING** evaluates to TRUE if an UPDATE statement is issued on the table, and the function **DELETING** evaluates to TRUE if a DELETE statement is issued on the table. Another Boolean function, **INSERTING**, also evaluates to TRUE when an **INSERT** statement is issued against the table.

This trigger inserts a record into the **AUDIT_TRAIL** table when an UPDATE or DELETE operation is issued against the **INSTRUCTOR** table. First, it determines which operation was issued against the **INSTRUCTOR** table via the CASE statement. The result of this evaluation is then assigned to the **v_trans_type** variable. Next, the trigger adds a new record to the **AUDIT_TRAIL** table.

Once this trigger is created on the **INSTRUCTOR** table, any UPDATE or DELETE operation causes the creation of new records in the **AUDIT_TRAIL** table. Furthermore, this trigger may be enhanced by calculating how many rows were updated or deleted from the **INSTRUCTOR** table.

You should use **AFTER** triggers in the following situations:

- When a trigger should fire after a DML statement is executed
- When a trigger performs actions not specified in a **BEFORE** trigger

Autonomous Transaction

As stated previously, when a trigger fires, all operations performed by the trigger become part of a transaction. When this transaction is committed or rolled back, the operations performed by the trigger are committed or rolled back as well. Consider an UPDATE statement against the **INSTRUCTOR** table as shown in [Listing 13.5](#).

Listing 13.5 UPDATE on the INSTRUCTOR Table

```
UPDATE instructor
   SET phone = '7181234567'
 WHERE instructor_id = 101;
```

When this UPDATE statement is executed, the **INSTRUCTOR_AUD** trigger fires and adds a single record to the **AUDIT_TRAIL** table as shown in [Listing 13.6](#).

Listing 13.6 SELECT from the AUDIT_TRAIL Table

[Click here to view code image](#)

```
SELECT *
  FROM audit_trail;

TABLE_NAME  TRANSACTION_NAME  TRANSACTION_USER  TRANSACTION_DATE
-----  -----  -----
INSTRUCTOR    UPDATE          STUDENT        05/07/2014
```

Next, consider rolling back the UPDATE statement just issued. In this case, the record inserted in the **AUDIT_TRAIL** table is rolled back as well, as shown in the [Listing 13.7](#).

Listing 13.7 Rolling Back UPDATE on the INSTRUCTOR Table

[Click here to view code image](#)

```
ROLLBACK;
```

```

SELECT *
  FROM audit_trail;

TABLE_NAME  TRANSACTION_NAME  TRANSACTION_USER  TRANSACTION_DATE
-----  -----  -----  -----

```

As you can see, the AUDIT_TRAIL table no longer contains any records. To circumvent such behavior, you may choose to employ autonomous transactions.

An autonomous transaction is an independent transaction started by another transaction that is usually referred to as the main transaction. In other words, an autonomous transaction may issue various DML statements and commit or roll them back, without committing or rolling back the DML statements issued by the main transaction.

To define an autonomous transaction, you employ the AUTONOMOUS_TRANSACTION pragma. You have already encountered one pragma, EXCEPTION_INIT, in [Chapter 10](#). Recall that a pragma is a special instruction to the PL/SQL compiler that is processed at the time of the compilation. The AUTONOMOUS_TRANSACTION pragma appears in the declaration section of a block, as shown in [Listing 13.8](#).

Listing 13.8 AUTONOMOUS_TRANSACTION Pragma

[Click here to view code image](#)

```

DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;

```

Now consider a modified version of the INSTRUCTOR_AUD trigger that includes an the autonomous transaction. Newly added statements are shown in bold.

For Example ch13_2b.sql

[Click here to view code image](#)

```

CREATE OR REPLACE TRIGGER instructor_aud
AFTER UPDATE OR DELETE ON INSTRUCTOR
DECLARE
  v_trans_type VARCHAR2(10);
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  v_trans_type := CASE
    WHEN UPDATING THEN 'UPDATE'
    WHEN DELETING THEN 'DELETE'
  END;
  INSERT INTO audit_trail
    (TABLE_NAME, TRANSACTION_NAME, TRANSACTION_USER, TRANSACTION_DATE)
  VALUES
    ('INSTRUCTOR', v_trans_type, USER, SYSDATE);
  COMMIT;
END;

```

In this version of the trigger, you added the AUTONOMOUS_TRANSACTION pragma to the declaration portion and the COMMIT statement to the executable portion of the trigger.

Now try issuing the UPDATE statement as in [Listing 13.5](#) and then rolling it back and querying the AUDIT_TRAIL table. Even though the changes on the INSTRUCTOR table were rolled back, the AUDIT_TRAIL table will continue to contain a record of the

attempted UPDATE operation.

Lab 13.2: Types of Triggers

After this lab, you will be able to

- [Use Row and Statement Triggers](#)
- [Use INSTEAD OF Triggers](#)

Row and Statement Triggers

In [Lab 13.1](#), you encountered the term *row trigger*. A row trigger is fired as many times as there are rows affected by the triggering statement. When the statement `FOR EACH ROW` is present in the `CREATE TRIGGER` clause, the trigger is a row trigger. Consider the code fragment shown in [Listing 13.9](#).

[Listing 13.9](#) Code Fragment of the COURSE_AU Trigger

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
...
```

In this code fragment, the statement `FOR EACH ROW` is present in the `CREATE TRIGGER` clause. Therefore, this trigger is a row trigger. Thus, if an `UPDATE` statement causes 20 records in the `COURSE` table to be modified, this trigger will fire 20 times.

A statement trigger is fired once for the triggering statement. In other words, a statement trigger fires once, regardless of the number of rows affected by the triggering statement. To create a statement trigger, you omit the `FOR EACH ROW` statement in the `CREATE TRIGGER` clause, as shown in [Listing 13.10](#).

[Listing 13.10](#) Code Fragment of the ENROLLMENT_AD Trigger

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER enrollment_ad
AFTER DELETE ON ENROLLMENT
...
```

This particular trigger fires once after a `DELETE` statement is issued against the `ENROLLMENT` table. Whether the `DELETE` statement removes one row, five rows, or 500 rows from the `ENROLLMENT` table, this trigger fires only once.

Statement triggers should be used when the operations performed by the trigger do not depend on the data in the individual records. For example, if you want to limit access to a table to business hours only, you might use a statement trigger. Consider the following example.

For Example ch13_3a.sql

[Click here to view code image](#)

```

CREATE OR REPLACE TRIGGER instructor_biud
BEFORE INSERT OR UPDATE OR DELETE ON INSTRUCTOR
DECLARE
    v_day VARCHAR2(10);
BEGIN
    v_day := RTRIM(TO_CHAR(SYSDATE, 'DAY'));

    IF v_day LIKE ('S%')
    THEN
        RAISE_APPLICATION_ERROR (-20000, 'A table cannot be modified during
off hours');
    END IF;
END;

```

This statement trigger on the **INSTRUCTOR** table fires before an **INSERT**, **UPDATE**, or **DELETE** statement is issued. First, the trigger determines the day of the week. If the day happens to be Saturday or Sunday, an error message is generated. For example, if the following **UPDATE** statement on the **INSTRUCTOR** table is issued on Saturday or Sunday

```

UPDATE instructor
    SET zip = 10025
  WHERE zip = 10015;

```

the trigger generates this error message:

[Click here to view code image](#)

```

update INSTRUCTOR
*
ERROR at line 1:
ORA-20000: A table cannot be modified during off hours
ORA-06512: at "STUDENT.INSTRUCTOR_BIUD", line 8
ORA-04088: error during execution of trigger 'STUDENT.INSTRUCTOR_BIUD'

```

Notice that this trigger checks for a specific day of the week, but it does not check the time of day. You can create a more sophisticated trigger that checks which day of the week it is and whether the current time is between 9:00 A.M. and 5:00 P.M. If the day falls in the business week and the time of the day is not between 9:00 A.M. and 5:00 P.M., the error is generated.

INSTEAD OF Triggers

So far, you have seen triggers that are defined on database tables. PL/SQL provides another kind of trigger that is defined on database views. A view is a custom representation of data that can be referred to as a “stored query.” Consider the following example of a view created against the **COURSE** table:

For Example *ch13_4a.sql*

[Click here to view code image](#)

```

CREATE VIEW course_cost
AS
    SELECT course_no, description, cost
      FROM course;

```

Watch Out!

You may find that you do not have privileges to create a view when logged in as STUDENT. In such a case, you need to log in as SYS and grant a CREATE VIEW privilege as follows:

```
GRANT CREATE VIEW TO student;
```

Did You Know?

Once a view is created, it does not contain or store any data. The data is derived from the SELECT statement associated with the view. In the preceding example, the COURSE_COST view contains three columns that are selected from the COURSE table.

Similar to tables, views can be manipulated via INSERT, UPDATE, and DELETE statements, with some restrictions. Be aware that when any of these statements are issued against a view, the corresponding data is modified in the underlying table. For example, consider an UPDATE statement against the COURSE_COST view.

For Example ch13_5a.sql

```
UPDATE course_cost
   SET cost = 2000
 WHERE course_no = 450;
```

Once this UPDATE statement is executed, both SELECT statements against the COURSE_COST view and SELECT statements against the COURSE table return the same value of the cost for course number 450, as shown in [Listing 13.11](#).

Listing 13.11 Selecting Data from the COURSE_COST View and the COURSE Table

[Click here to view code image](#)

```
SELECT *
  FROM course_cost
 WHERE course_no = 450;

COURSE_NO      DESCRIPTION          COST
-----  -----
450           DB Programming in Java    2000
SELECT course_no, cost
  FROM course
 WHERE course_no = 450;

COURSE_NO      COST
-----  -----
450           2000
```

As mentioned earlier, some restrictions are placed on the views in terms of whether they can be modified by INSERT, UPDATE, and DELETE statements. Specifically, these restrictions apply to the underlying SELECT statement, which is also referred to as a “view query.” Thus, if a view query performs any of the operations or contains any of the following constructs, a view cannot be modified by an UPDATE, INSERT, and DELETE

statements:

- Set operations such as UNION, UNION ALL, INTERSECT, and MINUS
- Group functions such as AVG, COUNT, MAX, MIN, and SUM
- GROUP BY or HAVING clauses
- CONNECT BY or START WITH clauses
- The DISTINCT operator
- The ROWNUM pseudocolumn

For example, consider a view created against the INSTRUCTOR and SECTION tables that summarizes how many courses are taught by an instructor.

For Example *ch13_6a.sql*

[Click here to view code image](#)

```
CREATE VIEW instructor_summary_view
AS
  SELECT i.instructor_id, COUNT(s.section_id) total_courses
    FROM instructor i
    LEFT OUTER JOIN section s
      ON (i.instructor_id = s.instructor_id)
   GROUP BY i.instructor_id;
```

Note that the SELECT statement is written in the ANSI 1999 SQL standard. It uses the outer join between the INSTRUCTOR and SECTION tables. The LEFT OUTER JOIN indicates that an instructor record in the INSTRUCTOR table that does not have a corresponding record in the SECTION table is included in the result set, with TOTAL_COURSES being equal to zero in this result.

Did You Know?

Detailed explanations and examples of statements using the new ANSI 1999 SQL standard may be found in Oracle's online help. Throughout this book we try to provide you with examples illustrating both standards; however, our main focus is on PL/SQL features rather than SQL.

This view is not updatable because it contains the group function, COUNT(). As a result, the DELETE statement

[Click here to view code image](#)

```
DELETE FROM instructor_summary_view
 WHERE instructor_id = 109;
```

causes the following error:

[Click here to view code image](#)

```
ORA-01732: data manipulation operation not legal on this view
01732. 00000 - "data manipulation operation not legal on this view"
```

Recall that PL/SQL provides a special kind of trigger that can be defined on database views. This INSTEAD OF trigger is created as a row trigger. An INSTEAD OF trigger

fires instead of the triggering statement (**INSERT**, **UPDATE**, **DELETE**) that has been issued against a view and directly modifies the underlying tables.

Consider an **INSTEAD OF** trigger defined on the **INSTRUCTOR_SUMMARY_VIEW**. This trigger deletes a record from the **INSTRUCTOR** table for the corresponding value of the instructor's ID.

For Example *ch13_7a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER instructor_summary_del
INSTEAD OF DELETE ON instructor_summary_view
FOR EACH ROW
BEGIN
    DELETE FROM instructor
    WHERE instructor_id = :OLD.INSTRUCTOR_ID;
END;
```

Note the usage of the **INSTEAD OF** clause in the trigger header. Once the trigger is created, the **DELETE** statement against the **INSTRUCTOR_SUMMARY_VIEW** does not generate any errors.

[Click here to view code image](#)

```
DELETE FROM instructor_summary_view
WHERE instructor_id = 109;

1 row deleted.
```

When this **DELETE** statement is issued, the trigger deletes a record from the **INSTRUCTOR** table corresponding to the specified value of **INSTRUCTOR_ID**.

Now consider the same **DELETE** statement with a different instructor ID:

[Click here to view code image](#)

```
DELETE FROM instructor_summary_view
WHERE instructor_id = 101;
```

When this **DELETE** statement is issued, it causes the following error:

[Click here to view code image](#)

```
ORA-02292: integrity constraint (STUDENT.SECT_INST_FK) violated - child
record found
ORA-06512: at "STUDENT.INSTRUCTOR_SUMMARY_DEL", line 2
ORA-04088: error during execution of trigger
'STUDENT.INSTRUCTOR_SUMMARY_DEL'
```

The **INSTRUCTOR_SUMMARY_VIEW** joins the **INSTRUCTOR** and **SECTION** tables based on the **INSTRUCTOR_ID** column that is present in both tables. The **INSTRUCTOR_ID** column in the **INSTRUCTOR** table has a primary key constraint defined on it. The **INSTRUCTOR_ID** column in the **SECTION** table has a foreign key constraint that references the **INSTRUCTOR_ID** column of the **INSTRUCTOR** table. Thus, the **SECTION** table is considered a child table of the **INSTRUCTOR** table.

The original **DELETE** statement does not cause any errors because there is no record in the **SECTION** table corresponding to the instructor ID of 109. In other words, the instructor with an ID of 109 does not teach any courses.

The second **DELETE** statement causes an error because the **INSTEAD OF** trigger tries to delete a record from the **INSTRUCTOR** table, the parent table. However, there is a corresponding record in the **SECTION** table, the child table, with an instructor ID of 101. This causes an integrity constraint violation error. It might seem that one more **DELETE** statement (highlighted in bold in the following example) should be added to the **INSTEAD OF** trigger.

For Example *ch13_7b.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER instructor_summary_del
INSTEAD OF DELETE ON instructor_summary_view
FOR EACH ROW
BEGIN
    DELETE FROM section
    WHERE instructor_id = :OLD.INSTRUCTOR_ID;
    DELETE FROM instructor
    WHERE instructor_id = :OLD.INSTRUCTOR_ID;
END;
```

Notice that the newly added **DELETE** statement removes records from the **SECTION** table before the **INSTRUCTOR** table because the **SECTION** table contains child records of the **INSTRUCTOR** table. However, the **DELETE** statement against the **INSTRUCTOR_SUMMARY_VIEW** causes yet another error:

[Click here to view code image](#)

```
DELETE FROM instructor_summary_view
WHERE instructor_id = 101;

ORA-02292: integrity constraint (STUDENT.GRTW_SECT_FK) violated - child
record found
ORA-06512: at "STUDENT.INSTRUCTOR_SUMMARY_DEL", line 2
ORA-04088: error during execution of trigger
'STUDENT.INSTRUCTOR_SUMMARY_DEL'
```

This time, the error refers to a different foreign key constraint that specifies the relationship between the **SECTION** and **GRADE_TYPE_WEIGHT** tables. In this case, the child records are found in the **GRADE_TYPE_WEIGHT** table. Thus, before deleting records from the **SECTION** table, the trigger must delete all corresponding records from the **GRADE_TYPE_WEIGHT** table. However, the **GRADE_TYPE_WEIGHT** table has child records in the **GRADE** table, so the trigger must delete records from the **GRADE** table first.

This example illustrates the complexity of designing an **INSTEAD OF** trigger. To ensure that such a trigger works as intended, you must be aware of two important factors: the relationships among tables in the database and the ripple effect that a particular design may introduce. This example suggests deleting records from four underlying tables. However, those tables contain information that relates not only to the instructors and the sections they teach, but also to the students and the sections in which they are enrolled.

Summary

In this chapter, you began learning about database triggers, including what they are, how they fire, which types of triggers are available, and how they may be used. You also learned how to define and employ autonomous transactions. In [Chapter 14](#), you will learn about compound triggers and their usage.

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.

14. Mutating Tables and Compound Triggers

In this Chapter, you will learn about

- [Mutating Tables](#)
- [Compound Triggers](#)

In [Chapter 13](#), you explored the concept of triggers. You learned about usage of triggers in the database, events that cause triggers to fire, and different types of triggers. In this chapter, you will continue exploring triggers. You will learn about mutating table issues and discover how triggers can be used to resolve these issues.

[Lab 14.1](#) describes mutating tables and explains how to resolve issues associated with them in Oracle database prior to the version 11g. [Lab 14.2](#) covers compound triggers, which were introduced in Oracle 11g, and discusses how they can be used to resolve mutating table issues.

Lab 14.1: Mutating Tables

After this lab, you will be able to

- [Understand Mutating Tables](#)
- [Resolve Mutating Tables Issues](#)

What Is a Mutating Table?

A table against which a DML statement is issued is called a mutating table. For a trigger, the mutating table is the one on which the trigger is defined. If a trigger tries to read or modify such a table, it causes a mutating table error. As a result, a SQL statement issued in the body of the trigger may not read or modify a mutating table. Note that this restriction applies to row-level triggers.

Watch Out!

A mutating table error is a runtime error. In other words, this error occurs not at the time of trigger creation (compilation), but rather when the trigger fires.

Consider the following example of a trigger causing a mutating table error.

For Example *ch14_1a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
DECLARE
    v_total NUMBER;
```

```

v_name  VARCHAR2(30);
BEGIN
  SELECT COUNT(*)
    INTO v_total
   FROM section - SECTION is MUTATING
  WHERE instructor_id = :NEW.instructor_id;

  -- check if the current instructor is overbooked
  IF v_total >= 10
  THEN
    SELECT first_name||' '||last_name
      INTO v_name
     FROM instructor
    WHERE instructor_id = :NEW.instructor_id;

    RAISE_APPLICATION_ERROR (-20000, 'Instructor, ''||v_name||', is
overbooked');
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RAISE_APPLICATION_ERROR (-20001, 'This is not a valid instructor');
END;

```

This trigger fires before an **INSERT** or **UPDATE** statement is issued on the **SECTION** table. The trigger checks whether the specified instructor is teaching too many sections. If the number of sections taught by an instructor is equal to or greater than 10, the trigger issues an error message stating that this instructor is teaching too many sections.

Now, consider the following **UPDATE** statement issued against the **SECTION** table:

```

UPDATE section
  SET instructor_id = 101
 WHERE section_id = 80;

```

When this **UPDATE** statement is issued against the **SECTION** table, the following error message is displayed:

[Click here to view code image](#)

```

ORA-04091: table STUDENT.SECTION is mutating, trigger/function may not see
it
ORA-06512: at "STUDENT.SECTION_BIU", line 5
ORA-04088: error during execution of trigger 'STUDENT.SECTION_BIU'

```

Notice that the error message states that the **SECTION** table is mutating and the trigger may not see it. This error message is generated because there is a **SELECT INTO** statement,

[Click here to view code image](#)

```

SELECT COUNT(*)
  INTO v_total
   FROM section
  WHERE instructor_id = :NEW.INSTRUCTOR_ID;

```

issued against the **SECTION** table that is being modified and, therefore, is mutating.

Resolving Mutating Table Issues

To correct mutating table error described earlier, the following steps must be taken when using an Oracle version prior to 11g:

1. To record the instructor's ID and name as described in the preceding example, two global variables must be declared with the help of a PL/SQL package. You will learn about global variables and packages in [Chapter 21](#).
 2. An existing trigger must be modified so that it records the instructor's ID, queries the INSTRUCTOR table, and records the instructor's name.
 3. A new trigger must be created on the SECTION table. This trigger should be a statement-level trigger that fires after the INSERT or UPDATE statement has been issued. It will check the number of courses that are taught by a particular instructor and will raise an error if that number is equal to or greater than 10.
-

Did You Know?

These steps are used to resolve mutating table errors in versions of Oracle prior to 11g. Starting with Oracle 11g, compound triggers are used to resolve this error. Compound triggers are covered in [Lab 14.2](#).

Consider the package specification shown in [Listing 14.1](#).

Listing 14.1 INSTRUCTOR_ADM Package Specification

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE instructor_adm
AS
    g_instructor_id    instructor.instructor_id%TYPE;
    g_instructor_name  varchar2(50);
END;
```

This package specification contains declarations for two global variables, `g_instructor_id` and `g_instructor_name`. Note that the `CREATE OR REPLACE` clause is similar to the clause used for a trigger. (Packages are covered in detail in [Chapter 21](#).)

Next, the existing trigger SECTION_BIU is modified as follows:

For Example ch14_1b.sql

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
BEGIN
    IF :NEW.instructor_id IS NOT NULL
    THEN
        BEGIN
            -- Assign new instructor ID to the global variable
            instructor_adm.g_instructor_id := :NEW.INSTRUCTOR_ID;

            SELECT first_name||' '||last_name
```

```

        INTO instructor_adm.g_instructor_name
        FROM instructor
        WHERE instructor_id = instructor_adm.g_instructor_id;

EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        RAISE_APPLICATION_ERROR (-20001, 'This is not a valid
instructor');
    END;
END IF;
END;

```

In this version of the trigger, the global variables `g_instructor_id` and `g_instructor_name` are initialized if the incoming value of the instructor's ID is not null. Notice that the variable names are prefixed by the package name—a convention called dot notation.

Finally, a new statement-level trigger is created on the `SECTION` table:

For Example *ch14_2a.sql*

[Click here to view code image](#)

```

CREATE OR REPLACE TRIGGER section_aiu
AFTER INSERT OR UPDATE ON section
DECLARE
    v_total INTEGER;
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM section
        WHERE instructor_id = instructor_adm.g_instructor_id;
    -- check if the current instructor is overbooked
    IF v_total >= 10
    THEN
        RAISE_APPLICATION_ERROR
            (-20000, 'Instructor, '||instructor_adm.g_instructor_name||', is
overbooked');
    END IF;
END;

```

This trigger fires after an `INSERT` or `UPDATE` statement is issued against the `SECTION` table. Because this is a statement-level trigger, the `FOR EACH ROW` clause is omitted from the trigger header. This trigger checks the number of courses that are taught by a particular instructor and raises an error if that number is equal to or greater than 10. This is accomplished with the help of two global variables, `g_instructor_id` and `g_instructor_name`. As mentioned earlier, these variables are populated by the `SECTION_BIU` trigger that fires before an `INSERT` or `UPDATE` statement is issued against the `SECTION` table.

As a result, the `UPDATE` statement used earlier

```

UPDATE section
    SET instructor_id = 101
    WHERE section_id = 80;

```

produces ORA-20000 error as expected

[Click here to view code image](#)

```
ORA-20000: Instructor, Fernand Hanks, is overbooked
ORA-06512: at "STUDENT.SECTION_AIU", line 12
ORA-04088: error during execution of trigger 'STUDENT.SECTION_AIU'
```

This error is generated by the trigger SECTION_AIU and does not contain any message about a mutating table.

Now consider a similar UPDATE statement for a different instructor ID that does not cause any errors:

```
UPDATE section
  SET instructor_id = 110
 WHERE section_id = 80;

1 row updated.
```

Lab 14.2: Compound Triggers

After this lab, you will be able to

- [Define a Compound Trigger](#)
- [Use Compound Triggers to Resolve Mutating Table Issues](#)

What Is a Compound Trigger?

A compound trigger allows you to combine different types of triggers into one trigger. Specifically, you are able to combine

- A statement trigger that fires before the firing statement
- A row trigger that fires before each row that the firing statement affects
- A row trigger that fires after each row that the firing statement affects
- A statement trigger that fires after the firing statement

For example, you can create a compound trigger on the STUDENT table with portions of code that would fire once before the insert, before the insert for each affected row, after the insert for each affected row, and once after the insert.

The structure of a compound trigger is shown in [Listing 14.2](#).

Listing 14.2 General Syntax for Creating a Compound Trigger

[Click here to view code image](#)

```
CREATE [OR REPLACE] TRIGGER trigger_name
triggering_event ON table_name
COMPOUND TRIGGER

  Declaration Statements

  BEFORE STATEMENT IS
  BEGIN
    Executable statements
```

```
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
  Executable statements
END BEFORE EACH ROW;

AFTER EACH ROW IS
BEGIN
  Executable statements
END AFTER EACH ROW;

AFTER STATEMENT IS
BEGIN
  Executable statements
END AFTER STATEMENT;

END;
```

First you specify the trigger header that includes the CREATE OR REPLACE clause, the triggering event, the table name for which the trigger is defined, and the COMPOUND TRIGGER clause that denotes that this is a compound trigger. Note the omission of the BEFORE or AFTER clause in the header of the compound trigger.

Next, you specify a declaration section that is common to all executable sections. In other words, any variable declared in this section can be referenced in any of the executable sections.

Finally, you specify the executable sections that fire at different timing points. Each of these sections is optional. Thus, if no action takes place after the firing statement, there is no AFTER STATEMENT section.

Watch Out!

Compound triggers have several restrictions:

- A compound trigger may be defined on a table or a view only.
- A triggering event of a compound trigger is limited to the DML statements.
- A compound trigger may not contain an autonomous transaction. In other words, its declaration portion cannot include `PRAGMA AUTOTONOMOUS_TRANSACTION`.
- An exception that occurs in one executable section must be handled within that section. For example, if an exception occurs in the `AFTER EACH ROW` section, it cannot propagate to the `AFTER STATEMENT` section; rather, it must be handled in the `AFTER EACH ROW` section.
- References to `:OLD` and `:NEW` pseudocolumns cannot appear in the declaration, `BEFORE STATEMENT`, and `AFTER STATEMENT` sections.
- The value of the `:NEW` pseudocolumn can be changed in the `BEFORE EACH ROW` section only.
- The firing order of the compound and simple triggers is not guaranteed. In other words, the firing of the compound trigger may interleave with the firing of the simple triggers.
- If a DML statement issued on a table that has a compound trigger defined on it fails (rolls back) due to an exception:
 - Variables declared in the compound trigger sections are reinitialized. In other words, any values assigned those variable are lost.
 - DML statements issued by the compound trigger are not rolled back.

Consider the following example of the compound trigger on the `STUDENT` table that has `BEFORE STATEMENT` and `BEFORE EACH ROW` sections only.

For Example `ch14_3a.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER student_compound
FOR INSERT ON STUDENT
COMPOUND TRIGGER

    -- Declaration section
    v_day  VARCHAR2(10);

    BEFORE STATEMENT IS
    BEGIN
        v_day := RTRIM(TO_CHAR(SYSDATE, 'DAY'));
        IF v_day LIKE ('S%')
        THEN
            RAISE_APPLICATION_ERROR
                (-20000, 'A table cannot be modified during off hours');
```

```

    END IF;
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
  :NEW.student_id      := STUDENT_ID_SEQ.NEXTVAL;
  :NEW.created_by     := USER;
  :NEW.created_date   := SYSDATE;
  :NEW.modified_by    := USER;
  :NEW.modified_date  := SYSDATE;
END BEFORE EACH ROW;

END;

```

This trigger has a declaration section and two executable sections only. Each of the executable sections is optional and is specified only because there is an action associated with it.

First, the declaration section contains the declaration of a single variable used in the BEFORE STATEMENT section. Second, the BEFORE STATEMENT section initializes the variable and contains an IF statement that prevents modification of the STUDENT table during off hours. This section fires once before an INSERT statement. Next, the BEFORE EACH ROW section initializes some of the columns of the STUDENT table to their default values.

Note that all references to the :NEW pseudorecord are placed in the BEFORE EACH ROW section of the trigger, as this section is available in the row-level section only. In fact, if you attempt to assign values to any of the members of the :NEW pseudorecord in the BEFORE STATEMENT section, the trigger compiles with the error message similar to one shown here:

[Click here to view code image](#)

```

PLS-00363: expression 'NEW.CREATED_BY' cannot be used as an assignment
target
PLS-00679: trigger binds not allowed in before/after statement section
PL/SQL: Statement ignored

```

Resolving Mutating Table Issues with Compound Triggers

In [Lab 14.1](#), you learned about mutating table issues and saw how they can be resolved in Oracle versions prior to 11g. In this lab, you will learn how to resolve mutating table issues by means of compound triggers, which were introduced in Oracle 11g. Recall the example of the trigger on the SECTION table from [Lab 14.1](#) that caused a mutating table error and the steps you took to resolve this error, as shown in [Listing 14.3](#).

Listing 14.3 Preventing Mutating Table Issue Prior to Oracle 11g

[Click here to view code image](#)

```

CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
DECLARE
  v_total NUMBER;
  v_name  VARCHAR2(30);

```

```

BEGIN
  SELECT COUNT(*)
    INTO v_total
    FROM section -- SECTION is MUTATING
   WHERE instructor_id = :NEW.instructor_id;

  -- check if the current instructor is overbooked
  IF v_total >= 10
  THEN
    SELECT first_name||' '||last_name
      INTO v_name
      FROM instructor
     WHERE instructor_id = :NEW.instructor_id;

    RAISE_APPLICATION_ERROR (-20000, 'Instructor, '||v_name||', is
overbooked');
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RAISE_APPLICATION_ERROR
      (-20001, 'This is not a valid instructor');
END;

```

To correct this problem, you took the following steps:

- You created a package where you declared two global variables.

[Click here to view code image](#)

```

CREATE OR REPLACE PACKAGE instructor_adm
  AS g_instructor_id  instructor.instructor_id%TYPE;
  g_instructor_name varchar2(50);
END;

```

- You modified the existing trigger to record the instructor's ID and name.

[Click here to view code image](#)

```

CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
BEGIN
  IF :NEW.instructor_id IS NOT NULL
  THEN
    BEGIN
      instructor_adm.g_instructor_id := :NEW.INSTRUCTOR_ID;

      SELECT first_name||' '||last_name
        INTO instructor_adm.g_instructor_name
        FROM instructor
       WHERE instructor_id = instructor_adm.g_instructor_id;
    EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
        RAISE_APPLICATION_ERROR (-20001, 'This is not a valid
instructor');
      END;
    END IF;
END;

```

- You created a new statement trigger that fires after the **INSERT** or **UPDATE**

statement has been issued.

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER section_aiu
AFTER INSERT OR UPDATE ON section
DECLARE
    v_total INTEGER;
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM section
       WHERE instructor_id = instructor_adm.v_instructor_id;

    -- check if the current instructor is overbooked
    IF v_total >= 10 THEN
        RAISE_APPLICATION_ERROR
            (-20000, 'Instructor, '||instructor_adm.v_instructor_name|||
             ', is overbooked');
    END IF;
END;
```

Now consider a compound trigger on the SECTION table that fires with an **INSERT** or **UPDATE** operation.

For Example *ch14_4a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TRIGGER section_compound
FOR INSERT OR UPDATE ON SECTION
COMPOUND TRIGGER

    -- Declaration Section
    v_instructor_id    INSTRUCTOR.INSTRUCTOR_ID%TYPE;
    v_instructor_name  VARCHAR2(50);
    v_total            INTEGER;

BEFORE EACH ROW IS
BEGIN
    IF :NEW.instructor_id IS NOT NULL
    THEN
        BEGIN
            v_instructor_id := :NEW.instructor_id;

            SELECT first_name||' '||last_name
                INTO v_instructor_name
                FROM instructor
               WHERE instructor_id = v_instructor_id;

        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                RAISE_APPLICATION_ERROR
                    (-20001, 'This is not a valid instructor');
        END;
    END IF;
END BEFORE EACH ROW;

AFTER STATEMENT IS
BEGIN
    SELECT COUNT(*)
```

```

    INTO v_total
    FROM section
    WHERE instructor_id = v_instructor_id;

    -- check if the current instructor is overbooked
    IF v_total >= 10
    THEN
        RAISE_APPLICATION_ERROR
            (-20000, 'Instructor,'||v_instructor_name||', is overbooked');
    END IF;
END AFTER STATEMENT;

END;

```

In this trigger, you declare three variables, two of which were previously declared in the package. Next, you place statements from two individual triggers into two corresponding sections of a compound trigger.

By using this compound trigger, you were able to resolve a mutating table issue with a simpler approach. You eliminated the need for the package that was used as a link between two triggers that fired at different times in a transaction.

Note that the `UPDATE` statement used earlier

```

UPDATE section
    SET instructor_id = 101
    WHERE section_id = 80;

```

still causes an ORA-20000 error:

[Click here to view code image](#)

```

ORA-20000: Instructor, Fernand Hanks, is overbooked
ORA-06512: at "STUDENT.SECTION_COMPOUND", line 38
ORA-04088: error during execution of trigger 'STUDENT.SECTION_COMPOUND'

```

This error is generated by the trigger `SECTION_COMPOUND` and does not contain any message about a mutating table.

Summary

In [Chapter 13](#), you began exploring various types of triggers supported in PL/SQL. In this chapter, you continued this exploration and learned about mutating table issues. You learned how such issues were resolved in Oracle versions prior to 11g. Finally, you learned about compound triggers, which were introduced in Oracle 11g, and saw how these types of triggers can be used to resolve mutating table issues.

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.

15. Collections

In this chapter, you will learn about

- [PL/SQL Tables](#)
- [Varrays](#)
- [Multilevel Collections](#)

Throughout this book you have explored different types of PL/SQL identifiers or variables that represent individual elements (for example, a variable that represents a grade for a particular student). However, often in your programs you want to have the ability to represent a group of elements (for example, the grades for a class of students). To support this technique, PL/SQL provides collection data types that work just like arrays available in other third-generation programming languages.

A collection is a group of elements of the same data type. Each element is identified by a unique subscript that represents its position in the collection. In this chapter you will learn about two collection data types: tables and varrays. You will also learn about multilevel collections.

Lab 15.1: PL/SQL Tables

After this lab, you will be able to

- [Use Associative Arrays](#)
- [Use Nested Tables](#)
- [Use Collection Methods](#)

A PL/SQL table is similar to a one-column database table. The rows of a PL/SQL table are not stored in any predefined order, yet when they are retrieved in a variable, each row is assigned a consecutive subscript starting at 1, as shown in [Figure 15.1](#).

10	Number (1)
1	Number (2)
39	Number (3)
57	Number (4)
3	Number (5)

Figure 15.1 PL/SQL Table

[Figure 15.1](#) shows a PL/SQL table consisting of integer numbers. Each number is assigned a unique subscript that corresponds to its position in the table. For example, the number 3 has the subscript 5 assigned to it because it is stored in the fifth row of the PL/SQL table.

There are two types of PL/SQL tables: associative arrays (formerly known as index-by tables) and nested tables. They have the same structure, and their rows are accessed in the same way—that is, via subscript notation. The main difference between these two types is that nested tables can be stored in a database column, whereas associative arrays cannot.

Associative Arrays

The general syntax for creating an associative array is shown in [Listing 15.1](#) (the reserved words and phrases surrounded by brackets are optional).

Listing 15.1 Associative Array

[Click here to view code image](#)

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
  INDEX BY index_type;
table_name TYPE_NAME;
```

Notice that the declaration of an associative array requires two steps. First, a table structure is defined using the **TYPE** statement, where *type_name* is the name of the type that is used in the second step to declare an actual table. An *element_type* is a data type of the individual elements in the arrays. The **INDEX BY** clause specifies which data type is used for indexing the associative array; it can be either a string type (for example, **VARCHAR2**) or **PLS_INTEGER**.

Did You Know?

An index of an associative array may be populated with any data type as long as the TO_CHAR function can convert it to VARCHAR2.

Second, the actual array variable is declared based on the type specified in the previous step. Consider the code fragment shown in [Listing 15.2](#).

Listing 15.2 Declaration of an Associative Array

[Click here to view code image](#)

```
DECLARE
    TYPE last_name_type IS TABLE OF student.last_name%TYPE
        INDEX BY PLS_INTEGER;
    last_name_tab last_name_type;
```

In this code fragment, type `last_name_type` is declared based on the column `LAST_NAME` of the `STUDENT` table that is indexed by `PLS_INTEGER`. Next, the actual associative array called `last_name_tab` is declared to be of a `last_name_type`.

As mentioned earlier, the individual elements of an associative array are referenced via subscript notation as follows:

```
array_name(subscript)
```

This technique is demonstrated in the following example.

For Example ch15_1a.sql

[Click here to view code image](#)

```
DECLARE
    CURSOR name_cur IS
        SELECT last_name
        FROM student
        WHERE rownum < 10;
    TYPE last_name_type IS TABLE OF student.last_name%TYPE
        INDEX BY PLS_INTEGER;
    last_name_tab last_name_type;

    v_index PLS_INTEGER := 0;
BEGIN
    FOR name_rec IN name_cur
    LOOP
        v_index := v_index + 1;
        last_name_tab(v_index) := name_rec.last_name;
        DBMS_OUTPUT.PUT_LINE ('last_name('||v_index||'):
'||last_name_tab(v_index));
    END LOOP;
END;
```

In this example, the associative array `last_name_tab` is populated with last names from the `STUDENT` table. The variable `v_index` is used as a subscript to reference individual table elements. This example produces the following output:

```
last_name(1): Kocka
last_name(2): Jung
last_name(3): Mulroy
```

```
last_name(4): Brendler
last_name(5): Carcia
last_name(6): Tripp
last_name(7): Frost
last_name(8): Snow
last_name(9): Scrittore
```

Watch Out!

Referencing a nonexistent row of the associative array raises the **NO_DATA_FOUND** exception as follows:

[Click here to view code image](#)

```
DECLARE
    CURSOR name_cur IS
        SELECT last_name
        FROM student
        WHERE rownum < 10;

    TYPE last_name_type IS TABLE OF student.last_name%TYPE
        INDEX BY PLS_INTEGER;
    last_name_tab last_name_type;

    v_index PLS_INTEGER := 0;
BEGIN
    FOR name_rec IN name_cur
    LOOP
        v_index := v_index + 1;
        last_name_tab(v_index) := name_rec.last_name;
        DBMS_OUTPUT.PUT_LINE ('last_name('|| v_index ||'):
        '||last_name_tab(v_index));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('last_name(10): '||last_name_tab(10));
END;
```

This version of the script produces the following output:

```
last_name(1): Kocka
last_name(2): Jung
last_name(3): Mulroy
last_name(4): Brendler
last_name(5): Carcia
last_name(6): Tripp
last_name(7): Frost
last_name(8): Snow
last_name(9): Scrittore
ORA-01403: no data found
ORA-06512: at line 19
```

Notice that the **DBMS_OUTPUT.PUT_LINE** statement shown in bold raises the **NO_DATA_FOUND** exception because it references the tenth row of the associative array, even though the array contains only nine rows.

Nested Tables

The general syntax for creating a nested table is shown in [Listing 15.3](#) (the reserved words and phrases surrounded by brackets are optional).

Listing 15.3 Nested Table

[Click here to view code image](#)

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
table_name TYPE_NAME;
```

Notice that this declaration is very similar to the declaration of an associative array except that there is no INDEX BY clause.

Consider the code fragment shown in [Listing 15.4](#).

Listing 15.4 Declaration of a Nested Table

[Click here to view code image](#)

```
DECLARE
  TYPE last_name_type IS TABLE OF student.last_name%TYPE;
  last_name_tab last_name_type;
```

In this code fragment, type `last_name_type` is declared based on the column `LAST_NAME` of the `STUDENT` table. Next, the actual nested table called `last_name_tab` is declared to be of a `last_name_type`.

Unlike an associative array, a nested table may also be defined as a stand-alone user-defined type via the `CREATE TYPE` statement. This scenario is illustrated in [Listing 15.5](#).

Listing 15.5 Define a Nested Table Type on the Schema Level

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE last_name_type AS TABLE OF VARCHAR2(30);
/
CREATE OR REPLACE TYPE last_name_table AS TABLE OF last_name_type;
/
```

In [Listing 15.5](#), you define two stand-alone types to be created in the `STUDENT` schema. The first type, `last_name_type`, is a nested table type; the individual elements of this type can contain strings up to 30 characters long. The second type, `last_name_table`, is the nested table itself, which is based on the `last_name_type`.

A nested table must be initialized before its individual elements can be referenced. Consider the modified version of the example given earlier in this lab. Notice that the `last_name_type` is defined as a nested table (there is no INDEX BY clause). Affected statements are shown in bold.

For Example ch15_1b.sql

[Click here to view code image](#)

```
DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;

TYPE last_name_type IS TABLE OF student.last_name%TYPE;
last_name_tab last_name_type;

  v_index PLS_INTEGER := 0;
```

```

BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_tab(v_index) := name_rec.last_name;
    DBMS_OUTPUT.PUT_LINE ('last_name'|| v_index ||');
    '||last_name_tab(v_index));
  END LOOP;
END;

```

This example causes the following error:

[Click here to view code image](#)

```

ORA-06531: Reference to uninitialized collection
ORA-06512: at line 15

```

It causes this error because a nested table is automatically NULL when it is declared. In other words, there are no individual elements yet, because the nested table itself is NULL. To reference the individual elements of the nested table, the table must be initialized with the help of a system-defined function called a constructor. The constructor has the same name as the nested table type.

For example, the statement

[Click here to view code image](#)

```
last_name_tab := last_name_type('Rosenzweig', 'Rakhimov');
```

initializes the `last_name_tab` table to two elements. Most of the time, you will not know in advance which values should constitute a particular nested table. In this scenario, the following statement produces an empty but non-NULL nested table:

[Click here to view code image](#)

```
last_name_tab := last_name_type();
```

Notice that there are no arguments passed to a constructor.

Now consider a modified version of the example shown previously. Changes are highlighted in bold.

For Example `ch15_1c.sql`

[Click here to view code image](#)

```

DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;

  TYPE last_name_type IS TABLE OF student.last_name%TYPE;
  last_name_tab last_name_type := last_name_type();

  v_index PLS_INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_tab.EXTEND;
    last_name_tab(v_index) := name_rec.last_name;
  END LOOP;
END;

```

```
DBMS_OUTPUT.PUT_LINE ('last_name('||v_index||'):'
'||last_name_tab(v_index));
END LOOP;
END;
```

In this version, the nested table is initialized at the time of the declaration. As a consequence, it is empty, but non-NULL. The cursor loop includes a statement with one of the collection methods, EXTEND. This method allows you to increase the size of the collection. Note that the EXTEND method cannot be used with associative arrays. The next section in this lab provides a detailed explanation of the various collection methods.

Next, the nested table is assigned values just like the associative array in the original version of the example. When run, this version of the example completes successfully and produces the following output:

```
last_name(1): Kocka
last_name(2): Jung
last_name(3): Mulroy
last_name(4): Brendler
last_name(5): Carcia
last_name(6): Tripp
last_name(7): Frost
last_name(8): Snow
last_name(9): Scrittore
```

Did You Know?

What is the difference between NULL and empty collections? If a collection has not been initialized, referencing its individual elements causes the following error:

[Click here to view code image](#)

```
DECLARE
    TYPE integer_type IS TABLE OF INTEGER;
    integer_tab integer_type;

    v_index PLS_INTEGER := 1;
BEGIN
    DBMS_OUTPUT.PUT_LINE (integer_tab(v_index));
END;

ORA-06531: Reference to uninitialized collection
ORA-06512: at line 7
```

If a collection has been initialized so that it is non-NULL yet is empty, referencing its individual elements causes a different error:

[Click here to view code image](#)

```
DECLARE
    TYPE integer_type IS TABLE OF INTEGER;
    integer_tab integer_type := integer_type();

    v_index PLS_INTEGER := 1;
BEGIN
    DBMS_OUTPUT.PUT_LINE (integer_tab(v_index));
END;

ORA-06533: Subscript beyond count
ORA-06512: at line 7
```

Collection Methods

In example ch15_1c.sql, you saw one of the collection methods—in this case, EXTEND. A collection method is either a built-in procedure or a function that is called using dot notation as shown in [Listing 15.6](#).

Listing 15.6 Invoking a Collection Method

```
collection_name.method_name
```

The following list identifies the collection methods that allow you to manipulate or gain information about a particular collection:

- **EXISTS:** This function returns TRUE if a specified element exists in a collection and can be used to avoid raising SUBSCRIPT_OUTSIDE_LIMIT exceptions.
- **COUNT:** This function returns the total number of elements in a collection.
- **EXTEND:** This procedure increases the size of a collection.
- **DELETE:** This procedure deletes either all elements, just the elements in the

specified range, or a particular element from a collection. PL/SQL keeps placeholders of the deleted elements.

- **FIRST** and **LAST**: These functions return subscripts of the first and last elements of a collection. If the first element of a nested table is deleted, the **FIRST** method returns a value greater than 1. If elements are deleted from the middle of a nested table, the **LAST** method returns a value greater than the **COUNT** method.
 - **PRIOR** and **NEXT**: These functions return subscripts that precede and succeed a specified collection subscript.
 - **TRIM**: This procedure removes either one or a specified number of elements from the end of a collection. PL/SQL does not keep placeholders for the trimmed elements.
-

Watch Out!

The **EXTEND** and **TRIM** methods cannot be used with associative arrays.

The following example illustrates the use of various collection methods.

For Example *ch15_2a.sql*

[Click here to view code image](#)

```
DECLARE
    TYPE index_by_type IS TABLE OF NUMBER
        INDEX BY PLS_INTEGER;
    index_by_table index_by_type;

    TYPE nested_type IS TABLE OF NUMBER;
    nested_table nested_type := nested_type(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

BEGIN
    -- Populate associative array
    FOR i IN 1..10
    LOOP
        index_by_table(i) := i;
    END LOOP;

    -- Check if the associative array has third element
    IF index_by_table.EXISTS(3)
    THEN
        DBMS_OUTPUT.PUT_LINE ('index_by_table(3) = '||index_by_table(3));
    END IF;

    -- Delete 10th element from associative array
    index_by_table.DELETE(10);
    -- Delete 10th element from nested table
    nested_table.DELETE(10);
    -- Delete elements 1 through 3 from nested table
    nested_table.DELETE(1,3);

    -- Get element counts for associative array and nested table
    DBMS_OUTPUT.PUT_LINE ('index_by_table.COUNT = '||index_by_table.COUNT);
    DBMS_OUTPUT.PUT_LINE ('nested_table.COUNT = '||nested_table.COUNT);
    -- Get first and last indexes of the associative array
```

```

-- and nested table
DBMS_OUTPUT.PUT_LINE ('index_by_table.FIRST ='||index_by_table.FIRST);
DBMS_OUTPUT.PUT_LINE ('index_by_table.LAST ='||index_by_table.LAST);
DBMS_OUTPUT.PUT_LINE ('nested_table.FIRST ='||nested_table.FIRST);
DBMS_OUTPUT.PUT_LINE ('nested_table.LAST ='||nested_table.LAST);

-- Get indexes that precede and succeed 2nd indexes of the associative
array
-- and nested table
DBMS_OUTPUT.PUT_LINE ('index_by_table.PRIOR(2) =
'||index_by_table.PRIOR(2));
DBMS_OUTPUT.PUT_LINE ('index_by_table.NEXT(2) =
'||index_by_table.NEXT(2));
DBMS_OUTPUT.PUT_LINE ('nested_table.PRIOR(2) =
'||nested_table.PRIOR(2));
DBMS_OUTPUT.PUT_LINE ('nested_table.NEXT(2) =
'||nested_table.NEXT(2));

-- Delete last two elements of the nested table
nested_table.TRIM(2);
-- Delete last element of the nested table
nested_table.TRIM;

-- Get last index of the nested table
DBMS_OUTPUT.PUT_LINE('nested_table.LAST ='||nested_table.LAST);
END;

```

Examine the output returned by the preceding example:

```

index_by_table(3)      = 3
index_by_table.COUNT   = 9
nested_table.COUNT     = 6
index_by_table.FIRST    = 1
index_by_table.LAST     = 9
nested_table.FIRST      = 4
nested_table.LAST       = 9
index_by_table.PRIOR(2) = 1
index_by_table.NEXT(2)  = 3
nested_table.PRIOR(2)   =
nested_table.NEXT(2)    = 4
nested_table.LAST       = 7

```

The first line of the output

```
index_by_table(3) = 3
```

demonstrates that the EXISTS method returns TRUE. As a result, the IF statement

```

IF index_by_table.EXISTS(3)
THEN
...
```

evaluates to TRUE as well.

The second and third lines of the output

```
index_by_table.COUNT = 9
nested_table.COUNT   = 6
```

show the results of the COUNT method after some elements were deleted from the associative array and nested table.

Next, the fourth through seventh lines of the output

```
index_by_table.FIRST = 1
index_by_table.LAST = 9
nested_table.FIRST = 4
nested_table.LAST = 9
```

show the results of the FIRST and LAST methods. Notice that the FIRST method applied to the nested table returns 4 because the first three elements were deleted earlier.

Next, lines 8 through 11 of the output

```
index_by_table.PRIOR(2) = 1
index_by_table.NEXT(2) = 3
nested_table.PRIOR(2) =
nested_table.NEXT(2) = 4
```

show the results of the PRIOR and NEXT methods. Notice that the PRIOR method applied to the nested table returns NULL because the first element was deleted earlier.

Finally, the last line of the output

```
nested_table.LAST = 7
```

shows the value of the last subscript after the last three elements of the nested table were removed. Once the DELETE method is issued, the PL/SQL keeps placeholders of the deleted elements. Therefore, the first call of the TRIM method removed the ninth and tenth elements from the nested table, and the second call of the TRIM method removed the eighth element from the nested table. As a result, the LAST method returned the value 7 as the last subscript of the nested table.

Lab 15.2: Varrays

After this lab, you will be able to

- Use Arrays

A varray—that is, a variable-size array—is another collection type. Similar to PL/SQL tables, each element of a varray is assigned a consecutive subscript starting at 1. [Figure 15.2](#) shows a varray consisting of five integer numbers, where each number is assigned a unique subscript that corresponds to its position in the varray.

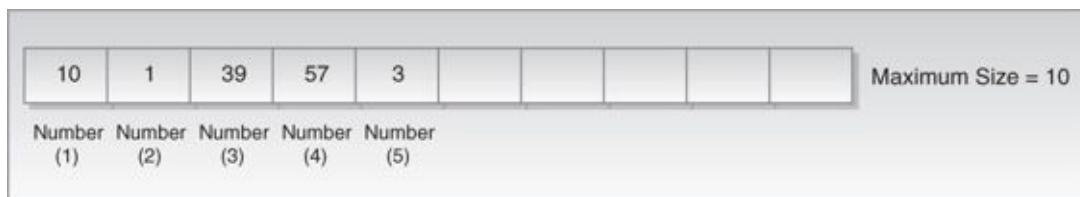


Figure 15.2 Varray

A varray has a maximum size. In other words, a subscript of a varray has a fixed lower bound equal to 1, and an upper bound that is extensible if such a need arises. In [Figure 15.2](#), the upper bound of a varray is 5, but it can be extended to 6, 7, and so on, up to 10. Therefore, a varray can contain a number of elements, varying from zero (empty array) to its maximum size. Recall that PL/SQL tables do not have a maximum size that must be

specified explicitly.

The general syntax for creating a varray is shown in [Listing 15.7](#) (the reserved words and phrases surrounded by brackets are optional).

Listing 15.7 Varray

[Click here to view code image](#)

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit) OF element_type  
[NOT NULL];  
varray_name TYPE_NAME;
```

First, a varray structure is defined using the TYPE statement, where *type_name* is the name of the type that is used in the second step to declare an actual varray. Notice that there are two variations of the type, VARRAY and VARYING ARRAY. A *size_limit* is a positive integer literal that specifies the upper bound of a varray.

Second, the actual varray is declared based on the type specified in the first step.

Consider the code fragment shown in [Listing 15.8](#).

Listing 15.8 Declaring a Varray

[Click here to view code image](#)

```
DECLARE  
  TYPE last_name_type IS VARRAY(10) OF student.last_name%TYPE;  
  last_name_varray last_name_type;
```

In this example, type *last_name_type* is declared as a varray of 10 elements based on the column LAST_NAME of the STUDENT table. Next, the actual varray *last_name_varray* is declared based on the *last_name_type*. Similarly to nested tables, a varray may be defined as a stand-alone user-defined type via the CREATE TYPE statement.

Just like a nested table, a varray is automatically NULL when it is declared and must be initialized before its individual elements can be referenced. In the following modified version of an example used in [Lab 15.1](#), ch15_1c.sql, instead of using a nested table, the script version uses varray (modified statements are highlighted in bold).

For Example ch15_1d.sql

[Click here to view code image](#)

```
DECLARE  
  CURSOR name_cur IS  
    SELECT last_name  
      FROM student  
     WHERE rownum < 10;  
  
  TYPE last_name_type IS VARRAY(10) OF student.last_name%TYPE;  
  last_name_varray last_name_type := last_name_type();  
  
  v_index PLS_INTEGER := 0;  
BEGIN  
  FOR name_rec IN name_cur  
  LOOP  
    v_index := v_index + 1;  
    last_name_varray.EXTEND;
```

```

last_name_varray(v_index) := name_rec.last_name;

DBMS_OUTPUT.PUT_LINE ('last_name(' || v_index || ')':
'|| last_name_varray(v_index));
END LOOP;
END;

```

This example produces the following output:

```

last_name(1): Kocka
last_name(2): Jung
last_name(3): Mulroy
last_name(4): Brendler
last_name(5): Carcia
last_name(6): Tripp
last_name(7): Frost
last_name(8): Snow
last_name(9): Scrittore

```

Based on the preceding example, you might realize that collection methods seen in [Lab 15.1](#) can be used with varrays as well. Consider the following example, which illustrates the use of various collection methods when applied to a varray:

For Example ch15_3a.sql

[Click here to view code image](#)

```

DECLARE
  TYPE varray_type IS VARRAY(10) OF NUMBER;
  varray varray_type := varray_type(1, 2, 3, 4, 5, 6);

BEGIN
  DBMS_OUTPUT.PUT_LINE ('varray.COUNT = ' || varray.COUNT);
  DBMS_OUTPUT.PUT_LINE ('varray.LIMIT = ' || varray.LIMIT);

  DBMS_OUTPUT.PUT_LINE ('varray.FIRST = ' || varray.FIRST);
  DBMS_OUTPUT.PUT_LINE ('varray.LAST = ' || varray.LAST);
  -- Append two copies of the 4th element to the varray
  varray.EXTEND(2, 4);
  DBMS_OUTPUT.PUT_LINE ('varray.LAST = ' || varray.LAST);
  DBMS_OUTPUT.PUT_LINE ('varray(' || varray.LAST || ') = '
    ' || varray(varray.LAST));

  -- Trim last two elements
  varray.TRIM(2);
  DBMS_OUTPUT.PUT_LINE('varray.LAST = ' || varray.LAST);
END;

```

This example returns the following output:

```

varray.COUNT = 6
varray.LIMIT = 10
varray.FIRST = 1
varray.LAST = 6
varray.LAST = 8
varray(8) = 4
varray.LAST = 6

```

The first two lines of the output

```

varray.COUNT = 6
varray.LIMIT = 10

```

show the results of the COUNT and LIMIT methods, respectively. Recall that the COUNT method returns the number of elements that a collection contains. This collection has been initialized to six elements, so the COUNT method returns a value of 6.

The next line of output corresponds to another collection method, LIMIT. This method returns the maximum number of elements that a collection can contain and is typically used with varrays because varrays have an upper bound specified at the time of declaration. This varray has an upper bound of 10, so the LIMIT method returns a value of 10.

Did You Know?

When the LIMIT method is used with associative arrays and nested tables, it returns NULL because those collection types do not have a maximum size.

The third and fourth lines of the output

```
varray.FIRST = 1  
varray.LAST  = 6
```

show the results of the FIRST and LAST methods. The fifth and six lines of the output

```
varray.LAST = 8  
varray(8)   = 4
```

show the results of the LAST method and the value of the eighth element of the collection after the EXTEND method has increased the size of the collection. Notice that the EXTEND method

```
varray.EXTEND(2, 4);
```

appends two copies of the fourth element to the collection. As a result, the seventh and eighth elements both contain a value of 4.

Finally, the last line of output

```
varray.LAST = 6
```

shows the value of the last subscript after the last two elements were removed via the TRIM method.

Watch Out!

You cannot use the **DELETE** method with a varray to remove its elements. Unlike PL/SQL tables, varrays are dense, and using the **DELETE** method causes an error, as illustrated in the following example:

[Click here to view code image](#)

```
DECLARE
    TYPE varray_type IS VARRAY(3) OF CHAR(1);
    varray varray_type := varray_type('A', 'B', 'C');
BEGIN
    varray.DELETE(3);
END;

ORA-06550: line 6, column 4:
PLS-00306: wrong number or types of arguments in call to 'DELETE'
ORA-06550: line 6, column 4:
PL/SQL: Statement ignored
```

You have now seen how to define and use all three collection data types: associative arrays, nested tables, and varrays. [Table 15.1](#) summarizes their similarities and differences.

Collection	Number of Elements	Indexed By	Status at the Time of Declaration	May Be Defined	Dense or Sparse
Associative array	Is not set	String or PLS_INTEGER	Empty	In PL/SQL block, package, procedure, or function	May be either
Nested table	Is not set	INTEGER	Null	In PL/SQL block, package, procedure, or function, and as stand-alone type at a schema level	Starts dense, but may become sparse
Varray	Set at the time of the declaration	INTEGER	Null	In PL/SQL block, package, procedure, or function, and as stand-alone type at a schema level	Always dense

Table 15.1 Associative Arrays, Nested Tables, and Varrays

Lab 15.3: Multilevel Collections

After this lab, you will be able to

- Use Multilevel Collections

So far, you have seen various examples of collections with the element type based on a scalar type, such as NUMBER and VARCHAR2. However, PL/SQL provides you with the ability to create collections whose element type is based on a collection type. Such collections are called multilevel collections.

[Figure 15.3](#) shows a varray of varrays, also called a nested varray. In this case, the varray of varrays consists of three elements, where each individual element is a varray

consisting of four integer numbers.

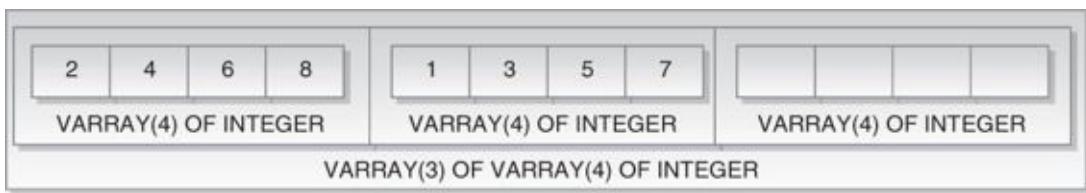


Figure 15.3 A Varray of Varrays

To reference an individual element of a varray of varrays, you use the notation as shown in [Listing 15.9](#).

Listing 15.9 Referencing an Element of a Nested Varray

[Click here to view code image](#)

```
varray_name(subscript of the outer varray)(subscript of the inner varray)
```

For example, `varray(1)(3)` in [Figure 15.3](#) equals 6; similarly, `varray(2)(1)` equals 1. Now consider the following example based on [Figure 15.3](#).

For Example ch15_4a.sql

[Click here to view code image](#)

```
DECLARE
    TYPE varray_type1 IS VARRAY(4) OF INTEGER;
    TYPE varray_type2 IS VARRAY(3) OF varray_type1;

    varray1 varray_type1 := varray_type1(2, 4, 6, 8);
    varray2 varray_type2 := varray_type2(varray1);

BEGIN
    DBMS_OUTPUT.PUT_LINE ('Varray of integers');
    FOR i IN 1..4
    LOOP
        DBMS_OUTPUT.PUT_LINE ('varray1'||i||': '||varray1(i));
    END LOOP;

    varray2.EXTEND;
    varray2(2) := varray_type1(1, 3, 5, 7);

    DBMS_OUTPUT.PUT_LINE (chr(10)||'Varray of varrays of integers');
    FOR i IN 1..2
    LOOP
        FOR j IN 1..4
        LOOP
            DBMS_OUTPUT.PUT_LINE ('varray2'||i||'('||j||'): '||varray2(i)(j));
        END LOOP;
    END LOOP;
END;
```

In the declaration portion of the example, you define two varray types. The first type, `varray_type1`, is based on the `INTEGER` data type and can contain up to four elements. The second type, `varray_type2`, is based on the `varray_type1` and can contain up to three elements, where each individual element may itself contain up to four elements. Next, you declare two varrays based on the types just described. The first varray, `varray1`, is declared as `varray_type1` and initialized so that its four elements are populated with the first four even numbers. The second varray, `varray2`, is declared

as `varray_type2`, so that each individual element is a varray consisting of four integer numbers, and initialized so that its first varray element is populated.

In the executable portion of the example, you display the values of the `varray1` on the screen. Next, you extend the upper bound of the `varray2` by 1, and populate its second element as follows:

[Click here to view code image](#)

```
varray2(2) := varray_type1(1, 3, 5, 7);
```

Here you are using a constructor corresponding to the `varray_type1` because each element of the `varray2` is based on the `varray1` collection. In other words, the same result could be achieved via the following two statements:

[Click here to view code image](#)

```
varray1(2) := varray_type1(1, 3, 5, 7);
varray2(2) := varray_type2(varray1);
```

Once the second element of the `varray2` is populated, you display the results on the screen via nested numeric FOR loops.

This example produces the following output:

```
Varray of integers
varray1(1): 2
varray1(2): 4
varray1(3): 6
varray1(4): 8

Varray of varrays of integers
varray2(1)(1): 2
varray2(1)(2): 4
varray2(1)(3): 6
varray2(1)(4): 8
varray2(2)(1): 1
varray2(2)(2): 3
varray2(2)(3): 5
varray2(2)(4): 7
```

Notice the blank line separating two portions of the example output. It is included for the readability purposes and created by adding Oracle's built-in function `CHR(10)` to the `DBMS_OUTPUT.PUT_LINE` statement. This function adds a line feed, which in turn separates two portions of the output with the blank line.

Summary

In this chapter, you learned about associative arrays, nested tables, and varrays collection types supported in PL/SQL. You also learned how to create stand-alone user-defined collection types. In addition, you discovered how to manipulate individual collection elements with the help of built-in procedures and functions designed specifically for this purpose called methods. Finally, you learned how different collection types may be nested inside another.

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.

16. Records

In this chapter, you will learn about

- [Record Types](#)
- [Nested Records](#)
- [Collections of Records](#)

In [Chapter 11](#), you were briefly introduced to the concept of a record type. You learned that a record is a composite data structure that allows you to combine different yet related data into a logical unit. You also learned that PL/SQL supports three kinds of record types: table based, cursor based, and user defined. In this chapter, you will revisit the table-based and cursor-based record types and learn about the user-defined record type. In addition, you will learn about records that contain collections and other records (called nested records) and collections of records.

Lab 16.1: Record Types

After this lab, you will be able to

- [Use Table-Based and Cursor-Based Records](#)
- [Use User-Defined Records](#)
- [Understand Record Compatibility](#)

A record structure is somewhat similar to a row of a database table. Each data item is stored in a field with its own name and data type. For example, suppose you have various data about a company, such as its name, address, and number of employees. A record containing a field for each of these items allows you to treat a company as a logical unit, thereby making it easier to organize and represent the company's information.

Table-Based and Cursor-Based Records

The %ROWTYPE attribute enables you to create table-based and cursor-based records. It is similar to the %TYPE attribute that is used to define scalar variables. Consider the following example of a table-based record.

For Example *ch16_1a.sql*

[Click here to view code image](#)

```
DECLARE
    course_rec course%ROWTYPE;
BEGIN
    SELECT *
      INTO course_rec
     FROM course
    WHERE course_no = 25;
```

```
DBMS_OUTPUT.PUT_LINE ('Course No: '||course_rec.course_no);
DBMS_OUTPUT.PUT_LINE ('Course Description: '||course_rec.description);
DBMS_OUTPUT.PUT_LINE ('Prerequisite: '||course_rec.prerequisite);
END;
```

The `course_rec` record has the same structure as a row in the `COURSE` table. As a result, there is no need to reference individual record fields when the `SELECT INTO` statement populates the `course_rec` record. However, a record does not have a value of its own; rather, each individual field holds a value. Therefore, to display record information on the screen, the individual fields are referenced using the dot notation, as shown in the `DBMS_OUTPUT.PUT_LINE` statements.

When run, this example produces the following output:

[Click here to view code image](#)

```
Course No: 25
Course Description: Intro to Programming
Prerequisite: 140
```

Watch Out!

A record does not have a value of its own. For this reason, you cannot test records for nullity, equality, or inequality. In other words, the statements

[Click here to view code image](#)

```
IF course_rec IS NULL THEN ...
IF course_rec1 = course_rec2 THEN ...
```

are illegal and will cause syntax errors.

Next, consider an example of a cursor-based record.

For Example `ch16_2a.sql`

[Click here to view code image](#)

```
DECLARE
  CURSOR student_cur IS
    SELECT first_name, last_name, registration_date
      FROM student
     WHERE rownum <= 4;

  student_rec student_cur%ROWTYPE;
BEGIN
  OPEN student_cur;
  LOOP
    FETCH student_cur INTO student_rec;
    EXIT WHEN student_cur%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE
      ('Name: '||student_rec.first_name||' '||student_rec.last_name);
    DBMS_OUTPUT.PUT_LINE
      ('Registration Date: '||to_char(student_rec.registration_date,
'MM/DD/YYYY'));
  END LOOP;
END;
```

The `student_rec` record has the same structure as the rows returned by the `student_cur` cursor. As a result, similar to the previous example, there is no need to reference the individual fields when data is fetched from the cursor to the record.

When run, this example produces the following output:

```
Name: George Kocka
Registration Date: 02/08/2007
Name: Janet Jung
Registration Date: 02/08/2007
Name: Kathleen Mulroy
Registration Date: 02/08/2007
Name: Joel Brendler
Registration Date: 02/08/2007
```

Because a cursor-based record is defined based on the rows returned by a select statement of a cursor, its declaration must be proceeded by a cursor declaration. In other words, *a cursor-based record is dependent on a particular cursor and cannot be declared prior to its cursor*.

Consider a modified version of the previous example. The cursor-based record variable is declared before the cursor (changes are shown in bold). In turn, when run, this example causes a syntax error.

For Example `ch16_2b.sql`

[Click here to view code image](#)

```
DECLARE
    student_rec student_cur%ROWTYPE;

CURSOR student_cur IS
    SELECT first_name, last_name, registration_date
    FROM student
    WHERE rownum <= 4;

BEGIN
    OPEN student_cur;
    LOOP
        FETCH student_cur INTO student_rec;
        EXIT WHEN student_cur%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE
            ('Name: '||student_rec.first_name||' '||student_rec.last_name);
        DBMS_OUTPUT.PUT_LINE
            ('Registration Date: '|| to_char(student_rec.registration_date,
'MM/DD/YYYY'));
    END LOOP;
END;
```

This example produces the following erroneous output:

[Click here to view code image](#)

```
ORA-06550: line 2, column 16:
PLS-00320: the declaration of the type of this expression is incomplete or
malformed
ORA-06550: line 2, column 16:
PL/SQL: Item ignored
ORA-06550: line 12, column 30:
```

```
PLS-00320: the declaration of the type of this expression is incomplete or
malformed
ORA-06550: line 12, column 7:
PL/SQL: SQL Statement ignored
ORA-06550: line 16, column 21:
PLS-00320: the declaration of the type of this expression is incomplete or
malformed
ORA-06550: line 15, column 7:
PL/SQL: Statement ignored
ORA-06550: line 18, column 40:
PLS-00320: the declaration of the type of this expression is incomplete or
malformed
ORA-06550: line 17, column 7:
PL/SQL: Statement ignored
```

User-Defined Records

So far, you have seen how to create records based on a table or a cursor. However, you may need to create a record that is not based on any table or any one cursor. For such situations, PL/SQL provides a user-defined record type that allows you to have complete control over the record structure.

The general syntax for creating a user-defined record is shown in [Listing 16.1](#) (the reserved words and phrases surrounded by brackets are optional).

Listing 16.1 User-Defined Record Type

[Click here to view code image](#)

```
TYPE type_name IS RECORD
  (field_name1 datatype1 [NOT NULL] [ := DEFAULT EXPRESSION],
   field_name2 datatype2 [NOT NULL] [ := DEFAULT EXPRESSION],
   ...
   field_nameN datatypeN [NOT NULL] [ := DEFAULT EXPRESSION]);
record_name TYPE_NAME;
```

First, a record structure is defined using the `TYPE` statement, where `type_name` is the name of the record type that is used in the second step to declare the actual record. Enclosed in the parentheses are declarations of each record field with its name and data type. You may also specify a `NOT NULL` constraint and/or assign a default value. Second, the actual record is declared based on the type specified in the previous step. Consider the following example.

For Example ch16_3a.sql

[Click here to view code image](#)

```
DECLARE
  TYPE time_rec_type IS RECORD
    (curr_date DATE,
     curr_day  VARCHAR2(12),
     curr_time VARCHAR2(8) := '00:00:00');

  time_rec TIME_REC_TYPE;
BEGIN
  SELECT sysdate
    INTO time_rec.curr_date
```

```

    FROM dual;

    time_rec.curr_day  := TO_CHAR(time_rec.curr_date, 'DAY');
    time_rec.curr_time := TO_CHAR(time_rec.curr_date, 'HH24:MI:SS');

    DBMS_OUTPUT.PUT_LINE ('Date: '||to_char(time_rec.curr_date, 'MM/DD/YYYY
HH24:MI:SS'));
    DBMS_OUTPUT.PUT_LINE ('Day: '||time_rec.curr_day);
    DBMS_OUTPUT.PUT_LINE ('Time: '||time_rec.curr_time);
END;

```

In this example, `time_rec_type` is a user-defined record type that contains three fields. The last field, `curr_time`, has been initialized to a particular value. Here, `time_rec` is a user-defined record based on the `time_rec_type`. Unlike in the previous examples, each record field is assigned a value individually. When run, the script produces the following output:

```

Date: 05/20/2014 10:26:32
Day: TUESDAY
Time: 10:26:32

```

As mentioned earlier, when declaring a record type, you may specify a `NOT NULL` constraint for individual fields. Such fields must be initialized. The following example causes a syntax error because a record field has not been initialized after a `NOT NULL` constraint has been defined on it.

For Example *ch16_4a.sql*

[Click here to view code image](#)

```

DECLARE
  TYPE sample_type IS RECORD
    (field1 NUMBER(3),
     field2 VARCHAR2(3) NOT NULL);
  sample_rec sample_type;

BEGIN
  sample_rec.field1 := 10;
  sample_rec.field2 := 'ABC';

  DBMS_OUTPUT.PUT_LINE ('sample_rec.field1 = '||sample_rec.field1);
  DBMS_OUTPUT.PUT_LINE ('sample_rec.field2 = '||sample_rec.field2);
END;

```

The preceding example produces this output:

[Click here to view code image](#)

```

ORA-06550: line 4, column 8:
PLS-00218: a variable declared NOT NULL must have an initialization
assignment

```

Now consider the correct version of this example (modified statements are highlighted in bold).

For Example *ch16_4b.sql*

[Click here to view code image](#)

```

DECLARE
  TYPE sample_type IS RECORD

```

```

(field1 NUMBER(3),
 field2 VARCHAR2(3) NOT NULL := 'ABC'); – initialize a NOT NULL field

sample_rec sample_type;

BEGIN
    sample_rec.field1 := 10;

    DBMS_OUTPUT.PUT_LINE ('sample_rec.field1 = '||sample_rec.field1);
    DBMS_OUTPUT.PUT_LINE ('sample_rec.field2 = '||sample_rec.field2);

END;

```

This version of the example produces the following output:

```

sample_rec.field1 = 10
sample_rec.field2 = ABC

```

Record Compatibility

You have seen that a record is defined by its name, structure, and type. Actually, two records may have the same structure yet be of a different type. In such a case, certain restrictions apply to the operations between the different record types. Consider the following example:

For Example *ch16_5a.sql*

[Click here to view code image](#)

```

DECLARE
    TYPE name_type1 IS RECORD
        (first_name VARCHAR2(15),
         last_name  VARCHAR2(30));
    TYPE name_type2 IS RECORD
        (first_name VARCHAR2(15),
         last_name  VARCHAR2(30));

    name_rec1 name_type1;
    name_rec2 name_type2;
BEGIN
    name_rec1.first_name := 'John';
    name_rec1.last_name := 'Smith';
    name_rec2 := name_rec1; – illegal assignment
END;

```

In this example, both records have the same structure, but each record is of a different type. As a result, these records are not compatible with each other on the record level. In other words, an aggregate assignment statement

[Click here to view code image](#)

```

name_rec2 := name_rec1; – illegal assignment

```

will cause an error:

[Click here to view code image](#)

```

ORA-06550: line 15, column 17:
PLS-00382: expression is of wrong type
ORA-06550: line 15, column 4:
PL/SQL: Statement ignored

```

To assign `name_rec1` to `name_rec2`, you can assign each field of `name_rec1` to the corresponding field of `name_rec2`, or you can declare `name_rec2` so that it has the same data type as `name_rec1` (changes are shown in bold).

For Example *ch16_5b.sql*

[Click here to view code image](#)

```
DECLARE
  TYPE name_type1 IS RECORD
    (first_name VARCHAR2(15),
     last_name  VARCHAR2(30));

  name_rec1 name_type1;
  name_rec2 name_type1;

BEGIN
  name_rec1.first_name := 'John';
  name_rec1.last_name := 'Smith';
  name_rec2 := name_rec1; – no longer illegal assignment
END;
```

The assignment restriction just mentioned applies to user-defined records. In other words, *you can assign a table-based or cursor-based record to a user-defined record as long as they have the same structure*. Consider the following example:

For Example *ch16_6a.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR course_cur IS
    SELECT *
      FROM course
     WHERE rownum < 2;

  TYPE course_type IS RECORD
    (course_no      NUMBER(38)
     ,description   VARCHAR2(50)
     ,cost          NUMBER(9,2)
     ,prerequisite  NUMBER(8)
     ,created_by    VARCHAR2(30)
     ,created_date  DATE
     ,modified_by   VARCHAR2(30)
     ,modified_date DATE);

  course_rec1 course%ROWTYPE; – table-based record
  course_rec2 course_cur%ROWTYPE; – cursor-based record
  course_rec3 course_type; – user-defined record

BEGIN
  – Populate table-based record
  SELECT *
    INTO course_rec1
    FROM course
   WHERE course_no = 10;

  – Populate cursor-based record
  OPEN course_cur;
  LOOP
    FETCH course_cur INTO course_rec2;
    EXIT WHEN course_cur%NOTFOUND;
```

```

END LOOP;

-- Assign COURSE_REC2 to COURSE_REC1 and COURSE_REC3
course_rec1 := course_rec2;
course_rec3 := course_rec2;

DBMS_OUTPUT.PUT_LINE (course_rec1.course_no||' - '
'||course_rec1.description);
DBMS_OUTPUT.PUT_LINE (course_rec2.course_no||' - '
'||course_rec2.description);
DBMS_OUTPUT.PUT_LINE (course_rec3.course_no||' - '
'||course_rec3.description);
END;

```

In this example, each record is of a different type; however, they are compatible with one another because all of the records have the same structure. As a result, this example does not cause any syntax errors and produces the following output:

```

10 - Technology Concepts
10 - Technology Concepts
10 - Technology Concepts

```

Lab 16.2: Nested Records

After this lab, you will be able to

- Use Nested Records

As mentioned in the introduction to this chapter, PL/SQL allows you to define nested records—that is, records that contain other records and collections. The record that contains a nested record or collection is called an enclosing record.

Consider the code fragment in [Listing 16.2](#).

Listing 16.2 Declaring a Nested Record

```

DECLARE
    TYPE name_type IS RECORD
        (first_name VARCHAR2(15),
         last_name  VARCHAR2(30));

    TYPE person_type IS
        (name   name_type,
         street  VARCHAR2(50),
         city    VARCHAR2(25),
         state   VARCHAR2(2),
         zip     VARCHAR2(5));

    person_rec person_type;

```

This code fragment contains two user-defined record types. The second user-defined record type, `person_type`, is a nested record type because its field `name` is a record of the `name_type` type (highlighted in bold).

Next, consider the complete version of the script based on the declaration of the nested record in [Listing 16.2](#). References to the nested record are shown in bold.

For Example *ch16_7a.sql*

[Click here to view code image](#)

```
DECLARE
    TYPE name_type IS RECORD
        (first_name VARCHAR2(15),
         last_name  VARCHAR2(30));

    TYPE person_type IS RECORD
        (name      name_type,
         street   VARCHAR2(50),
         city     VARCHAR2(25),
         state    VARCHAR2(2),
         zip      VARCHAR2(5));

    person_rec person_type;

BEGIN
    SELECT first_name, last_name, street_address, city, state, zip
        INTO person_rec.name.first_name, person_rec.name.last_name,
              person_rec.street, person_rec.city, person_rec.state,
              person_rec.zip
        FROM student
        JOIN zipcode USING (zip)
        WHERE rownum < 2;

    DBMS_OUTPUT.PUT_LINE ('Name:    ' ||
        person_rec.name.first_name||' '||person_rec.name.last_name);
    DBMS_OUTPUT.PUT_LINE ('Street:  '||person_rec.street);
    DBMS_OUTPUT.PUT_LINE ('City:    '||person_rec.city);
    DBMS_OUTPUT.PUT_LINE ('State:   '||person_rec.state);
    DBMS_OUTPUT.PUT_LINE ('Zip:     '||person_rec.zip);
END;
```

In this example, the `person_rec` record is a user-defined nested record. To reference its field `name`, which is a record with two fields, you use the syntax shown in [Listing 16.3](#). The parentheses are included in this listing solely for readability purposes.

Listing 16.3 Referencing Individual Fields of a Nested Record

[Click here to view code image](#)

```
enclosing_record.(nested_record or nested_collection).field_name
```

In this case, `person_rec` is the enclosing record because it contains the `name` record as one of its fields. In other words, the `name` record is nested in the `person_rec` record.

This example produces the following output:

```
Name:    George Kocka
Street:  24 Beaufield St.
City:    Dorchester
State:   MA
Zip:     02124
```

A nested record may also contain a collection as one of its fields. In the following example, given a value of a ZIP code, the names of the students residing in that ZIP code area are displayed on the screen.

For Example *ch16_8a.sql*

[Click here to view code image](#)

```
DECLARE
    TYPE last_name_type IS TABLE OF student.last_name%TYPE
        INDEX BY PLS_INTEGER;

    TYPE zip_info_type IS RECORD
        (zip      VARCHAR2(5),
         last_name_tab last_name_type);

    CURSOR name_cur (p_zip VARCHAR2) IS
        SELECT last_name
        FROM student
        WHERE zip = p_zip;

    zip_info_rec zip_info_type;
    v_zip          VARCHAR2(5) := '&sv_zip';
    v_index        PLS_INTEGER := 0;
BEGIN
    zip_info_rec.zip := v_zip;
    DBMS_OUTPUT.PUT_LINE ('ZIP: '||zip_info_rec.zip);

    FOR name_rec IN name_cur (v_zip)
    LOOP
        v_index := v_index + 1;
        zip_info_rec.last_name_tab(v_index) := name_rec.last_name;
        DBMS_OUTPUT.PUT_LINE
            ('Names('||v_index||'): '||zip_info_rec.last_name_tab(v_index));
    END LOOP;
END;
```

The declaration section of this example contains declarations of the associative array type, `last_name_type`; record type, `zip_info_type`; and nested user-defined record, `zip_info_rec`. The field, `last_name_tab`, of the `zip_info_rec` is an associative array that is populated with the help of the cursor, `name_cur`. In addition, the declaration portion contains two variables, `v_zip` and `v_index`. The variable `v_zip` is used to store the incoming value of the ZIP code provided at run time. The variable `v_index` is used to populate the associative array, `last_name_tab`. The executable portion of the script assigns values to the individual record fields, `zip` and `last_name_tab`. The `last_name_tab` is an associative array, which is populated via the cursor `FOR` loop.

When the value of 11368 is provided for the ZIP code at run time, this script produces the following output:

```
ZIP: 11368
Names(1): Lasseter
Names(2): Miller
Names(3): Boyd
Names(4): Griffen
Names(5): Huthesing
Names(6): Chatman
```

Lab 16.3: Collections of Records

After this lab, you will be able to

- Use Collections of Records

In [Lab 16.2](#), you saw an example of a nested record in which one of the record fields was defined as an associative array. PL/SQL also gives you the ability to define a collection of records (for example, an associative array where the element type is a cursor-based record). The following example illustrates this usage.

For Example *ch16_9a.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR name_cur IS
    SELECT first_name, last_name
      FROM student
     WHERE ROWNUM <= 4;

  TYPE name_type IS TABLE OF name_cur%ROWTYPE
    INDEX BY PLS_INTEGER;

  name_tab name_type;
  v_index  INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;

    name_tab(v_index).first_name := name_rec.first_name;
    name_tab(v_index).last_name  := name_rec.last_name;

    DBMS_OUTPUT.PUT_LINE('First Name('||v_index||'): ' ||
      name_tab(v_index).first_name);
    DBMS_OUTPUT.PUT_LINE('Last Name('||v_index||'): ' ||
      name_tab(v_index).last_name);
  END LOOP;
END;
```

The declaration section of this example contains a definition of the `name_cur` cursor, which returns the first and last names of four students. In addition, it defines an associative array type. The element type of the associative array is a cursor-based record defined as `%ROWTYPE`. In addition, this script defines an associative array variable and the index variable that is used later to reference individual rows of the associative array.

The executable section of the example populates the associative array and displays its records on screen. The notation used in the preceding example to reference individual elements of the array is shown in [Listing 16.4](#).

Listing 16.4 Referencing a Collection of Records

[Click here to view code image](#)

```
collection_name(index).record_field_name1
collection_name(index).record_field_name2
```

```
...  
collection_name(index).record_field_nameN
```

To reference each row of the array, you use the index variable just as in all of the previous examples that employed collections. However, because each row of this associative array is a record, you must also reference individual fields of the underlying record.

This example produces the following output:

```
First Name(1): George  
Last Name(1): Kocka  
First Name(2): Janet  
Last Name(2): Jung  
First Name(3): Kathleen  
Last Name(3): Mulroy  
First Name(4): Joel  
Last Name(4): Brendler
```

Next, consider a modified version of the preceding example. In this version, the collection type has been changed from an associative array to a nested table (all changes are shown in bold).

For Example *ch16_9b.sql*

[Click here to view code image](#)

```
DECLARE  
  CURSOR name_cur IS  
    SELECT first_name, last_name  
      FROM student  
     WHERE ROWNUM <= 4;  
  
  TYPE name_type IS TABLE OF name_cur%ROWTYPE;  
  
  name_tab name_type := name_type();  
  v_index INTEGER := 0;  
BEGIN  
  FOR name_rec IN name_cur  
  LOOP  
    v_index := v_index + 1;  
    name_tab.EXTEND;  
  
    name_tab(v_index).first_name := name_rec.first_name;  
    name_tab(v_index).last_name := name_rec.last_name;  
  
    DBMS_OUTPUT.PUT_LINE('First Name('||v_index||'): '||  
                         name_tab(v_index).first_name);  
    DBMS_OUTPUT.PUT_LINE('Last Name('||v_index||'): '||  
                         name_tab(v_index).last_name);  
  END LOOP;  
END;
```

The only differences in regard to the previous version of the script are the collection type declaration and methods required for the collection initialization. All references to the record and its individual fields remain unchanged. This version of the script produces the same output as the earlier version:

```
First Name(1): George  
Last Name(1): Kocka
```

```
First Name(2): Janet
Last Name(2): Jung
First Name(3): Kathleen
Last Name(3): Mulroy
First Name(4): Joel
Last Name(4): Brendler
```

So far, you have seen examples where a collection of records was defined on the cursor-based record type. Next, consider an example where a collection of records is defined on the user-defined record type.

For Example *ch16_10a.sql*

[Click here to view code image](#)

```
DECLARE
    CURSOR enroll_cur IS
        SELECT first_name, last_name, COUNT(*) total
            FROM student
                JOIN enrollment USING (student_id)
                    GROUP BY first_name, last_name;
    TYPE enroll_rec_type IS RECORD
        (first_name  VARCHAR2(15),
         last_name   VARCHAR2(30),
         enrollments INTEGER);

    TYPE enroll_array_type IS TABLE OF enroll_rec_type
        INDEX BY PLS_INTEGER;

    enroll_tab enroll_array_type;
    v_index    INTEGER := 0;
BEGIN
    FOR enroll_rec IN enroll_cur
    LOOP
        v_index := v_index + 1;

        enroll_tab(v_index).first_name  := enroll_rec.first_name;
        enroll_tab(v_index).last_name   := enroll_rec.last_name;
        enroll_tab(v_index).enrollments := enroll_rec.total;

        IF v_index <= 4
        THEN
            DBMS_OUTPUT.PUT_LINE('First Name('||v_index||'): ' ||
                enroll_tab(v_index).first_name);
            DBMS_OUTPUT.PUT_LINE('Last Name('||v_index||'): ' ||
                enroll_tab(v_index).last_name);
            DBMS_OUTPUT.PUT_LINE('Enrollments('||v_index||'): ' ||
                enroll_tab(v_index).enrollments);
            DBMS_OUTPUT.PUT_LINE ('-----');
        END IF;
    END LOOP;
END;
```

The declaration section of the script contains a user-defined record type, `enroll_rec_type`, which is subsequently used in the declaration of the associative array type, `enroll_array_type`. Finally, the associative array, `enroll_tab`, is declared based on the `enroll_array_type`.

In the executable portion of the script, the associative array, `enroll_tab`, is

populated via the cursor FOR loop and the first four records of the associative array are displayed on the screen.

When run, this script produces the following output:

```
First Name(1): Judy
Last Name(1): Sethi
Enrollments(1): 1
-----
First Name(2): Larry
Last Name(2): Walter
Enrollments(2): 2
-----
First Name(3): Winsome
Last Name(3): Laporte
Enrollments(3): 2
-----
First Name(4): Hiedi
Last Name(4): Lopez
Enrollments(4): 1
-----
```

Summary

In this chapter, you learned about the different types of records supported in PL/SQL and saw how to manipulate individual record elements. You have also learned about record compatibility and explored how it affects your ability to assign or compare records to each other. In addition, you discovered how different record types may be nested inside one another and learned how to define and manipulate a record that contains a collection element. Finally, you learned how to define and handle collections of records.

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.

17. Native Dynamic SQL

In this chapter, you will learn about

- [EXECUTE IMMEDIATE Statements](#)
- [OPEN-FOR, FETCH, and CLOSE Statements](#)

Generally, PL/SQL applications perform a specific task and manipulate a static set of tables. For example, a stored procedure might accept a student ID and return the student's first and last names. In such a procedure, a SELECT statement is known in advance and is compiled as part of the procedure. Such SELECT statements are called static because they do not change from execution to execution.

Now, consider a different type of PL/SQL application where SQL statements are built on the fly, based on a set of parameters specified at run time. For example, an application might need to build various reports based on SQL statements where the table and column names are not known in advance or the sorting and grouping of data are specified by the user requesting a report. Similarly, another application might need to create or drop tables or other database objects based on the actions specified by a user at run time. Because these SQL statements are generated on the fly and might change from time to time, they are called *dynamic*.

PL/SQL has a feature called *native dynamic SQL* ("dynamic SQL" for short) that helps you build applications similar to those described previously. The use of dynamic SQL makes such applications flexible, versatile, and concise because it eliminates the need for complicated programming approaches. Native dynamic SQL is more convenient to use than the Oracle-supplied package DBMS_SQL, which has similar functionality. In this chapter you will learn how to create and use dynamic SQL.

Lab 17.1: EXECUTE IMMEDIATE Statements

After this lab, you will be able to

- [Use the EXECUTE IMMEDIATE Statement](#)
- [Avoid ORA Errors When Using EXECUTE IMMEDIATE Statements](#)

Generally, dynamic SQL statements are built by your program and stored as character strings based on the parameters specified at run time. These strings must contain valid SQL statements or PL/SQL code. Consider the following example of a dynamic SQL statement:

[Click here to view code image](#)

```
'SELECT first_name, last_name FROM student  
WHERE student_id = :student_id'
```

This SELECT statement returns a student's first and last names for a given student ID. The

value of the student ID is not known in advance and is specified with the help of a *bind argument*, `:student_id`. The bind argument acts as a placeholder for an undeclared identifier, and its name must be prefixed by a colon. As a result, PL/SQL does not differentiate between the following statements:

[Click here to view code image](#)

```
'SELECT first_name, last_name
  FROM student WHERE
student_id = :student_id'
'SELECT first_name, last_name
  FROM student WHERE student_id = :id'
```

To process dynamic SQL statements, you use `EXECUTE IMMEDIATE` or `OPEN-FOR`, `FETCH`, and `CLOSE` statements. `EXECUTE IMMEDIATE` is used for a single-row `SELECT` statement, all DML statements, and DDL statements, whereas `OPEN-FOR`, `FETCH`, and `CLOSE` statements are used for multirow `SELECT` statements and reference cursors.

Did You Know?

To improve performance of dynamic SQL statements, you can also use `BULK EXECUTE IMMEDIATE`, `BULK FETCH`, `FORALL`, and `COLLECT INTO` statements. However, these statements are outside the scope of this book and are not covered here. You can find detailed explanations and examples of their usage in Oracle's online help.

Using the `EXECUTE IMMEDIATE` Statement

The `EXECUTE IMMEDIATE` statement parses a dynamic statement or a PL/SQL block for immediate execution and has the structure shown here (the reserved words and phrases surrounded by brackets are optional):

[Click here to view code image](#)

```
EXECUTE IMMEDIATE dynamic_SQL_string
  [INTO defined_variable1, defined_variable2, ...]
  [USING [IN | OUT | IN OUT] bind_argument1, bind_argument2,
  ...][{RETURNING | RETURN} field1, field2,
  ... INTO bind_argument1, bind_argument2, ...]
```

The `dynamic_SQL_string` is a string that contains a valid SQL statement or a PL/SQL block. The `INTO` clause contains the list of predefined variables that hold values returned by the `SELECT` statement. This clause is used when a dynamic SQL statement returns a single row, similar to a static `SELECT INTO` statement. Next, the `USING` clause contains a list of bind arguments whose values are passed to the dynamic SQL statement or PL/SQL block. The `IN`, `OUT`, and `IN OUT` options are modes for bind arguments. If no mode is specified, all bind arguments listed in the `USING` clause default to the `IN` mode. Finally, the `RETURNING INTO` or `RETURN` clause contains a list of bind arguments that store values returned by the dynamic SQL statement or PL/SQL block. Similar to the `USING` clause, the `RETURNING INTO` clause may also contain various argument modes; however, if no mode is specified, all bind arguments default to the `OUT` mode.

Did You Know?

When an EXECUTE IMMEDIATE statement contains both USING and RETURNING INTO clauses, the USING clause may specify only IN arguments.

The following script contains several examples of dynamic SQL.

For Example

[Click here to view code image](#)

```
DECLARE
    sql_stmt VARCHAR2(100);
    plsql_block VARCHAR2(300);
    v_zip VARCHAR2(5) := '11106';
    v_total_students NUMBER;
    v_new_zip VARCHAR2(5);
    v_student_id NUMBER := 151;
BEGIN
    -- Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ' ||
                'AS SELECT * FROM student WHERE zip = '||v_zip;
    EXECUTE IMMEDIATE sql_stmt;

    -- Select total number of records from MY_STUDENT table
    -- and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM my_student'
    INTO v_total_students;
    DBMS_OUTPUT.PUT_LINE ('Students added: '||v_total_students);

    -- Select current date and display it on the screen
    plsql_block := 'DECLARE ' ||
                   '    v_date DATE; ' ||
                   'BEGIN ' ||
                   '    SELECT SYSDATE INTO v_date FROM DUAL; ' ||
                   '    DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_date,
                                         "DD-MON-YYYY"));' ||
                   'END;';
    EXECUTE IMMEDIATE plsql_block;

    -- Update record in MY_STUDENT table
    sql_stmt := 'UPDATE my_student SET zip = 11105 WHERE student_id =
               :1 ' ||
               'RETURNING zip INTO :2';
    EXECUTE IMMEDIATE sql_stmt USING v_student_id RETURNING INTO
        v_new_zip;
    DBMS_OUTPUT.PUT_LINE ('New zip code: '||v_new_zip);
END;
```

First, this script creates the table MY_STUDENT and populates it with records for a specified value of the ZIP code. Here, the variable v_zip is concatenated with the CREATE statement instead of being passed in as a bind argument. This point is illustrated in the next example.

Second, the script selects the total number of students added to the MY_STUDENT table and displays it on the screen. The INTO option is used with the EXECUTE IMMEDIATE

statement because the `SELECT` statement returns a single row.

Third, the script includes a simple PL/SQL block that selects the current date and displays it on the screen. Because the PL/SQL block does not contain any bind arguments, the `EXECUTE IMMEDIATE` statement is used in its simplest form.

Finally, the script updates the `MY_STUDENT` table for a given student ID and returns a new ZIP code value via the `RETURNING` statement.

Thus, this `EXECUTE IMMEDIATE` command contains both `USING` and `RETURNING INTO` options. The `USING` option allows you to pass a value for the student ID to the `UPDATE` statement at run time, and the `RETURNING INTO` option allows you to pass a new ZIP code value from the `UPDATE` statement into the program.

When run, this example produces the following output:

[Click here to view code image](#)

```
Students added: 4
22-JUN-2003
New zip code: 11105

PL/SQL procedure successfully completed.
```

How to Avoid Common ORA Errors When Using `EXECUTE IMMEDIATE`

Next, consider the simplified yet incorrect version of the preceding example. Changes are shown in bold.

For Example

[Click here to view code image](#)

```
DECLARE
    sql_stmt VARCHAR2(100);
    v_zip VARCHAR2(5) := '11106';
    v_total_students NUMBER;
BEGIN
    – Drop table MY_STUDENT
    EXECUTE IMMEDIATE 'DROP TABLE my_student';
    – Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ||
                'AS SELECT * FROM student ||
                'WHERE zip = :zip';
    EXECUTE IMMEDIATE sql_stmt USING v_zip;

    – Select total number of records from MY_STUDENT table
    – and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM my_student'
    INTO v_total_students;

    DBMS_OUTPUT.PUT_LINE ('Students added: '|| v_total_students);
END;
```

First, this script drops the table `MY_STUDENT` created in the previous version of the example. Next, it recreates the `MY_STUDENT` table, but in this case uses a bind argument

to pass a value for the ZIP code to the CREATE statement at run time.

When run, this example produces the following error:

[Click here to view code image](#)

```
DECLARE
*
ERROR at line 1:
ORA-01027: bind variables not allowed for data definition operations
ORA-06512: at line 12
```

By the Way

A CREATE TABLE statement is a data definition statement. As a result, it cannot accept any bind arguments.

Next, consider another simplified version of the same example that also causes a syntax error. In this version, the table name is passed as a bind argument to the SELECT statement. Changes are shown in bold.

For Example

[Click here to view code image](#)

```
DECLARE
    sql_stmt  VARCHAR2(100);
    v_zip    VARCHAR2(5) := '11106';
    v_total_students NUMBER;
BEGIN
    -- Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ' ||
                'AS SELECT * FROM student ' || 'WHERE zip =' || v_zip;
    EXECUTE IMMEDIATE sql_stmt;
    -- Select total number of records from MY_STUDENT table
    -- and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM :my_table'
    INTO v_total_students
    USING 'my_student';
    DBMS_OUTPUT.PUT_LINE ('Students added: ' || v_total_students);
END;
```

When run, this example causes the following error:

[Click here to view code image](#)

```
DECLARE
*
ERROR at line 1:
ORA-00903: invalid table name
ORA-06512: at line 13
```

This example causes an error because *you cannot pass names of schema objects to dynamic SQL statements as bind arguments*. To provide a table name at the run time, you need to concatenate it with the SELECT statement, as shown here:

[Click here to view code image](#)

```
EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM '||my_table
    INTO v_total_students;
```

As mentioned earlier, a dynamic SQL string can contain any SQL statement or PL/SQL block. However, unlike static SQL statements, a dynamic SQL statement should not be terminated by the semicolon (;). Similarly, a dynamic PL/SQL block should not be terminated by the forward slash (/). Consider a different version of the same example where the SELECT statement is terminated by the semicolon. Changes are shown in bold. Note that if you have created the MY_STUDENT table based on the earlier corrected version of the script, you need to drop it prior to running the following script. Otherwise, the error message generated by the example will differ from the error message shown here.

For Example

[**Click here to view code image**](#)

```
DECLARE
    sql_stmt VARCHAR2(100);
    v_zip VARCHAR2(5) := '11106';
    v_total_students NUMBER;
BEGIN
    -- Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ' ||
                'AS SELECT * FROM student ' ||
                'WHERE zip = ' || v_zip;
    EXECUTE IMMEDIATE sql_stmt;

    -- Select total number of records from MY_STUDENT table
    -- and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM my_student;'
    INTO v_total_students;
    DBMS_OUTPUT.PUT_LINE ('Students added: ' || v_total_students);
END;
```

When run, this example produces the following error:

```
DECLARE
*
ERROR at line 1:
ORA-00903: invalid character
ORA-06512: at line 13
```

The semicolon added to the `SELECT` statement is treated as an invalid character when the statement is created dynamically. A somewhat similar error is generated when a PL/SQL block is terminated by a forward slash, as demonstrated in the next example. Changes are shown in bold.

For Example

[Click here to view code image](#)

```
DECLARE
  plsql_block VARCHAR2(300);
BEGIN
  -- Select current date and display it on the screen
  plsql_block := 'DECLARE '
                 '    v_date DATE; '
                 'BEGIN '
                 '    SELECT SYSDATE INTO v_date FROM DUAL; '
                 '    DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_date,
                                         "DD-MON-YYYY"));'
                 'END;'

  |||
```

```
'//';
EXECUTE IMMEDIATE plsql_block;
END;
```

When run, this example produces the following error:

[Click here to view code image](#)

```
DECLARE
*
ERROR at line 1:
ORA-06550: line 1, column 133:
PLS-00103: Encountered the symbol "/" The symbol "/" was ignored.
ORA-06512: at line 12
```

Passing NULLS

In some cases you may need to pass a **NULL** value to a dynamic SQL statement as a value for a bind argument. For example, you need to update the COURSE table so that the PREREQUISITE column is set to **NULL**. You can accomplish this with the following dynamic SQL and the EXECUTE IMMEDIATE statement.

For Example

[Click here to view code image](#)

```
DECLARE
    sql_stmt VARCHAR2(100);
BEGIN
    sql_stmt := 'UPDATE course' ||
                '    SET prerequisite = :some_value';
    EXECUTE IMMEDIATE sql_stmt
    USING NULL;
END;
```

When run, this script causes the following error:

[Click here to view code image](#)

```
USING NULL;
*
ERROR at line 7:
ORA-06550: line 7, column 10:
PLS-00457: expressions have to be of SQL types
ORA-06550: line 6, column 4:
PL/SQL: Statement ignored
```

This error is generated because the literal **NULL** in the **USING** clause is not recognized as one of the SQL types. To pass a **NULL** value to the dynamic SQL statement, this example should be modified as follows (changes are shown in bold):

For Example

[Click here to view code image](#)

```
DECLARE
    sql_stmt VARCHAR2(100);
v_null VARCHAR2(1);
BEGIN
    sql_stmt := 'UPDATE course' ||
                '    SET prerequisite = :some_value';
    EXECUTE IMMEDIATE sql_stmt
```

```
    USING v_null;
END;
```

Putting It All Together

To correct the script, you add an initialized variable `v_null` and replace the literal `NULL` in the `USING` clause with this variable. Because the variable `v_null` has not been initialized, its value remains `NULL`, and it is passed to the dynamic `UPDATE` statement at run time. As a result, this version of the script completes without any errors.

For Example *ch17_1.sql, version 1.0*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    sql_stmt VARCHAR2(200);
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
BEGIN
    sql_stmt := 'SELECT first_name, last_name' ||
                ' FROM student' ||
                ' WHERE student_id = :1';
    EXECUTE IMMEDIATE sql_stmt
    INTO v_first_name, v_last_name
    USING v_student_id;

    DBMS_OUTPUT.PUT_LINE ('First Name: ' || v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: ' || v_last_name);
END;
```

In the proceeding example, the declaration section of the script includes a declaration of the string that contains the dynamic SQL statement as well as declarations of three variables to hold the student's ID, first name, and last name, respectively. The executable portion of the script contains a dynamic SQL statement with one bind argument that is used to pass the value of the student ID to the `SELECT` statement at run time. The dynamic SQL statement is executed via the `EXECUTE IMMEDIATE` statement with two options, `INTO` and `USING`. The `INTO` clause contains two variables, `v_first_name` and `v_last_name`. These variables contain results returned by the `SELECT` statement. The `USING` clause contains the variable `v_student_id`, which is used to pass a value to the `SELECT` statement at run time. Finally, two `DBMS_OUTPUT.PUT_LINE` statements are used to display the results of the `SELECT` statement on the screen.

When run, this script will prompt the user for a value. If, for example, 105 is entered, it produces the following output:

[Click here to view code image](#)

```
Enter value for sv_student_id: 105

old      3:      v_student_id NUMBER := &sv_student_id;
new      3:      v_student_id NUMBER := 105;
First Name: Angel
Last Name: Moskowitz

PL/SQL procedure successfully completed
```

In the next example, the script is modified so that the student's address (street, city, state, and ZIP code) are displayed on the screen as well.

For Example ch17_1.sql, version 2.0

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    sql_stmt VARCHAR2(200);
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
    v_street VARCHAR2(50);
    v_city VARCHAR2(25);
    v_state VARCHAR2(2);
    v_zip VARCHAR2(5);
BEGIN
    sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
                '           ,b.city, b.state, b.zip' ||
                '      FROM student a, zipcode b' ||
                '     WHERE a.zip = b.zip' ||
                '       AND student_id = :1';
    EXECUTE IMMEDIATE sql_stmt
    INTO v_first_name, v_last_name, v_street, v_city, v_state, v_zip
    USING v_student_id;
    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
    DBMS_OUTPUT.PUT_LINE ('Street: '||v_street);
    DBMS_OUTPUT.PUT_LINE ('City: '||v_city);
    DBMS_OUTPUT.PUT_LINE ('State: '||v_state);
    DBMS_OUTPUT.PUT_LINE ('Zip Code: '||v_zip);
END;
```

In this script, four new variables are declared: `v_street`, `v_city`, `v_state`, and `v_zip`. Next, the dynamic SQL statement is modified so that it can return the student's address. In turn, the `INTO` clause is modified by adding the new variables to it. Next, `DBMS_OUTPUT.PUT_LINE` statements are added to display the student's address on the screen.

When run, this script produces the following output:

[Click here to view code image](#)

```
Enter value for sv_student_id: 105
old   3:    v_student_id NUMBER := &sv_student_id;
new   3:    v_student_id NUMBER := 105;
First Name: Angel
Last Name: Moskowitz
Street: 320 John St.
City: Ft. Lee
State: NJ
Zip Code: 07024
```

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

Note that the order of variables listed in the `INTO` clause must follow the order of the columns listed in the `SELECT` statement. In other words, if the `INTO` clause listed variables so that `v_zip` and `v_state` were misplaced while the `SELECT` statement

remains unchanged, the scripts would generate an error. The following example demonstrates this case.

For Example *ch17_1.sql*, version 3.0

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    sql_stmt VARCHAR2(200);
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
    v_street VARCHAR2(50);
    v_city VARCHAR2(25);
    v_state VARCHAR2(2);
    v_zip VARCHAR2(5);
BEGIN
    sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
                '           ,b.city,b.state, b.zip' ||
                '      FROM studenta, zipcode b' ||
                '     WHERE a.zip = b.zip' ||
                '       AND student_id = :1';
    EXECUTE IMMEDIATE sql_stmt
    -- variables v_state and v_zip are misplaced
    INTO v_first_name, v_last_name, v_street, v_city, v_zip, v_state
    USING v_student_id;
    DBMS_OUTPUT.PUT_LINE ('First Name: ' || v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: ' || v_last_name);
    DBMS_OUTPUT.PUT_LINE ('Street:      ' || v_street);
    DBMS_OUTPUT.PUT_LINE ('City:        ' || v_city);
    DBMS_OUTPUT.PUT_LINE ('State:       ' || v_state);
    DBMS_OUTPUT.PUT_LINE ('Zip Code:    ' || v_zip);
END;
```

When run, this script produces the following error:

[Click here to view code image](#)

```
Enter value for sv_student_id: 105

old   3:      v_student_id NUMBER := &sv_student_id;
new   3:      v_student_id NUMBER := 105;

DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 16
```

This error is generated because the variable **v_state** can hold up to two characters. However, in this example, it is trying to store a ZIP code that contains five characters.

Next, this script is modified so that the **SELECT** statement can be run against either the **STUDENT** or **INSTRUCTOR** table. In other words, the user can specify the table name to be used in the **SELECT** statement at run time. The changes for this version are highlighted in bold.

For Example *ch17_1.sql*, version 4.0

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    sql_stmt VARCHAR2(200);
    v_table_name VARCHAR2(20) := '&sv_table_name';
    v_id NUMBER := &sv_id;
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
    v_street VARCHAR2(50);
    v_city VARCHAR2(25);
    v_state VARCHAR2(2);
    v_zip VARCHAR2(5);
BEGIN
    sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
                ', b.city, b.state, b.zip' ||
                ' FROM'||v_table_name||' a, zipcode b' ||
                ' WHERE a.zip = b.zip' ||
                ' AND'||v_table_name||'_id = :1';
    EXECUTE IMMEDIATE sql_stmt
    -INTO v_first_name, v_last_name, v_street, v_city, v_state, v_zip
    USING v_id;

    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
    DBMS_OUTPUT.PUT_LINE ('Street: '||v_street);
    DBMS_OUTPUT.PUT_LINE ('City: '||v_city);
    DBMS_OUTPUT.PUT_LINE ('State: '||v_state);
    DBMS_OUTPUT.PUT_LINE ('Zip Code: '||v_zip);
END;
```

The declaration portion of the script contains a new variable, **v_table_name** that holds the name of a table provided at run time by the user. In addition, the variable **v_student_id** has been replaced by the variable **v_id** since it is not known in advance which table, STUDENT or INSTRUCTOR, will be accessed at run time.

The executable portion of the script contains a modified dynamic SQL statement. Notice that the statement does not contain any information specific to the STUDENT or INSTRUCTOR tables. In other words, the dynamic SQL statement used by the previous version (ch17_1.sql version 3.0)

[Click here to view code image](#)

```
sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
            ', b.city, b.state, b.zip' ||
            ' FROM student a, zipcode b' ||
            ' WHERE a.zip = b.zip' ||
            ' AND student_id = :1';
```

has been replaced by

[Click here to view code image](#)

```
sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
            ', b.city, b.state, b.zip' ||
            ' FROM'||v_table_name||' a, zipcode b' ||
            ' WHERE a.zip = b.zip' ||
            ' AND'||v_table_name||'_id = :1';
```

The table name (STUDENT) has been replaced by the variable **v_table_name** in the

FROM and WHERE clauses.

Did You Know?

In the last two versions of the script, you have used generic table aliases, **a** and **b**, instead of **s** and **z** or **i** and **z**, which are more descriptive. This technique allows you to create generic SQL statements that are not based on a specific table because you do not always know which table is appropriate in advance.

This version of the script produces the following output. The first run is made against the STUDENT table.

[Click here to view code image](#)

```
Enter value for sv_table_name: student
old  3:      v_table_name VARCHAR2(20) := '&sv_table_name';
new  3:      v_table_name VARCHAR2(20) := 'student';
Enter value for sv_id: 105
old  4:      v_id NUMBER := &sv_id;
new  4:      v_id NUMBER := 105;
First Name: Angel
Last Name: Moskowitz
Street:     320 John St.
City:       Ft. Lee
State:      NJ
Zip Code:   07024
PL/SQL procedure successfully completed.
```

The second run is against the INSTRUCTOR table:

[Click here to view code image](#)

```
Enter value for sv_table_name: instructor
old  3:      v_table_name VARCHAR2(20) := '&sv_table_name';
new  3:      v_table_name VARCHAR2(20) := 'instructor';
Enter value for sv_id: 105
old  4:      v_id NUMBER := &sv_id;
new  4:      v_id NUMBER := 105;
First Name: Anita
Last Name: Morris
Street:     34 Maiden Lane
City:       New York
State:      NY
Zip Code:   10015
PL/SQL procedure successfully completed.
```

Lab 17.2: OPEN-FOR, FETCH, and CLOSE Statements

After this lab, you will be able to

- Use OPEN-FOR statements
- Use FETCH statements
- Use CLOSE statements

The OPEN-FOR, FETCH, and CLOSE statements are used for multirow queries or cursors. This concept is very similar to the static cursor processing that you encountered in [Chapter 11](#). Just as in the case of static cursors, first you associate a cursor variable with a query. Next, you open the cursor variable so that it points to the first row of the result set. Next, you fetch one row at a time from the result set. Finally, when all rows have been processed, you close the cursor (cursor variable).

Opening Cursor

In the case of a dynamic SQL, the OPEN-FOR statement has an optional USING clause that allows you to pass values to the bind arguments at run time. The general syntax for an OPEN-FOR statement is as follows (the reserved words and phrases surrounded by brackets are optional):

[Click here to view code image](#)

```
OPEN cursor_variable FOR dynamic_SQL_string  
[USING bind_argument1, bind_argument2, ...]
```

The *cursor_variable* is a variable of a weak REF CURSOR type, and the *dynamic_SQL_string* is a string that contains a multirow query.

For Example

[Click here to view code image](#)

```
DECLARE  
    TYPE student_cur_type IS REF CURSOR;  
    student_cur student_cur_type;  
    v_zip VARCHAR2(5) := '&sv_zip';  
    v_first_name VARCHAR2(25);  
    v_last_name VARCHAR2(25);  
BEGIN  
    OPEN student_cur FOR  
        'SELECT first_name, last_name FROM student'||'WHERE zip = :1'  
        USING v_zip;  
    ...
```

This code fragment first defines a weak cursor type, *student_cur_type*. Next, it defines a cursor variable, *student_cur*, based on the REF CURSOR type specified in the previous step. At run time, the *student_cur* variable is associated with the SELECT statement that returns the first and last names of students for a given value of *zip*.

Fetching from a Cursor

As mentioned earlier, the FETCH statement returns a single row from the result set into a list of variables defined in a PL/SQL block and moves the cursor to the next row. If a loop is being processed and there are no more rows to fetch, the EXIT WHEN statement evaluates to TRUE, and control of the execution passes outside the cursor loop. The general syntax for a FETCH statement is as follows:

[Click here to view code image](#)

```
FETCH cursor_variable
```

```
INTO defined_variable1, defined_variable2, ...
EXIT WHEN cursor_variable%NOTFOUND;
```

Continuing the previous example, you fetch the student's first and last names into variables specified in the declaration section of the PL/SQL block. Next, you evaluate if there are more records to process via an `EXIT WHEN` statement. As long as there are more records to process, the student's first and last names are displayed on the screen. Once the last row is fetched, the cursor loop terminates. The changes necessary for these steps are shown in bold.

For Example

[Click here to view code image](#)

```
DECLARE
    TYPE student_cur_type IS REF CURSOR;
    student_cur student_cur_type;
    v_zip VARCHAR2(5) := '&sv_zip';
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
BEGIN
    OPEN student_cur FOR
        'SELECT first_name, last_name FROM student '||'WHERE zip = :1'
    USING v_zip;

    LOOP
        FETCH student_cur INTO v_first_name, v_last_name;
        EXIT WHEN student_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
        DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
    END LOOP;
    ...

```

The number of variables listed in the `INTO` clause must correspond to the number of columns returned by the cursor. Furthermore, the variables in the `INTO` clause must be type compatible with the cursor columns.

Closing a Cursor

The `CLOSE` statement disassociates the cursor variable with the multirow query. As a result, after the `CLOSE` statement executes, the result set becomes undefined. The general syntax for a `CLOSE` statement is as follows:

```
CLOSE cursor_variable;
```

Now consider the complete version of the example shown previously. Changes are shown in bold.

For Example

[Click here to view code image](#)

```
DECLARE
    TYPE student_cur_type IS REF CURSOR;
    student_cur student_cur_type;
    v_zip VARCHAR2(5) := '&sv_zip';
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
```

```

BEGIN
  OPEN student_cur FOR
    'SELECT first_name, last_name FROM student '||'WHERE zip = :1'
    USING v_zip;
  LOOP
    FETCH student_cur INTO v_first_name, v_last_name;
    EXIT WHEN student_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
  END LOOP;
  CLOSE student_cur;
EXCEPTION
  WHEN OTHERS THEN
    IF student_cur%ISOPEN THEN
      CLOSE student_cur;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
END;

```

The IF statement in the exception-handling section evaluates to TRUE if an exception is encountered before the cursor processing is completed. In such a case, it is considered good practice to check whether a cursor is still open and close it, if necessary, so that all resources associated with the cursor will be freed before the program terminates.

When run, this example produces the following output:

[Click here to view code image](#)

```

Enter value for sv_zip: 11236
old  5:      v_zip VARCHAR2(5) := '&sv_zip';
new  5:      v_zip VARCHAR2(5) := '11236';
First Name: Derrick
Last Name: Baltazar
First Name: Michael
Last Name: Lefbowitz
First Name: Bridget
Last Name: Hagel

```

PL/SQL procedure successfully completed.

In the following example, pay close attention to the use of spaces.

For Example *ch17_2.sql, version 1.0*

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
  TYPE zip_cur_type IS REF CURSOR;
  zip_cur zip_cur_type;
  sql_stmt VARCHAR2(500);
  v_zip VARCHAR2(5);
  v_total NUMBER;
  v_count NUMBER;
BEGIN
  sql_stmt := 'SELECT zip, COUNT(*) total'|||
              '   FROM student '|||
              ' GROUP BY zip';
  v_count := 0;
  OPEN zip_cur FOR sql_stmt;
  LOOP

```

```

    FETCH zip_cur INTO v_zip, v_total;
    EXIT WHEN zip_cur%NOTFOUND;
    -- Limit the number of lines printed on the
    -- screen to 10
    v_count := v_count + 1;
    IF v_count <= 10 THEN
        DBMS_OUTPUT.PUT_LINE ('Zip code: '||v_zip||
                               ' Total: '||v_total);
    END IF;
END LOOP;
CLOSE zip_cur;
EXCEPTION
    WHEN OTHERS THEN
        IF zip_cur%ISOPEN THEN
            CLOSE zip_cur;
        END IF;
        DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
    END;

```

Consider the use of spaces in the SQL statements generated dynamically. In the preceding script, the string that holds the dynamic SQL statement consists of three strings concatenated together, where each string is written on a separate line.

[Click here to view code image](#)

```

sql_stmt := 'SELECT zip, COUNT(*) total' ||
            ' FROM student' ||
            'GROUP BY zip';

```

This format of the dynamic SELECT statement is very similar to the format of any static SELECT statement that you have seen throughout this book. However, there is a subtle difference. In one instance, extra spaces have been added for formatting reasons. For example, the FROM keyword is prefixed by two spaces so that it is aligned with the SELECT keyword. Yet, in another instance, a space has been added to separate out a reserved phrase. In this case, a space has been added after the STUDENT table to separate out the GROUP BY clause. This step is necessary because once the strings are concatenated, the resulting SELECT statement looks as follows:

[Click here to view code image](#)

```
SELECT zip, COUNT(*) total  FROM student GROUP BY zip
```

If no space is added after the STUDENT table, the resulting SELECT statement

[Click here to view code image](#)

```
SELECT zip, COUNT(*) total  FROM studentGROUP BY zip
```

causes the following error:

[Click here to view code image](#)

```
ERROR: ORA-00933: SQL command not properly ended
```

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

In the declaration portion of the example script, a weak cursor type is defined as `zip_cur_type`, and a cursor variable `zip_cur` of the `zip_cur_type` type is also defined. Next, a string variable to hold a dynamic SQL statement is defined as well as two variables `v_zip` and `v_total`, which hold data returned by the cursor. Finally, a

counter variable is defined so that only the first 10 rows returned by the cursor are displayed on the screen.

In the executable portion of the script, a dynamic SQL statement is generated, which is then associated with the cursor variable, `zip_cur`. The cursor is opened. Then, for each row returned by the cursor, the values of a ZIP code and the total number of students living in that ZIP code area are populated into the variables `v_zip` and `v_total`, respectively. Next, the script checks whether there are more rows to fetch from the cursor. If there are more rows to process, the value of the counter variable is incremented by 1. As long as the value of the counter is less than or equal to 10, the row returned by the cursor is displayed on the screen. If there are no more rows to fetch, the cursor is closed.

The exception-handling section of the script checks whether the cursor is open. If it is, the script closes the cursor and displays an error message on the screen before terminating.

When run, the script should produce output similar to that shown here:

[Click here to view code image](#)

```
Zip code: 01247 Total: 1
Zip code: 02124 Total: 1
Zip code: 02155 Total: 1
Zip code: 02189 Total: 1
Zip code: 02563 Total: 1
Zip code: 06483 Total: 1
Zip code: 06605 Total: 1
Zip code: 06798 Total: 1
Zip code: 06820 Total: 3
Zip code: 06830 Total: 3
PL/SQL procedure successfully completed.
```

The script `ch17_2.sql`, version 1.0, from the previous example is now modified so that the `SELECT` statement can be run against either the `STUDENT` or `INSTRUCTOR` table. In other words, the user can specify the table name used in the `SELECT` statement at run time.

For Example `ch17_2.sql, version 2.0`

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
  TYPE zip_cur_type IS REF CURSOR;
  zip_cur zip_cur_type;
  v_table_name VARCHAR2(20) := '&sv_table_name';
  sql_stmt VARCHAR2(500);
  v_zip VARCHAR2(5);
  v_total NUMBER;
  v_count NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Totals from ||v_table_name||'
    ' table');

  sql_stmt := 'SELECT zip, COUNT(*) total||'
    ' FROM'||v_table_name||' ||'
    ' GROUP BY zip';

  v_count := 0;
```

```

OPEN zip_cur FOR sql_stmt;
LOOP
    FETCH zip_cur INTO v_zip, v_total;
    EXIT WHEN zip_cur%NOTFOUND;
    -- Limit the number of lines printed on the
    -- screen to 10
    v_count := v_count + 1;
    IF v_count <= 10 THEN
        DBMS_OUTPUT.PUT_LINE ('Zip code: '||v_zip||
                               ' Total: '||v_total);
    END IF;
END LOOP;
CLOSE zip_cur;
EXCEPTION
    WHEN OTHERS THEN
        IF zip_cur%ISOPEN THEN
            CLOSE zip_cur;
        END IF;
        DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
END;

```

In this version of the script, the variable `v_table_name` has been added to hold the name of a table provided at the run time. A `DBMS_OUTPUT.PUT_LINE` has been added to display a message indicating the table from which the total numbers are coming. The dynamic SQL statement was also modified as follows:

[Click here to view code image](#)

```

sql_stmt := 'SELECT zip, COUNT(*) total' ||
            ' FROM '||v_table_name||' ' ||
            'GROUP BY zip';

```

The variable `v_table_name` has been inserted in place of the actual table name (`STUDENT`). Note that a space is concatenated to the variable `v_table_name`, so that the `SELECT` statement does not cause any errors.

When run, this script produces the following output. The first run is based on the `STUDENT` table.

[Click here to view code image](#)

```

Enter value for sv_table_name: student
old      5:      v_table_name VARCHAR2(20) := '&sv_table_name';
new      5:      v_table_name VARCHAR2(20) := 'student';
Totals from student table
Zip code: 01247 Total: 1
Zip code: 02124 Total: 1
Zip code: 02155 Total: 1
Zip code: 02189 Total: 1
Zip code: 02563 Total: 1
Zip code: 06483 Total: 1
Zip code: 06605 Total: 1
Zip code: 06798 Total: 1
Zip code: 06820 Total: 3
Zip code: 06830 Total: 3

```

PL/SQL procedure successfully completed.

The second run is based on the `INSTRUCTOR` table.

[Click here to view code image](#)

```
Enter value for sv_table_name: instructor
old   5:      v_table_name VARCHAR2(20) := '&sv_table_name';
new   5:      v_table_name VARCHAR2(20) := 'instructor';
Totals from instructor table
Zip code: 10005 Total: 1
Zip code: 10015 Total: 3
Zip code: 10025 Total: 4
Zip code: 10035 Total: 1

PL/SQL procedure successfully completed.
```

So far, you have seen that values returned by the dynamic SQL statements are stored in individual variables such as `v_last_name` or `v_first_name`. In such cases, you list variables in the order of the corresponding columns returned by the `SELECT` statement. This approach becomes somewhat cumbersome when a dynamic SQL statement returns more than a few columns. As a result, PL/SQL allows you to store values returned by the dynamic `SELECT` statements in the variables of the record type.

Consider the modified version of the script used in this lab. In this version, instead of creating separate variables, a user-defined record is created. This record is then used to fetch data from the cursor and display it on the screen. Changes are shown in bold.

For Example *ch17_2.sql, version 3.0*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    TYPE zip_cur_type IS REF CURSOR;
    zip_cur zip_cur_type;

    TYPE zip_rec_type IS RECORD
        (zip VARCHAR2(5),
         total NUMBER);
    zip_rec zip_rec_type;

    v_table_name VARCHAR2(20) := '&sv_table_name';
    sql_stmt VARCHAR2(500);
    v_count NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Totals from '||v_table_name||
                          ' table');
    sql_stmt := 'SELECT zip, COUNT(*) total' ||
               ' FROM '||v_table_name||' ' ||
               'GROUP BY zip';
    v_count := 0;
    OPEN zip_cur FOR sql_stmt;
    LOOP
        FETCH zip_cur INTO zip_rec;
        EXIT WHEN zip_cur%NOTFOUND;

        -- Limit the number of lines printed on the
        -- screen to 10
        v_count := v_count + 1;
        IF v_count <= 10 THEN
            DBMS_OUTPUT.PUT_LINE ('Zip code: '||zip_rec.zip||
                                  ' Total: '||zip_rec.total);
        END IF;
    END LOOP;
END;
```

```

        END IF;
    END LOOP;
    CLOSE zip_cur;
EXCEPTION
    WHEN OTHERS THEN
        IF zip_cur%ISOPEN THEN
            CLOSE zip_cur;
        END IF;
        DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
END;

```

When version 3 of this script is run, it produces the following results for the STUDENT table:

[Click here to view code image](#)

```

Enter value for sv_table_name: student
old 10:      v_table_name VARCHAR2(20) := '&sv_table_name';
new 10:      v_table_name VARCHAR2(20) := 'student';
Totals from student table
Zip code: 01247 Total: 1
Zip code: 02124 Total: 1
Zip code: 02155 Total: 1
Zip code: 02189 Total: 1
Zip code: 02563 Total: 1
Zip code: 06483 Total: 1
Zip code: 06605 Total: 1
Zip code: 06798 Total: 1
Zip code: 06820 Total: 3
Zip code: 06830 Total: 3

PL/SQL procedure successfully completed.

```

The same script produces the following results for the INSTRUCTOR table:

[Click here to view code image](#)

```

Enter value for sv_table_name: instructor
old 10:      v_table_name VARCHAR2(20) := '&sv_table_name';
new 10:      v_table_name VARCHAR2(20) := 'instructor';
Totals from instructor table
Zip code: 10005 Total: 1
Zip code: 10015 Total: 3
Zip code: 10025 Total: 4
Zip code: 10035 Total: 1

PL/SQL procedure successfully completed.

```

Summary

In this chapter, you learned how to build native dynamic SQL statements in PL/SQL; such statements are used when you need to build flexibility into your code. Dynamic SQL allows for varying the SQL statements that are executed at run time, thereby allowing for various elements of the SQL to change, such as the table and the columns. The first method covered in this chapter was the EXECUTE IMMEDIATE statement; you also saw how to avoid various Oracle errors when using this method. The OPEN-FOR, FETCH, and CLOSE statements were then explained in detail; these approaches allow for multirow queries.

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.

18. Bulk SQL

In this chapter, you will learn about

- [FORALL Statements](#)
- [The BULK COLLECT Clause](#)
- [Binding Collections in SQL Statements](#)

In [Chapter 1](#), you learned that the PL/SQL engine sends SQL statements to the SQL engine, which then returns results back to the PL/SQL engine. The communication between the PL/SQL and SQL engines is also known as a context switch. A certain performance overhead is associated with these context switches. However, the PL/SQL language has a number of features that can minimize the performance overhead, which are collectively known as bulk SQL. Generally, if a SQL statement affects four or more rows, bulk SQL may improve performance significantly. Bulk SQL supports batch processing of SQL statements and their results, and it consists of two features, the **FORALL** statement and the **BULK COLLECT** clause.

Starting with Oracle 12c, support for collection data types and bulk SQL has been extended to dynamic SQL. As a consequence, you are able to bind collection variables when using an **EXECUTE IMMEDIATE** statement or **OPEN-FOR**, **FETCH**, and **CLOSE** statements. This ability is covered in detail in [Lab 18.3](#).

Lab 18.1: FORALL Statements

After this Lab, you will be able to

- [Use FORALL Statements](#)
- [Use the SAVE EXCEPTIONS Option](#)
- [Use the INDICES OF Option](#)
- [Use the VALUES OF Option](#)

Consider an **INSERT** statement enclosed by a numeric **FOR** loop that iterates 10 times, as shown in [Listing 18.1](#).

Listing 18.1 *INSERT Statement Enclosed by a Numeric FOR Loop*

```
FOR i IN 1..10
LOOP
  INSERT INTO table_name
  VALUES (...);
END LOOP;
```

This **INSERT** statement will be sent from the PL/SQL engine to the SQL engine 10 times. In other words, 10 context switches take place. If the numeric **FOR** loop is replaced with a **FORALL** statement, however, the **INSERT** statement is sent only once from

PL/SQL engine to the SQL engine, yet it is still executed 10 times. In this case, there is only one context switch between PL/SQL and SQL.

Using FORALL Statements

The FORALL statement sends INSERT, UPDATE, or DELETE statements in batches from the PL/SQL engine to the SQL engine instead of one statement at a time. It has the structure shown in [Listing 18.2](#) (the reserved words and phrases surrounded by brackets are optional).

Listing 18.2 FORALL Statement Syntax

[Click here to view code image](#)

```
FORALL loop_counter IN bounds_clause
    SQL_STATEMENT [SAVE EXCEPTIONS];
```

where *BOUNDS_CLAUSE* is one of the following:

[Click here to view code image](#)

```
lower_limit..upper_limit
INDICES OF collection_name BETWEEN lower_limit..upper_limit
VALUES OF collection_name
```

The FORALL statement has an implicitly defined loop counter variable associated with it. The values of this loop counter variable and the number of loop iterations are controlled by the *BOUNDS_CLAUSE*, which has three forms. The first form specifies lower and upper limits for the loop counter; this syntax is very similar to the numeric FOR loop. The second form, INDICES OF... references subscripts of the individual elements of a particular collection. This collection may be a nested table or an associative array that has numeric subscripts. The third form of the *BOUNDS_CLAUSE*, VALUES OF... references values of the individual elements of a particular collection that is either a nested table or an associative array.

By the Way

A collection referenced by the INDICES OF clause may be sparse. In other words, some of its elements have been deleted.

Watch Out!

When using the `VALUES OF` option, the following restrictions apply:

- If the collection used in the `VALUES OF` clause is an associative array, it must be indexed by a `PLS_INTEGER`.
 - The elements of the collection used in the `VALUES OF` clause must be `PLS_INTEGER`.
 - When a collection referenced by the `VALUES OF` clause is empty, the `FORALL` statement causes an exception.
-

Next, the `SQL_STATEMENT` is a static or dynamic `INSERT`, `UPDATE`, or `DELETE` statement that references one or more collections. Finally, the `SAVE EXCEPTION` clause is optional; it allows the `FORALL` statement to continue even when `SQL_STATEMENT` causes an exception.

The following example illustrates how the `FORALL` statement may be used. This example, as well as other examples in this chapter, use a `TEST` table created specifically for this purpose. The rows from the `TEST` table can be easily inserted, updated, or deleted without affecting the `STUDENT` schema or violating any integrity constraints.

For Example `ch18_1a.sql`

[Click here to view code image](#)

```
CREATE TABLE test
  (row_num NUMBER
   ,row_text VARCHAR2(10));

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10)    INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;
  v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Populate TEST table
  FORALL i IN 1..10
    INSERT INTO test (row_num, row_text)
    VALUES (row_num_tab(i), row_text_tab(i));

  COMMIT;
  -- Check how many rows were inserted in the TEST table
  -- display it on the screen
  SELECT COUNT(*)
```

```

    INTO v_rows
    FROM TEST;

    DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' rows in the TEST table');
END;

```

As mentioned earlier, when SQL statements are used with FORALL statements, they reference collection elements. Thus, in this script, you define two collection types, `row_num_type` and `row_text_type`, as associative arrays. In addition, two collections, `row_num_tab` and `row_text_tab`, are populated via the numeric FOR loop. Next, you populate the TEST table with the data from these two collections.

When run, this example produces the following output:

[Click here to view code image](#)

There are 10 rows in the TEST table

The next example demonstrates the performance gain realized through use of the FORALL statement. This script compares the execution times of the INSERT statements issued against the TEST table. The first 100 INSERT statements are enclosed by the numeric FOR loop and the second 100 INSERT statements are enclosed by the FORALL statement.

For Example *ch18_2a.sql*

[Click here to view code image](#)

```

TRUNCATE TABLE test;

DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10)   INDEX BY PLS_INTEGER;

    row_num_tab row_num_type;
    row_text_tab row_text_type;

    v_start_time INTEGER;
    v_end_time   INTEGER;
BEGIN
    -- Populate collections
    FOR i IN 1..100
    LOOP
        row_num_tab(i) := i;
        row_text_tab(i) := 'row'||i;
    END LOOP;

    -- Record start time
    v_start_time := DBMS_UTLILITY.GET_TIME;

    -- Insert first 100 rows
    FOR i IN 1..100
    LOOP
        INSERT INTO test (row_num, row_text)
        VALUES (row_num_tab(i), row_text_tab(i));
    END LOOP;

    -- Record end time

```

```

v_end_time := DBMS_UTILITY.GET_TIME;

-- Calculate and display elapsed time
DBMS_OUTPUT.PUT_LINE ('Duration of the FOR LOOP: ' ||
(v_end_time - v_start_time));

-- Record start time
v_start_time := DBMS_UTILITY.GET_TIME;

-- Insert second 100 rows
FORALL i IN 1..100
  INSERT INTO test (row_num, row_text)
  VALUES (row_num_tab(i), row_text_tab(i));

-- Record end time
v_end_time := DBMS_UTILITY.GET_TIME;

-- Calculate and display elapsed time
DBMS_OUTPUT.PUT_LINE ('Duration of the FORALL statement: ' ||
(v_end_time - v_start_time));

COMMIT;
END;

```

To calculate the execution times for the numeric FOR loop and the FORALL statement, the script employs the GET_TIME function defined in the DBMS_UTILITY package that is owned by the Oracle user SYS. The GET_TIME function returns the current time in 100ths of a second. Here is the output produced by the preceding example:

[Click here to view code image](#)

```

Duration of the FOR LOOP: 1
Duration of the FORALL statement: 0

```

SAVE EXCEPTIONS Option

The SAVE EXCEPTIONS option enables the FORALL statement to continue even when a corresponding SQL statement causes an exception. These exceptions are stored in the cursor attribute called SQL%BULK_EXCEPTIONS. The SQL%BULK_EXCEPTIONS attribute is a collection of records in which each record consists of two fields, ERROR_INDEX and ERROR_CODE. The ERROR_INDEX field stores the number of the iteration of the FORALL statement during which an exception was encountered, and the ERROR_CODE stores the Oracle error code corresponding to the raised exception.

The number of exceptions that occurred during the execution of the FORALL statement can be retrieved via SQL%BULK_EXCEPTIONS.COUNT. Although the individual error messages are not saved, they can be looked up via the SQLERRM function.

The following example uses a FORALL statement with the SAVE EXCEPTIONS option.

For Example *ch18_3a.sql*

[Click here to view code image](#)

```
TRUNCATE TABLE TEST;
```

```

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(11)    INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;

  -- Define user-defined exception and associated Oracle
  -- error number with it
  errors EXCEPTION;
  PRAGMA EXCEPTION_INIT(errors, -24381);

  v_rows NUMBER;
BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Modify 1, 5, and 7 elements of the V_ROW_TEXT collection
  -- These rows will cause exceptions in the FORALL statement
  row_text_tab(1) := RPAD(row_text_tab(1), 11, ' ');
  row_text_tab(5) := RPAD(row_text_tab(5), 11, ' ');
  row_text_tab(7) := RPAD(row_text_tab(7), 11, ' ');

  -- Populate TEST table
  FORALL i IN 1..10 SAVE EXCEPTIONS
    INSERT INTO test (row_num, row_text)
    VALUES (row_num_tab(i), row_text_tab(i));
  COMMIT;

EXCEPTION
  WHEN errors
  THEN
    -- Display total number of records inserted in the TEST table
    SELECT count(*)
      INTO v_rows
       FROM test;
    DBMS_OUTPUT.PUT_LINE ('There are '||v_rows||' records in the TEST
table');

    -- Display total number of exceptions encountered
    DBMS_OUTPUT.PUT_LINE ('There were '||SQL%BULK_EXCEPTIONS.COUNT||'
exceptions');

    -- Display detailed exception information
    FOR i in 1.. SQL%BULK_EXCEPTIONS.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE ('Record '||
        SQL%BULK_EXCEPTIONS(i).error_index||' caused error '||i||': '||
        SQL%BULK_EXCEPTIONS(i).error_code||' ||
        SQLERRM(-SQL%BULK_EXCEPTIONS(i).error_code));
    END LOOP;
END;

```

This example declares a user-defined exception and associates it with the ORA-24381 exception. This exception occurs when errors are encountered in an array DML statement.

—in this case, the `INSERT` statement that uses collection elements.

In the execution section of the script, the first, fifth, and seventh elements of the `row_text_tab` collection are expanded to store 11 characters instead of 10, thereby causing exceptions in the `INSERT` statement applied against the `TEST` table. Note the presence of the `SAVE EXCEPTION` clause in the `FORALL` statement. As mentioned earlier, the `SAVE EXCEPTION` clause allows the `FORALL` statement to execute to completion.

The exception-handling section checks how many records were added to the `TEST` table and how many exception records are present in the `SQL%BULK_EXCEPTIONS` collection. The latter task is accomplished by employing the `COUNT` method. In addition, this section of the script displays detailed exception information, such as the record number that caused an exception and the error message associated with this exception.

To display the number of the record that caused an exception, the `error_index` field is referenced in the `DBMS_OUTPUT.PUT_LINE` statement as follows:

[Click here to view code image](#)

```
SQL%BULK_EXCEPTIONS(i).error_index
```

To display the error message itself, the `error_code` field is passed as an input parameter to the `SQLERRM` function. Note that when the `error_code` is passed to the `SQLERRM` function, it is prefixed by a minus sign.

[Click here to view code image](#)

```
SQLERRM( -SQL%BULK_EXCEPTIONS(i).ERROR_CODE )
```

When run, this script produces the following output:

[Click here to view code image](#)

```
There are 7 records in the TEST table
There were 3 exceptions
Record 1 caused error 1: 12899 ORA-12899: value too large for
column  (actual: , maximum: )
Record 5 caused error 2: 12899 ORA-12899: value too large for
column  (actual: , maximum: )
Record 7 caused error 3: 12899 ORA-12899: value too large for
column  (actual: , maximum: )
```

As mentioned previously, adding the `SAVE EXCEPTIONS` clause to the `FORALL` statement enables this statement to execute to completion. As a result, the `INSERT` statement was able to add seven records to the `TEST` table successfully.

INDICES OF Option

As stated previously, the `INDICES OF` option enables you to loop through a sparse collection. Recall that such collection may be a nested table or an associative array. The use of the `INDICES OF` option is illustrated in the following example.

For Example `ch18_4a.sql`

[Click here to view code image](#)

```
TRUNCATE TABLE TEST;
```

```

DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10)   INDEX BY PLS_INTEGER;

    row_num_tab row_num_type;
    row_text_tab row_text_type;

    v_rows NUMBER;
BEGIN
    -- Populate collections
    FOR i IN 1..10
    LOOP
        row_num_tab(i) := i;
        row_text_tab(i) := 'row '||i;
    END LOOP;

    -- Delete 1, 5, and 7 elements of collections
    row_num_tab.DELETE(1); row_text_tab.DELETE(1);
    row_num_tab.DELETE(5); row_text_tab.DELETE(5);
    row_num_tab.DELETE(7); row_text_tab.DELETE(7);

    -- Populate TEST table
    FORALL i IN INDICES OF row_num_tab
        INSERT INTO test (row_num, row_text)
        VALUES (row_num_tab(i), row_text_tab(i));
    COMMIT;

    SELECT COUNT(*)
        INTO v_rows
        FROM test;

    DBMS_OUTPUT.PUT_LINE ('There are '||v_rows||' rows in the TEST table');
END;

```

To make the associative arrays sparse, the first, fifth, and seventh elements are deleted from both collections. As a result, the **FORALL** statement iterates seven times, and seven rows are added to the **TEST** table. This outcome is illustrated by the following output:

[Click here to view code image](#)

There are 7 rows in the TEST table

VALUES OF Option

The **VALUES OF** option specifies that the values of the loop counter in the **FORALL** statement are based on the values of the elements of the specified collection. Essentially, this collection is a group of indices that the **FORALL** statement can loop through. Furthermore, these indices do not need to be unique and can be listed in arbitrary order. The following example demonstrates the use of the **VALUES OF** option.

For Example *ch18_5a.sql*

[Click here to view code image](#)

```

CREATE TABLE TEST_EXC
  (row_num  NUMBER
  ,row_text VARCHAR2(50));

```

```

TRUNCATE TABLE TEST;

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER      INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(11) INDEX BY PLS_INTEGER;
  TYPE exc_ind_type IS TABLE OF PLS_INTEGER  INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;
  exc_ind_tab exc_ind_type;

  -- Define user-defined exception and associated Oracle
  -- error number with it
  errors EXCEPTION;
  PRAGMA EXCEPTION_INIT(errors, -24381);

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Modify 1, 5, and 7 elements of the ROW_TEXT_TAB collection
  -- These rows will cause exceptions in the FORALL statement
  row_text_tab(1) := RPAD(row_text_tab(1), 11, ' ');
  row_text_tab(5) := RPAD(row_text_tab(5), 11, ' ');
  row_text_tab(7) := RPAD(row_text_tab(7), 11, ' ');

  -- Populate TEST table
  FORALL i IN 1..10 SAVE EXCEPTIONS
    INSERT INTO test (row_num, row_text)
    VALUES (row_num_tab(i), row_text_tab(i));
  COMMIT;

EXCEPTION
  WHEN errors
  THEN
    -- Populate EXC_IND_TAB collection to be used in the VALUES OF
    -- clause
    FOR i in 1..SQL%BULK_EXCEPTIONS.COUNT
    LOOP
      exc_ind_tab(i) := SQL%BULK_EXCEPTIONS(i).error_index;
    END LOOP;
    -- Insert records that caused exceptions in the TEST_EXC table
    FORALL i in VALUES OF exc_ind_tab
      INSERT INTO test_exc (row_num, row_text)
      VALUES (row_num_tab(i), row_text_tab(i));
    COMMIT;
END;

```

This script employs the **TEST_EXC** table, which has the same structure as the **TEST** table but expanded data type sizes. The newly created table is used to store records that cause exceptions when they are inserted in the **TEST** table. Next, the new collection data type, **exc_ind_type**, is defined as an associative array of **PLS_INTEGERS**. Finally,

the new collection variable, `exc_ind_tab`, is defined based on the `exc_ind_type`. This new collection is referenced by the `VALUES OF` clause in the exception-handling section of the script.

To cause exceptions in the `FORALL` statement, the first, fifth, and seventh elements of the `row_text_tab` associative array are modified to contain 11 characters instead of 10. Then, in the exception-handling section of the script, the `exc_ind_tab` collection is populated with index values of the rows that caused the exceptions. In this example, these index values are 1, 5, and 7, and they are stored in the `error_index` field of the `SQL%BULK_EXCEPTION` collection. Once the `exc_ind_tab` is populated, it is used to iterate through the `row_num_tab` and `row_test_tab` collections again and insert erroneous records in the `TEST_EXC` table.

When this script is executed, the `TEST` and `TEST_EXC` tables contain the following records:

[Click here to view code image](#)

```
select *
  from test;

  ROW_NUM      ROW_TEXT
----- -----
      2        row 2
      3        row 3
      4        row 4
      6        row 6
      8        row 8
      9        row 9
     10        row 10
```

```
select *
  from test_exc;

  ROW_NUM      ROW_TEXT
----- -----
      1        row 1
      5        row 5
      7        row 7
```

Lab 18.2: The BULK COLLECT Clause

After this lab, you will be able to

- Use the `BULK COLLECT` Clause

The `BULK COLLECT` clause fetches the batches of results and brings them back from the SQL engine to the PL/SQL engine. For example, consider a cursor against the `STUDENT` table that returns the student's ID, first name, and last name. Once this cursor is opened, the rows are fetched one by one until all rows have been processed. Then the cursor is closed. These steps are illustrated in the following example.

For Example *ch18_6a.sql*

[Click here to view code image](#)

```
DECLARE
    CURSOR student_cur IS
        SELECT student_id, first_name, last_name
        FROM student;
BEGIN
    FOR rec IN student_cur
    LOOP
        DBMS_OUTPUT.PUT_LINE ('student_id: '||rec.student_id);
        DBMS_OUTPUT.PUT_LINE ('first_name: '||rec.first_name);
        DBMS_OUTPUT.PUT_LINE ('last_name: '||rec.last_name);
    END LOOP;
END;
```

Recall that the cursor **FOR** loop opens and closes the cursor and fetches cursor records implicitly.

The same task of fetching records from the **STUDENT** table can be accomplished by employing the **BULK COLLECT** clause. The difference here is that the **BULK COLLECT** clause will fetch all rows from the **STUDENT** table at once. Because **BULK COLLECT** fetches multiple rows, these rows are stored in collection variables.

In the following modified version of the preceding example, cursor processing is replaced by the **BULK COLLECT** clause.

For Example *ch18_6b.sql*

[Click here to view code image](#)

```
DECLARE
    -- Define collection type and variables to be used by the
    -- BULK COLLECT clause
    TYPE student_id_type IS TABLE OF student.student_id%TYPE;
    TYPE first_name_type IS TABLE OF student.first_name%TYPE;
    TYPE last_name_type IS TABLE OF student.last_name%TYPE;
    student_id_tab student_id_type;
    first_name_tab first_name_type;
    last_name_tab last_name_type;

BEGIN
    -- Fetch all student data at once via BULK COLLECT clause
    SELECT student_id, first_name, last_name
    BULK COLLECT INTO student_id_tab, first_name_tab, last_name_tab
    FROM student;

    FOR i IN student_id_tab.FIRST..student_id_tab.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE ('student_id: '||student_id_tab(i));
        DBMS_OUTPUT.PUT_LINE ('first_name: '||first_name_tab(i));
        DBMS_OUTPUT.PUT_LINE ('last_name: '||last_name_tab(i));
    END LOOP;
END;
```

This script declares three nested table types and variables. These variables are used to store data returned by the **SELECT** statement with the **BULK COLLECT** clause. Because this version of the script is using the **BULK COLLECT** clause, there is no need to declare

and process a cursor.

Did You Know?

When nested tables are populated via **SELECT** with the **BULK COLLECT** clause, they are initialized and extended automatically. Recall that a nested table must usually be initialized prior to its use by calling a constructor function that has the same name as its nested table type. Once it has been initialized, the nested table must be extended via the **EXTEND** method before the next value can be assigned to it.

To display the data that has been selected into the individual collections, the script loops through them via the numeric **FOR** loop. Notice that the lower and upper limits for the loop counter are specified via the **FIRST** and **LAST** methods.

Did You Know?

The **BULK COLLECT** clause is similar to a cursor loop in that it does not raise a **NO_DATA_FOUND** exception when the **SELECT** statement does not return any records. As a result, it is considered good practice to check whether the resulting collection contains any data.

Because the **BULK COLLECT** clause does not restrict the size of a collection and extends it automatically, it is also a good idea to limit the result set when a **SELECT** statement returns large amounts of data. This can be achieved by using **BULK COLLECT** with a cursor **SELECT** statement and by adding the **LIMIT** option.

For Example *ch18_6c.sql*

[Click here to view code image](#)

```
DECLARE
  CURSOR student_cur IS
    SELECT student_id, first_name, last_name
      FROM student;
  -- Define collection type and variables to be used by the
  -- BULK COLLECT clause
  TYPE student_id_type IS TABLE OF student.student_id%TYPE;
  TYPE first_name_type IS TABLE OF student.first_name%TYPE;
  TYPE last_name_type IS TABLE OF student.last_name%TYPE;

  student_id_tab student_id_type;
  first_name_tab first_name_type;
  last_name_tab last_name_type;

  -- Define variable to be used by the LIMIT clause
  v_limit PLS_INTEGER := 50;

BEGIN
  OPEN student_cur;
  LOOP
    -- Fetch 50 rows at once
    FETCH student_cur
      BULK COLLECT INTO student_id_tab, first_name_tab, last_name_tab
```

```

        LIMIT v_limit;

    EXIT WHEN student_id_tab.COUNT = 0;

    FOR i IN student_id_tab.FIRST..student_id_tab.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE ('student_id: '||student_id_tab(i));
        DBMS_OUTPUT.PUT_LINE ('first_name: '||first_name_tab(i));
        DBMS_OUTPUT.PUT_LINE ('last_name: '||last_name_tab(i));
    END LOOP;
    END LOOP;
    CLOSE student_cur;
END;

```

This version of the script employs the **BULK COLLECT** clause with the **LIMIT** option to fetch 50 rows from the **STUDENT** table at once. In other words, each collection will contain at most 50 records. To accomplish this task, the **BULK COLLECT** clause is used in conjunction with the cursor loop. In this case, the exit condition of the loop is based on the number of records in the collection rather than the **student_cur%NOTFOUND** attribute.

Note how the numeric **FOR** loop that displays information on the screen has been moved inside the cursor loop. This is done because every new batch of 50 records fetched by the **BULK COLLECT** clause will replace the previous batch of 50 records fetched in the previous iteration.

So far, you have seen examples of the **BULK COLLECT** clause fetching data into collections where the underlying elements are simple data types such as **NUMBER** or **VARCHAR2**. However, the **BULK COLLECT** clause can also be used to fetch data into collections of records or objects. Collections of objects are discussed in [Chapter 23](#). In the following modified version of the previous example, student data is fetched into a collection of user-defined records.

For Example *ch18_6d.sql*

[Click here to view code image](#)

```

DECLARE
    CURSOR student_cur IS
        SELECT student_id, first_name, last_name
        FROM student;

    -- Define record type
    TYPE student_rec IS RECORD
        (student_id student.student_id%TYPE,
         first_name student.first_name%TYPE,
         last_name student.last_name%TYPE);

    -- Define collection type
    TYPE student_type IS TABLE OF student_rec;

    -- Define collection variable
    student_tab student_type;

    -- Define variable to be used by the LIMIT clause
    v_limit PLS_INTEGER := 50;

```

```

BEGIN
  OPEN student_cur;
  LOOP
    -- Fetch 50 rows at once
    FETCH student_cur BULK COLLECT INTO student_tab LIMIT v_limit;

    EXIT WHEN student_tab.COUNT = 0;

    FOR i IN student_tab.FIRST..student_tab.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE ('student_id: '||student_tab(i).student_id);
      DBMS_OUTPUT.PUT_LINE ('first_name: '||student_tab(i).first_name);
      DBMS_OUTPUT.PUT_LINE ('last_name: '||student_tab(i).last_name);
    END LOOP;
  END LOOP;
  CLOSE student_cur;
END;

```

In this version of the script, the result set returned by the cursor is fetched into collection of user-defined records, `student_tab`. As a consequence, the `FETCH` statement with the `BULK COLLECTION` option does not need to reference individual record elements.

All versions of this example produce the same output, a portion of which is shown here:

```

student_id: 230
first_name: George
last_name : Kocka
student_id: 232
first_name: Janet
last_name : Jung
student_id: 233
first_name: Kathleen
last_name : Mulroy
student_id: 234
first_name: Joel
last_name : Brendler
...

```

So far you have seen how to use the `BULK COLLECT` clause with the `SELECT` statement. However, oftentimes `BULK COLLECT` is used with `INSERT`, `UPDATE`, and `DELETE` statements. In this case, the `BULK COLLECT` clause may be used in conjunction with the `RETURNING` clause, as shown in the following example.

For Example *ch18_7a.sql*

[Click here to view code image](#)

```

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;

BEGIN
  DELETE FROM test
  RETURNING row_num, row_text

```

```

        BULK COLLECT INTO row_num_tab, row_text_tab;

        DBMS_OUTPUT.PUT_LINE ('Deleted'||SQL%ROWCOUNT||' rows:');

        FOR i IN row_num_tab.FIRST..row_num_tab.LAST
        LOOP
            DBMS_OUTPUT.PUT_LINE
            ('row_num ='||row_num_tab(i)||' row_text = '||row_text_tab(i));
        END LOOP;

        COMMIT;
    END;

```

This script deletes records from the TEST table created and populated in [Lab 18.1](#). The DELETE statement returns the ROW_NUM and ROW_TEXT values via the RETURNING clause. These values are then fetched by the BULK COLLECT clause into two collections, row_num_tab and row_text_tab. Next, to display the data that has been fetched into the individual collections, they are looped through via the numeric FOR loop.

When run, this script produces the following output:

[Click here to view code image](#)

```

Deleted 7 rows:
row_num = 2 row_text = row 2
row_num = 3 row_text = row 3
row_num = 4 row_text = row 4
row_num = 6 row_text = row 6
row_num = 8 row_text = row 8
row_num = 9 row_text = row 9
row_num = 10 row_text = row 10

```

As mentioned previously, the BULK COLLECT clause is similar to the cursor loop in that it does not generate a NO_DATA_FOUND exception when no rows are returned by the SELECT statement. This is illustrated by the following example.

For Example ch18_8a.sql

[Click here to view code image](#)

```

DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER           INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

    row_num_tab row_num_type;
    row_text_tab row_text_type;

BEGIN
    SELECT row_num, row_text
    BULK COLLECT INTO row_num_tab, row_text_tab
    FROM test;

    FOR i IN row_num_tab.FIRST..row_num_tab.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE
        ('row_num ='||row_num_tab(i)||' row_text = '||row_text_tab(i));
    END LOOP;
END;

```

In this example, the data is selected from the TEST table and populated into two collections, `row_num_tab` and `row_text_tab`. This is accomplished via the `BULK COLLECT` clause. Next, the collection data is displayed on the screen through the numeric `FOR` loop, with the values returned by the `row_num_tab.FIRST` and `row_num_tab.LAST` methods being used as lower and upper bounds of the loop.

At first glance, this example seems very similar to the example `ch18_6b.sql` that appeared earlier in this lab, as it follows the same steps. First, collection types and variables are declared. Second, the data is selected in the collection variables. Third, the data in the collection variables is displayed on the screen. However, when it is run, this script raises the following exception:

[Click here to view code image](#)

```
ORA-06502: PL/SQL: numeric or value error  
ORA-06512: at line 14
```

This error is caused by the

[Click here to view code image](#)

```
FOR i IN row_num_tab.FIRST..row_num_tab.LAST
```

statement, as the collection variables do not have any data in them. This situation occurs because the data from the TEST table was deleted in the `ch18_7a.sql` example. To remedy this problem, the example should be modified as follows (affected statements are shown in bold):

For Example `ch18_8b.sql`

[Click here to view code image](#)

```
DECLARE  
  -- Define collection types and variables  
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;  
  TYPE row_text_type IS TABLE OF VARCHAR2(10)    INDEX BY PLS_INTEGER;  
  row_num_tab row_num_type;  
  row_text_tab row_text_type;  
  
BEGIN  
  SELECT row_num, row_text  
  BULK COLLECT INTO row_num_tab, row_text_tab  
  FROM test;  
  
  IF row_num_tab.COUNT != 0  
  THEN  
    FOR i IN row_num_tab.FIRST..row_num_tab.LAST  
    LOOP  
      DBMS_OUTPUT.PUT_LINE  
      ('row_num = '||row_num_tab(i)||' row_text = ' ||row_text_tab(i));  
    END LOOP;  
  ELSE  
    DBMS_OUTPUT.PUT_LINE ('row_num_tab.COUNT = '||row_num_tab.COUNT);  
    DBMS_OUTPUT.PUT_LINE ('row_text_tab.COUNT = '||row_text_tab.COUNT);  
  END IF;  
END;
```

When run, this version of the script does not cause any exception. The `IF` statement evaluates to `FALSE` when the `COUNT` method returns 0, as illustrated by the output:

```
row_num_tab.COUNT = 0
row_text_tab.COUNT = 0
```

Throughout this chapter, you have seen how to use the FORALL statement and BULK COLLECT clause. Now we will consider an example that combines both techniques. This example uses the MY_ZIPCODE table, which is created based on the ZIPCODE table. Note that the CREATE TABLE statement creates an empty table because the criteria specified in the WHERE clause do not return any records.

For Example *ch18_9a.sql*

[Click here to view code image](#)

```
CREATE TABLE my_zipcode AS
SELECT *
  FROM zipcode
 WHERE 1 = 2;

DECLARE
  -- Declare collection types
  TYPE string_type IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
  TYPE date_type   IS TABLE OF DATE           INDEX BY PLS_INTEGER;

  -- Declare collection variables to be used by the FORALL statement
  zip_tab      string_type;
  city_tab     string_type;
  state_tab    string_type;
  cr_by_tab    string_type;
  cr_date_tab  date_type;
  mod_by_tab   string_type;
  mod_date_tab date_type;

  v_rows INTEGER := 0;
BEGIN
  -- Populate individual collections
  SELECT *
    BULK COLLECT INTO zip_tab, city_tab, state_tab, cr_by_tab,
                     cr_date_tab, mod_by_tab, mod_date_tab
    FROM zipcode
   WHERE state = 'CT';

  -- Populate MY_ZIPCODE table
  FORALL i IN 1..zip_tab.COUNT
    INSERT INTO my_zipcode
      (zip, city, state, created_by, created_date, modified_by,
       modified_date)
      VALUES
        (zip_tab(i), city_tab(i), state_tab(i), cr_by_tab(i),
         cr_date_tab(i), mod_by_tab(i), mod_date_tab(i));
  COMMIT;

  -- Check how many records were added to MY_ZIPCODE table
  SELECT COUNT(*)
    INTO v_rows
    FROM my_zipcode;

  DBMS_OUTPUT.PUT_LINE (v_rows||' records were added to MY_ZIPCODE
table');
END;
```

This script populates the MY_ZIPCODE table with the records selected from the ZIPCODE table. To enable use of the BULK COLLECT and FORALL statements, it employs seven collections. Note that there are only two collection types associated with these seven collections. This is because the individual collections store only two data types, VARCHAR2 and DATE. When run, this example produces the following output:

[Click here to view code image](#)

```
19 records were added to MY_ZIPCODE table
```

Next, consider another example where the FORALL statement and BULK COLLECT clause are used together with the DELETE statement. In this example, the records from the MY_ZIPCODE table are deleted for a few ZIP codes, and the corresponding city names along with the deleted ZIP codes are stored in the city_tab and zip_tab collections, respectively.

For Example ch18_10a.sql

[Click here to view code image](#)

```
DECLARE
    -- Declare collection types
    TYPE string_type IS TABLE OF VARCHAR2(100);

    -- Declare collection variables to be used by the FORALL statement
    -- and BULK COLLECT clause
    zip_codes string_type := string_type ('06401', '06455', '06483',
    '06520', '06605');
    zip_tab   string_type;
    city_tab  string_type;

    v_rows INTEGER := 0;
BEGIN
    -- Delete some records from MY_ZIPCODE table
    FORALL i in zip_codes.FIRST..zip_codes.LAST
        DELETE FROM my_zipcode
        WHERE zip = zip_codes(i)
        RETURNING zip, city
        BULK COLLECT INTO zip_tab, city_tab;
    COMMIT;

    DBMS_OUTPUT.PUT_LINE ('The following records were deleted from
    MY_ZIPCODE table:');
    FOR i in zip_tab.FIRST..zip_tab.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE ('Zip code '||zip_tab(i)||', city
        '||city_tab(i));
    END LOOP;
END;
```

In this script, the FORALL statement runs the DELETE statement for a given list of ZIP code values stored in the zip_codes collection. Also, the DELETE statement contains the RETURNING clause with the BULK COLLECT clause, which in turn stores ZIP codes and city names in the zip_tab and city_tab collections, respectively. Finally, the numeric FOR loop is used to display the data stored in the zip_tab and city_tab collections, as illustrated by the output from the script:

[Click here to view code image](#)

```
The following records were deleted from MY_ZIPCODE table:  
Zip code 06401, city Ansonia  
Zip code 06455, city Middlefield  
Zip code 06483, city Oxford  
Zip code 06520, city New Haven  
Zip code 06605, city Bridgeport
```

Lab 18.3: Binding Collections in SQL Statements

After this lab, you will be able to

- [Bind Collections When Using EXECUTE IMMEDIATE Statements](#)
- [Bind Collections When Using OPEN-FOR, FETCH, and CLOSE Statements](#)

As mentioned previously, the ability to bind collection data types when employing dynamic SQL has been added in Oracle 12c. Recall that dynamic SQL was covered in [Chapter 17](#), where you learned how to use the EXECUTE IMMEDIATE statement and OPEN-FOR, FETCH, and CLOSE statements.

Binding Collections with EXECUTE IMMEDIATE Statements

In [Chapter 17](#), you saw numerous examples of the EXECUTE IMMEDIATE statement. All of these examples have one thing in common: The data types of the bind variables are known SQL types. In other words, these data types are all supported by SQL, such as NUMBER and VARCHAR2. In Oracle 12c, it is possible to use bind variables based on the collection and record types, albeit with one restriction applied: *The collection or record data type must be declared in the package specification.*

Consider the package called TEST_ADM_PKG, shown in [Listing 18.3](#). This package contains definitions of two collection types and three procedures that insert, update, and delete records from the TEST table. (Procedures, functions, and packages are covered in detail in [Chapters 19](#) through [21](#).)

Listing 18.3 TEST_ADM_PKG Package with Collection Types

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE test_adm_pkg
AS
    -- Define collection types
    TYPE row_num_type IS TABLE OF NUMBER           INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10)     INDEX BY PLS_INTEGER;

    -- Define procedures
    PROCEDURE populate_test (row_num_tab ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE);

    PROCEDURE update_test (row_num_tab ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE);

    PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE);
END test_adm_pkg;
```

```

/
CREATE OR REPLACE PACKAGE BODY test_adm_pkg
AS
    PROCEDURE populate_test (row_num_tab ROW_NUM_TYPE
                            ,row_num_type ROW_TEXT_TYPE)
    IS
    BEGIN
        FORALL i IN 1..10
            INSERT INTO test (row_num, row_text)
            VALUES (row_num_tab(i), row_num_type(i));
    END populate_test;

    PROCEDURE update_test (row_num_tab ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE)
    IS
    BEGIN
        FORALL i IN 1..10
            UPDATE test
            SET row_text = row_num_type(i)
            WHERE row_num = row_num_tab(i);
    END update_test;

    PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE)
    IS
    BEGIN
        FORALL i IN 1..10
            DELETE from test
            WHERE row_num = row_num_tab(i);
    END delete_test;

END test_adm_pkg;
/

```

This package has both a package specification and a package body. The package specification contains declarations of two associative array types (ROW_NUM_TYPE and ROW_TEXT_TYPE) and three procedures (POPULATE_TEST, UPDATE_TEST, and DELETE_TEST). Each procedure has parameters ROW_NUM_TAB and ROW_TEXT_TAB that are based on the collection types defined in this package. The package body contains the code for the procedures declared in the package specification.

The following example uses this newly created package. References to the package objects are highlighted in bold.

For Example *ch18_11a.sql*

[Click here to view code image](#)

```

DECLARE
    row_num_tab test_adm_pkg.row_num_type;
    row_text_tab test_adm_pkg.row_text_type;

    v_rows NUMBER;

BEGIN
    -- Populate collections
    FOR i IN 1..10
    LOOP
        row_num_tab(i) := i;

```

```

    row_text_tab(i) := 'row '||i;
END LOOP;

-- Delete previously added data from the TEST table
test_adm_pkg.delete_test (row_num_tab);

-- Populate TEST table
test_adm_pkg.populate_test (row_num_tab, row_text_tab);
COMMIT;

-- Check how many rows where inserted in the TEST table
-- and display this number on the screen
SELECT COUNT(*)
  INTO v_rows
  FROM TEST;

DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' rows in the TEST table');
END;

```

This example is very similar to the example ch18_1a.sql used in [Lab 18.1](#). It populates the TEST table, checks how many records were added to the TEST table, and displays this information on the screen. The main difference is that it references the TEST_ADM_PKG package when declaring two collection variables and it calls the DELETE_TEST and POPULATE_TEST procedures to delete previously added records to the TEST table and repopulate the table with the new data. Note that all of the references to the packaged objects are prefixed by the package name.

When run, this example produces the following output:

[Click here to view code image](#)

```
There are 10 rows in the TEST table
```

Now consider a modified version of this example where calls to the procedures DELETE_TEST and POPULATE_TEST are embedded in the dynamic SQL. All changes are shown in bold.

For Example ch18_11b.sql

[Click here to view code image](#)

```

DECLARE
  row_num_tab  test_adm_pkg.row_num_type;
  row_text_tab test_adm_pkg.row_text_type;

  v_dyn_sql VARCHAR2(1000);
  v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Delete previously added data from the TEST table
  v_dyn_sql := 'begin test_adm_pkg.delete_test (:row_num_tab); end;';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab;

```

```

-- Populate TEST table
v_dyn_sql := 'begin test_adm_pkg.populate_test (:row_num_tab,
:row_text_tab); end;';
EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab, row_text_tab;
COMMIT;

-- Check how many rows where inserted in the TEST table
-- display it on the screen
SELECT COUNT(*)
  INTO v_rows
  FROM TEST;

DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' rows in the TEST table');
END;

```

This version of the script declares a new variable, `v_dyn_sql`, that is used to store dynamic SQL statement. Next, the calls to the `DELETE_TEST` and `POPULATE_TEST` procedures are replaced by the dynamic SQL statements that are executed by the `EXECUTE IMMEDIATE` statement.

Notice the syntax of the dynamic SQL statements. In the original example, the packaged procedures are invoked in this way:

[Click here to view code image](#)

- Delete previously added data from the TEST table
`test_adm_pkg.delete_test (row_num_tab);`
- Populate TEST table
`test_adm_pkg.populate_test (row_num_tab, row_text_tab);`

In the modified version of the example, calls to these procedures are placed between the `BEGIN` and `END` statements:

[Click here to view code image](#)

- Delete previously added data from the TEST table
`v_dyn_sql := 'begin test_adm_pkg.delete_test (:row_num_tab); end;';`
`EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab;`
- Populate TEST table
`v_dyn_sql := 'begin test_adm_pkg.populate_test (:row_num_tab,`
`:row_text_tab); end;';`
`EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab, row_text_tab;`

This approach is used because each dynamic SQL statement is executed as an anonymous PL/SQL block and, therefore, requires `BEGIN` and `END` statements. If these `BEGIN` and `END` statements are omitted from the dynamic SQL, the script will not execute successfully. This is illustrated by the following example (affected statements are shown in bold):

For Example *ch18_11c.sql*

[Click here to view code image](#)

```

DECLARE
  row_num_tab  test_adm_pkg.row_num_type;
  row_text_tab test_adm_pkg.row_text_type;

```

```

v_dyn_sql VARCHAR2(1000);
v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Delete previously added data from the TEST table
  v_dyn_sql := 'test_adm_pkg.delete_test (:row_num_tab);';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab;

  -- Populate TEST table
  v_dyn_sql := 'test_adm_pkg.populate_test (:row_num_tab,
:row_text_tab)';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab, row_text_tab;
  COMMIT;

  -- Check how many rows where inserted in the TEST table
  -- display it on the screen
  SELECT COUNT(*)
    INTO v_rows
   FROM TEST;

  DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' rows in the TEST table');
END;

```

This version of the script produces the following error:

[Click here to view code image](#)

```

ORA-00900: invalid SQL statement
ORA-06512: at line 18

```

As mentioned previously, starting with Oracle 12c you can also use bind variables based on the record types. Similar to the collection types, the record types must be defined in the package specification. Consider the modified version of **TEST_ADM_PKG** shown in [Listing 18.4](#). The package now contains a definition of a record type and a new procedure that populates a record variable from the TEST table using the newly created record type. Newly added items are shown in bold.

Listing 18.4 TEST_ADM_PKG Package with Record Type

[Click here to view code image](#)

```

CREATE OR REPLACE PACKAGE test_adm_pkg
AS
  -- Define collection types
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

  -- Define record type
  TYPE rec_type IS RECORD
    (row_num NUMBER
     ,row_text VARCHAR2(10));

  -- Define procedures

```

```

PROCEDURE populate_test (row_num_tab  ROW_NUM_TYPE
                        ,row_num_type ROW_TEXT_TYPE);

PROCEDURE update_test (row_num_tab  ROW_NUM_TYPE
                        ,row_num_type ROW_TEXT_TYPE);

PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE);

PROCEDURE populate_test_rec (row_num_val IN NUMBER
                             ,test_rec    OUT REC_TYPE);
END test_adm_pkg;
/

CREATE OR REPLACE PACKAGE BODY test_adm_pkg
AS
  PROCEDURE populate_test (row_num_tab  ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE);
  IS
  BEGIN
    FORALL i IN 1..10
      INSERT INTO test (row_num, row_text)
      VALUES (row_num_tab(i),row_num_type(i));
  END populate_test;

  PROCEDURE update_test (row_num_tab  ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE);
  IS
  BEGIN
    FORALL i IN 1..10
      UPDATE test
        SET row_text = row_num_type(i)
      WHERE row_num = row_num_tab(i);
  END update_test;

  PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE)
  IS
  BEGIN
    FORALL i IN 1..10
      DELETE from test
      WHERE row_num = row_num_tab(i);
  END delete_test;
  PROCEDURE populate_test_rec (row_num_val IN NUMBER
                             ,test_rec    OUT REC_TYPE)
  IS
  BEGIN
    SELECT *
      INTO test_rec
      FROM test
     WHERE row_num = row_num_val;
  END populate_test_rec;

END test_adm_pkg;
/

```

This version of the package contains definitions of the user-defined record type, `rec_type`, and a new procedure, `POPULATE_TEST_REC`, that selects a record from the `TEST` table into the `test_rec` parameter based on the `row_num` value provided at run time. Note the `IN` and `OUT` parameter modes specified in the

`POPULATE_TEST_REC` procedure header: They denote that the `row_num_val` parameter is used to pass a value into the procedure, and that the `test_rec` parameter is used to pass a value from the procedure. (Parameter modes are covered in detail in [Chapters 19](#) through [21](#).)

The next example uses the newly created record type and procedure to display a record from the TEST table.

For Example *ch18_12a.sql*

[Click here to view code image](#)

```
DECLARE
    test_rec test_adm_pkg.rec_type;

    v_dyn_sql VARCHAR2(1000);

BEGIN
    -- Select record from the TEST table
    v_dyn_sql := 'begin test_adm_pkg.populate_test_rec (:val, :rec); end;';
    EXECUTE IMMEDIATE v_dyn_sql USING IN 10, OUT test_rec;
    COMMIT;

    -- Display newly selected record
    DBMS_OUTPUT.PUT_LINE ('test_rec.row_num = '||test_rec.row_num);
    DBMS_OUTPUT.PUT_LINE ('test_rec.row_text = '||test_rec.row_text);
END;
```

In this example, the `USING` clause in the `EXECUTE IMMEDIATE` statement contains parameter modes. This is done to ensure that the variables used by the `EXECUTE IMMEDIATE` statement have the same modes as the parameters in the procedure. When run, this example produces the following output:

[Click here to view code image](#)

```
test_rec.row_num = 10
test_rec.row_text = row 10
```

Binding Collections with OPEN-FOR, FETCH, and CLOSE Statements

Recall that the OPEN-FOR, FETCH, and CLOSE statements are used with multirow queries or cursors. This is illustrated by the example below.

For Example *ch18_13a.sql*

[Click here to view code image](#)

```
DECLARE
    TYPE student_cur_typ IS REF CURSOR;

    student_cur student_cur_typ;
    student_rec student%ROWTYPE;

    v_zip_code student.zip%TYPE := '06820';

BEGIN
    OPEN student_cur
```

```

FOR 'SELECT * FROM student WHERE zip = :my_zip' USING v_zip_code;

LOOP
  FETCH student_cur INTO student_rec;
  EXIT WHEN student_cur%NOTFOUND;

  -- Display student ID, first and last names
  DBMS_OUTPUT.PUT_LINE ('student_rec.student_id =
'||student_rec.student_id);
  DBMS_OUTPUT.PUT_LINE ('student_rec.first_name =
'||student_rec.first_name);
  DBMS_OUTPUT.PUT_LINE
('student_rec.last_name = '||student_rec.last_name);
END LOOP;
CLOSE student_cur;
END;

```

The declaration portion of this script specifies the cursor type, `student_cur_typ`, defined as `REF CURSOR`, and a cursor variable, `student_cur`, based on this type. Next, it defines a record variable, `student_rec`, based on the `STUDENT` table.

The executable portion of the script associates the `SELECT` statement with the `STUDENT` table for a given ZIP code with the `student_cur` variable and opens it. Next, each row returned by the `SELECT` statement is fetched into the `student_rec` variable and the student's ID, first name, and last name are displayed on the screen. Once all the records returned by the `SELECT` statement are fetched, the cursor terminates.

When run, this example produces the following output:

[Click here to view code image](#)

```

student_rec.student_id = 240
student_rec.first_name = Z.A.
student_rec.last_name = Scrittore
student_rec.student_id = 326
student_rec.first_name = Piotr
student_rec.last_name = Padel
student_rec.student_id = 360
student_rec.first_name = Calvin
student_rec.last_name = Kiraly

```

Next, consider a modified version of this example where all student records for a given ZIP code are fetched at once into a collection of records. Recall that to bind a collection or a record type, the following restriction must be respected: *The collection or record data type must be declared in the package specification.* To comply with this rule, the `STUDENT_ADMIN_PKG` package shown in [Listing 18.5](#) is created specifically for this purpose.

Listing 18.5 STUDENT_ADMIN_PKG Package with Record and Collection Types

[Click here to view code image](#)

```

CREATE OR REPLACE PACKAGE student_adm_pkg
AS
  -- Define collection type
  TYPE student_tab_type IS TABLE OF student%ROWTYPE INDEX BY PLS_INTEGER;

  -- Define procedures

```

```

PROCEDURE populate_student_tab (zip_code      IN VARCHAR2
                               ,student_tab OUT student_tab_type);

PROCEDURE display_student_info (student_rec student%ROWTYPE);

END student_adm_pkg;
/

CREATE OR REPLACE PACKAGE BODY student_adm_pkg
AS
    PROCEDURE populate_student_tab (zip_code      IN VARCHAR2
                                    ,student_tab OUT student_tab_type)

    IS
        BEGIN
            SELECT *
            BULK COLLECT INTO student_tab
            FROM student
            WHERE zip = zip_code;
        END populate_student_tab;

    PROCEDURE display_student_info (student_rec student%ROWTYPE)
    IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE ('student_rec.zip =      '||student_rec.zip);
            DBMS_OUTPUT.PUT_LINE ('student_rec.student_id = '
            '||student_rec.student_id);
            DBMS_OUTPUT.PUT_LINE ('student_rec.first_name = '
            '||student_rec.first_name);
            DBMS_OUTPUT.PUT_LINE ('student_rec.last_name = '
            '||student_rec.last_name);
        END display_student_info;

    END student_adm_pkg;
/

```

The package specification declares the associative array type, `student_tab_type`, where each element of the collection is a record of the `student%ROWTYPE`. Next, it declares the `populate_student_tab` procedure, which accepts the value of a ZIP code and returns a collection of records, `student_tab`, populated via the `BULK COLLECT INTO` statement. Note the parameter modes specified in the procedure declaration. The `zip_code` is specified as an `IN` parameter. Based on its value, the `OUT` parameter `student_tab` is populated via the `SELECT` statement from the `STUDENT` table.

The second procedure, `display_student_info`, accepts one input parameter, `student_rec` based on the `STUDENT` table and displays the student's ZIP code, ID, first name, and last name on the screen.

The package body contains the executable code of the `populate_student_tab` and `display_student_info` procedures.

The following modified version of example `ch18_13a.sql` employs this newly created package. References to packages objects are highlighted in bold.

For Example `ch18_13b.sql`

[Click here to view code image](#)

```
DECLARE
    TYPE student_cur_typ IS REF CURSOR;
    student_cur student_cur_typ;

    -- Collection and record variables
    student_tab student_adm_pkg.student_tab_type;
    student_rec student%ROWTYPE;

BEGIN
    -- Populate collection of records
    student_adm_pkg.populate_student_tab ('06820', student_tab);

    OPEN student_cur
        FOR 'SELECT * FROM TABLE(:my_table)' USING student_tab;

    LOOP
        FETCH student_cur INTO student_rec;
        EXIT WHEN student_cur%NOTFOUND;

        student_adm_pkg.display_student_info (student_rec);
    END LOOP;
    CLOSE student_cur;
END;
```

This version of the script declares a collection of records, `student_tab`, based on the collection type defined in the `STUDENT_ADM_PKG` package. The execution section of the script populates the `student_tab` collection with the records from the `STUDENT` table for a particular ZIP code. This is accomplished by calling the `populate_student_tab` procedure defined in the `STUDENT_ADM_PKG` package. Next, the student records are selected from the newly populated `student_tab` collection. Note the usage of the built-in `TABLE` function in the `SELECT` statement.

Did You Know?

The `TABLE` function allows you to query a collection like a physical database table. Essentially, it accepts a collection as its input parameter and returns the appropriate result set based on the `SELECT` statement. Note that an input parameter can also be a `REF CURSOR`.

When run, this version of the script produces the following output:

[Click here to view code image](#)

```
student_rec.zip      = 06820
student_rec.student_id = 240
student_rec.first_name = Z.A.
student_rec.last_name = Scrittorale
student_rec.zip      = 06820
student_rec.student_id = 326
student_rec.first_name = Piotr
student_rec.last_name = Padel
student_rec.zip      = 06820
student_rec.student_id = 360
student_rec.first_name = Calvin
student_rec.last_name = Kiraly
```

Summary

In this chapter, you learned how to optimize PL/SQL code with features known as bulk SQL. Fundamentally, you discovered how to batch SQL statements and their results so as to minimize the performance overhead associated with the number of context switches between the PL/SQL and SQL engines. Specifically, you learned about the **FORALL** statement and the **BULK COLLECT** clause. In addition, you learned that starting with Oracle 12c, you can employ bulk SQL and collection data types along with dynamic SQL.

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.

19. Procedures

In this chapter, you will learn about

- [Creating Procedures](#)
- [Passing Parameters IN and OUT of Procedures](#)

All the PL/SQL that you have written up to this point has been anonymous blocks that were run as scripts and compiled by the database server at run time. Now you will begin to use modular code. Modular code is a methodology to build a program from distinct parts (modules), each of which performs a specific function or task toward the final objective of the program. Once modular code is stored on the database server, it becomes a database object, or subprogram, that is available to other program units for repeated execution. To save code into the database, the source code needs to be sent to the server so that it can be compiled into p-code and stored in the database. This process will be covered in the following three chapters. This chapter is short: It simply introduces stored procedures. [Chapter 20](#) covers the basics of stored functions, and [Chapter 21](#) is a lengthy chapter that pulls all the material together to cover packages.

In the first lab of this chapter, you will learn more about stored code and discover how to write one type of stored code known as procedures. In the second lab, you will learn about passing parameters into and out of procedures. Prior to covering the details of stored procedures, you will be introduced to the benefits of module code.

Benefits of Modular Code

A PL/SQL module is any complete logical unit of work. There are five types of PL/SQL modules: (1) anonymous blocks that are run with a text script (the type you have used until now), (2) procedures, (3) functions, (4) packages, and (5) triggers. There are two main benefits to using modular code: (1) It is more reusable and (2) it is more manageable.

You create a procedure either in SQL*Plus or in one of the many tools for creating and debugging stored PL/SQL code. If you are using SQL*Plus, you will need to write your code in a text editor and then run it at the SQL*Plus prompt.

Block Structure

The same block structure is used for all the module types. The block begins with a header (for named blocks only), which consists of (1) the name of the module and (2) a parameter list (if used).

The declaration section defines variables, cursors, and sub-blocks that will be needed in the next section.

The main part of the module is the execution section, where all of the calculations and processing are performed. This will contain executable code such as IF-THEN-ELSE statements, loops, calls to other PL/SQL modules, and so on.

The last section of the module is an optional exception handler, which contains the code to handle exceptions.

Anonymous Blocks

Until this chapter, you have written only anonymous blocks. Anonymous blocks are very much like modules, except that anonymous blocks do not have headers. There are important distinctions, though. As the name implies, anonymous blocks have no names and, therefore, cannot be called by another block. They are not stored in the database and must be compiled and then run each time the script is loaded.

The PL/SQL block in a subprogram is a named block that can accept parameters and can be invoked from an application that can communicate with the Oracle database server. A subprogram can be compiled and stored in the database. This allows the programmer to reuse the program. It also provides for easier maintenance of code. Subprograms may be either procedures or functions.

Lab 19.1: Creating Procedures

After this lab, you will be able to

- [Put Procedure Creation Syntax into Practice](#)
- [Query the Data Dictionary for Information on Procedures](#)

A procedure is a module performing one or more actions; it does not need to return any values. The syntax for creating a procedure is as follows:

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE name
  [(parameter[, parameter, ...])]

AS
  [local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
```

A procedure may have zero to many parameters (this topic is covered in [Lab 19.2](#)). Every procedure has two parts: (1) the header portion, which comes before the AS (or sometimes IS—they are interchangeable) keyword and contains the procedure name and the parameter list, and (2) the body, which is everything after the AS (IS) keyword. The word REPLACE is optional. When this keyword is not included in the header of the procedure, to change the code in the procedure, you must first drop the procedure and then recreate it. Because it is very common to change the code of the procedure, especially when it is under development, it is strongly recommended that you use the OR REPLACE option.

Putting Procedure Creation Syntax into Practice

The following script demonstrates the syntax for creating a procedure. When this script is run, it creates a procedure named **Discount** that is compiled into p-code and stored in the database for later execution.

For Example *ch19_1.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE Discount
AS
  CURSOR c_group_discount
  IS
    SELECT distinct s.course_no, c.description
      FROM section s, enrollment e, course c
     WHERE s.section_id = e.section_id
       AND c.course_no = s.course_no
    GROUP BY s.course_no, c.description,
             e.section_id, s.section_id
   HAVING COUNT(*) >=8;
BEGIN
  FOR r_group_discount IN c_group_discount
  LOOP
    UPDATE course
      SET cost = cost * .95
     WHERE course_no = r_group_discount.course_no;
    DBMS_OUTPUT.PUT_LINE
      ('A 5% discount has been given to ' ||
       r_group_discount.course_no||' ' ||
       r_group_discount.description
      );
  END LOOP;
END;
```

To execute the stored procedure **Discount**, the following syntax is used:

```
EXECUTE Procedure_name
```

Executing the **Discount** procedure yields the following result:

[Click here to view code image](#)

```
5% discount has been given to 25 Adv. Word Perfect
.... (through each course with an enrollment over 8)
PL/SQL procedure successfully completed.
```

There is no **COMMIT** in this procedure, which means the procedure will not update the database. A **COMMIT** command needs to be issued after the procedure is run, if you want the changes to be made. Alternatively, you can enter a **COMMIT** command either before or after the end loop. If you put the **COMMIT** statement before the end loop, then you are committing changes after every loop. If you put the **COMMIT** statement after the end loop, then the changes will not be committed until after the procedure is near completion. It is wiser to follow the second option, as it leaves you better prepared for handling errors.

By the Way

If you receive an error in SQL*Plus, type the following command:

Show error

You can also add to the command,

[Click here to view code image](#)

L start_line_number end_line_number

to see a portion of the code and isolate errors.

Querying the Data Dictionary for Information on Procedures

Two main views in the data dictionary provide information on stored code: the `USER_OBJECTS` view, which gives information about the objects, and the `USER_SOURCE` view, which gives the text of the source code. The data dictionary also has `ALL_` and `DBA_` versions of these views.

The following `SELECT` statement gets pertinent information from the `USER_OBJECTS` view about the `Discount` procedure you just wrote:

[Click here to view code image](#)

```
SELECT object_name, object_type, status
  FROM user_objects
 WHERE object_name = 'DISCOUNT';
```

The result would be the following, assuming the only object in the database is the new `Discount` procedure:

[Click here to view code image](#)

OBJECT_NAME	OBJECT_TYPE	STATUS
DISCOUNT	PROCEDURE	VALID

The status indicates where the procedure was compiled successfully. An invalid procedure cannot be executed.

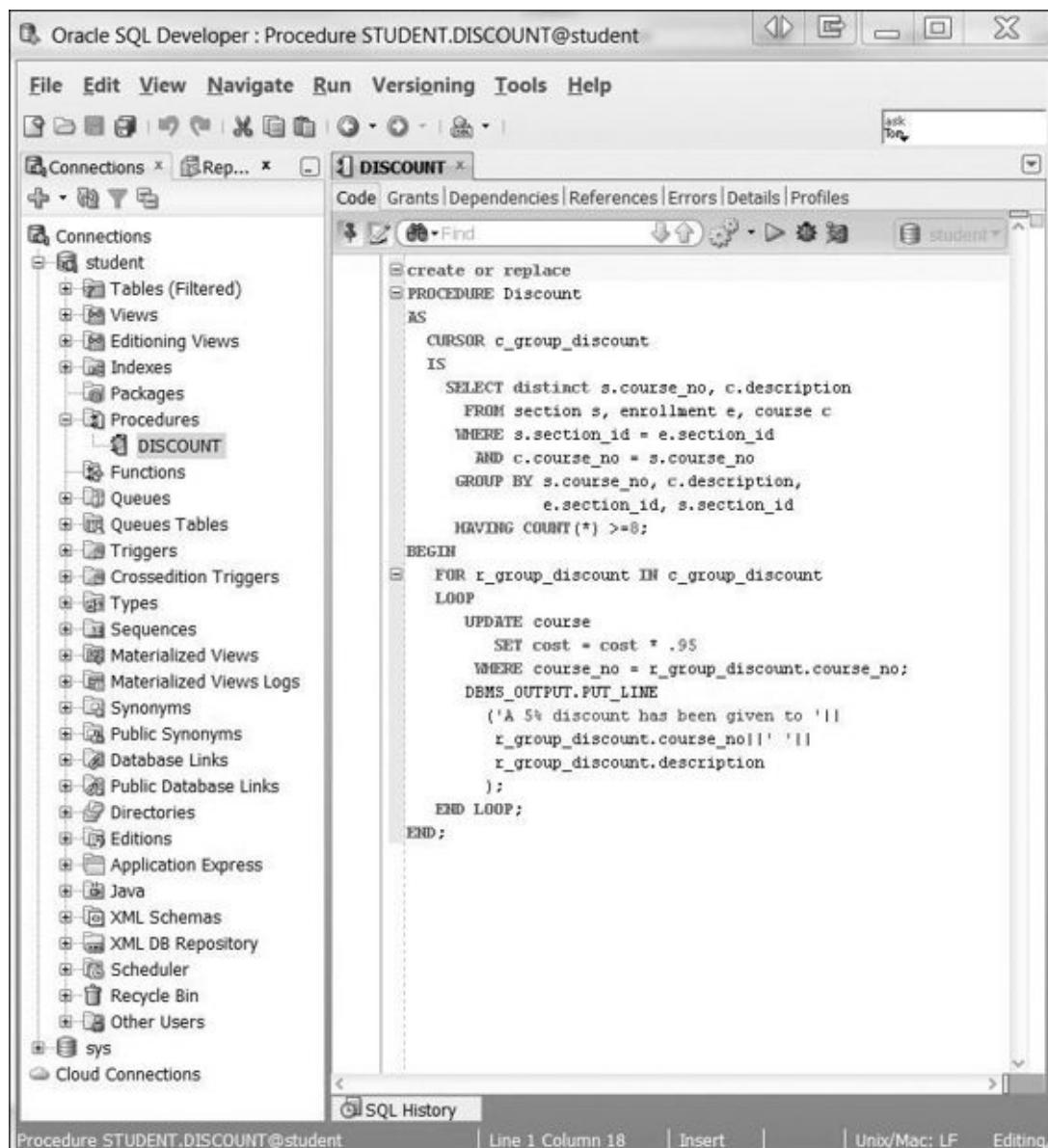
The following `SELECT` statement displays the source code from the `USER_SOURCE` view for the `Discount` procedure:

[Click here to view code image](#)

```
SELECT TO_CHAR(line, 99)||'>', text
  FROM user_source
 WHERE name = 'DISCOUNT'
```

Stored procedures in the database can also be seen in Oracle SQL Developer. If you expand the nodes under the appropriate database connection, you will see under the Procedure node all procedures in the database for the user specified in the database connection. The node will show both valid and invalid procedures. [Figure 19.1](#), for instance, shows the valid `Discount` procedure. The default tab that opens shows the code; the code can be modified and recompiled in this tab. Additionally, tabs on grants and

dependencies are available. If the procedure was invalid, it will have a red X next to it. There is also a tool in Oracle SQL Developer that can be utilized to write new procedures, which can be accessed by right-clicking on the Procedure node.



The screenshot shows the Oracle SQL Developer interface. The title bar reads "Oracle SQL Developer : Procedure STUDENT.DISCOUNT@student". The menu bar includes File, Edit, View, Navigate, Run, Versioning, Tools, and Help. The toolbar has various icons for file operations. The left sidebar shows a tree view of database objects under "Connections" (student), including Tables (Filtered), Views, Editioning Views, Indexes, Packages, Procedures (with "DISCOUNT" selected), Functions, Queues, Queue Tables, Triggers, Crossedition Triggers, Types, Sequences, Materialized Views, Materialized Views Logs, Synonyms, Public Synonyms, Database Links, Public Database Links, Directories, Editions, Application Express, Java, XML Schemas, XML DB Repository, Scheduler, Recycle Bin, Other Users, sys, and Cloud Connections. The main pane displays the "DISCOUNT" procedure code:

```
create or replace
PROCEDURE Discount
AS
CURSOR c_group_discount
IS
SELECT distinct s.course_no, c.description
  FROM section s, enrollment e, course c
 WHERE s.section_id = e.section_id
   AND c.course_no = s.course_no
 GROUP BY s.course_no, c.description,
          e.section_id, s.section_id
 HAVING COUNT(*) >=8;
BEGIN
FOR r_group_discount IN c_group_discount
LOOP
  UPDATE course
    SET cost = cost * .95
  WHERE course_no = r_group_discount.course_no;
DBMS_OUTPUT.PUT_LINE
  ('A 5% discount has been given to '''
  || r_group_discount.course_no || ''')
  || r_group_discount.description
);
END LOOP;
END;
```

Figure 19.1 Discount Procedure Seen in Oracle SQL Developer

By the Way

A procedure can become invalid if the table it is based on is deleted or changed. You can recompile an invalid procedure with the following command:

[Click here to view code image](#)

```
alter procedure procedure_name compile
```

Lab 19.2: Passing Parameters IN and OUT of Procedures

After this lab, you will be able to

- [Use IN and OUT Parameters with Procedures](#)

Using IN and OUT Parameters with Procedures

Parameters are the means to pass values from the calling environment to the server, and vice versa. These values are processed or returned via the execution of the procedure. There are three parameter modes: IN, OUT, and IN OUT.

Modes

Modes specify whether the parameter passed is read in or acts as a receptacle for what comes out. [Figure 19.2](#) illustrates the relationship between the parameters when they are in the procedure header versus when the procedure is executed.

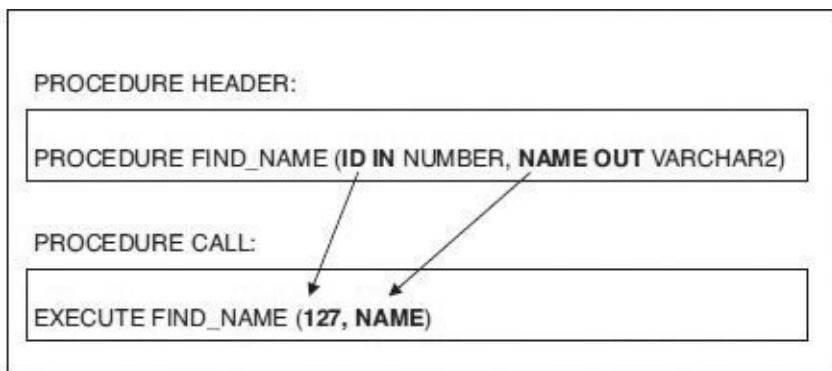


Figure 19.2 Matching a Procedure Call to a Procedure Header

Formal and Actual Parameters

Formal parameters are the names specified within parentheses as part of the header of a module. Actual parameters are the value expressions specified within parentheses as a parameter list when a call is made to the module. The formal parameter and the related actual parameter must be of the same or compatible data types. [Table 19.1](#) explains the three types of parameters.

Mode	Description	Usage
IN	Passes a value into the program Constants, literals, expressions Cannot be changed within program Default mode	Read-only value
OUT	Passes a value back from the program Cannot assign default values Must be a variable A value is assigned only if the program is successful	Write-only value
IN OUT	Both passes values in and sends values back	Must be a variable

Table 19.1 Three Types of Parameters

Passing of Constraints (Data Types) with Parameter Values

Formal parameters do not require constraints on the data type. For example, instead of specifying a constraint such as VARCHAR2(60), you can just issue VARCHAR2 against the parameter name in the formal parameter list. The constraint is passed with the value when a call is made.

Matching Actual and Formal Parameters

Two methods can be used to match actual and formal parameters: positional notation and named notation. Positional notation is simply association by position; that is, the order of the parameters used when executing the procedure matches the order in the procedure's header exactly. Named notation is explicit association using the symbol =>. It has the following syntax:

[Click here to view code image](#)

```
formal_parameter_name => argument_value
```

In named notation, the order does not matter. If you mix notation, however, you should list the positional notation before the named notation.

Default values can be used if a call to the program does not include a value in the parameter list. Note that it makes no difference which style is used; both work in similar fashion.

For Example ch19_2.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE find_sname
  (i_student_id IN NUMBER,
   o_first_name OUT VARCHAR2,
   o_last_name OUT VARCHAR2
  )
AS
BEGIN
  SELECT first_name, last_name
    INTO o_first_name, o_last_name
    FROM student
   WHERE student_id = i_student_id;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('Error in finding student_id:
      '||i_student_id);
END find_sname;
```

This procedure takes in a `student_id` via the parameter named `i_student_id`. It passes out the parameters `o_first_name` and `o_last_name`. The procedure is a simple `SELECT` statement that retrieves the `first_name` and `last_name` from the `STUDENT` table when the `student_id` matches the value of `i_student_id`, which is the only `IN` parameter that exists in the procedure. To call the procedure, a value must be passed in for the `i_student_id` parameter.

For Example ch19_3.sql

[Click here to view code image](#)

```
DECLARE
    v_local_first_name student.first_name%TYPE;
    v_local_last_name student.last_name%TYPE;
BEGIN
    find_sname
        (145, v_local_first_name, v_local_last_name);
    DBMS_OUTPUT.PUT_LINE
        ('Student 145 is: '||v_local_first_name||
         ' '|| v_local_last_name||'.'
        );
END;
```

When calling the procedure `find_sname`, a valid `student_id` should be passed in for the `i_student_id`. If it is not a valid `student_id`, an exception will be raised. Two variables must also be listed when calling the procedure. These variables, `v_local_first_name` and `v_local_last_name`, are used to hold the values of the parameters that are being passed out. After the procedure has been executed, the local variables will have values and can then be displayed with a `DBMS_OUTPUT.PUT_LINE` statement.

Summary

In this chapter, you learned how to create procedures. First, you saw how to create a basic procedure that has no parameters. Then, in the second part of the chapter, you saw how to add parameters to the procedure to narrow the transaction process taking place within that procedure.

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.

20. Functions

In this chapter, you will learn about

- [Creating Functions](#)
- [Using Functions in SQL Statements](#)
- [Optimizing Function Execution in SQL](#)

A function that is stored in the database is much like a procedure, in that it is a named PL/SQL block that can take parameters and be invoked. There are key differences both in the way it is created and how it is used, however. In this short chapter, you will learn the basics of how to create, use, and drop a function. In [Chapter 21](#), you will learn how to extend functions when they are placed into packages.

Lab 20.1: Creating Functions

After this lab, you will be able to

- [Create Stored Functions](#)
- [Make Use of Functions](#)

Functions are another type of stored code and are very similar to procedures. The significant difference between the two is that a function is a PL/SQL block that returns a single value. Functions can accept one, many, or no parameters, but they must have a return clause in their execution section. The data type of the return value must be declared in the header of the function. A function is not a stand-alone executable in the same way that a procedure is; that is, a function must always be used in some context. You can think of it as equivalent to a sentence fragment. A function produces output that needs to be assigned to a variable, or it can be used in a SELECT statement.

Creating Stored Functions

This section covers the basic function syntax and demonstrates how to create a function. The syntax for creating a function is as follows:

[Click here to view code image](#)

```
CREATE [OR REPLACE] FUNCTION function_name
  (parameter list)
    RETURN datatype
IS
BEGIN
  <body>
  RETURN (return_value);
END;
```

The function does not necessarily have any parameters, but it must have a RETURN value declared in the header, and it must return values for all of the possible execution

streams. The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception). A function may have IN, OUT, or IN OUT parameters—although you will rarely see anything except IN parameters, because it is bad programming practice to use the other parameters.

The following example shows the script for creating a function named `show_description`.

For Example `ch20_1.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE FUNCTION show_description
  (i_course_no course.course_no%TYPE)
RETURN varchar2
AS
  v_description varchar2(50);
BEGIN
  SELECT description
    INTO v_description
   FROM course
  WHERE course_no = i_course_no;
  RETURN v_description;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RETURN('The Course is not in the database');
  WHEN OTHERS
  THEN
    RETURN('Error in running show_description');
END;
```

When the function has been created in SQL Developer, the following message will be displayed in the script editor:

[Click here to view code image](#)

```
FUNCTION SHOW_DESCRIPTION compiled
```

This message indicates that the function was successfully compiled. It can also be seen in the Function node of the Database Objects in SQL Developer. See [Figure 20.1](#).

```

create or replace
FUNCTION show_description
    (i_course_no course.course_no%TYPE)
RETURN varchar2
AS
    v_description varchar2(50);
BEGIN
    SELECT description
        INTO v_description
        FROM course
        WHERE course_no = i_course_no;
    RETURN v_description;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        RETURN('The Course is not in the database');
    WHEN OTHERS
    THEN
        RETURN('Error in running show_description');
END;

```

Figure 20.1 Show_Description Function as seen in SQL Developer

The example's function heading indicates that the function takes in a parameter of the number data type and returns a VARCHAR2. The function makes use of a VARCHAR2(5) variable called `v_description`. It gives this variable the value of the description of the course, whose number is passed into the function. The return value is then the variable.

There are two exceptions in the function. The first is the `WHEN NO_DATA_FOUND` exception, which is the one most likely to occur. The second exception, `WHEN OTHERS`, which is used as a catchall for any other error that may occur.

The `RETURN` clause is one of the last statements in the function. The reason for this positioning is that the program focus will return to the calling environment once the `RETURN` clause is issued.

The following example demonstrates the syntax for creating a function named `id_is_good`. The output returned in this example is a Boolean value.

For Example *ch20_2.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE FUNCTION id_is_good
  (i_student_id IN NUMBER)
  RETURN BOOLEAN
AS
  v_id_cnt NUMBER;
BEGIN
  SELECT COUNT(*)
    INTO v_id_cnt
   FROM student
  WHERE student_id = i_student_id;
  RETURN 1 = v_id_cnt;
EXCEPTION
  WHEN OTHERS
  THEN
    RETURN FALSE;
END id_is_good;
```

The function **id_is_good** checks whether the ID passed in exists in the database. It takes in a number (which is assumed to be a student ID) and returns a **BOOLEAN** value. The function uses the variable **v_id_cnt** as a means to process the data. The **SELECT** statement obtains a count of the number of students with the numeric value that was passed in. If a student is found in the database, using the **student_id** as the primary key, the value of **v_id_cnt** will be 1. If the student is not in the database, the **SELECT** statement passes the focus to the exception-handling section, where the function returns a value of **FALSE**. The function makes use of a very interesting method to return **TRUE**. If the student is in the database, then **v_id_cnt** will equal 1, so the code **RETURN 1 = v_id_cnt** will actually return a value of **TRUE**.

Making Use of Functions

This section demonstrates how to make use of the stored function **show_description** that was created in the last section. The following example demonstrates how to call the stored function **show_description** in a PL/SQL block.

For Example *ch20_3.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
  v_description VARCHAR2(50);
BEGIN
  v_description := show_description(&sv_cnumber);
  DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

A lexical parameter in the PL/SQL block of **&cnumber** causes the user to be prompted as follows:

```
Enter value for cnumber:
```

In SQL Developer, a pop-up window will be displayed, which requests that the user enter a value for the substitution variable, as shown in [Figure 20.2](#).

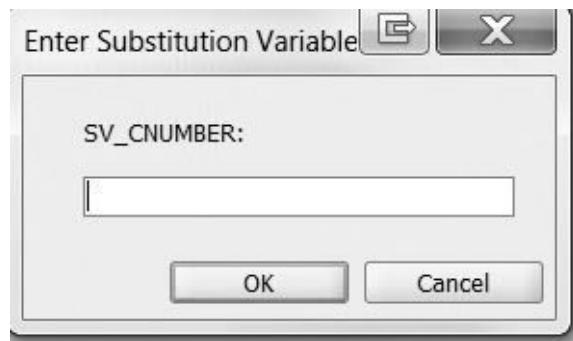


Figure 20.2 Enter Substitution Dialog Box in SQL Developer

If you enter “350,” you will see “Java Developer II” in the DBMS Output window of SQL Developer. In the Script Output window, you will see the old version of the script with the substitution variable &sv_cnumber and then you will see the new version of the script where the substitution variable &sv_cnumber has been replaced by 350. See [Figure 20.3](#).

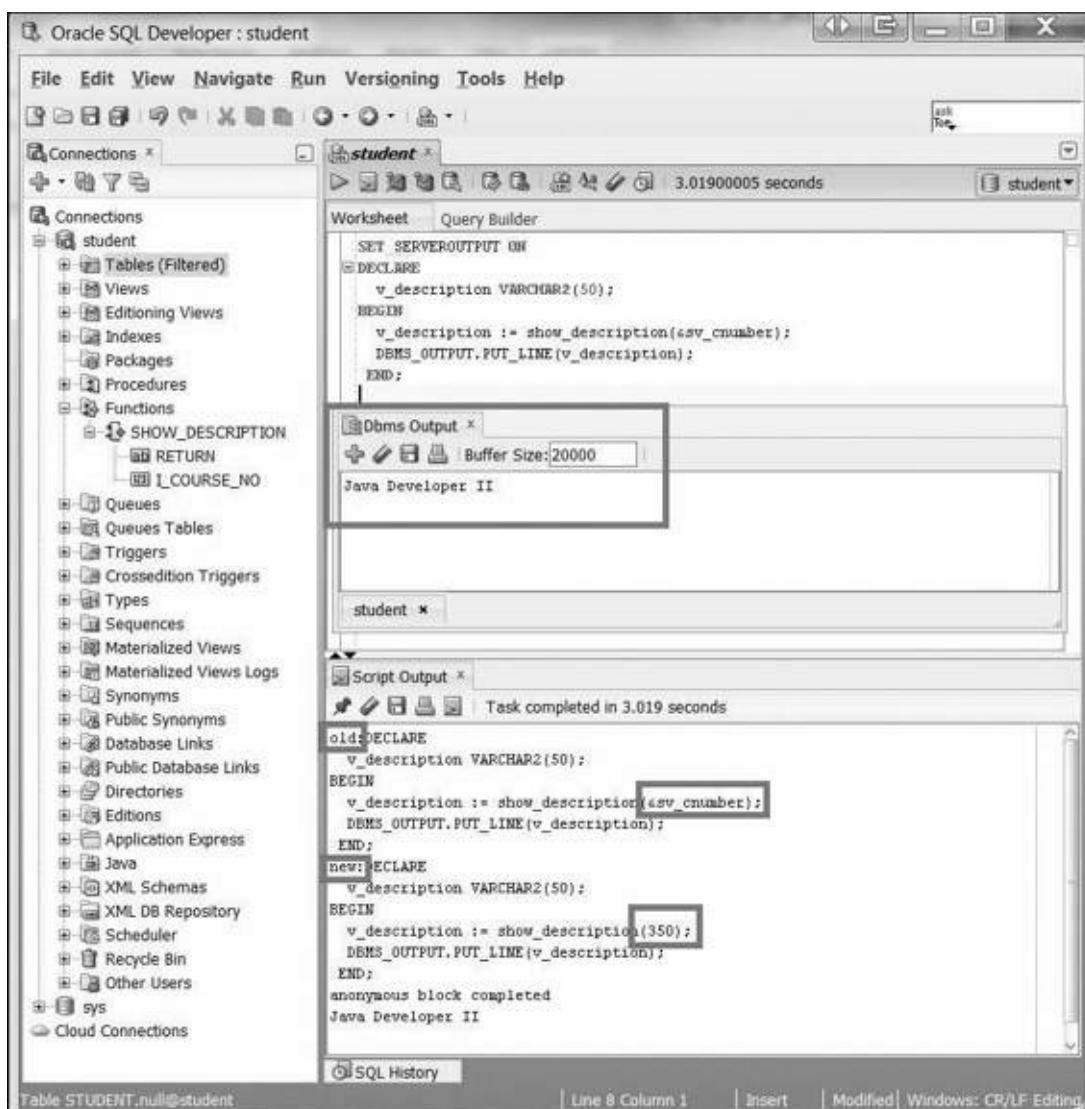


Figure 20.3 Execution of Function Show_Description

When this script is run in SQL*Plus, you will see the following output:

[Click here to view code image](#)

```
old  4:  v_descript := show_description();
new  4:  v_descript := show_description(350);
```

```
Java Developer II
PL/SQL procedure successfully completed.
```

This message means that the value for `&sv_cnumber` has been replaced with 350. The function `show_description` returns a VARCHAR2 value, which is the course description for the course number that is passed in. The PL/SQL block initializes the value of `v_description` with the value returned by the `show_description` function. This value is then displayed with the `DBMS_OUTPUT` package.

The following example is an anonymous block that makes use of the function `id_is_good`.

For Example `ch20_4.sql`

[Click here to view code image](#)

```
DECLARE
    v_id number;
BEGIN
    v_id := &id;
    IF id_is_good(v_id)
    THEN
        DBMS_OUTPUT.PUT_LINE
            ('Student ID: ''||v_id||' is a valid.');
    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('Student ID: ''||v_id||' is not valid.');
    END IF;
END;
```

This PL/SQL block evaluates the return from the function and then determines which output to project. Because the function `id_is_good` returns a Boolean value, the easiest way to make use of this function is to run it and use the result (which will be either TRUE or FALSE) in an IF statement. When testing the Boolean function `id_is_good`, the line `IF id_is_good(v_id)` means that if the function `id_is_good` for the variable results in a return of TRUE, another set of statements will be executed. The ELSE clause covers what will happen if the function returns FALSE.

Lab 20.2: Using Functions in SQL Statements

After this lab, you will be able to

- [Invoke Functions in SQL Statements](#)
- [Write Complex Functions](#)

Invoking Functions in SQL Statements

Functions return a single value and can be very useful in a SELECT statement. In particular, they can help you avoid repeated complex SQL statements within a SELECT statement. The following statement demonstrates the use of the `show_description` function in a SELECT statement for every course in the COURSE table:

[Click here to view code image](#)

```
SELECT course_no, show_description(course_no)
  FROM course;
```

It is identical to the following SELECT statement:

[Click here to view code image](#)

```
SELECT course_no, description
  FROM course;
```

Functions can also be used in a SQL statement. In fact, you have been using them all along; you just might not have realized it. As a simple example, imagine using the function **UPPER** in a SELECT statement:

[Click here to view code image](#)

```
SELECT UPPER('bill') FROM DUAL;
```

The Oracle-supplied function **UPPER** returns the uppercase value of the parameter that was passed in.

For a user-defined function to be called in a SQL expression, it must be a ROW function, not a GROUP function, and the data types must be SQL data types. The data types cannot be PL/SQL data types like a Boolean value, table, or record. Additionally, the function is not allowed to include any DML statements (**INSERT**, **UPDATE**, **DELETE**).

To use a function in a SQL SELECT statement, the function must have a certain level of purity, which is accomplished with the **PRAGMA RESTRICT_REFERENCES** clause. This topic is discussed in detail in [Chapter 21](#) in the context of functions within packages.

Writing Complex Functions

This section introduces a more complex function that will be used in [Chapter 21](#) on packages. As the following example demonstrates, functions can become very elaborate and complex.

For Example *ch20_5.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE FUNCTION new_instructor_id
  RETURN instructor.instructor_id%TYPE
AS
  v_new_instid instructor.instructor_id%TYPE;
BEGIN
  SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
    INTO v_new_instid
    FROM dual;
  RETURN v_new_instid;
EXCEPTION
  WHEN OTHERS
  THEN
    DECLARE
      v_sqlerrm VARCHAR2(250)
      := SUBSTR(SQLERRM,1,250);
    BEGIN
      RAISE_APPLICATION_ERROR(-20003,
        'Error in      instructor_id: ' || v_sqlerrm);
    END;
END;
```

```
END new_instructor_id;
```

This is a function that generates a new instructor ID. If the sequence fails to generate a new instructor ID, then a SQL error is returned.

Lab 20.3: Optimizing Function Execution in SQL

After this lab, you will be able to

- [Defining a Function Using the WITH Clause](#)
- [Create a Function with the UDF Pragma](#)

In Oracle 12.1, a few new features were introduced that allow for improved optimization of functions that are used in SQL statements. In particular, function definition can take place in the same statement as the SELECT operation.

Defining a Function Using the WITH Clause

Starting with Oracle Database 12.1, you can define functions as well as procedures within the same SQL statement in which the SELECT statement appears. This alleviates the context switch between the PL/SQL and SQL engines by allowing both steps to take place in the SQL engine and, in turn, provides for a performance gain. The function or procedure needs to be defined using the WITH clause. In previous versions of the Oracle platform, only subqueries could be defined in the WITH clause.

The following example demonstrates how the `show_description` function, which was developed earlier in this chapter, can be used in the WITH clause. The function has been renamed `show_descript` to ensure that it is not confused with the previous version, `show_description`.

For Example `ch20_6.sql`

[Click here to view code image](#)

```
WITH
  FUNCTION show_descript
    (i_course_no course.course_no%TYPE)
  RETURN varchar2
AS
  v_description varchar2(50);
BEGIN
  SELECT description
    INTO v_description
    FROM course
   WHERE course_no = i_course_no;
  RETURN v_description;
END;
SELECT course_no, show_descript(course_no), cost
  FROM COURSE
```

The WITH FUNCTION feature is useful in many different situations. The main downside to this feature is that you lose the benefits of a reusable function in favor of obtaining improved performance through reduced context shifts between the SQL and

PL/SQL engines. Before deciding which approach to use, it is advisable to do a cost analysis and weigh the benefits against the possible need to reuse the function in other contexts.

Creating a Function with the UDF Pragma

Functions can be created by adding the UDF pragma syntax, which notifies the compiler that a user-defined function will be used in SQL statements. A pragma is basically a hint to the compiler that allows it to optimize the function appropriately. When the UDF pragma syntax is used, the function will have higher performance when used in SQL. Very little needs to be done to apply the pragma, except to add the phrase **pragma UDF** prior to the variable declaration, as shown in bold in the following example:

For Example *ch20_7.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE FUNCTION show_description
  (i_course_no course.course_no%TYPE)
RETURN varchar2
AS
  pragma UDF;
  v_description varchar2(50);
BEGIN
  SELECT description
    INTO v_description
   FROM course
  WHERE course_no = i_course_no;
  RETURN v_description;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RETURN('The Course is not in the database');
  WHEN OTHERS
  THEN
    RETURN('Error in running show_description');
END;
```

Summary

In this chapter, you learned how to create and execute functions. You also learned how to include functions in SQL statements. Finally, you learned two methods to optimize function execution in SQL: use of the **WITH** function and use of the **pragma UDF** syntax.

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.

21. Packages

In this chapter, you will learn about

- [Creating Packages](#)
- [Cursors Variables](#)
- [Extending the Package](#)
- [Package Instantiation and Initialization](#)
- [SERIALLY REUSABLE Packages](#)

A package is a collection of PL/SQL objects grouped together under one package name. Packages may include procedures, functions, cursors, declarations, types, and variables. Collecting objects into a package has numerous benefits. In this chapter, you will learn what these benefits are and how to take advantage of them.

Lab 21.1: Creating Packages

After this lab, you will be able to

- [Create Package Specifications](#)
- [Create Package Bodies](#)
- [Call Stored Packages](#)
- [Create Private Objects](#)

There are numerous benefits of using packages as a method to bundle your functions and procedures, the first being that a well-designed package is a logical grouping of objects such as functions, procedures, global variables, and cursors. All of the code (parse tree and pseudocode [p-code]) is loaded into memory (shared global area [SGA] of the Oracle server) on the first call of the package. This means that the first call to the package is very expensive (it involves a lot of processing on the server), but all subsequent calls will result in improved performance. For this reason, packages are often used in applications where procedures and functions are called repeatedly.

Example of a Basic Currency Conversion

Once you have the same calculation written in multiple places, you have a large maintenance job every time the calculation is enhanced in complexity. For example, basic currency conversion is fairly simple: An amount is multiplied by an exchange rate. In actuality, currency conversion has become more complex. Once the European Union was formed, individual national currencies were phased out when a country adopted the euro as its currency. The European Union then adopted a complex policy on how these “dead” currencies would be converted. This consideration would be important if contracts were set up when the currency was in place but later the currency was phased out. If you had an old contract in German deutschemarks that needed to be converted into U.S. dollars, for example, it would have to go through this process. First it would be converted from German deutschemarks to euros based on the prevailing rate. Then it would be rounded based on a standard rounding mechanism for German deutschemarks to euros, and then it would be converted from euros to U.S. dollars at the prevailing rate. If your programs had many places where currency was converted, it would make more sense to encapsulate the conversion process into one function that encompassed this euro scenario. This function could be a public or private (explained later in this chapter) function that all other procedures in the same package called.

Packages allow you to make use of some of the concepts involved in object-oriented programming, even though PL/SQL is not a “true” object-oriented programming language. With the PL/SQL package, you can collect functions and procedures and provide them with a context. Because all the package code is loaded into memory, you can also write your code so that similar code is placed into the package in a manner that allows multiple procedures and functions to call them. You would want to do this if the logic for calculation is fairly intensive and you want to keep it in one place.

Creating Package Specifications

An additional level of security applies when using packages. When a user executes a procedure in a package (or stored procedures and functions), the procedure operates with the same permissions as its owner. Packages allow the creation of private functions and procedures, which can be called only from other functions and procedures in the package. This enforces information hiding. The structure of the package thus encourages top-down design.

The Package Specification

The package specification contains information about the contents of the package, but not the code for the procedures and functions. It also contains declarations of global/public variables. Anything placed in the declaration section of a PL/SQL block may be coded in a package specification. All objects placed in the package specification are called public objects. Any function or procedure not in the package specification but coded in a package body is called a private function or procedure.

When public procedures and functions are being called from a package, the programmer writing the “calling” process needs only the information in the package specification, as it provides all the information needed to call one of the procedures or functions within the package. The syntax for the package specification is as follows:

[Click here to view code image](#)

```
PACKAGE package_name
IS
  [ declarations of variables and types ]
  [ specifications of cursors ]
  [ specifications of modules ]
END [ package_name ];
```

The Package Body

The package body contains the actual executable code for the objects described in the package specification. It contains the code for all procedures and functions described in the specification and may additionally contain code for objects not declared in the specification; the latter type of packaged object is invisible outside the package and is referred to as “hidden.” When creating stored packages, the package specification and body can be compiled separately.

[Click here to view code image](#)

```
PACKAGE BODY package_name
IS
  [ declarations of variables and types ]
  [ specification and SELECT statement of cursors ]
  [ specification and body of modules ]
  [ BEGIN
    executable statements ]
  [ EXCEPTION
    exception handlers ]
END [ package_name ];
```

Rules for the Package Body

A number of rules must be followed in package body code. First, there must be an exact match between the cursor and module headers and their definitions in package specification. Second, declarations of variables, exceptions, type, or constants in the specification cannot be repeated in the body. Third, any element declared in the specification can be referenced in the body.

Referencing Package Elements

You use the following notation when calling packaged elements from outside the package:
`package_name.element`.

You do not need to qualify elements when they are declared and referenced inside the body of the package or when they are declared in a specification and referenced inside the body of the same package.

The following example shows the package specification for the package `manage_students`. Later in this chapter, a section will describe the creation of the body of the same package.

For Example *ch21_1.sql*

[Click here to view code image](#)

```
1 CREATE OR REPLACE PACKAGE manage_students
2 AS
3   PROCEDURE find_sname
4     (i_student_id IN student.student_id%TYPE,
5      o_first_name OUT student.first_name%TYPE,
6      o_last_name OUT student.last_name%TYPE
7    );
8   FUNCTION id_is_good
9     (i_student_id IN student.student_id%TYPE)
10    RETURN BOOLEAN;
11 END manage_students;
```

Upon running this script, the specification for the package `manage_students` will be compiled into the database. The specification for the package now indicates that there is one procedure and one function. The procedure `find_sname` requires one `IN` parameter, the student ID; it returns two `OUT` parameters, the student's first name and the student's last name. The function `id_is_good` takes in a single parameter, a student ID, and returns a Boolean value (true or false). Although the body has not yet been entered into the database, the package is still available for other applications. For example, if you included a call to one of these procedures in another stored procedure, that procedure would compile (but would not execute). This is illustrated by the following example.

For Example *ch21_2.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
  v_first_name student.first_name%TYPE;
  v_last_name student.last_name%TYPE;
BEGIN
  manage_students.find_sname
    (125, v_first_name, v_last_name);
  DBMS_OUTPUT.PUT_LINE(v_first_name||' '||v_last_name);
END;
```

This procedure cannot run because only the specification for the procedure exists in the database, not the body. The SQL*Plus session returns the following output:

[Click here to view code image](#)

```
ERROR at line 1:
```

```
ORA-04068: existing state of packages has been discarded
ORA-04067: not executed, package body
    "STUDENT.MANAGE_STUDENTS" does not exist
ORA-06508: PL/SQL: could not find program
    unit being called
ORA-06512: at line 5
```

The following example creates a package specification for a package named `school_api`. This package contains the procedure `discount_cost` from [Chapter 19](#) and the function `new_instructor_id` from [Chapter 20](#).

For Example `ch21_3.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE school_api AS
    PROCEDURE discount_cost;
    FUNCTION new_instructor_id
        RETURN instructor.instructor_id%TYPE;
END school_api;
```

Creating Package Bodies

Now we will create the body of the `manage_students` and `school_api` packages, which were specified in the previous section.

For Example `ch21_4.sql`

[Click here to view code image](#)

```
1  CREATE OR REPLACE PACKAGE BODY manage_students
2  AS
3      PROCEDURE find_sname
4          (i_student_id IN student.student_id%TYPE,
5           o_first_name OUT student.first_name%TYPE,
6           o_last_name OUT student.last_name%TYPE
7          )
8      IS
9          v_student_id student.student_id%TYPE;
10     BEGIN
11         SELECT first_name, last_name
12             INTO o_first_name, o_last_name
13             FROM student
14             WHERE student_id = i_student_id;
15     EXCEPTION
16         WHEN OTHERS
17         THEN
18             DBMS_OUTPUT.PUT_LINE
19             ('Error in finding student_id: '||v_student_id);
20     END find_sname;
21     FUNCTION id_is_good
22         (i_student_id IN student.student_id%TYPE)
23         RETURN BOOLEAN
24     IS
25         v_id_cnt number;
26     BEGIN
27         SELECT COUNT(*)
28             INTO v_id_cnt
29             FROM student
30             WHERE student_id = i_student_id;
```

```

31      RETURN 1 = v_id_cnt;
32      EXCEPTION
33      WHEN OTHERS
34      THEN
35          RETURN FALSE;
36      END id_is_good;
37  END manage_students;

```

This script compiles the package `manage_students` into the database. The specification for the package indicates that there is one procedure and one function. The procedure `find_sname` requires one `IN` parameter, the student ID; it returns two `OUT` parameters, the student's first name and the student's last name. The function `id_is_good` takes in a single parameter of a student ID and returns a Boolean value (true or false). Although the body has not yet been entered into the database, the package is still available for other applications. For example, if you included a call to one of these procedures in another stored procedure, that procedure would compile (but would not execute).

The next example creates the package body for the package named `school_api` that was created in the previous example. It contains the procedure `discount_cost` from [Chapter 19](#) and the function `new_instructor_id` from [Chapter 20](#).

For Example *ch21_5.sql*

[Click here to view code image](#)

```

1 CREATE OR REPLACE PACKAGE BODY school_api AS
2     PROCEDURE discount_cost
3     IS
4         CURSOR c_group_discount
5         IS
6             SELECT distinct s.course_no, c.description
7                 FROM section s, enrollment e, course c
8                 WHERE s.section_id = e.section_id
9                 GROUP BY s.course_no, c.description,
10                         e.section_id, s.section_id
11                 HAVING COUNT(*) >=8;
12     BEGIN
13         FOR r_group_discount IN c_group_discount
14         LOOP
15             UPDATE course
16                 SET cost = cost * .95
17                 WHERE course_no = r_group_discount.course_no;
18             DBMS_OUTPUT.PUT_LINE
19                 ('A 5% discount has been given to'
20                  ||r_group_discount.course_no||
21                  ||r_group_discount.description);
22         END LOOP;
23     END discount_cost;
24     FUNCTION new_instructor_id
25         RETURN instructor.instructor_id%TYPE
26     IS
27         v_new_instid instructor.instructor_id%TYPE;
28     BEGIN
29         SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
30             INTO v_new_instid
31             FROM dual;

```

```
32      RETURN v_new_instid;
33  EXCEPTION
34      WHEN OTHERS
35      THEN
36          DECLARE
37              v_sqlerrm VARCHAR2(250) :=
38                  SUBSTR(SQLERRM,1,250);
39          BEGIN
40              RAISE_APPLICATION_ERROR(-20003,
41                  'Error in    instructor_id: '||v_sqlerrm);
42          END;
43      END new_instructor_id;
44  END school_api;
```

Calling Stored Packages

Now we will use elements of the `manage_students` package in another code block.

For Example *ch21_6.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
DECLARE
    v_first_name student.first_name%TYPE;
    v_last_name student.last_name%TYPE;
BEGIN
    IF manage_students.id_is_good(&&v_id)
    THEN
        manage_students.find_sname(&&v_id, v_first_name,
                                    v_last_name);
        DBMS_OUTPUT.PUT_LINE('Student No. '||&v_id||' is '
                            ||v_last_name||', '||v_first_name);
    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('Student ID: '||&v_id||' is not in the database.');
    END IF;
END;
```

This is a correct PL/SQL block for running the function and the procedure in the package `manage_students`. If an existing `student_id` is entered, then the name of the student is displayed. If the student ID is not valid, then an error message is displayed. The following example shows the result when `145` is entered for the variable `v_id` in SQL Developer. The script output shows the original script and then the script once all variables have been replaced with the number entered (in this case `145`). The final line (in bold) is the result.

Click here to view code image

```
old:DECLARE
  v_first_name student.first_name%TYPE;
  v_last_name student.last_name%TYPE;
BEGIN
  IF manage_students.id_is_good(&&v_id)
  THEN
    manage_students.find_sname(&&v_id, v_first_name,
      v_last_name);
  DBMS_OUTPUT.PUT_LINE('Student No. '||&v_id||' is '
    ||v_last_name||'. '||v_first_name);
```

```

ELSE
    DBMS_OUTPUT.PUT_LINE
        ('Student ID: '||amp;&v_id||' is not in the database.');
END IF;
END;
new:DECLARE
    v_first_name student.first_name%TYPE;
    v_last_name student.last_name%TYPE;
BEGIN
    IF manage_students.id_is_good(145)
    THEN
        manage_students.find_sname(145, v_first_name,
            v_last_name);
        DBMS_OUTPUT.PUT_LINE('Student No. '||145||' is '
            ||v_last_name||', '||v_first_name);
    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('Student ID: '||145||' is not in the database.');
    END IF;
END;
anonymous block completed
Student No. 145 is Lefkowitz, Paul

```

The function `id_is_good` returns TRUE for an existing `student_id` such as 145. Control then passes to the first part of the IF statement and the procedure `manage_students.find_sname` finds the first and last names for `student_id` of 145—specifically, Paul Lefkowitz.

The following is an example of a test script for the `school_api` package.

For Example *ch21_7.sql*

[Click here to view code image](#)

```

SET SERVEROUTPUT ON
DECLARE
    V_instructor_id instructor.instructor_id%TYPE;
BEGIN
    School_api.Discount_Cost;
    v_instructor_id := school_api.new_instructor_id;
    DBMS_OUTPUT.PUT_LINE
        ('The new id is: '||v_instructor_id);
END;

```

Creating Private Objects

Public elements are elements defined in the package specification. If an object is defined only in the package body, then it is private. Private elements cannot be accessed directly by any programs outside the package. You can think of the package specification as being a “menu” of packaged items that are available to users; there may be other objects working behind the scenes, but they aren’t accessible. They cannot be called or utilized in any way; they are available as part of the internal “menu” of the package and can be called only by other elements of the package.

The following steps show how to transform the package `manage_students` so that the function `student_count_priv` becomes a private function. The public procedure

`display_student_count` then calls this private function.

Step 1: Replace the last lines of the `manage_students` package specification with the following code and recompile the package specification:

[Click here to view code image](#)

```
11 PROCEDURE display_student_count;
12 END manage_students;
```

Step 2: Replace the end of the body with the following code and recompile the package body. Lines 1–36 are unchanged from lines 1–36 in example ch21_4.sql:

[Click here to view code image](#)

```
37 FUNCTION student_count_priv
38     RETURN NUMBER
39 IS
40     v_count NUMBER;
41 BEGIN
42     select count(*)
43     into v_count
44     from student;
45     return v_count;
46 EXCEPTION
47     WHEN OTHERS
48         THEN
49             return(0);
50 END student_count_priv;
51 PROCEDURE display_student_count
52 IS
53     v_count NUMBER;
54 BEGIN
55     v_count := student_count_priv;
56     DBMS_OUTPUT.PUT_LINE
57     ('There are '||v_count||' students.');
58 END display_student_count;
59 END manage_students;
```

Now run the following script:

[Click here to view code image](#)

```
DECLARE
    V_count NUMBER;
BEGIN
    V_count := Manage_students.student_count_priv;
    DBMS_OUTPUT.PUT_LINE(v_count);
END;
```

Because the private function `student_count_priv` cannot be called from outside the package, you will receive the following error message:

[Click here to view code image](#)

```
ERROR at line 1:
ORA-06550: line 4, column 31:
PLS-00302: component 'STUDENT_COUNT_PRIV' must be declared
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```

It appears as if the private function does not exist. This point is important to keep in

mind—it can be useful when you are writing PL/SQL packages used by other developers. Those developers need to see only the package specification, not the inner workings of the package. That is, they need to know what is being passed into the procedures and functions and what is being returned. If a number of procedures will make use of the same logic, it may make more sense to put that logic into a private function called by the procedures. This is also a good approach to keep in mind if one calculation will be used by many other procedures in the same package. For example, we just created a function to count students. Perhaps other procedures will need to make use of this function—such as if a change in the price of all courses should occur once the student count reaches a certain number.

The following example shows a valid method of running a procedure. The result would be a line indicating the number of students in the database. Note that the procedure in the package `manage_students` uses the private function `student_count_priv` to retrieve the student count.

[Click here to view code image](#)

```
SET SERVEROUTPUT ON
Execute manage_students.display_student_count;
```

If you forget to include a procedure or function in a package specification, it becomes private. If you declare a procedure or function in the package specification, but then do not define it when you create the body, you will receive the following error message:

[Click here to view code image](#)

```
PLS-00323: subprogram or cursor 'procedure_name' is
declared in a package specification and must be
defined in the package body
```

The following updated script for the `manage_students` package adds a private function to the `school_api` called `get_course_descript_private`. It accepts a `course.course_no%TYPE` and returns a `course.description%TYPE`. It searches for and returns the course description for the course number passed to it. If the course does not exist or if an error occurs, it returns `NULL`. Nothing needs to be added to the package specification, because you are simply adding a private object.

For Example *ch21_8.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE manage_students
AS
  PROCEDURE find_sname
    (i_student_id IN student.student_id%TYPE,
     o_first_name OUT student.first_name%TYPE,
     o_last_name OUT student.last_name%TYPE
    );
  FUNCTION id_is_good
    (i_student_id IN student.student_id%TYPE)
    RETURN BOOLEAN;
  PROCEDURE display_student_count;
END manage_students;
```

The package body for `manage_students` now has the following form:

For Example ch21_9.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE BODY manage_students
AS
    PROCEDURE find_sname
        (i_student_id IN student.student_id%TYPE,
         o_first_name OUT student.first_name%TYPE,
         o_last_name OUT student.last_name%TYPE
        )
    IS
        v_student_id student.student_id%TYPE;
    BEGIN
        SELECT first_name, last_name
        INTO o_first_name, o_last_name
        FROM student
        WHERE student_id = i_student_id;
    EXCEPTION
        WHEN OTHERS
        THEN
            DBMS_OUTPUT.PUT_LINE
            ('Error in finding student_id: '||v_student_id);
    END find_sname;
    FUNCTION id_is_good
        (i_student_id IN student.student_id%TYPE)
        RETURN BOOLEAN
    IS
        v_id_cnt number;
    BEGIN
        SELECT COUNT(*)
        INTO v_id_cnt
        FROM student
        WHERE student_id = i_student_id;
        RETURN 1 = v_id_cnt;
    EXCEPTION
        WHEN OTHERS
        THEN
            RETURN FALSE;
    END id_is_good;
    FUNCTION student_count_priv
        RETURN NUMBER
    IS
        v_count NUMBER;
    BEGIN
        select count(*)
        into v_count
        from student;
        return v_count;
    EXCEPTION
        WHEN OTHERS
        THEN
            return(0);
    END student_count_priv;
    PROCEDURE display_student_count
        is
        v_count NUMBER;
    BEGIN
        v_count := student_count_priv;
        DBMS_OUTPUT.PUT_LINE
```

```

('There are'||v_count||' students.');
END display_student_count;
FUNCTION get_course_descript_private
  (i_course_no course.course_no%TYPE)
  RETURN course.description%TYPE
IS
  v_course_descript course.description%TYPE;
BEGIN
  SELECT description
    INTO v_course_descript
    FROM course
   WHERE course_no = i_course_no;
  RETURN v_course_descript;
EXCEPTION
  WHEN OTHERS
  THEN
    RETURN NULL;
END get_course_descript_private;
END manage_students;

```

Lab 21.2: Cursor Variables

After this lab, you will be able to

- Make Use of Cursor Variables

Up to this point in this book, you have seen cursors used to gather specific data from a single `SELECT` statement. At the beginning of this chapter, you learned how to bring a number of procedures together into a large program called a package. A package may have one cursor that is used by a few procedures. In this case, each of the procedures that uses the same cursor would have to declare, open, fetch, and close the cursor. In the current version of PL/SQL, cursors can be declared and manipulated like any other PL/SQL variable. This type of variable is called a cursor variable or a `REF CURSOR`. A cursor variable is just a reference or a handle to a static cursor. It permits a programmer to pass this reference to the same cursor among all the program's units that need access to the cursor. A cursor variable binds the cursor's `SELECT` statement dynamically at run time.

Explicit cursors are used to name a work area that holds the information of a multirow query. A cursor variable may be used to point to the area in memory where the result of a multirow query is stored. The cursor always refers to the same information in a work area, whereas a cursor variable can point to different work areas. Cursors are static, but cursor variables can be seen as dynamic because they are not tied to any one specific query. Cursor variables give you easy access to centralized data retrieval.

You can use a cursor variable to pass the result set of a query between stored procedures and various clients. A query work area remains accessible as long as a cursor variable points to it. As a consequence, you can freely pass a cursor variable from one scope to another. Two types of cursor variables exist: strong and weak.

To execute a multirow query, the Oracle server opens a work area called a cursor to store processing information. To access this information, you either name the work area or use a cursor variable that points to the work area. A cursor always refers to the same work

area, whereas a cursor variable can refer to different work areas. Hence, cursors and cursor variables are not interoperable. An explicit cursor is static and is associated with one SQL statement. A cursor variable, in contrast, can be associated with different statements at run time. Primarily you use a cursor variable to pass a pointer to query result sets between PL/SQL stored subprograms and various clients, such as a client Oracle Developer Forms application. None of them owns the result set; they simply share a pointer to the query work area that stores the result set. You can declare a cursor variable on the client side, open and fetch from it on the server side, and then continue to fetch from it on the client side.

Cursor variables differ from cursors in much the same way that constants differ from variables. A cursor is static; a cursor variable is dynamic. In PL/SQL, a cursor variable has a **REF CURSOR** data type, where **REF** stands for reference and **CURSOR** stands for the class of the object. You will now learn the syntax for declaring and using a cursor variable.

To create a cursor variable, you first need to define a **REF CURSOR** type and then declare a variable of that type. Before you declare the **REF CURSOR** to be of a strong type, you must declare a record that has the data types of the result set of the **SELECT** statement that you plan to use (note that this step is not necessary for a weak **REF CURSOR**).

[Click here to view code image](#)

```
TYPE inst_city_type IS RECORD
  (first_name instructor.first_name%TYPE;
  last_name   instructor.last_name%TYPE;
  city        zipcode.city%TYPE;
  state       zipcode.state%TYPE)
```

Second, you must declare a composite data type for the cursor variable that is of the type **REF CURSOR**. The syntax is as follows:

[Click here to view code image](#)

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

The **ref_type_name** is a type specified in subsequent declarations. The **return type** represents a record type for a strong cursor; a weak cursor does not have a specific return type but can handle any combination of data items in a **SELECT** statement. The **REF CURSOR** keywords indicate that the new type will be a pointer to the defined type. The **return_type** indicates the types of **SELECT** lists that are eventually returned by the cursor variable. The return type must be a record type.

[Click here to view code image](#)

```
TYPE inst_city_cur IS REF CURSOR RETURN inst_city_type;
```

A cursor variable can be strong (restrictive) or weak (nonrestrictive). A strong cursor variable is a **REF CURSOR** type definition that specifies a **return_type**; a weak definition does not. PL/SQL enables you to associate a strong type with type-comparable queries only, while a weak type can be associated with any query. This makes a strong cursor variable less error prone but renders weak **REF CURSOR** types more flexible.

Following are the key steps for handling a cursor variable:

1. Define and declare the cursor variable.

Open the cursor variable. Associate a cursor variable with a multirow **SELECT** statement, execute the query, and identify the result set. An **OPEN FOR** statement can open the same cursor variable for different queries. You do not need to close a cursor variable before reopening it. Keep in mind that when you reopen a cursor variable for a different query, the previous query is lost. A good programming technique is to close the cursor variables before reopening them later on in the program.

2. Fetch rows from the result set.

Retrieve rows from the result set, one at a time. Note that the return type of the cursor variable must be compatible with the variable named in the **INTO** clause of the **FETCH** statement.

The **FETCH** statement retrieves rows from the result set, one at a time. PL/SQL verifies that the return type of the cursor variable is compatible with the **INTO** clause of the **FETCH** statement. For each query column value returned, there must be a type-comparable variable in the **INTO** clause. Also, the number of query column values must equal the number of variables. In case of a mismatch in number or type, an error occurs at compile time for strongly typed cursor variables and at run time for weakly typed cursor variables.

3. Close the cursor variable.

The next example shows the use of a cursor variable in a package.

For Example *ch21_10.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE course_pkg AS
  TYPE course_rec_typ IS RECORD
    (first_name  student.first_name%TYPE,
     last_name   student.last_name%TYPE,
     course_no   course.course_no%TYPE,
     description course.description%TYPE,
     section_no  section.section_no%TYPE
    );
  TYPE course_cur IS REF CURSOR RETURN course_rec_typ;
  PROCEDURE get_course_list
    (p_student_id    NUMBER ,
     p_instructor_id NUMBER ,
     course_list_cv IN OUT course_cur);
END course_pkg;
/

CREATE OR REPLACE PACKAGE BODY course_pkg AS
  PROCEDURE get_course_list
    (p_student_id    NUMBER ,
     p_instructor_id NUMBER ,
     course_list_cv IN OUT course_cur)
  IS
  BEGIN
    IF p_student_id IS NULL AND p_instructor_id
      IS NULL THEN
```

```

OPEN course_list_cv FOR
  SELECT 'Please choose a student-' First_name,
         'instructor combination' Last_name,
        NULL course_no,
        NULL description,
        NULL section_no
       FROM dual;
ELSIF p_student_id IS NULL THEN
  OPEN course_list_cv FOR
    SELECT s.first_name first_name,
           s.last_name last_name,
           c.course_no course_no,
           c.description description,
           se.section_no section_no
      FROM instructor i, student s,
           section se, course c, enrollment e
     WHERE i.instructor_id = p_instructor_id
       AND i.instructor_id = se.instructor_id
       AND se.course_no = c.course_no
       AND e.student_id = s.student_id
       AND e.section_id = se.section_id
      ORDER BY c.course_no, se.section_no;
ELSIF p_instructor_id IS NULL THEN
  OPEN course_list_cv FOR
    SELECT i.first_name first_name,
           i.last_name last_name,
           c.course_no course_no,
           c.description description,
           se.section_no section_no
      FROM instructor i, student s,
           section se, course c, enrollment e
     WHERE s.student_id = p_student_id
       AND i.instructor_id = se.instructor_id
       AND se.course_no = c.course_no
       AND e.student_id = s.student_id
       AND e.section_id = se.section_id
      ORDER BY c.course_no, se.section_no;
END IF;
END get_course_list;
END course_pkg;

```

You can pass query result sets between PL/SQL stored subprograms and various clients. This approach works because PL/SQL and its clients share a pointer to the query work area identifying the result set. This can be done in a client program like SQL*Plus by defining a host variable with a data type of **REF CURSOR** to hold the query result generated from a **REF CURSOR** in a stored program. To see what is being stored in the SQL*Plus variable, use the SQL*Plus **PRINT** command. Optionally, you can use the SQL*Plus command **SET AUTOPRINT ON** to display the query results automatically.

In script ch21_10, the package specification includes two declarations of a **TYPE**. The first is for the record type **course_rec_type**. This record type is declared to define the result set of the **SELECT** statements that will be used for the cursor variable. When data items in a record do not match a single table, it is necessary to create a record type. The second **TYPE** declaration is for the cursor variable, **REF CURSOR**. This variable has the name **course_cur** and is declared as a strong cursor, meaning that it can be used only

for a single record type. The record type is `course_rec_type`. The procedure `get_course_list` in the `course_pkg` returns a cursor variable that holds three different result sets. Each of the result sets is of the same record type.

- The first type is for when both `IN` parameters—that is, the student ID and instructor ID—are null. This will produce a result set that consists of a message, “Please choose a student-instructor combination.”
- The second way the procedure runs is if the `instructor_id` is passed in but the `student_id` is null (note that the logic of the procedure is a reverse negative; saying in the second clause of the `IF` statement `p_student_id IS NULL`, means “when the `instructor_id` is passed in”). This will run a `SELECT` statement to populate the cursor variable that holds a list of all courses this instructor teaches and the students enrolled in these classes.
- The third way this procedure runs is with a `student_id` and no `instructor_id`. This will produce a result set containing all the courses the student is enrolled in and the instructor for each section.

Be aware that once the cursor variable is opened, it is not closed until you specifically close it.

The following SQL statement will create a variable that is a cursor variable type:

[Click here to view code image](#)

```
VARIABLE course_cv REF CURSOR
```

There are three ways to execute this procedure. The first way would be to pass a student ID and not an instructor ID:

[Click here to view code image](#)

```
exec course_pkg.get_course_list(102, NULL, :course_cv);
```

The contents of the variable `course_cv` can then be displayed in SQL*Plus with the following command:

[Click here to view code image](#)

```
SQL> print course_cv
```

FIRST_NAME	LAST_NAME	COURSE_NO	DESCRIPTION	SECTION_NO
Charles	Lowry	25	Intro to Programming	2
Nina	Schorin	25	Intro to Programming	5

The next method would be to pass an instructor ID and not a student ID:

[Click here to view code image](#)

```
SQL> exec course_pkg.get_course_list(NULL, 102, :course_cv);  
PL/SQL procedure successfully completed.
```

```
SQL> print course_cv
```

FIRST_NAME	LAST_NAME	COURSE_NO	DESCRIPTION	SECTION_NO
Jeff	Runyan	10	Technology Concepts	2
Dawn	Dennis	25	Intro to Programming	4
May	Jodoin	25	Intro to Programming	4

Jim	Joas	25	Intro to Programming	4
Arun	Griffen	25	Intro to Programming	4
Alfred	Hutheesing	25	Intro to Programming	4
Lula	Oates	100	Hands-On Windows	1
Regina	Bose	100	Hands-On Windows	1
Jenny	Goldsmith	100	Hands-On Windows	1
Roger	Snow	100	Hands-On Windows	1
Rommel	Frost	100	Hands-On Windows	1
Debra	Boyce	100	Hands-On Windows	1
Janet	Jung	120	Intro to Java Programming	4
John	Smith	124	Advanced Java Programming	1
Charles	Caro	124	Advanced Java Programming	1
Sharon	Thompson	124	Advanced Java Programming	1
Evan	Fielding	124	Advanced Java Programming	1
Ronald	Tangaribuan	124	Advanced Java Programming	1
N	Kuehn	146	Java for C/C++ Programmers	2
Derrick	Baltazar	146	Java for C/C++ Programmers	2
Angela	Torres	240	Intro to the Basic Language	2

The last method would be not to pass either the student ID or the instructor ID:

[Click here to view code image](#)

```
SQL> exec course_pkg.get_course_list(NULL, NULL, :course_cv);
PL/SQL procedure successfully completed.
```

```
SQL> print course_cv
```

FIRST_NAME	LAST_NAME	C DESCRIPTION	S
----- - -----			
Please choose a student- instructor combination			

The next example creates another package called **student_info_pkg** that has a single procedure called **get_student_info**. The **get_student_info** package will have three parameters: the **student_id**, a number called **p_choice**, and a weak cursor variable. The **p_choice** parameter indicates which information will be delivered about the student. If it is 1, then the procedure will return the information about the student from the **STUDENT** table. If it is 2, then the procedure will list all the courses in which the student is enrolled, along with the student names of the fellow students enrolled in the same section as the student with the **student_id** that was passed in. If it is 3, then the procedure will return the instructor name for that student, with the information about the courses in which the student is enrolled.

For Example *ch21_11.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE student_info_pkg AS
  TYPE student_details IS REF CURSOR;
  PROCEDURE get_student_info
    (p_student_id NUMBER ,
     p_choice      NUMBER ,
     details_cv IN OUT student_details);
END student_info_pkg;
/
CREATE OR REPLACE PACKAGE BODY student_info_pkg AS
  PROCEDURE get_student_info
    (p_student_id NUMBER ,
     p_choice      NUMBER ,
```

```

        details_cv IN OUT student_details)
IS
BEGIN
    IF p_choice = 1 THEN
        OPEN details_cv FOR
            SELECT s.first_name first_name,
                   s.last_name last_name,
                   s.street_address address,
                   z.city      city,
                   z.state     state,
                   z.zip       zip
            FROM student s, zipcode z
            WHERE s.student_id = p_student_id
                  AND z.zip = s.zip;
    ELSIF p_choice = 2 THEN
        OPEN details_cv FOR
            SELECT c.course_no course_no,
                   c.description           description,
                   se.section_no          section_no,
                   s.first_name first_name,
                   s.last_name last_name
            FROM student s, section se,
                   course c, enrollment e
            WHERE se.course_no = c.course_no
                  AND e.student_id = s.student_id
                  AND e.section_id = se.section_id
                  AND se.section_id in (SELECT e.section_id
                                         FROM student s,
                                         enrollment e
                                         WHERE s.student_id =
                                               p_student_id
                                         AND s.student_id =
                                               e.student_id)
            ORDER BY c.course_no;
    ELSIF p_choice = 3 THEN
        OPEN details_cv FOR
            SELECT i.first_name first_name,
                   i.last_name last_name,
                   c.course_no course_no,
                   c.description description,
                   se.section_no section_no
            FROM instructor i, student s,
                   section se, course c, enrollment e
            WHERE s.student_id = p_student_id
                  AND i.instructor_id = se.instructor_id
                  AND se.course_no = c.course_no
                  AND e.student_id = s.student_id
                  AND e.section_id = se.section_id
            ORDER BY c.course_no, se.section_no;
    END IF;
END get_student_info;

END student_info_pkg;

```

To execute the `get_student_info` procedure, you would first have to create a session variable:

[Click here to view code image](#)

```
VARIABLE student_cv REF CURSOR
```

Then execute the procedure with the appropriate values:

[Click here to view code image](#)

```
SQL> execute student_info_pkg.GET_STUDENT_INFO  
(102, 1, :student_cv);
```

Finally display the results:

[Click here to view code image](#)

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

```
SQL> print student_cv  
FIRST_ LAST_NAM ADDRESS          CITY          ST ZIP  
-- -- - - - -  
Fred   Crocitto 101-09 120th St.  Richmond Hill NY 11419
```

```
SQL> execute student_info_pkg.GET_STUDENT_INFO  
(102, 2, :student_cv);  
PL/SQL procedure successfully completed.
```

```
SQL> print student_cv  
COURSE_NO DESCRIPTION      SECTION_NO FIRST_NAME LAST_NAME  
--- - - - -  
    25 Intro to Programming      2 Fred        Crocitto  
    25 Intro to Programming      2 Judy        Sethi  
     5 Intro to Programming      2 Jenny       Goldsmith  
    25 Intro to Programming      2 Barbara     Robichaud  
    25 Intro to Programming      2 Jeffrey     Citron  
    25 Intro to Programming      2 George      Kocka  
    25 Intro to Programming      5 Fred        Crocitto  
    25 Intro to Programming      5 Hazel       Lasseter  
    25 Intro to Programming      5 James       Miller  
    25 Intro to Programming      5 Regina     Gates  
    25 Intro to Programming      5 Arlyne     Sheppard  
    25 Intro to Programming      5 Thomas     Edwards  
    25 Intro to Programming      5 Sylvia     Perrin  
    25 Intro to Programming      5 M.         Diokno  
    25 Intro to Programming      5 Edgar      Moffat  
    25 Intro to Programming      5 Bessie     Heedles  
    25 Intro to Programming      5 Walter     Boremann  
    25 Intro to Programming      5 Lorrane    Velasco
```

```
SQL> execute student_info_pkg.GET_STUDENT_INFO  
(214, 3, :student_cv);  
PL/SQL procedure successfully completed.
```

```
SQL> print student_cv  
FIRST_NAME LAST_NAME      COURSE_NO DESCRIPTION      SECTION_NO  
--- - - - -  
Marilyn   Frantzen      120 Intro to Java Programming    1  
Fernand   Hanks         122 Intermediate Java Programming 5  
Gary      Pertz          130 Intro to Unix           2  
Marilyn   Frantzen      145 Internet Protocols      1
```

Early versions of Oracle offered the use of only **REF CURSOR**, where first a type of **REF CURSOR** would be created with a particular record set and then another variable would have to be created of that type to make use of **REF CURSOR** in stored procedures and functions. Later versions of Oracle introduced the **SYS_REFCURSOR** as a predefined

type (of type REF CURSOR) that behaves in a similar manner. SYS_REFCURSOR is weakly typed, which means any SELECT statement can be used with different FROM or WHERE clauses, as well as different number and types of columns. The examples in [Chapter 24](#) in the section that covers DBMS_SQL include a syntax example that uses SYS_REFCURSOR instead of REF CURSOR.

Rules for Using Cursor Variables

- The cursor variable cannot be defined in a package specification.
 - You cannot use cursor variables with remote subprograms on another server, so you cannot pass cursor variables to a procedure that is called through a database link.
 - Do not use FOR UPDATE with OPEN FOR in processing a cursor variable.
 - You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
 - A cursor variable cannot be assigned a null value.
 - A REF CURSOR type cannot be used in CREATE TABLE or VIEW statements as there is no equivalent data type for a database column.
 - A stored procedure that uses a cursor variable can be used only as a query block data source; it cannot be used for a DML block data source. Using a REF CURSOR is ideal for queries that are dependent only on variations in SQL statements and not on PL/SQL statements.
 - You cannot store cursor variables in an associative array, nested table, or varray.
 - If you pass a host cursor variable to PL/SQL, you cannot fetch from it on the server side unless you also open it there on the same server call.
-

Lab 21.3: Extending the Package

After this lab, you will be able to

- [Extend the Package with Additional Procedures](#)
-

In this lab, you will make use of previously introduced concepts to both extend the packages you have created and create new ones. Only by completing extensive exercises will you become more comfortable with programming in PL/SQL. It is very important when writing your PL/SQL code that you carefully take into consideration all aspects of the business requirements. A good rule of thumb is to think ahead and write your code in reusable components so that it will be easy to extend and maintain that PL/SQL code.

Extending the Package with Additional Procedures

This section provides more examples of writing packages by working through a complex package with various complex functions and procedures. It is always a best practice to build up large packages one step at a time and to test each section you create to ensure that it works properly and does not contain any syntax errors. The following set of examples show you how to build a package step by step.

Creating the Manage_Grades Package Specification

The following script creates a new package specification called `manage_grades`. This package will perform a number of calculations on grades and will need two package cursors. The first step is to create a cursor called `c_grade_Type` that has an `IN` parameter of a section ID and provides a list of all grade types for a given section; this information is necessary to calculate a student's grade in that section. The return items from the cursor will be (1) the grade type code; (2) the number of that grade type for this section; (3) the percentage of the final grade; and (4) the drop lowest indicator (a flag).

The first thing you should always do when building a package cursor is to write the `SELECT` statement and test it on a known result set. In other words, you hard-code a value for the variable—for example, a `student_id` and `section_id`—and then replace the hard-coded values with the appropriate variables. You continue to build the package one step at a time in this manner. Try to build each component of the package with the smallest testable unit of code. Once that unit of code is returning the correct result and the syntax is free of errors, you can then turn to building the next unit.

The following example contains only the SQL `SELECT` statement. You are well advised to write the SQL `SELECT` statement first and then test it for a known value. In this case, the `student_id` is 145 and the `section_id` is 106.

For Example `ch21_12.sql`

[Click here to view code image](#)

```
SELECT GRADE_TYPE_CODE,
       NUMBER_PER_SECTION,
       PERCENT_OF_FINAL_GRADE,
       DROP_LOWEST
  FROM grade_Type_weight
 WHERE section_id = 106
   AND section_id IN (SELECT section_id
                      FROM grade
                     WHERE student_id = 145)
```

This `SELECT` statement is now put into the package:

For Example `ch21_13.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
  -- Cursor to loop through all grade types for a given section
  CURSOR c_grade_type
    (pc_section_id  section.section_id%TYPE,
     PC_student_ID  student.student_id%TYPE)
  IS
```

```

SELECT GRADE_TYPE_CODE,
       NUMBER_PER_SECTION,
       PERCENT_OF_FINAL_GRADE,
       DROP_LOWEST
  FROM grade_Type_weight
 WHERE section_id = pc_section_id
   AND section_id IN (SELECT section_id
                      FROM grade
                     WHERE student_id = pc_student_id);
END MANAGE_GRADES;

```

Creating the c_grade Cursor

The next example shows the expansion of the `manage_grades` package through the addition of a section cursor called `c_grades`. This cursor will take a grade type code, a student ID, and a section ID and return all the grades for that student for that section of that grade type. For example, if Alice was enrolled in the Introduction to Java section, this cursor could be used to gather all of her quiz grades.

For Example *ch21_14.sql*

[Click here to view code image](#)

```

CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
  -- Cursor to loop through all grade types for a given section.
  CURSOR c_grade_type
    (pc_section_id  section.section_id%TYPE,
     PC_student_ID  student.student_id%TYPE)
  IS
  SELECT GRADE_TYPE_CODE,
         NUMBER_PER_SECTION,
         PERCENT_OF_FINAL_GRADE,
         DROP_LOWEST
    FROM grade_Type_weight
   WHERE section_id = pc_section_id
     AND section_id IN (SELECT section_id
                        FROM grade
                           WHERE student_id = pc_student_id);
  -- Cursor to loop through all grades for a given student
  -- in a given section.
  CURSOR c_grades
    (p_grade_type_code
     grade_Type_weight.grade_type_code%TYPE,
      pc_student_id  student.student_id%TYPE,
      pc_section_id  section.section_id%TYPE) IS
  SELECT grade_type_code,grade_code_occurrence,
         numeric_grade
    FROM grade
   WHERE student_id = pc_student_id
     AND section_id = pc_section_id
     AND grade_type_code = p_grade_type_code;
END MANAGE_GRADES;

```

Creating the Function final_grade

The next step is to add a function to this package specification called `final_grade`. This function will have two IN parameters: the student ID and the section ID. It will return a number—that student's final grade in that section—plus an exit code. The reason you add an exit code instead of raise exceptions is because this approach makes the procedure more flexible and allows the calling program to choose how to proceed depending on the specific error code generated.

For Example `ch21_15.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
    -- Cursor to loop through all grade types for a given section.
    CURSOR c_grade_type
        (pc_section_id section.section_id%TYPE,
         PC_student_ID student.student_id%TYPE)
    IS
        SELECT GRADE_TYPE_CODE,
               NUMBER_PER_SECTION,
               PERCENT_OF_FINAL_GRADE,
               DROP_LOWEST
        FROM grade_Type_weight
       WHERE section_id = pc_section_id
         AND section_id IN (SELECT section_id
                             FROM grade
                            WHERE student_id = pc_student_id);
    -- Cursor to loop through all grades for a given student
    -- in a given section.
    CURSOR c_grades
        (p_grade_type_code
         grade_Type_weight.grade_type_code%TYPE,
         pc_student_id student.student_id%TYPE,
         pc_section_id section.section_id%TYPE) IS
        SELECT grade_type_code,grade_code_occurrence,
               numeric_grade
        FROM grade
       WHERE student_id = pc_student_id
         AND section_id = pc_section_id
         AND grade_type_code = p_grade_type_code;
    -- Function to calculate a student's final grade
    -- in one section
    Procedure final_grade
        (P_student_id   IN student.student_id%type,
         P_section_id   IN section.section_id%TYPE,
         P_Final_grade  OUT enrollment.final_grade%TYPE,
         P_Exit_Code    OUT CHAR);
END MANAGE_GRADES;
```

The next step is to add the function to the package body. To perform this calculation, you will need a number of variables to hold values as the calculation is carried out.

This exercise is also a very good review of the data relationships among the student tables. Before you read through the next step, review [Appendix B](#), which has an entity–relationship diagram (ERD) of the STUDENT schema and descriptions of the tables and their columns.

When calculating the final grade, there are many things that you must keep in mind:

- Each student is enrolled in a course, and this information is captured in the enrollment table.
- The table holds the final grade only for each student enrolled in one section.
- Each section has its own set of elements that are evaluated to calculate the final grade.
- All grades for these elements (which have been entered, meaning there is no NULL value in the database) are in the grade table.
- Every grade has a grade type code. These codes represent the grade type. For example, the grade type QZ stands for quiz. The description of each GRADE_TYPE comes from the GRADE_TYPE table.
- The GRADE_TYPE_WEIGHT table holds key information for this calculation. There is one entry in this table for each grade type that is utilized in a given section (not all grade types exist for each section).
- In the GRADE_TYPE_WEIGHT table, the NUMBER_PER_SECTION column lists how many times a grade type should be entered to compute the final grade for a particular student in a particular section of a particular course. This helps you determine whether all grades for a given grade type have been entered and whether too many grades for a given grade type have been entered.
- You must take into consideration the drop_lowest flag. The drop_lowest flag can hold a value of Y or N. If the drop lowest flag is Y [Y = Yes, N = No], then you must drop the lowest grade from the grade type when calculating the final grade. The PERCENT_OF_FINAL_GRADE column refers to all the grades for a given grade type. If the homework element represents 20 percent of the final grade, and there are five homework assignments and a drop_lowest flag, then each remaining homework is worth 5 percent. When calculating the final grade, you divide the PERCENT_OF_FINAL_GRADE by the NUMBER_PER_SECTION (note that would be NUMBER_PER_SECTION – 1 if drop_lowest = Y).

Exit codes should be defined as one of the following five:

S = Success, the final grade has been computed. If the grade cannot be computed, then the final grade will be NULL and the exit code will be one of the other four options.

I = Incomplete; not all the required grades have been entered for this student in this section.

T = Too many grades exist for this student. For example, there should be only four homework grades, but instead there are six.

N = No grades have been entered for this student in this section.

E = There was a general computation error (exception When_others). This kind of exit code allows the procedure to compute final grades when it can; if an Oracle

error is somehow raised by some of them, the calling program can still proceed with the grades that have been computed.

To calculate the final grade, you will need a number of variables to hold temporary values during the calculation. Initially the code will create all the variables for the procedure `final_grade`, but then it will leave the main block with just the statement `NULL`; this allows you to compile the procedure and check all of the syntax for the variable declaration one step at a time.

The `student_id`, `section_id`, and `grade_type_code` will be values carried from one part of the program to another—which is why you created a variable for each of them. Each instance of a grade will be computed to determine what percentage of the final grade it represents. A counter is needed while processing each individual grade to ensure there are enough grades for the given grade count. A lowest grade variable holds each grade during the examination to see whether it is the lowest. In the end, once the lowest grade is known for a given grade type, it can be removed from the final grade. Additionally, two variables are used as row counters to determine whether the cursor was opened.

The next example shows the package body in a stub format; that is, this example includes all of the necessary variables but no actual processing code has been written. The reason you start with this step when writing the package body is to ensure that all of the syntax is correct. Once this stub compiles without errors, you can then work on the rest of the code for the package body.

For Example *ch21_16.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE BODY MANAGE_GRADES AS
    Procedure final_grade
        (P_student_id    IN student.student_id%type,
         P_section_id    IN section.section_id%TYPE,
         P_Final_grade   OUT enrollment.final_grade%TYPE,
         P_Exit_Code     OUT CHAR)
    IS
        v_student_id          student.student_id%TYPE;
        v_section_id          section.section_id%TYPE;
        v_grade_type_code     grade_type_weight.grade_type_code%TYPE;
        v_grade_percent       NUMBER;
        v_final_grade         NUMBER;
        v_grade_count         NUMBER;
        v_lowest_grade        NUMBER;
        v_exit_code           CHAR(1) := 'S';
        v_no_rows1             CHAR(1) := 'N';
        v_no_rows2             CHAR(1) := 'N';
        e_no_grade             EXCEPTION;
    BEGIN
        NULL;
    END;
END MANAGE_GRADES;
```

The full package body is provided in the next example. Comments have been placed inside the code to explain what is being done at each step. It is a good idea to include comments within your code to help the next person who has to make changes to the

package.

For Example *ch21_17.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE BODY MANAGE_GRADES AS
    Procedure final_grade
        (P_student_id  IN student.student_id%type,
         P_section_id  IN section.section_id%TYPE,
         P_Final_grade OUT enrollment.final_grade%TYPE,
         P_Exit_Code    OUT CHAR)
    IS
        v_student_id      student.student_id%TYPE;
        v_section_id      section.section_id%TYPE;
        v_grade_type_code grade_type_weight.grade_type_code%TYPE;
        v_grade_percent   NUMBER;
        v_final_grade     NUMBER;
        v_grade_count     NUMBER;
        v_lowest_grade    NUMBER;
        v_exit_code       CHAR(1) := 'S';
        v_no_rows1        CHAR(1) := 'N';
        v_no_rows2        CHAR(1) := 'N';
        e_no_grade        EXCEPTION;
    BEGIN
        v_section_id := p_section_id;
        v_student_id := p_student_id;
        -- Start loop of grade types for the section.
        FOR r_grade IN c_grade_type(v_section_id, v_student_id)
        LOOP
            -- Since cursor is open it has a result
            -- set; change indicator.
            v_no_rows1 := 'Y';
            -- To hold the number of grades per section,
            -- reset to 0 before detailed cursor loops.
            v_grade_count := 0;
            v_grade_type_code := r_grade.GRADE_TYPE_CODE;
            -- Variable to hold the lowest grade.
            -- 500 will not be the lowest grade.
            v_lowest_grade := 500;
            -- Determine what to multiply a grade by to
            -- compute final grade; must take into consideration
            -- if the drop lowest grade indicator is Y.
            SELECT (r_grade.percent_of_final_grade /
                    DECODE(r_grade.drop_lowest, 'Y',
                           (r_grade.number_per_section - 1),
                           r_grade.number_per_section
                    ))* 0.01
            INTO v_grade_percent
            FROM dual;
            -- Open cursor of detailed grade for a student in a
            -- given section.
            FOR r_detail IN c_grades(v_grade_type_code,
                                      v_student_id, v_section_id) LOOP
                -- Since cursor is open it has a result
                -- set; change indicator.
                v_no_rows2 := 'Y';
                v_grade_count := v_grade_count + 1;
                -- Handle the situation where there are more
                -- entries for grades of a given grade type
```

```

    - than there should be for that section.
        If v_grade_count > r_grade.number_per_section THEN
            v_exit_code := 'T';
            raise e_no_grade;
        END IF;
    - If drop lowest flag is Y determine which is lowest
    - grade to drop.
        IF r_grade.drop_lowest = 'Y' THEN
            IF nvl(v_lowest_grade, 0) >=
                r_detail.numeric_grade
            THEN
                v_lowest_grade := r_detail.numeric_grade;
            END IF;
        END IF;
    - Increment the final grade with percentage of current
    - grade in the detail loop.
        v_final_grade := nvl(v_final_grade, 0) +
            (r_detail.numeric_grade * v_grade_percent);
    END LOOP;
    - Once detailed loop is finished, if the number of grades
    - for a given student for a given grade type and section
    - is less than the required amount, raise an exception.
        IF v_grade_count < r_grade.NUMBER_PER_SECTION THEN
            v_exit_code := 'I';
            raise e_no_grade;
        END IF;
    - If the drop lowest flag was Y, then you need to take
    - the lowest grade out of the final grade; it was not
    - known when it was added which was the lowest grade
    - to drop until all grades were examined.
        IF r_grade.drop_lowest = 'Y' THEN
            v_final_grade := nvl(v_final_grade, 0) -
                (v_lowest_grade * v_grade_percent);
        END IF;
    END LOOP;
    - If either cursor had no rows then there is an error.
    IF v_no_rows1 = 'N' OR v_no_rows2 = 'N' THEN
        v_exit_code := 'N';
        raise e_no_grade;
    END IF;
    P_final_grade := v_final_grade;
    P_exit_code := v_exit_code;
EXCEPTION
    WHEN e_no_grade THEN
        P_final_grade := null;
        P_exit_code := v_exit_code;
    WHEN OTHERS THEN
        P_final_grade := null;
        P_exit_code := 'E';
    END final_grade;
END MANAGE_GRADES;

```

The following example is an anonymous block to test the `final_grade` procedure. The block asks for a `student_id` and a `section_id` and returns the final grade and an exit code.

It is often a good idea to review the parameter order for the procedure before you write the anonymous block to run the code. In SQL*Plus, this can be done by running a describe

command on a procedure.

[Click here to view code image](#)

```
SQL> desc manage_grades
PROCEDURE FINAL_GRADE
```

Argument Name	Type	In/Out	Default?
P_STUDENT_ID	NUMBER(8)	IN	
P_SECTION_ID	NUMBER(8)	IN	
P_FINAL_GRADE	NUMBER(3)	OUT	
P_EXIT_CODE	CHAR	OUT	

In SQL Developer, you can expand the node for packages and hover your cursor over the procedure to obtain more details. By doing so, you can see both what has been declared in the package header and what is compiled in the package body. This is illustrated in [Figure 21.1](#).

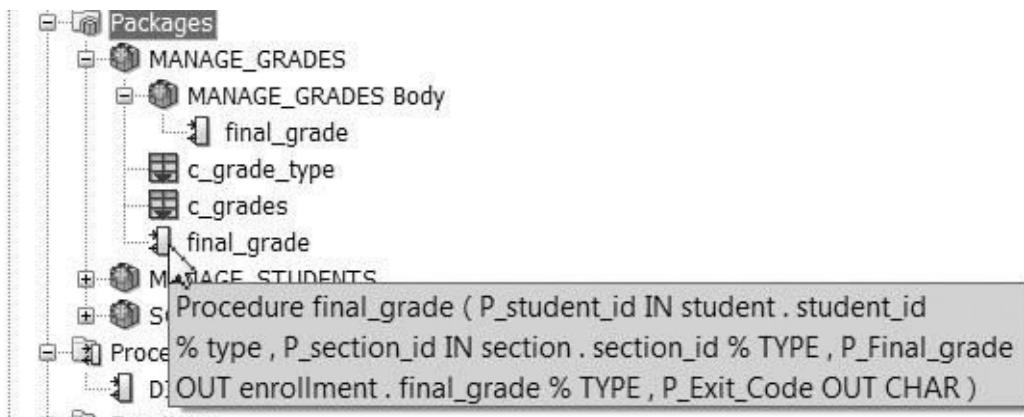


Figure 21.1 Manage_Grades Package as Seen in SQL Developer

The following example is an anonymous block that can be used to run the package `manage_grades`.

For Example *ch21_18.sql*

[Click here to view code image](#)

```
SET SERVEROUTPUT ON

DECLARE
  v_student_id    student.student_id%TYPE := &sv_student_id;
  v_section_id    section.section_id%TYPE := &sv_section_id;
  v_final_grade   enrollment.final_grade%TYPE;
  v_exit_code     CHAR;
BEGIN
  manage_grades.final_grade(v_student_id, v_section_id,
    v_final_grade, v_exit_code);
  DBMS_OUTPUT.PUT_LINE('The Final Grade is '||v_final_grade);
  DBMS_OUTPUT.PUT_LINE('The Exit Code is '||v_exit_code);
END;
```

If you were to run this script for a `student_id` of 102 and a `section_id` of 89, you would get the following result in SQL*Plus. In SQL Developer, you would see the full code as you ran it and with the variables substituted for 102 and 89. Both outputs have the same final lines that appear after the anonymous block completes.

[Click here to view code image](#)

```

Enter value for sv_student_id: 102
old 2: v_student_id student.student_id%TYPE := &sv_student_id;
new 2: v_student_id student.student_id%TYPE := 102;
Enter value for sv_section_id: 86
old 3: v_section_id section.section_id%TYPE := &sv_section_id;
new 3: v_section_id section.section_id%TYPE := 86;
The Final Grade is 89
The Exit Code is S
PL/SQL procedure successfully completed.

```

The next step is to add a function to the `manage_grades` package specification called `median_grade` that takes in a course number (`p_course_number`), a section number (`p_section_number`), and a grade type (`p_grade_type`) and returns a `work_grade.grade%TYPE`. Cursors that will be used by this function also need to be added as well as any types that will be required by the function.

For Example *ch21_19.sql*

[Click here to view code image](#)

```

CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
    -- Cursor to loop through all grade types for a given section.
    CURSOR c_grade_type
        (pc_section_id section.section_id%TYPE,
         PC_student_ID student.student_id%TYPE)
    IS
        SELECT GRADE_TYPE_CODE,
               NUMBER_PER_SECTION,
               PERCENT_OF_FINAL_GRADE,
               DROP_LOWEST
          FROM grade_Type_weight
         WHERE section_id = pc_section_id
           AND section_id IN (SELECT section_id
                               FROM grade
                              WHERE student_id = pc_student_id);
    -- Cursor to loop through all grades for a given student
    -- in a given section.
    CURSOR c_grades
        (p_grade_type_code
         grade_Type_weight.grade_type_code%TYPE,
         pc_student_id student.student_id%TYPE,
         pc_section_id section.section_id%TYPE) IS
        SELECT grade_type_code,grade_code_occurrence,
               numeric_grade
          FROM grade
         WHERE student_id = pc_student_id
           AND section_id = pc_section_id
           AND grade_type_code = p_grade_type_code;
    -- Function to calculate a student's final grade
    -- in one section.
    Procedure final_grade
        (P_student_id   IN student.student_id%type,
         P_section_id   IN section.section_id%TYPE,
         P_Final_grade  OUT enrollment.final_grade%TYPE,
         P_Exit_Code    OUT CHAR);
    -----
    -- Function to calculate the median grade .
    FUNCTION median_grade
        (p_course_number section.course_no%TYPE,

```

```

        p_section_number section.section_no%TYPE,
        p_grade_type grade.grade_type_code%TYPE)
    RETURN grade.numeric_grade%TYPE;
CURSOR c_work_grade
    (p_course_no section.course_no%TYPE,
     p_section_no section.section_no%TYPE,
     p_grade_type_code grade.grade_type_code%TYPE
    )IS
    SELECT distinct numeric_grade
        FROM grade
       WHERE section_id = (SELECT section_id
                            FROM section
                           WHERE course_no= p_course_no
                             AND section_no = p_section_no)
         AND grade_type_code = p_grade_type_code
    ORDER BY numeric_grade;
TYPE t_grade_type IS TABLE OF c_work_grade%ROWTYPE
    INDEX BY BINARY_INTEGER;
t_grade t_grade_type;
END MANAGE_GRADES;

```

The next step is to add a function to the `manage_grades` package specification called `median_grade` that takes in a course number (`p_cnumber`), a section number (`p_snumber`), and a grade type (`p_grade_type`). This function will return the median grade (`work_grade.grade%TYPE` data type) based on those three components. For example, one might use this function to answer the question, “What is the median grade of homework assignments in Introduction to Java section 2?” A true median can contain two values. Because this function can return only one value, if the median is made of two values, then the function will return the average of the two.

For Example ch21_20.sql

[Click here to view code image](#)

```

CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
CREATE OR REPLACE PACKAGE BODY MANAGE_GRADES AS
    Procedure final_grade
        (P_student_id  IN student.student_id%type,
         P_section_id  IN section.section_id%TYPE,
         P_Final_grade OUT enrollment.final_grade%TYPE,
         P_Exit_Code    OUT CHAR)
    IS
        v_student_id      student.student_id%TYPE;
        v_section_id      section.section_id%TYPE;
        v_grade_type_code grade_type_weight.grade_type_code%TYPE;
        v_grade_percent   NUMBER;
        v_final_grade     NUMBER;
        v_grade_count     NUMBER;
        v_lowest_grade    NUMBER;
        v_exit_code       CHAR(1) := 'S';
        - Next two variables are used to calculate whether a cursor
        - has no result set.
        v_no_rows1          CHAR(1) := 'N';
        v_no_rows2          CHAR(1) := 'N';
        e_no_grade          EXCEPTION;
    BEGIN
        v_section_id := p_section_id;
        v_student_id := p_student_id;

```

```

    - Start loop of grade types for the section.
        FOR r_grade IN c_grade_type(v_section_id, v_student_id)
        LOOP
    - Since cursor is open it has a result
    - set; change indicator.
        v_no_rows1 := 'Y';
    - To hold the number of grades per section,
    - reset to 0 before detailed cursor loops.
        v_grade_count := 0;
        v_grade_type_code := r_grade.GRADE_TYPE_CODE;
    - Variable to hold the lowest grade.
    - 500 will not be the lowest grade.
        v_lowest_grade := 500;
    - Determine what to multiply a grade by to
    - compute final grade; must take into consideration
    - if the drop lowest grade indicator is Y.
        SELECT (r_grade.percent_of_final_grade /
                DECODE(r_grade.drop_lowest, 'Y',
                        (r_grade.number_per_section - 1),
                        r_grade.number_per_section
                )) * 0.01
        INTO v_grade_percent
        FROM dual;
    - Open cursor of detailed grade for a student in a
    - given section.
        FOR r_detail IN c_grades(v_grade_type_code,
                                v_student_id, v_section_id) LOOP
    - Since cursor is open it has a result
    - set; change indicator.
        v_no_rows2 := 'Y';
        v_grade_count := v_grade_count + 1;
    - Handle the situation where there are more
    - entries for grades of a given grade type
    - than there should be for that section.
        IF v_grade_count > r_grade.number_per_section THEN
            v_exit_code := 'T';
            raise e_no_grade;
        END IF;
    - If drop lowest flag is Y determine which is lowest
    - grade to drop.
        IF r_grade.drop_lowest = 'Y' THEN
            IF nvl(v_lowest_grade, 0) >=
                r_detail.numeric_grade
            THEN
                v_lowest_grade := r_detail.numeric_grade;
            END IF;
        END IF;
    - Increment the final grade with percentage of current
    - grade in the detail loop.
        v_final_grade := nvl(v_final_grade, 0) +
            (r_detail.numeric_grade * v_grade_percent);
    END LOOP;
    - Once detailed loop is finished, if the number of grades
    - for a given student for a given grade type and section
    - is less than the required amount, raise an exception.
        IF v_grade_count < r_grade.NUMBER_PER_SECTION THEN
            v_exit_code := 'I';
            raise e_no_grade;
        END IF;

```

```

    - If the drop lowest flag was Y then you need to take
    - the lowest grade out of the final grade. It was not
    - known when it was added which was the lowest grade
    - to drop until all grades were examined.
        IF r_grade.drop_lowest = 'Y' THEN
            v_final_grade := nvl(v_final_grade, 0) -
                (v_lowest_grade * v_grade_percent);
        END IF;
    END LOOP;
    - If either cursor had no rows then there is an error.
    IF v_no_rows1 = 'N' OR v_no_rows2 = 'N' THEN
        v_exit_code := 'N';
        raise e_no_grade;
    END IF;
    P_final_grade := v_final_grade;
    P_exit_code := v_exit_code;
EXCEPTION
    WHEN e_no_grade THEN
        P_final_grade := null;
        P_exit_code := v_exit_code;
    WHEN OTHERS THEN
        P_final_grade := null;
        P_exit_code := 'E';
END final_grade;

FUNCTION median_grade
    (p_course_number section.course_no%TYPE,
    p_section_number section.section_no%TYPE,
    p_grade_type grade.grade_type_code%TYPE)
RETURN grade.numeric_grade%TYPE
IS
BEGIN
    FOR r_work_grade
        IN c_work_grade(p_course_number, p_section_number, p_grade_type)
    LOOP
        t_grade(NVL(t_grade.COUNT, 0) + 1).numeric_grade :=
    r_work_grade.numeric_grade;
    END LOOP;
    IF t_grade.COUNT = 0
    THEN
        RETURN NULL;
    ELSE
        IF MOD(t_grade.COUNT, 2) = 0
        THEN
            - There is an even number of work grades. Find the middle
            - two and average them.
            RETURN (t_grade(t_grade.COUNT / 2).numeric_grade +
                    t_grade((t_grade.COUNT / 2) + 1).numeric_grade
                    ) / 2;
        ELSE
            - There is an odd number of grades. Return the one in the middle.
            RETURN t_grade(TRUNC(t_grade.COUNT / 2, 0) + 1).numeric_grade;
        END IF;
    END IF;
EXCEPTION
    WHEN OTHERS
    THEN
        RETURN NULL;
END median_grade;

```

```
END MANAGE_GRADES;
```

The following example is a **SELECT** statement that makes use of the function **median_grade** and shows the median grade for all grade types in sections 1 and 2 of course 25.

For Example *ch21_21.sql*

[Click here to view code image](#)

```
SELECT COURSE_NO,
       COURSE_NAME,
       SECTION_NO,
       GRADE_TYPE,
       manage_grades.median_grade
          (COURSE_NO,
           SECTION_NO,
           GRADE_TYPE)
       median_grade
  FROM
  (SELECT DISTINCT
            C.COURSE_NO          COURSE_NO,
            C.DESCRIPTION        COURSE_NAME,
            S.SECTION_NO         SECTION_NO,
            G.GRADE_TYPE_CODE   GRADE_TYPE
       FROM SECTION S, COURSE C, ENROLLMENT E, GRADE G
      WHERE C.course_no = s.course_no
        AND s.section_id = e.section_id
        AND e.student_id = g.student_id
        AND c.course_no = 25
        AND s.section_no between 1 and 2
    ORDER BY 1, 4, 3) grade_source
```

The results of the **SELECT** statement using the **median_grade** function for all grade types in sections 1 and 2 of course 25 would be as follows:

[Click here to view code image](#)

COURSE_NO	COURSE_NAME	SECTION_NO	GRADE_TYPE	MEDIAN_GRADE
25	Intro to Programming	1	FI	98
25	Intro to Programming	2	FI	71
25	Intro to Programming	1	HM	76
25	Intro to Programming	2	HM	83
25	Intro to Programming	1	MT	86
25	Intro to Programming	2	MT	89
25	Intro to Programming	1	PA	91
25	Intro to Programming	2	PA	97
25	Intro to Programming	1	QZ	71
25	Intro to Programming	2	QZ	78

10 rows selected.

Lab 21.4: Package Instantiation and Initialization

After this lab, you will be able to

- [Create Package Variables During Initialization](#)

The first time a session makes any reference to a package, Oracle instantiates the package. If multiple sessions are connected to the database at the same time, each will have its own instantiation of that package. The package is loaded into the SGA of the database instance, which makes all elements of the package available in memory. Anything in the SGA will be accessed more quickly than if the database needs to query tables.

The instantiation process means the following steps will take place:

1. Public constants in the package will be assigned an initial value.
2. Public variables, which have a declaration section, will be assigned an initial value.
3. If there is an initialization section in the package body, it will be executed.

Creating Package Variables During Initialization

The first time a package is called within a user session, the code in the initialization section of the package will be executed if it exists. This step is done only once; it is not repeated if the user calls other procedures or functions for that package. The initialization section encompasses everything between the BEGIN statement and the END statement of the package body. Variables, cursors, and user-defined data types used by numerous procedures and functions can be declared once at the beginning of the package specification and then be used by the functions and procedures within the package without having to declare them again.

The following example creates a package global variable called `v_current_date` in the `student_api` package.

For Example `ch21_22.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE school_api AS
    v_current_date DATE;
    PROCEDURE Discount_Cost;
    FUNCTION new_instructor_id
        RETURN instructor.instructor_id%TYPE;
END school_api;
```

The following script adds an initialization section that assigns the current system date to the variable `v_current_date`. This variable can then be used in any procedure in the package that needs to make use of the current date.

For Example `ch21_23.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE BODY school_api AS
    PROCEDURE discount_cost
    IS
        CURSOR c_group_discount
        IS
            SELECT distinct s.course_no, c.description
            FROM section s, enrollment e, course c
            WHERE s.section_id = e.section_id
            GROUP BY s.course_no, c.description,
                    e.section_id, s.section_id
```

```

HAVING COUNT(*) >=8;
BEGIN
  FOR r_group_discount IN c_group_discount
  LOOP
    UPDATE course
      SET cost = cost * .95
      WHERE course_no = r_group_discount.course_no;
    DBMS_OUTPUT.PUT_LINE
      ('A 5% discount has been given to'
      ||r_group_discount.course_no||
      ||r_group_discount.description);
  END LOOP;
END discount_cost;
FUNCTION new_instructor_id
  RETURN instructor.instructor_id%TYPE
IS
  v_new_instid instructor.instructor_id%TYPE;
BEGIN
  SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
    INTO v_new_instid
    FROM dual;
  RETURN v_new_instid;
EXCEPTION
  WHEN OTHERS
  THEN
    DECLARE
      v_sqlerrm VARCHAR2(250) :=
        SUBSTR(SQLERRM,1,250);
    BEGIN
      RAISE_APPLICATION_ERROR(-20003,
        'Error in instructor_id: '||v_sqlerrm);
    END;
END new_instructor_id;
BEGIN
  SELECT trunc(sysdate, 'DD')
    INTO v_current_date
    FROM dual;
END school_api;

```

Lab 21.5: SERIALLY_REUSABLE Packages

After this lab, you will be able to

- [Use the SERIALLY REUSABLE Pragma](#)

In the last section, you learned how to load objects into the SGA as part of the instantiation process. This was done to help improve performance of the package. This process has some drawbacks, however. The objects are held in memory and can produce some undesirable side effects and errors if, for example, a package cursor is left open. Moreover, if package cursors are large, they can hold onto a large amount of the session's memory and then fail to release it. To avoid these side effects, you can make use of the SERIALLY_REUSABLE pragma.

Using the SERIALLY_REUSABLE Pragma

The SERIALLY_REUSABLE pragma must be used in both the package specification and the package body if you want to take advantage of what it has to offer. This pragma identifies the package as serially reusable. When a package is marked as such, then the package state can be reduced from the entire session to just a call of a program in the package. The result is the opposite of the initialization advantage; it means the values of package variables and other elements will not persist. The syntax to invoke this pragma is to add the following line after IS in the package header and body:

```
PRAGMA SERIALLY_REUSABLE;
```

Here are some points to keep in mind when using serialized packages:

- The global memory for serialized packages is allocated in the SGA, not in the user global area (UGA). This approach allows the package work area to be reused. Each time the package is reused, its package-level variables are initialized to their default values or to NULL, and its initialization section is reexecuted.
- The maximum number of work areas needed for a serialized package is determined by the number of concurrent users of that package. The increased use of SGA memory is offset by the decreased use of the UGA or program memory. Moreover, the database ages out work areas not in use if it needs to reclaim memory from the SGA for other requests.
- If a package is not SERIALLY_REUSABLE, its package state is stored in the UGA for each user. Therefore, the amount of UGA memory needed increases linearly with the number of users, limiting scalability. The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days.

The following script is an extremely simple example that illustrates how the SERIALLY_REUSABLE pragma operates (a longer example would be needed to more clearly show a use case where this pragma would be necessary).

For Example ch21_24.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE show_date
IS
    PRAGMA SERIALLY_REUSABLE;
    the_date DATE := SYSDATE + 4;
    PROCEDURE display_DATE;
    PROCEDURE set_date;
END show_date;
/
CREATE OR REPLACE PACKAGE BODY show_date
IS
    PRAGMA SERIALLY_REUSABLE;
    PROCEDURE display_DATE IS
        BEGIN
            DBMS_OUTPUT.PUT_LINE ('The date is ' || show_date.the_date);
        END;
    -- Initialize package state
```

```

PROCEDURE set_date IS
BEGIN
    show_date.the_date := sysdate;
END;
END show_date;

```

The next example shows a PL/SQL block to execute this procedure and illustrate how it behaves.

For Example *ch21_25.sql*

[Click here to view code image](#)

```

begin
    -- initialize and print the package variable
    show_date.display_DATE;
    -- change the value of the variable the_date
    show_date.set_date;
    -- Display the new value of variable the_date
    show_date.display_DATE;
end;
/
begin
    show_date.display_DATE;
end;
/

```

If this script were run on July 27, 2014, the result of this would be as follows:

```

anonymous block completed
The date is 31-JUL-14

The date is 27-JUL-14

anonymous block completed
The date is 31-JUL-14

```

The example shows how the value of the variable `the_date` changes depending on how it is called. When the `SERIALLY_REUSABLE` pragma is not used, the value of a package variable persists in memory and does not change until a program changes it programmatically. In this case, because the package uses the `SERIALLY_REUSABLE` pragma, the behavior is different. The first time the package is called in a PL/SQL block, the initialization section of the package is called and the value of the variable `the_date` is set to the system date plus four days. This value is then displayed. Next, the procedure `show_date.set_date` is executed and the value of `the_date` is reset to be the system date. Because the `SERIALLY_REUSABLE` pragma has been used, the value of `the_date` is not retained. The next time the package is referenced in a second PL/SQL block, the value of `the_date` is reset by the initialization section of the package.

When the package uses the pragma `SERIALLY_REUSABLE`, however, the package state is kept in the work area of the system global area. The package state will persist only for the duration of a server call. Once that call completes, the work area is flushed. If another server call references the same package, Oracle will reinstantiate the package—which means it reinitializes the package. Anything that changed the variables in the package will be lost. Once a unit of work is complete, the Oracle database takes care of the following tasks:

- Closes any open cursors.
- Frees some nonreusable memory
- Returns the package instantiation to the pool of reusable instantiations kept for this package

Database triggers, stand-alone SQL statements, and any other type of PL/SQL subprogram cannot access a package that makes use of the `SERIALLY_REUSABLE` pragma.

Summary

In this chapter, you learned how to create packages. You first investigated the details of the package specification and the package body. You also learned how to call the stored package and explored the various types of package components such as private objects and cursor variables. Then you were introduced to an elaborate package that pulled together many of the concepts discussed in this and other chapters. Initialization of the package was addressed in terms of initialization of variables. Additionally, you saw how to prevent Oracle from holding onto memory by using the `SERIALLY_REUSABLE` pragma in the package definition.

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.

22. Stored Code

In this chapter, you will learn about

- [Gathering Information about Stored Code](#)

In [Chapter 19](#) you learned about procedures, in [Chapter 20](#) you learned about functions, and in [Chapter 21](#) you learned about the process of grouping functions and procedures into a package. Now you will learn more about what it means to have code bundled into a package. Numerous data dictionary views can be accessed to gather information about the objects in a package.

Functions in packages are required to meet additional restrictions to be used in a SELECT statement. In this chapter, you will learn what those restrictions are and how to enforce them. You will also learn an advanced technique to overload a function or procedure so that it executes different code depending on the type of parameter passed in.

Lab 22.1: Gathering Information about Stored Code

After this lab, you will be able to

- [Get Stored Code Information from the Data Dictionary](#)
- [Overload Modules](#)

Stored programs are held in a compiled form in the database. Information about such stored programs is accessible through various data dictionary views. In [Chapter 19](#), you learned about two data dictionary views: `USER_OBJECTS` and `USER_SOURCE`. In [Chapter 13](#), you learned about another view, `USER_TRIGGERS`. A few other data dictionary views are also useful for obtaining information about stored code. In this lab, you will learn how to take advantage of these options.

Getting Stored Code Information from the Data Dictionary

The Oracle data dictionary contains system views that can be used to examine all the stored procedures, functions, and packages in the current schema of the database. They also provide the current status of the stored code. The primary view to be used for this purpose is the `USER_OBJECTS` view you encountered in [Chapter 11](#). This view has information about all database objects in the schema of the current user. In contrast, if you want to see all the objects in other schemas to which the current user has access, you would use the `ALL_OBJECTS` view. There is also a `DBA_OBJECTS` view that lists all objects in the database regardless of privilege. The status of each object will be marked as either `VALID` or `INVALID`. That status can change from `VALID` to `INVALID` if an underlying table is altered or if privileges on a referenced object are revoked by the creator of the function, procedure, or package.

The following SELECT statement lists all functions, procedures, and packages that are

in the schema of the current user.

For Example *ch22_1.sql*

[Click here to view code image](#)

```
SELECT OBJECT_TYPE, OBJECT_NAME, STATUS
  FROM USER_OBJECTS
 WHERE OBJECT_TYPE IN
      ('FUNCTION', 'PROCEDURE', 'PACKAGE',
       'PACKAGE_BODY')
 ORDER BY OBJECT_TYPE;
```

The `user_source` view in the data dictionary can be used to extract the source code for procedures, functions, and packages. The column `TEXT` holds the actual source code text, `NAME` holds the name, and `TYPE` indicates if it is a function, procedure, package, or package body. The text is listed in order by line number in the column `line`.

The following example creates a function called `scode_at_line` that provides an easy mechanism for retrieving the text from a stored program for a specified line number.

For Example *ch22_2.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE FUNCTION scode_at_line
  (i_name_in IN VARCHAR2,
   i_line_in IN INTEGER := 1,
   i_type_in IN VARCHAR2 := NULL)
RETURN VARCHAR2
IS
  CURSOR scode_cur IS
    SELECT text
      FROM user_source
     WHERE name = UPPER (i_name_in)
       AND (type = UPPER (i_type_in)
            OR i_type_in IS NULL)
       AND line = i_line_in;
  scode_rec scode_cur%ROWTYPE;
BEGIN
  OPEN scode_cur;
  FETCH scode_cur INTO scode_rec;
  IF scode_cur%NOTFOUND
    THEN
      CLOSE scode_cur;
      RETURN NULL;
    ELSE
      CLOSE scode_cur;
      RETURN scode_rec.text;
    END IF;
END;
```

This function is useful if a developer receives a compilation error message referring to a particular line number in an object. The developer can call this function to find out which text is the source of the error.

The `scode_at_line` function uses three parameters:

`name_in` The name of the stored object.

line_in The line number of the line you wish to retrieve. The default value is 1.

type_in The type of object you want to view. The default for **type_in** is NULL.

The default values are designed to make this function as easy as possible to use.

By the Way

The output from a call to **SHOW ERRORS** in SQL*Plus displays the line number in which an error occurred, but the line number doesn't correspond to the line in your text file. Instead, it relates directly to the line number stored with the source code in the **USER_SOURCE** view.

You can use the **USER_ERRORS** view to get more details about compilation errors that occur when you are writing code. This view stores current errors on the user's stored objects. The text file contains the text of the error—a handy feature when you are trying to pin down the details of a compilation error. Following are the columns for the **USER_ERRORS** view that you would see if you entered the command **DESC USER_ERRORS** in SQL*Plus.

[Click here to view code image](#)

Name	Null?	Type
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(12)
SEQUENCE	NOT NULL	NUMBER
LINE	NOT NULL	NUMBER
POSITION	NOT NULL	NUMBER
TEXT	NOT NULL	VARCHAR2(2000)

The following code fragment produces a forced error so that we can demonstrate the various methods used to debug a problem:

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE FORCE_ERROR
as
BEGIN
  SELECT course_no
  INTO v_temp
  FROM course;
END;
```

In SQL Developer, the errors would then be seen in the compiler log screen. In SQL*Plus, you need to type **SHO ERR** to see the same information. In either case, the errors will be shown as follows:

[Click here to view code image](#)

```
Errors for PROCEDURE FORCE_ERROR:
LINE/COL ERROR
-----
4/4          PL/SQL: SQL Statement ignored
5/9          PLS-00201: identifier 'V_TEMP' must be declared
6/4          PL/SQL: ORA-00904: : invalid identifier
```

You can use a **SELECT** statement to retrieve information from the **USER_ERRORS**

view:

[Click here to view code image](#)

```
SELECT line||'/'||position "LINE/COL", TEXT "ERROR"
  FROM user_errors
 WHERE name = 'FORCE_ERROR'
```

It is important to know how to retrieve this information from the **USER_ERRORS** view because the **SHO ERR** command simply brings up the most recent errors. If you run a script creating a number of objects, then you must rely on the **USER_ERRORS** view to identify all of the errors.

The **USER_DEPENDENCIES** view is useful for analyzing how table changes or changes to other stored procedures affect other parts of the script. If you plan to redesign tables, for example, you might want to assess their impact by examining the information in this view. The **ALL_DEPENDENCIES** and **DBA_DEPENDENCIES** views show all dependencies for procedures, functions, package specifications, and package bodies. Entering the command **DESC USER_DEPENDENCIES** in SQL&*Plus produces the following output:

[Click here to view code image](#)

Name	Null?	Type
-----	-----	-----
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(12)
REFERENCED_OWNER		VARCHAR2(30)
REFERENCED_NAME	NOT NULL	VARCHAR2(30)
REFERENCED_TYPE		VARCHAR2(12)
REFERENCED_LINK_NAME		VARCHAR2(30)

The following **SELECT** statement demonstrates the dependencies for the **school_api** package:

```
SELECT referenced_name
  FROM user_dependencies
 WHERE name = 'SCHOOL_API';
```

This is the result of running the **SELECT** statement:

REFERENCED_NAME

STANDARD
STANDARD
DUAL
DBMS_STANDARD
DBMS_OUTPUT
COURSE
ENROLLMENT
INSTRUCTOR
INSTRUCTOR
INSTRUCTOR_ID_SEQ
SCHOOL_API
SECTION

This list of dependencies for the **school_api** package lists all objects referenced in the package. It includes tables, sequences, and procedures (even Oracle-supplied packages). This information is very useful when you are planning a change to a database structure.

You can easily pinpoint what the ramifications are for any database changes.

The DESC command in SQL*Plus is used to describe the columns in a table as well as to identify procedures, packages, and functions. This command shows all the parameters with their default values and indicates whether they are IN or OUT. If the object is a function, then the return data type is displayed. This is very different from the USER_DEPENDENCIES view, which provides information on all the objects that are referenced in a package, function, or procedure. In SQL Developer, the same information can be obtained by finding the name of the object in the tree and hovering the cursor over the name.

Overloading Modules

When you overload modules, you give two or more modules the same name. The parameter lists of the modules must differ in a manner significant enough for the compiler (and run-time engine) to distinguish between the different versions.

You can overload modules in three contexts:

1. In a local module in the same PL/SQL block
2. In a package specification
3. In a package body

The following changes to the school_api package demonstrate how module overloading can be used.

For Example ch22_3.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE school_api AS
    v_current_date DATE;
    PROCEDURE Discount_Cost;
    FUNCTION new_instructor_id
        RETURN instructor.instructor_id%TYPE;
        FUNCTION total_cost_for_student
            (i_student_id IN student.student_id%TYPE)
        RETURN course.cost%TYPE;
        PRAGMA RESTRICT_REFERENCES
            (total_cost_for_student, WNDS, WNPS, RNPS);
    PROCEDURE get_student_info
        (i_student_id    IN student.student_id%TYPE,
         o_last_name     OUT student.last_name%TYPE,
         o_first_name    OUT student.first_name%TYPE,
         o_zip           OUT student.zip%TYPE,
         o_return_code   OUT NUMBER);
    PROCEDURE get_student_info
        (i_last_name    IN student.last_name%TYPE,
         i_first_name   IN student.first_name%TYPE,
         o_student_id   OUT student.student_id%TYPE,
         o_zip          OUT student.zip%TYPE,
         o_return_code  OUT NUMBER);
END school_api;
```

In this example of an overloaded procedure, the specification has two procedures with

the same name and different IN parameters (different both in number and in data type). The OUT parameters are also different in number and data type. This overloaded function accepts either of the two sets of IN parameters and performs the version of the function corresponding to the data type passed in. The next example contains the package body.

For Example ch22_4.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE BODY school_api AS
  PROCEDURE discount_cost
  IS
    CURSOR c_group_discount
    IS
      SELECT distinct s.course_no, c.description
        FROM section s, enrollment e, course c
       WHERE s.section_id = e.section_id
     GROUP BY s.course_no, c.description,
              e.section_id, s.section_id
    HAVING COUNT(*) >=8;
  BEGIN
    FOR r_group_discount IN c_group_discount
    LOOP
      UPDATE course
        SET cost = cost * .95
       WHERE course_no = r_group_discount.course_no;
      DBMS_OUTPUT.PUT_LINE
        ('A 5% discount has been given to'
         ||r_group_discount.course_no||
         ||r_group_discount.description);
    END LOOP;
  END discount_cost;
  FUNCTION new_instructor_id
    RETURN instructor.instructor_id%TYPE
  IS
    v_new_instid instructor.instructor_id%TYPE;
  BEGIN
    SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
      INTO v_new_instid
      FROM dual;
    RETURN v_new_instid;
  EXCEPTION
    WHEN OTHERS
    THEN
      DECLARE
        v_sqlerrm VARCHAR2(250) := SUBSTR(SQLERRM,1,250);
      BEGIN
        RAISE_APPLICATION_ERROR(-20003,
          'Error in instructor_id: '||v_sqlerrm);
      END;
  END new_instructor_id;
  FUNCTION total_cost_for_student
    (i_student_id IN student.student_id%TYPE)
    RETURN course.cost%TYPE
  IS
    v_cost course.cost%TYPE;
  BEGIN
    SELECT sum(cost)
```

```

        INTO v_cost
        FROM course c, section s, enrollment e
        WHERE c.course_no = s.course_no
          AND e.section_id = s.section_id
          AND e.student_id = i_student_id;
      RETURN v_cost;
EXCEPTION
  WHEN OTHERS THEN
    RETURN NULL;
END total_cost_for_student;

PROCEDURE get_student_info
  (i_student_id  IN student.student_id%TYPE,
   o_last_name    OUT student.last_name%TYPE,
   o_first_name   OUT student.first_name%TYPE,
   o_zip          OUT student.zip%TYPE,
   o_return_code  OUT NUMBER)
IS
BEGIN
  SELECT last_name, first_name, zip
    INTO o_last_name, o_first_name, o_zip
    FROM student
   WHERE student.student_id = i_student_id;
  o_return_code := 0;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Student ID is not valid.');
    o_return_code := -100;
    o_last_name := NULL;
    o_first_name := NULL;
    o_zip := NULL;
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Error in procedure get_student_info');
END get_student_info;
PROCEDURE get_student_info
  (i_last_name   IN student.last_name%TYPE,
   i_first_name  IN student.first_name%TYPE,
   o_student_id  OUT student.student_id%TYPE,
   o_zip         OUT student.zip%TYPE,
   o_return_code OUT NUMBER)
IS
BEGIN
  SELECT student_id, zip
    INTO o_student_id, o_zip
    FROM student
   WHERE UPPER(last_name) = UPPER(i_last_name)
     AND UPPER(first_name) = UPPER(i_first_name);
  o_return_code := 0;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Student name is not valid.');
    o_return_code := -100;
    o_student_id := NULL;

```

```

    o_zip := NULL;
WHEN OTHERS
THEN
    DBMS_OUTPUT.PUT_LINE
        ('Error in procedure get_student_info');
END get_student_info;
BEGIN
    SELECT TRUNC(sysdate, 'DD')
        INTO v_current_date
        FROM dual;
END school_api;

```

In this version of the `school_api`, a single function name, `get_student_info`, accepts either a single `IN` parameter of `student_id` or two parameters consisting of a student's `last_name` and `first_name`. If a number is passed in, then the procedure looks for the name and ZIP code of the student. If it finds them, they are returned along with a return code of 0. If they cannot be found, then null values are returned along with a return code of 100. If two `VARCHAR2` parameters are passed in, then the procedure searches for the `student_id` corresponding to the names passed in. As with the other version of this procedure, if a match is found, the procedure returns a `student_id`, the student's ZIP code, and a return code of 0. If a match is not found, then the values returned are null and the exit code is 100.

PL/SQL uses overloading in many common functions and built-in packages. For example, `TO_CHAR` converts both numbers and dates to strings. Overloading makes it easy for other programmers to use your code in an API.

The main benefits of overloading are threefold. First, overloading simplifies the call interface of packages and reduces many program names to one. Second, modules are easier to use and hence more likely to be used. The software determines the context. Third, the volume of code is reduced because the code required for different data types is often the same.

Watch Out!

The rules for overloading are as follows: (1) The compiler must be able to distinguish between the two calls at run time. Distinguishing between the uses of the overloaded module is what is important—not solely the differences in the specification or header. (2) The formal parameters must differ in number, order, or data type family. (3) You cannot overload the names of stand-alone modules. (4) Functions differing only in `RETURN` data types cannot be overloaded.

The following PL/SQL block shows how this overloaded function can be used:

[Click here to view code image](#)

```

DECLARE
    v_student_ID  student.student_id%TYPE;
    v_last_name   student.last_name%TYPE;
    v_first_name  student.first_name%TYPE;
    v_zip         student.zip%TYPE;
    v_return_code NUMBER;

```

```

BEGIN
    school_api.get_student_info
        (&&p_id, v_last_name, v_first_name,
         v_zip, v_return_code);
    IF v_return_code = 0
    THEN
        DBMS_OUTPUT.PUT_LINE
            ('Student with ID '||&p_id||' is '||v_first_name
             ||' '||v_last_name
            );
    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('The ID '||&p_id||' is not in the database'
            );
    END IF;
    school_api.get_student_info
        (&&p_last_name , &&p_first_name, v_student_id,
         v_zip , v_return_code);
    IF v_return_code = 0
    THEN
        DBMS_OUTPUT.PUT_LINE
            (&&p_first_name||' '|| &&p_last_name||
             ' has an ID of '||v_student_id      );
    ELSE
        DBMS_OUTPUT.PUT_LINE
            (&&p_first_name||' '|| &&p_last_name||
             ' is not in the database'
            );
    END IF;
END;

```

When you run this script, you will be prompted for these three values. Here is an example of a valid value to enter as the input:

[Click here to view code image](#)

```

Enter value for p_id: 149
Enter value for p_last_name: 'Prochaska'
Enter value for p_first_name: 'Judith'

```

This example demonstrates the benefits of using a `&&` variable. The value for the variable need be entered only once, but if you run the code a second time, you will not be prompted to enter the value again because it is now in memory.

Here are a few points to keep in mind when you overload functions or procedures. These two procedures cannot be overloaded:

[Click here to view code image](#)

```

PROCEDURE calc_total (reg_in IN CHAR);
PROCEDURE calc_total (reg_in IN VARCHAR2).

```

In these two versions of `calc_total`, the two different `IN` variables cannot be distinguished from each other. In the following example, an anchored type (`%TYPE`) is relied on to establish the data type of the second `calc`'s parameter.

[Click here to view code image](#)

```

DECLARE
    PROCEDURE calc (comp_id_IN IN NUMBER)
        IS

```

```
BEGIN ... END;
PROCEDURE calc
(comp_id_IN IN company.comp_id%TYPE)
IS
BEGIN ... END;
```

PL/SQL does not find a conflict at compile time with overloading even though `comp_id` is a numeric column. Instead, you will see the following message at run time:

[Click here to view code image](#)

```
PLS-00307: too many declarations of '<program>' match this call
```

Summary

In this chapter, you learned about the various data dictionary views that can be used to gather information about stored code. These views enable you to obtain information about the parameters and dependencies of the functions, procedures, and packages. You also learned about how to overload functions and procedures so that the same object can be used in different ways depending on how many and which type of values are passed to the calling function or procedure.

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.

23. Object Types in Oracle

In this chapter, you will learn about

- [Object Types](#)
- [Object Type Methods](#)

In Oracle, object types are the main ingredient of object-oriented programming. They are used to model real-world tangible entities, such as students, instructors, and bank accounts, as well as abstract entities, such as ZIP codes, geometrical shapes, and chemical reactions.

In this chapter, you will learn how to create object types, and how to nest object types within collection types. In addition, you will learn about different kinds of object type methods and their usage.

This chapter is introductory in nature and does not cover more advanced topics such as object type inheritance and evolution, REF modifiers, and object type tables (not to be confused with collections). These topics, along with many others, are covered in Oracle's documentation—specifically, Oracle's *Database Object-Relational Developer's Guide*.

Lab 23.1: Object Types

After this lab, you will be able to

- [Create Object Types](#)
- [Use Object Types with Collections](#)

Object types generally consist of two parts: attributes (data) and methods (functions and procedures). Attributes are essential characteristics that describe the object type. For example, some attributes of the student object type may be first and last names, contact information, and enrollment information. Methods are functions and procedures defined in an object type; they are optional. They represent actions that are likely to be performed on the object attributes. For example, methods of the student object type might update the student contact information, get the student's name, or display the student's information.

By combining attributes and methods, object types facilitate encapsulation of data with the operations that may be performed on that data. As an example, [Figure 23.1](#) shows the object type `Student`. Some of the attributes of the `Student` object type are `Student ID`, `First Name`, `Zip`, and `Employer`, and some of the methods are `Update Contact Info`, `Get Student ID`, and `Get Student Name`. [Figure 23.1](#) also shows two instances of the object type, `Student 1` and `Student 2`. An object instance is a value of an object type. In other words, the instances `Student 1` and `Student 2` of the `Student` object type contain actual student data so that the `Get Student ID` method returns student ID 102 for instance `Student 1` and 103 for

instance Student 2.

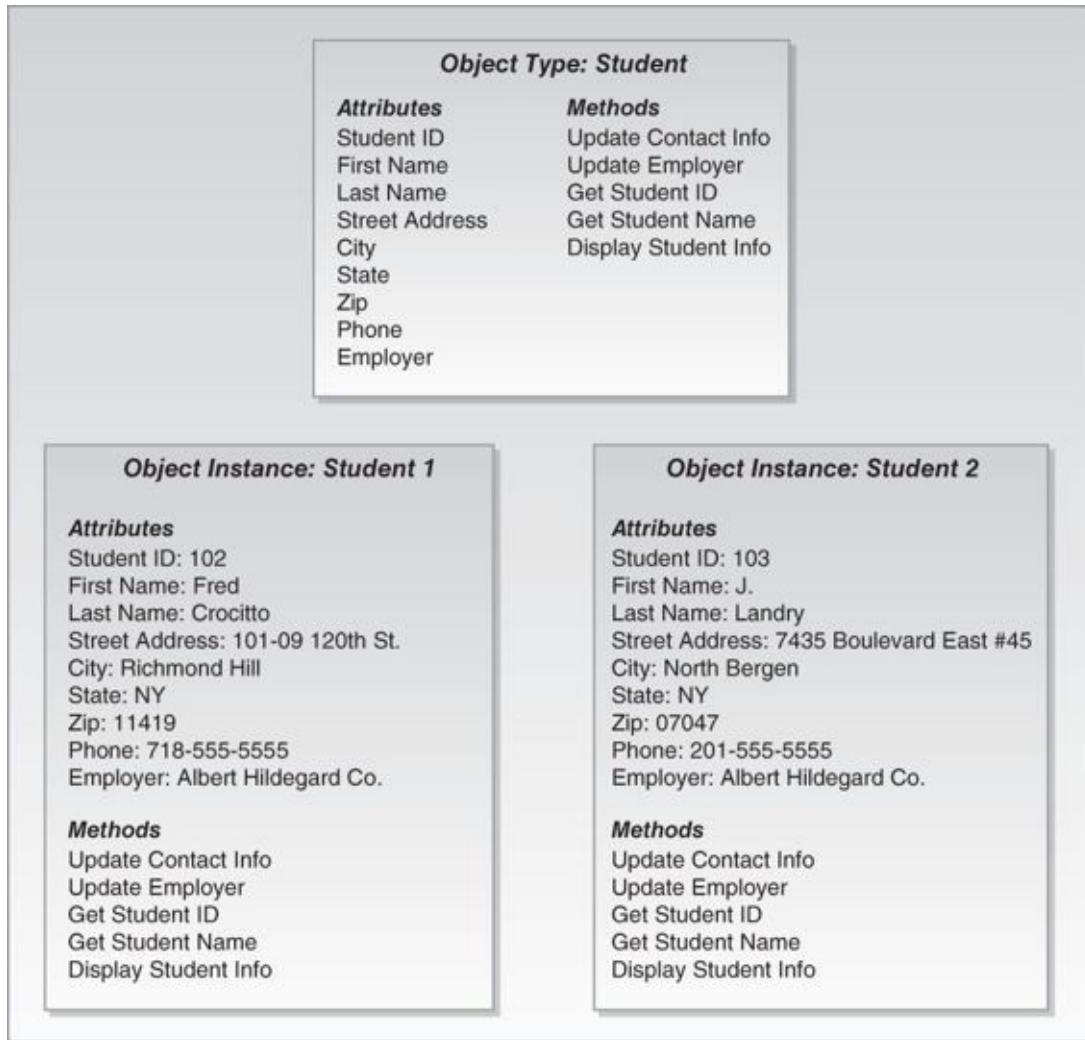


Figure 23.1 Object Type Student

Did You Know?

An object instance is often referred to simply as an object.

In Oracle, an object type is created with the CREATE OR REPLACE TYPE clause and is stored in the database schema. As a consequence, object types cannot be created within a PL/SQL block or stored subprogram. Once an object type has been created and stored in the database schema, a PL/SQL block or subprogram may reference that object type.

Creating Object Types

The general syntax for creating an object type is shown in [Listing 23.1](#) (the reserved words and phrases surrounded by brackets are optional):

Listing 23.1 Create Object Type

[Click here to view code image](#)

```
CREATE [OR REPLACE] TYPE type_name AS OBJECT  
  (attribute_name1 attribute_type,  
   attribute_name2 attribute_type,  
   ...)
```

```

attribute_nameN attribute_type,
[method1 specification],
[method2 specification],
...
[methodN specification]);
[CREATE [OR REPLACE] TYPE BODY type_name AS
method1 body;
method2 body;
...
methodN body;]
END;

```

Notice that the creation of an object type includes two parts: the object type specification and the object type body. The object type specification contains declarations of attributes as well as any methods that may be used with that object. The **attribute_type** may be a built-in PL/SQL type such as NUMBER or VARCHAR2, or it may be a complex user-defined type such as a collection, record, or other object type. The method specification consists of the method type, its name, and any input and output parameters the method needs.

Object specification is required when creating an object type. Any attributes and methods defined in the object type specification are visible to the outside world (such as a PL/SQL block, subprogram, or Java application). The object type specification is also called a public interface, and the methods defined in it are called public methods. As mentioned earlier, methods are optional when creating object types. However, if an object type has a method specification, it requires an object type body.

The object type body is optional when creating an object type. This part of the script contains the bodies (executable statements) of the methods defined in the object type specification. In addition, the object type body may contain methods that have not been defined in the object type specification. These methods are private—that is, they are not visible to the outside world. Some types of methods that might be specified as private are constructor, member, and static methods. Different method types, their usage, and restrictions are discussed in detail in [Lab 23.2](#).

Note that the concepts explained to this point are very similar to those you learned about in [Chapter 21](#), dealing with packages. Thus, rules that apply to the package specification and body mostly apply to the object type specification and body as well. For example, the header of the method defined in the object type specification must match the method header in the object type body.

Consider the following example of the zipcode_obj_type object type specification.

For Example ch23_1a.sql

[Click here to view code image](#)

```

CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
(zip                  VARCHAR2(5)
,city                VARCHAR2(25)
,state               VARCHAR2(2)
,created_by         VARCHAR2(30)
,created_date       DATE
,modified_by        VARCHAR2(30)

```

```
,modified_date      DATE);
```

This object type does not have any methods associated with it, and its syntax is somewhat similar to the `CREATE TABLE` syntax.

Once this object type has been created, it can be used as demonstrated in the following example:

For Example *ch23_2a.sql*

[Click here to view code image](#)

```
DECLARE
    zip_obj zipcode_obj_type;
BEGIN
    SELECT zipcode_obj_type(zip, city, state, null, null, null, null)
        INTO zip_obj
        FROM zipcode
       WHERE zip = '06883';

    DBMS_OUTPUT.PUT_LINE ('Zip:    '||zip_obj.zip);
    DBMS_OUTPUT.PUT_LINE ('City:   '||zip_obj.city);
    DBMS_OUTPUT.PUT_LINE ('State:  '||zip_obj.state);
END;
```

This script defines an instance `zip_obj` of the object type `zipcode_obj_type`. It then initializes some of the object attributes and displays those values on the screen.

The object attributes are initialized via the `SELECT INTO` statement. Note how the `SELECT` clause uses an object type constructor. Recall that you learned about constructors for nested table types in [Chapter 15](#). Default constructors for object types are similar in that they are system-defined functions that have the same name as the corresponding object type. In [Lab 23.2](#), you will learn how to define your own constructor functions.

When run, the preceding script produces this output:

```
Zip: 06883
City: Weston
State: CT
```

When an object instance is defined, its value is null. This means that not only its individual attributes are null, but the object itself is also null. The object remains null until its constructor method is called, as illustrated in the next example.

For Example *ch23_3a.sql*

[Click here to view code image](#)

```
DECLARE
    zip_obj zipcode_obj_type;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Object instance has not been initialized');

    IF zip_obj IS NULL
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj instance is null');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('zip_obj instance is not null');
    END IF;
```

```

IF zip_obj.zip IS NULL
THEN
  DBMS_OUTPUT.PUT_LINE ('zip_obj.zip is null');
END IF;

-- Initialize zip_obj_instance
zip_obj := zipcode_obj_type(null, null, null, null, null, null);

DBMS_OUTPUT.PUT_LINE ('Object instance has been initialized');

IF zip_obj IS NULL
THEN
  DBMS_OUTPUT.PUT_LINE ('zip_obj instance is null');
ELSE
  DBMS_OUTPUT.PUT_LINE ('zip_obj instance is not null');
END IF;

IF zip_obj.zip IS NULL
THEN
  DBMS_OUTPUT.PUT_LINE ('zip_obj.zip is null');
END IF;
END;

```

When run, this script produces the following output:

[Click here to view code image](#)

```

Object instance has not been initialized
zip_obj instance is null
zip_obj.zip is null
Object instance has been initialized
zip_obj instance is not null
zip_obj.zip is null

```

As you can see, both the object instance and its attributes are null prior to the initialization. Once the object instance has been initialized with the help of its default constructor, it is no longer null, even though its individual attributes remain null.

Watch Out!

Referencing individual attributes of an uninitialized object instance causes an ORA-06530 exception, “Reference to uninitialized composite error”:

[Click here to view code image](#)

```

DECLARE
  zip_obj zipcode_obj_type;
BEGIN
  zip_obj.zip := '12345';
END;

ORA-06530: Reference to uninitialized composite
ORA-06512: at line 4

```

It is a good practice to always initialize any newly created object type instance.

Using Object Types with Collections

As mentioned previously, object types and collection types may be nested inside one another. Consider the following example, which includes a collection of ZIP code objects.

For Example *ch23_4a.sql*

[Click here to view code image](#)

```
DECLARE
  TYPE zip_type IS TABLE OF zipcode_obj_type INDEX BY PLS_INTEGER;
  zip_tab zip_type;
BEGIN
  SELECT zipcode_obj_type(zip, city, state, null, null, null, null)
    BULK COLLECT INTO zip_tab
    FROM zipcode
   WHERE rownum <= 5;

  IF zip_tab.COUNT > 0
  THEN
    FOR i in 1..zip_tab.count
    LOOP
      DBMS_OUTPUT.PUT_LINE ('Zip:  '||zip_tab(i).zip);
      DBMS_OUTPUT.PUT_LINE ('City: '||zip_tab(i).city);
      DBMS_OUTPUT.PUT_LINE ('State: '||zip_tab(i).state);
      DBMS_OUTPUT.PUT_LINE ('-----');
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Collection of objects is empty');
  END IF;
END;
```

This example declares an associative array type `zip_type` of object type `zipcode_obj_type`. Next, it declares a collection variable `zip_tab` based on the newly created associative array type. It then populates this collection of objects via a `BULK SELECT` statement. Finally, it checks whether the collection has been populated via the `IF` statement and displays its data on the screen.

Note how individual object type attributes are referenced by the `DBMS_OUTPUT.PUT_LINE` statement. Each attribute is prefixed by the collection name and row subscript without any reference to the object type itself.

When run, this example produces the following output:

```
Zip: 00914
City: Santurce
State: PR
-----
Zip: 01247
City: North Adams
State: MA
-----
Zip: 02124
City: Dorchester
State: MA
-----
Zip: 02155
City: Tufts Univ. Bedford
```

```
State: MA
-----
Zip: 02189
City: Weymouth
State: MA
-----
```

In this example, you saw how to populate an associative array of objects with data. PL/SQL also supports selecting the data from collection of objects. In such a case, *the collection type should be a nested table or varray type that is created and stored in the database schema* just like its corresponding object type. This usage is illustrated by the following example.

For Example *ch23_5a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE zip_tab_type AS TABLE OF zipcode_obj_type;
/
DECLARE
    zip_tab zip_tab_type := zip_tab_type();
    v_zip   VARCHAR2(5);
    v_city  VARCHAR2(20);
    v_state VARCHAR2(2);
BEGIN
    SELECT zipcode_obj_type(zip, city, state, null, null, null, null)
        BULK COLLECT INTO zip_tab
        FROM zipcode
        WHERE rownum <= 5;

    SELECT zip, city, state
        INTO v_zip, v_city, v_state
        FROM TABLE(zip_tab)
        WHERE rownum < 2;

    DBMS_OUTPUT.PUT_LINE ('Zip: '||v_zip);
    DBMS_OUTPUT.PUT_LINE ('City: '||v_city);
    DBMS_OUTPUT.PUT_LINE ('State: '||v_state);
END;
```

First, this script creates a nested table type, `zip_tab_type`, in the `STUDENT` schema. This table type is then used by the PL/SQL block. Creating and storing a nested table type in the `STUDENT` schema enables you to use the `TABLE` function in the `SELECT INTO` statement to select data from the collection of objects into the `v_zip`, `v_city`, and `v_state` variables. Recall that the `TABLE` function enables you to query a collection as if it were a physical database table.

When run, this example produces the following output:

```
Zip: 00914
City: Santurce
State: PR
```

So far, you have seen various examples of collections of objects. PL/SQL also supports nesting a collection type within an object type. Similarly to the previous examples, both collection and object data types should be created and stored in the database schema. Consider an example of a state object type that has two collection attributes, `cities`, and

ZIP codes.

For Example *ch23_6a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE city_tab_type AS TABLE OF VARCHAR2(25);
/
CREATE OR REPLACE TYPE zip_tab_type AS TABLE OF VARCHAR2(5);
/
CREATE OR REPLACE TYPE state_obj_type AS OBJECT
  (state VARCHAR2(2)
   ,city  city_tab_type
   ,zip   zip_tab_type);
/
```

This script creates two nested table types, `city_tab_type` and `zip_tab_type`, in the `STUDENT` schema. Next, it creates an object type, `state_obj_type`, that has three attributes. Note that the `city` and `zip` attributes are based on the nested table types created earlier.

Next consider an example that employs the newly created collection and object types.

For Example *ch23_7a.sql*

[Click here to view code image](#)

```
DECLARE
  city_tab  city_tab_type;
  zip_tab   zip_tab_type;

  state_obj state_obj_type := state_obj_type(null, city_tab_type(),
                                             zip_tab_type());

BEGIN
  SELECT city, zip
    BULK COLLECT INTO city_tab, zip_tab
      FROM zipcode
     WHERE state = 'NY'
       AND rownum <= 5;

  state_obj := state_obj_type ('NY', city_tab, zip_tab);

  DBMS_OUTPUT.PUT_LINE ('State: '||state_obj.state);
  DBMS_OUTPUT.PUT_LINE ('-----');

  IF state_obj.city.COUNT > 0
  THEN
    FOR i in state_obj.city.FIRST..state_obj.city.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE ('City:  '||state_obj.city(i));
      DBMS_OUTPUT.PUT_LINE ('Zip:   '||state_obj.zip(i));
    END LOOP;
  END IF;
END;
```

In the declaration portion of the script, there are definitions of the two nested table variables, `city_tab` and `zip_tab`, based on the `city_tab_type` and the `zip_tab_type`, respectively. Also, there is a declaration and initialization of the object instance `state_obj` based on the `state_obj_type`. Because the `city` and `zip`

attributes of `state_obj` are nested tables, they are initialized via their default constructor methods, as highlighted here:

[Click here to view code image](#)

```
state_obj state_obj_type := state_obj_type(null, city_tab_type(),  
zip_tab_type());
```

In the executable portion of the example, the two nested tables, `city_tab` and `zip_tab`, are populated via a `SELECT` statement with a `BULK COLLECT INTO` clause. Recall that when nested tables are populated in such manner, there is no need to initialize them by invoking their default constructor methods. Next, the `state_obj` instance is populated by invoking its default constructor method:

[Click here to view code image](#)

```
state_obj := state_obj_type ('NY', city_tab, zip_tab);
```

In this case, there is no need to employ default constructor methods for the two nested table attributes, `city` and `zip`. Instead, the `city_tab` and `zip_tab` nested tables are simply passed into the constructor method `state_obj_type`.

When run, this script produces the following output:

```
State: NY  
-----  
City: Irvington  
Zip: 07111  
City: Franklin Lakes  
Zip: 07417  
City: Alpine  
Zip: 07620  
City: Oradell  
Zip: 07649  
City: New York  
Zip: 10004
```

Lab 23.2: Object Type Methods

After this lab, you will be able to

- [Use Constructor Methods](#)
- [Use Member Methods](#)
- [Use Static Methods](#)
- [Compare Objects Via Map and Order Methods](#)

In [Lab 23.1](#), you learned that object type methods are functions and procedures that specify actions that may be performed on the object type attributes and are defined in the object type specification. Also, you saw how to use the default system-defined constructor methods. A constructor is only one of the method types supported by PL/SQL. Some other method types are member, static, map, and order. The method type is typically determined by the actions that a particular method performs. For example, constructor methods are used to initialize object instances, whereas map and order methods are used for comparing

and sorting object instances, respectively.

Oftentimes object type methods use a built-in parameter called SELF. This parameter represents a particular instance of the object type. As such, it is available to the methods that are invoked on that object type instance. You will see various examples of the SELF parameter in the discussions that follow.

Constructor Methods

A constructor method is a default method that is implicitly created by the system whenever a new object type is created. This function has the same name as its object type. Its input parameters have the same names and data types as the object type attributes and are listed in the same order as the object type attributes. A constructor method returns a new instance of the object type. In other words, it initializes the new object instance and assigns values to the object attributes.

[Listing 23.2](#) illustrates calls to the default constructor method for the `zipcode_obj_type` created in [Lab 23.1](#).

Listing 23.2 Default Constructor Method for Zipcode_Obj_Type

[Click here to view code image](#)

```
zip_obj1 := ZIPCODE_OBJ_TYPE('00914', 'Santurce', 'PR', USER, SYSDATE,  
USER, SYSDATE);
```

or

[Click here to view code image](#)

```
zip_obj2 := ZIPCODE_OBJ_TYPE(NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL);
```

The first call to the constructor method returns a new instance, `zip_obj1`, of the `zipcode_obj_type` with attributes initialized to non-null values. The second call creates a new instance, `zip_obj2`, with NULL attribute values. Note that both calls produce non-null instances of the `zipcode_obj_type`. *The difference between them lies in the values assigned to the individual attributes.*

In [Listing 23.2](#), both calls to the default constructor method use positional notation. Recall that positional notation associates values with corresponding parameters based on their positions in the header of the function, procedure, or (in this case) constructor. Next, consider the call to the default constructor method that uses named notation. In this case, the order of parameters does not correspond to the order of attributes in the `zipcode_obj_type`, as they are referenced by their names as shown in [Listing 23.3](#).

Listing 23.3 Using Positional Notation with the Default Constructor Method for Zipcode_Obj_Type

[Click here to view code image](#)

```
,state          => 'PR');
```

As noted earlier, PL/SQL also provides you with the ability to create your own (user-defined) constructors. User-defined constructors offer flexibility that the default constructors lack. For example, you might want to define a constructor on the `zipcode_obj_type` that initializes only some of the attributes of the newly created object instance. In this case, any attributes for which you do not specify values will be initialized to `NULL` by the system. In addition, you can control the number and type of parameters that your constructor may require.

Consider the following example of the user-defined constructors for the `zipcode_obj_type`.

For Example *ch23_8a.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
  (zip          VARCHAR2(5)
   ,city         VARCHAR2(25)
   ,state        VARCHAR2(2)
   ,created_by   VARCHAR2(30)
   ,created_date DATE
   ,modified_by  VARCHAR2(30)
   ,modified_date DATE

   ,CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY
zipcode_obj_type
                           ,zip      VARCHAR2)
   RETURN SELF AS RESULT

   ,CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY
zipcode_obj_type
                           ,zip      VARCHAR2
                           ,city    VARCHAR2
                           ,state   VARCHAR2)
   RETURN SELF AS RESULT);
/

CREATE OR REPLACE TYPE BODY zipcode_obj_type AS

  CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY
zipcode_obj_type
                           ,zip      VARCHAR2)
   RETURN SELF AS RESULT
  IS
  BEGIN
    SELF.zip := zip;
    SELECT city, state
      INTO SELF.city, SELF.state
      FROM zipcode
     WHERE zip = SELF.zip;

    RETURN;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN;
  END;
```

```

CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY
zipcode_obj_type
          , zip      VARCHAR2
          , city    VARCHAR2
          , state   VARCHAR2)

RETURN SELF AS RESULT
IS
BEGIN
  SELF.zip    := zip;
  SELF.city   := city;
  SELF.state  := state;

  RETURN;
END;
END;
/

```

This script expands the definition of the `zipcode_obj_type` by providing two versions of the default constructor method. In programming terms, such approach is called overloading. Essentially, overloading allows two methods or subprograms to use the same name as long as their parameters differ in terms of either their data types or their number. In the preceding example, the first constructor method expects two parameters, and the second constructor method expects four parameters.

Both constructors use the default parameter `SELF` as an `IN OUT` parameter and as a return data type in the `RETURN` clause. As stated previously, `SELF` references a particular object type instance. Note the use of the `NOCOPY` compiler hint. This hint is typically used with `OUT` and `IN OUT` parameters. By default, `OUT` and `IN OUT` parameters are passed by value. As a consequence, the values of these parameters are copied prior to the execution of the subprogram or method. Then, during the execution, temporary variables are used to hold values of the `OUT` parameters. For parameters that represent complex data types such as collections, records, and object type instances, this copying step can add significant processing overhead. By adding the `NOCOPY` hint, you instruct PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference and eliminate the copying step altogether.

In the type body, both constructor methods populate the `city`, `state`, and `zip` attributes. The first constructor method accomplishes this work via a `SELECT INTO` statement, and the second constructor method assigns incoming values to the object attributes. Notice how the attributes are referenced via the `SELF` parameter in the constructor methods.

Member Methods

Member methods provide access to the object instance data. As such, a member method should be defined for each action that an object type must perform. For example, you may need to return city, state, and ZIP code values associated with an object instance to the calling application, as shown in the next example. Note that this example shows only the newly added member method.

For Example `ch23_8b.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
...
, MEMBER PROCEDURE get_zipcode_info (out_zip    OUT VARCHAR2
                                     ,out_city   OUT VARCHAR2
                                     ,out_state  OUT VARCHAR2));
/
CREATE OR REPLACE TYPE BODY zipcode_obj_type AS
...
MEMBER PROCEDURE get_zipcode_info (out_zip    OUT VARCHAR2
                                     ,out_city   OUT VARCHAR2
                                     ,out_state  OUT VARCHAR2)
IS
BEGIN
    out_zip  := SELF.zip;
    out_city := SELF.city;
    out_state := SELF.state;
END;
END;
/
```

This version of the object type definition contains a new member procedure that returns the ZIP code, city, and state values associated with a particular instance of the `zipcode_obj_type` object type. The reference to the `SELF` parameter in this procedure is optional, however, so the assignment statements can be modified as follows:

```
out_zip  := zip;
out_city := city;
out_state := state;
```

These statements initialize `OUT` parameters associated with individual attributes of a particular object instance, just like the statements that include the reference to the `SELF` parameter.

Static Methods

Static methods are created for actions that do not need to access data associated with a particular object instance. As such, these methods are created for the object type itself and describe actions that are global to that object type. Because static methods do not have access to the data associated with a particular object type instance, they may not reference the default parameter `SELF`.

Consider the following example of a static method that displays ZIP code information. Note that this example shows only the newly added static method.

For Example `ch23_8c.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
...
, STATIC PROCEDURE display_zipcode_info (in_zip_obj IN
```

```

zipcode_obj_type));
/

CREATE OR REPLACE TYPE BODY zipcode_obj_type AS

...

STATIC PROCEDURE display_zipcode_info (in_zip_obj IN zipcode_obj_type)
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Zip: ' ||in_zip_obj.zip);
    DBMS_OUTPUT.PUT_LINE ('City: ' ||in_zip_obj.city);
    DBMS_OUTPUT.PUT_LINE ('State: '||in_zip_obj.state);
END;
END;
/

```

In this version of the script, the static method `display_zipcode_info` displays the values of the individual attributes for the ZIP code object on the screen. Even though this method references data associated with some object instance, the object instance is created elsewhere (i.e., another PL/SQL script, function, or procedure) and then passed into this method as an input parameter.

Comparing Objects

In PL/SQL, element data types such `VARCHAR2`, `NUMBER`, or `DATE` have a predefined order that enables them to be compared to each other or sorted. For example, the comparison operator (`>`) determines which variable contains a greater value and the `IF-THEN-ELSE` statement evaluates to `TRUE`, `FALSE`, or `NULL` accordingly:

```

IF v_num1 > v_num2 THEN
    — Do something
ELSE
    — Do something else
END IF;

```

In contrast, an object type may contain multiple attributes of different data types and, as a result, does not have a predefined order. Thus, to be able to compare and sort object instances of the same object type, you must specify how these object instances should be compared and ordered. This can be accomplished via two types of optional member methods, `map` and `order`.

Map Methods

Map methods compare and order object instances essentially by mapping an object instance to an element (scalar) data type such as `DATE`, `NUMBER`, or `VARCHAR2`. This mapping is then used to position object instance on the axis (`DATE`, `NUMBER`, or `VARCHAR2`) used for the comparison.

A map method is a member function that does not accept any parameters and returns an element data type, as demonstrated in the next example. Note that this example shows only the newly added map method.

For Example `ch23_8d.sql`

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
...
,
    ,MAP MEMBER FUNCTION zipcode RETURN VARCHAR2);
/
CREATE OR REPLACE TYPE BODY zipcode_obj_type AS
...
MAP MEMBER FUNCTION zipcode RETURN VARCHAR2
IS
BEGIN
    RETURN (zip);
END;
END;
/
```

In this version of the script, the map member function returns the value of the `zip` attribute that has been defined as `VARCHAR2`.

Once a map method is added to the object type, object type instances may be compared or ordered similarly to the element data types. For example, if `zip_obj1` and `zip_obj2` are two instances of the `zipcode_obj_type`, they can be compared as follows:

```
zip_obj1 > zip_obj2
```

or

[Click here to view code image](#)

```
zip_obj1.zipcode() > zip_obj2.zipcode()
```

The second statement uses dot notation to reference the map function.

The next example demonstrates how the various object type methods created so far may be used.

For Example *ch23_9a.sql*

[Click here to view code image](#)

```
DECLARE
    zip_obj1 zipcode_obj_type;
    zip_obj2 zipcode_obj_type;
BEGIN
    -- Initialize object instances with user-defined constructor methods
    zip_obj1 := zipcode_obj_type (zip => '12345'
                                ,city => 'Some City'
                                ,state => 'AB');

    zip_obj2 := zipcode_obj_type (zip => '48104');

    -- Compare object instances via map methods
    IF zip_obj1 > zip_obj2
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is greater than zip_obj2');
    ELSE
```

```
DBMS_OUTPUT.PUT_LINE
    ('zip_obj1 is not greater than zip_obj2');
END IF;
END;
```

When the user-defined constructors are invoked, there is no reference to the SELF default parameter in the call statements.

When run, this script produces the following output:

[Click here to view code image](#)

```
v_zip_obj1 is not greater than v_zip_obj2
```

Order Methods

Order methods use a different technique for comparing and ordering object instances. They do not map object instances to an external axis such as NUMBER or DATE. Instead, an order method compares the current object instance with another object instance of the same object type based on some criteria specified in the method.

An order method is a member function with a single IN parameter of the same object type that returns INTEGER as its return type. Furthermore, the method must return a negative number, zero, or a positive number, which indicates that the object instance referenced by the SELF parameter is less than, equal to, or greater than the object instance referenced by the IN parameter, respectively.

Watch Out!

The following restrictions apply to the map and order methods:

- An object type may contain either an order method or a map method. If it has both, the following error is raised at the time of its creation:

[Click here to view code image](#)

```
PLS-00154: An object type may have only 1 MAP or 1 ORDER method.
```

- An object type derived from another object type may not define an order method.

Consider the following example of an order method for the zipcode_obj_type. Similarly to the previous examples, this version shows only the newly added order method.

For Example ch23_8e.sql

[Click here to view code image](#)

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
...
, ORDER MEMBER FUNCTION zipcode (zip_obj zipcode_obj_type) RETURN
INTEGER);
/
CREATE OR REPLACE TYPE BODY zipcode_obj_type AS
```

```

...
ORDER MEMBER FUNCTION zipcode (zip_obj zipcode_obj_type) RETURN INTEGER
IS
BEGIN
    IF      zip < zip_obj.zip THEN RETURN -1;
    ELSIF zip = zip_obj.zip THEN RETURN  0;
    ELSIF zip > zip_obj.zip THEN RETURN  1;
    END IF;
END;
END;
/

```

In this version of the script, the map member function is replaced by the order member function. Much like the map method, the order method uses the `zip` attribute as the basis of comparison for the two object type instances.

The following example demonstrates how an order method may be used.

For Example *ch23_10a.sql*

[Click here to view code image](#)

```

DECLARE
    zip_obj1 zipcode_obj_type;
    zip_obj2 zipcode_obj_type;

    v_result  INTEGER;
BEGIN
    -- Initialize object instances with user-defined constructor methods
    zip_obj1 := zipcode_obj_type ('12345', 'Some City', 'AB');
    zip_obj2 := zipcode_obj_type ('48104');

    -- Compare objects instances via ORDER method
    v_result := zip_obj1.zipcode(zip_obj2);
    DBMS_OUTPUT.PUT_LINE ('The result of comparison is'||v_result);

    IF v_result = 1
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is greater than zip_obj2');

    ELSIF v_result = 0
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is equal to zip_obj2');

    ELSIF v_result = -1
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is less than zip_obj2');
    END IF;
END;

```

In this script, the result of the order method is assigned to the `v_result` variable, which is defined as an `INTEGER`. Note how the order method is invoked:

[Click here to view code image](#)

```
v_result := zip_obj1.zipcode(zip_obj2);
```

The order method associated with the instance `zip_obj1` accepts the instance `zip_obj2` as its input parameter.

When run, this script produces the following output:

[Click here to view code image](#)

```
The result of comparison is -1  
zip_obj1 is less than zip_obj2
```

It is important to recognize which object instance is being used to invoke the order method, as different instances may yield different results. For example, consider what happens if we transpose the object instances when calling the order method (the affected statements are shown in bold):

For Example *ch23_10b.sql*

[Click here to view code image](#)

```
DECLARE  
    zip_obj1 zipcode_obj_type;  
    zip_obj2 zipcode_obj_type;  
  
    v_result  INTEGER;  
BEGIN  
    -- Initialize object instances with user-defined constructor methods  
    zip_obj1 := zipcode_obj_type ('12345', 'Some City', 'AB');  
    zip_obj2 := zipcode_obj_type ('48104');  
  
    -- Compare objects instances via ORDER method  
    v_result := zip_obj2.zipcode(zip_obj1);  
    DBMS_OUTPUT.PUT_LINE ('The result of comparison is '||v_result);  
    IF v_result = 1  
    THEN  
        DBMS_OUTPUT.PUT_LINE ('zip_obj2 is greater than zip_obj1');  
  
    ELSIF v_result = 0  
    THEN  
        DBMS_OUTPUT.PUT_LINE ('zip_obj2 is equal to zip_obj1');  
  
    ELSIF v_result = -1  
    THEN  
        DBMS_OUTPUT.PUT_LINE ('zip_obj2 is less than zip_obj1');  
    END IF;  
END;
```

The text displayed by the DBMS_OUTPUT.PUT_LINE statements has been altered in this script to produce the correct output. While this is a very simple example, it is able to demonstrate that changing how the order method is invoked (via a change in the object instance) affects how its return value is evaluated.

This version of the script produces different output as shown:

[Click here to view code image](#)

```
The result of comparison is 1  
zip_obj2 is greater than zip_obj1
```

Summary

In this chapter, you learned how to define and use objects in Oracle. Overall, object types in Oracle are similar to classes created in Java. They consist of attributes and methods, where attributes represent different data elements of an object and methods are used to perform various actions on these data elements. You also learned how to implement and use different types of methods to initialize, compare and sort objects. In addition, you discovered how to use objects with collections.

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.

24. Oracle-Supplied Packages

In this chapter, you will learn about

- [Extending Functionality with Oracle-Supplied Packages](#)
- [Error Reporting with Oracle-Supplied Packages](#)

Oracle has built into its database hundreds of packages that extend what you can achieve with PL/SQL. Each new version of the database comes with new supplied packages. With version 12c, Oracle introduced 18 brand-new packages and added new procedures in many existing packages. These packages offer functionality that you would not be able to achieve with PL/SQL alone. The reason is that the Oracle-supplied packages make use of the C programming language, which is not something that you can do with ordinary PL/SQL packages. As a consequence, Oracle-supplied packages have full access to the operating system and other aspects of the Oracle server that are not available to ordinary PL/SQL packages. You are already familiar with the DBMS_OUTPUT package's procedure PUT_LINE, which is used to gather debugging information in the buffer for output. This chapter serves as an introduction to a few key Oracle-supplied packages; you will learn about their basic features and discover how to make use of them.

Lab 24.1: Extending Functionality with Oracle-Supplied Packages

After this lab, you will be able to

- [Access Files within PL/SQL with UTL_FILE](#)
- [Schedule Jobs with DBMS_JOB](#)
- [Generate an Explain Plan with DBMS_XPLAN](#)
- [Generate Implicit Statement Results with DBMS_SQL](#)

Accessing Files within PL/SQL with UTL_FILE

The UTL_FILE package provides text file input and output capabilities within PL/SQL. Oracle introduced this package with database version 7.3. It enables you to read input from the operating system files and write to operating system files. This capability could prove useful if you want to load data from another system into the database. For instance, if you want to store logs from a web server in your data warehouse, the UTL_FILE package would enable you to read the text file logs and then parse them so as to load the data in the correct tables and columns in the data warehouse. This package also allows you to write data out to a file. This capability is useful if you want to produce logs or capture current information about the database and store it in a text file, or extract data into a text file that another application can process.

The **UTL_FILE** package provides server-side text file access, so it cannot read binary files. For that purpose, you would use the **DBMS_LOB** package. The files that you access must be mapped to a drive on the server. The security settings that determine which directories you can access are controlled in the **INIT.ORA** file; you set the drives that can be accessed with the **UTL_FILE_DIR** initialization parameter.

```
UTL_FILE_DIR = 'C:\WORKING'
```

You can also bypass all server-side security and allow *all* files to be accessed with the **UTL_FILE** package with the following setting:

```
UTL_FILE_DIR = *
```

If you do not have access to the **INIT.ORA** file on the database server, you can query the data dictionary to find the value that has been set in your database with the following SQL code:

```
SELECT name, value
FROM V$SYSTEM_PARAMETER
WHERE name = 'utl_file_dir'
```

By the Way

It is not advisable to allow **UTL_FILE** access to all files in a production environment. This setting means that all files, including important files that manage the operation of the database, are accessible. Developers may potentially write a procedure that corrupts the database in such a case.

To use the **UTL_FILE** file package, you open the text file, process the file by writing to it and getting lines from the file, and close the file. If you do not close the file, the operating system will think that the file is in use and will not allow you to write to the file until it is closed. [Table 24.1](#) lists the major functions, procedures, and data types in the **UTL_FILE** package. [Table 24.2](#) identifies the exceptions found in this package.

Function, Procedure, or Data Type	Column Heading
FILE_TYPE	Data type for a file handle.
IS_OPEN	This function has a return data type of BOOLEAN; it returns TRUE if the file is open and FALSE if the file is closed.
FOPEN	This function is used to open a file for input or output. The function return value is the form handle in the FILE_TYPE data type. The modes to open a file are: "R"—read mode "W"—write mode "A"—append mode
FCLOSE	This procedure closes a file that is open.
FCLOSE_ALL	This procedure closes all files that are open in the current session. (It is a good idea to place this procedure in your exception to make sure you don't leave any files locked.)
FFLUSH	This procedure takes all the data buffered in memory and writes it to a file.
GET_LINE	This procedure gets one line of text from the opened file and places the text into the OUT parameter of the procedure.
PUT_LINE	This procedure writes a string of text from the IN parameter to the opened file. Afterward, a line terminator is placed into the text file.
PUT	This procedure is the same as PUT_LINE except that no line terminator is placed in the open file.
PUTF	This procedure puts formatted text into the opened file.
NEW_LINE	This procedure inserts a new line terminator in the opened text file.

Table 24.1 UTL_FILE Functions, Procedures, and Data Types

Exception Name	Description
INVALID_PATH	The file location or the filename is not valid.
INVALID_MODE	This exception is for FOPEN only; the mode for the OPEN_MODE parameter is not valid.
INVALID_FILEHANDLE	The file handle is not valid.
INVALID_OPERATION	The file could not be opened or operated on in the manner requested.
READ_ERROR	An operating system error prevented the read file operation from occurring.
WRITE_ERROR	An operating system error prevented the write file operation from occurring.
INTERNAL_ERROR	An unspecified PL/SQL error occurred.

Table 24.2 UTL_FILE Exceptions

The following example demonstrates a procedure that writes to a log file the date, time, and number of users who are currently logged on. To make use of this example, the user STUDENT needs to have privileges to access the v\$session table. Access can be granted by the database administrator (DBA) to STUDENT as follows:

[Click here to view code image](#)

```
GRANT SELECT ON sys.v_$session TO student;
ch24_1.sql"
```

For Example ch24_1.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE BODY school_api AS
CREATE OR REPLACE PROCEDURE LOG_USER_COUNT
(PI_DIRECTORY IN VARCHAR2,
 PI_FILE_NAME IN VARCHAR2)
AS
  V_File_handle UTL_FILE.FILE_TYPE;
  V_user_count number;
BEGIN
  SELECT count(*)
  INTO   V_user_count
  FROM   v$session
  WHERE  username is not null;

  V_File_handle :=
    UTL_FILE.FOPEN(PI_DIRECTORY, PI_FILE_NAME, 'A');
  UTL_FILE.NEW_LINE(V_File_handle);
  UTL_FILE.PUT_LINE(V_File_handle , '-- User log --');
  UTL_FILE.NEW_LINE(V_File_handle);
  UTL_FILE.PUT_LINE(V_File_handle , 'on ' ||
    TO_CHAR(SYSDATE, 'MM/DD/YY HH24:MI'));
  UTL_FILE.PUT_LINE(V_File_handle ,
    'Number of users logged on: '|| V_user_count);
  UTL_FILE.PUT_LINE(V_File_handle , '-- End log --');
  UTL_FILE.NEW_LINE(V_File_handle);
  UTL_FILE.FCLOSE(V_File_handle);

EXCEPTION
  WHEN UTL_FILE.INVALID_FILENAME THEN
    DBMS_OUTPUT.PUT_LINE('File is invalid');
  WHEN UTL_FILE.WRITE_ERROR THEN
    -DBMS_OUTPUT.PUT_LINE('Oracle is not able to write to file');
END;
```

The LOG_USER_COUNT procedure can be executed to log the number of users into the file c :\working\user.log.

[Click here to view code image](#)

```
SQL> exec LOG_USER_COUNT('C:\working', 'USER.LOG');
```

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

Here are the resulting USER.LOG contents:

```
-- User log --
on 07/05/03 13:09
Number of users logged on: 1
-- End log --
```

Access Files with UTL_FILE

The following PL/SQL script creates a procedure to read a file and display the contents. The exception WHEN NO_DATA_FOUND will be raised when the last line of the file has been read and there are no more lines to read.

For Example *ch24_2.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE READ_FILE
  (PI_DIRECTORY  IN VARCHAR2,
   PI_FILE_NAME  IN VARCHAR2)
AS
  V_File_Handle  UTL_FILE.FILE_TYPE;
  V_File_Line    VARCHAR2(1024);
BEGIN
  V_File_Handle := UTL_FILE.FOPEN(PI_DIRECTORY, PI_FILE_NAME, 'R');
  LOOP
    UTL_FILE.GET_LINE( V_File_Handle , v_file_line);
    DBMS_OUTPUT.PUT_LINE(v_file_line);
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND
    THEN UTL_FILE.FCLOSE( V_File_Handle );
END;
```

Scheduling Jobs with DBMS_JOB

The Oracle-supplied package DBMS_JOB allows you to schedule the execution of a PL/SQL procedure. It was first introduced in PL/SQL version 2.2. DBMS_JOB is an Oracle PL/SQL package provided to users. A job is submitted to a job queue and runs at the specified time. The user can also input a parameter that specifies how often the job should run. A job can consist of any PL/SQL code. The DBMS_JOB package has procedures for submitting jobs for scheduled execution, executing a job that has been submitted outside of its schedule, changing the execution parameters of a previously submitted job, suspending a job, and removing jobs from the schedule ([Table 24.3](#)). The primary reasons you might want to use this feature would be to run a batch program during off hours when there are fewer users logged into the system or to maintain a log.

Procedure Name	Description
SUBMIT	This procedure enters a PL/SQL procedure as a job into the job queue.
REMOVE	This procedure removes a previously submitted PL/SQL procedure from the job queue.
CHANGE	This procedure changes the parameters that have been set for a previously submitted job (description, next run time, or interval).
BROKEN	This procedure disables a job in the job queue.
INTERVAL	This procedure alters the interval set for an existing job in the job queue.
NEXT_DATE	This procedure changes the next time an existing job is set to run.
RUN	This procedure forces a job in the job queue to be run regardless of the schedule for that job.

Table 24.3 The Main Procedures in the DBMS_JOB Package

The job queue is governed by the SNP (Snapshot Process) process that runs in the background. This process is used to implement data snapshots as well as job queues. If it fails, the database will attempt to restart the process. The database initialization parameter

JOB_QUEUE_PROCESSES (set in the INIT.ORA file and viewable in the DBA view V\$SYSTEM_PARAMETER) determines how many processes can start. It must be set to a number greater than 0 (the default is 0).

Watch Out!

SNP background processes will not execute jobs if the system has been started in restricted mode. It is expected behavior for jobs not to be executed while the database is in restricted mode. However, you can use the ALTER SYSTEM command to turn this behavior on and off as follows:

[Click here to view code image](#)

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

Submitting Jobs

An important first step when submitting jobs to the queue is to ensure that your PL/SQL procedure is valid and executes the way you expect it to run. Prior to submitting a PL/SQL procedure, thoroughly test the procedure's functionality. Job submission assumes your job is valid. The SUBMIT procedure will take four IN parameters and return one OUT parameter ([Table 24.4](#)). The OUT parameter is the job number of the job you have submitted. This job number is also visible in the DBA_JOBS view.

Parameter Name	Mode	Description
JOB	OUT	The unique number that identifies the job in the job queue.
WHAT	IN	The PL/SQL procedure and parameters that will execute as part of this job.
NEXT_DATE	IN	The next execution date for the job.
INTERVAL	IN	The calculation to compute the next date of the job. (This can make use of SYSDATE and any date function.)
NO_PARSE	IN	A Boolean indicator as to whether to run the DEFAULT: FALSE job at job submission

Table 24.4 Parameters for the DBMS_JOB.SUBMIT Procedure

The following example will submit the LOG_USER_COUNT procedure (created with ch24_3.sql) and set it to run every 6 hours.

For Example *ch24_3.sql*

[Click here to view code image](#)

```
DECLARE
  V_JOB_NO NUMBER;
BEGIN
  DBMS_JOB.SUBMIT( JOB      => v_job_no,
                    WHAT     -=> 'LOG_USER_COUNT
                                ("C:\WORKING'', "USER.LOG");',
                    NEXT_DATE => SYSDATE,
                    INTERVAL  => 'SYSDATE + 1/4 ');
  Commit;
```

```
DBMS_OUTPUT.PUT_LINE(v_job_no);
END;
```

To see this job in the queue, query the DBA_JOB view. For the STUDENT user to be able to perform this query, the DBA needs to perform the following grant:

[Click here to view code image](#)

```
GRANT SELECT on DBA_JOBS to STUDENT;
```

Running the SELECT statement

[Click here to view code image](#)

```
SELECT JOB, NEXT_DATE, NEXT_SEC, BROKEN, WHAT
FROM   DBA_JOBS;
```

then produces the following result:

[Click here to view code image](#)

```
JOB    NEXT_DATE    NEXT_SEC    B    WHAT
---  -----
1    05-JUL-03 16:56:30 N  LOG_USER_COUNT('D:\WORKING', 'USER.LOG');
```

To force job number 1 to run or to change, use the RUN or CHANGE procedure. To remove job number 1 from the job queue, use the REMOVE procedure.

[Click here to view code image](#)

- execute job number 1
exec dbms_job.run(1);
- remove job number 1 from the job queue
exec dbms_job.remove(1);
- change job number 1 to run immediately and then every hour of
– the day
exec DBMS_JOB.CHANGE(1, null, SYSDATE, 'SYSDATE + 1/24 '');

Once the job has failed, it will be marked as broken in the job queue. Broken jobs do not run. You can also force a job to be flagged as broken. You may want to do this if you have entered all the parameters correctly yet do not want the job to run on its normal cycle while you are in the middle of altering one of its dependencies. You can then run the job again by forcing the broken flag off.

[Click here to view code image](#)

- set job 1 to be broken
exec dbms_job.BROKEN(1, TRUE);
- set job 1 not to be broken
exec dbms_job.BROKEN(1, FALSE);

When jobs are running, you will see their activity in the view DBA_JOBS_RUNNING. Once the run has completed, it will no longer be visible in this view.

In the following example, the procedure DELETE_ENROLL will delete a student's enrollment if there are no grades in the GRADE table for that student and the start date of the section is already one month past the current system date.

For Example *ch24_4.sql*

[Click here to view code image](#)

```
CREATE or REPLACE procedure DELETE_ENROLL
AS
  CURSOR C_NO_GRADES is
    SELECT st.student_id, se.section_id
      FROM student st,
            enrollment e,
            section se
     WHERE st.student_id = e.student_id
       AND e.section_id = se.section_id
       AND se.start_date_time < ADD_MONTHS(SYSDATE, -1)
       AND NOT EXISTS (SELECT g.student_id, g.section_id
                          FROM grade g
                         WHERE g.student_id = st.student_id
                           AND g.section_id = se.section_id);
BEGIN
  FOR R in C_NO_GRADES LOOP
    DELETE enrollment
      WHERE section_id = r.section_id
        AND student_id = r.student_id;
  END LOOP;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

The following script shows how to submit the procedure **DELETE_ENROLL** to the job queue so that it will execute once a month:

[Click here to view code image](#)

```
SQL> VARIABLE V_JOB NUMBER
SQL> EXEC DBMS_JOB.SUBMIT(:v_job, 'DELETE_ENROLL;', SYSDATE,
  'ADD_MONTHS(SYSDATE, 1)');
PL/SQL procedure successfully completed
SQL> commit;

Commit complete.

SQL> print v_job

      V_JOB
-----
      2
```

Generating an Explain Plan with DBMS_XPLAN

The **DBMS_XPLAN** package became available in Oracle version 9.2. This package helps to display the execution plan of an SQL statement as the output of the explain plan command. It provides the output in an easier-to-understand format than was possible in prior versions of Oracle. The SQL execution plan and runtime statistics that are stored in **V\$SQL_PLAN**, **V\$SQL**, and **PLAN_STATISTICS** are displayed with the **DBMS_XPLAN** package. The SQL command for creating an explain plan takes this information and uses it to populate the **PLAN_TABLE**. You must know a great deal about query optimization to make the most effective use of an explain plan.

By the Way

For details on SQL optimization and use of the results in an explain plan, see *Oracle SQL by Example* by Alice Rischert (ISBN-10: 0137142838; ISBN-13: 978-0137142835).

The DBMS_XPLAN package depends on PLAN_TABLE—a table that holds the results from running an explain plan on a SELECT statement. The following DDL is used to create the PLAN_TABLE:

[Click here to view code image](#)

```
create table PLAN_TABLE (
    statement_id          varchar2(30),
    plan_id               number,
    timestamp             date,
    remarks               varchar2(4000),
    operation              varchar2(30),
    options                varchar2(255),
    object_node            varchar2(128),
    object_owner            varchar2(30),
    object_name             varchar2(30),
    object_alias            varchar2(65),
    object_instance         numeric,
    object_type              varchar2(30),
    optimizer                varchar2(255),
    search_columns           number,
    id                      numeric,
    parent_id                numeric,
    depth                    numeric,
    position                  numeric,
    cost                      numeric,
    cardinality              numeric,
    bytes                     numeric,
    other_tag                varchar2(255),
    partition_start           varchar2(255),
    partition_stop             varchar2(255),
    partition_id              numeric,
    other                      long,
    distribution              varchar2(30),
    cpu_cost                  numeric,
    io_cost                    numeric,
    temp_space                 numeric,
    access_predicates        varchar2(4000),
    filter_predicates        varchar2(4000),
    projection                varchar2(4000),
    time                      numeric,
    qblock_name              varchar2(30),
    other_xml                  clob
);
```

By the Way

The RDBMS/ADMIN/ subdirectory under your Oracle home directory will always contain the most up-to-date DDL script to create a PLAN_TABLE. You can connect as the SYSDBA to create this table and make it available to all users. The following statements will create the PLAN_TABLE under the SYS schema, create a public schema, and allow all users to make use of the PLAN_TABLE:

[Click here to view code image](#)

```
SQL> CONN sys/password AS SYSDBA
Connected
SQL> @$ORACLE_HOME/rdbms/admin/utlxplan.sql
SQL> GRANT ALL ON sys.plan_table TO public;
SQL> CREATE PUBLIC SYNONYM plan_table FOR sys.plan_table;
```

By default, if several plans in the plan table match the statement_id parameter that is passed to the display table function (the default value is NULL), only the plan corresponding to the last EXPLAIN PLAN command is displayed. Hence, there is no need to purge the plan table after each EXPLAIN PLAN is created. However, you should purge the plan table regularly (for example, by using the TRUNCATE TABLE command) to ensure good performance in the execution of the DISPLAY table function.

In prior versions of Oracle, a number of options were available. For example, you could use the SQL*Plus command SET AUTOTRACE TRACE EXPLAIN to generate an immediate explain plan.

[Click here to view code image](#)

```
SQL> SET AUTOTRACE TRACE EXPLAIN
  1  SELECT s.course_no,
  2        c.description,
  3        i.first_name,
  4        i.last_name,
  5        s.section_no,
  6        TO_CHAR(-s.start_date_time, 'Mon-DD-YYYY HH:MIAM'),
  7        s.location
  8  FROM section s,
  9       course c,
 10      instructor i
 11 WHERE s.course_no    = c.course_no
 12* AND   s.instructor_id= i.instructor_id

Execution Plan
-----
 0  SELECT STATEMENT Optimizer=CHOOSE (Cost=9 Card=78 Bytes=4368)
 1  0  HASH JOIN (Cost=9 Card=78 Bytes=4368)
 2  1   HASH JOIN (Cost=6 Card=78 Bytes=2574)
 3  2    TABLE ACCESS (FULL) OF 'INSTRUCTOR' (Cost=3 Card=10 Bytes=140)
 4  2    TABLE ACCESS (FULL) OF 'SECTION' (Cost=3 Card=78 Bytes=1482)
 5  1    TABLE ACCESS (FULL) OF 'COURSE' (Cost=3 Card=30 Bytes=690)
```

You can also generate an explain plan that will be stored in the PLAN_TABLE and then query the results of an explain plan.

[Click here to view code image](#)

```
SQL> explain plan for
  2  SELECT s.course_no,
  3          c.description,
  4          i.first_name,
  5          i.last_name,
  6          s.section_no,
  7          TO_CHAR(s.start_date_time,'Mon-DD-YYYY HH:MIAM'),
  8          s.location
  9  FROM section s,
 10      course c,
 11      instructor i
12 WHERE s.course_no      = c.course_no
13 AND   s.instructor_id= i.instructor_id;
```

Explained.

```
select rtrim( lpad( ' ', 2*level ) ||
              rtrim( operation ) || ' ' ||
              rtrim( options ) || ' ' ||
              object_name || ' ' ||
              partition_start || ' ' ||
              partition_stop || ' ' ||
              to_char( partition_id ) )
       the_query_plan
  from plan_table
 connect by prior id = parent_id
 start with id = 0;
```

THE_QUERY_PLAN

```
-----
      SELECT STATEMENT
        HASH JOIN
        HASH JOIN
          TABLE ACCESS BY INDEX ROWID SECTION
            INDEX FULL SCAN SECT_INST_FK_I
          SORT JOIN
            TABLE ACCESS FULL INSTRUCTOR
          TABLE ACCESS FULL COURSE
```

To make use of the DBMS_XPLAN procedure, use the `SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY)` command to generate the explain plan.

[Click here to view code image](#)

```
SQL> explain plan for
  2  SELECT s.course_no,
  3          c.description,
  4          i.first_name,
  5          i.last_name,
  6          s.section_no,
  7          TO_CHAR(s.start_date_time,'Mon-DD-YYYY HH:MIAM'),
  8          s.location
  9  FROM section s,
 10      course c,
 11      instructor i
12 WHERE s.course_no      = c.course_no
13 AND   s.instructor_id= i.instructor_id;
```

Explained.

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)
Time						
0	SELECT STATEMENT		78	4368	9	(34)
*00:00:01						
1	HASH JOIN		78	4368	9	(34)
00:00:01						
2	HASH JOIN		78	2574	6	(34)
00:00:01						
3	TABLE ACCESS FULL	INSTRUCTOR	10	140	3	(34)
00:00:01						
4	TABLE ACCESS FULL	SECTION	78	1482	3	(34)
00:00:01						
5	TABLE ACCESS FULL	COURSE	30	690	3	(34)
00:00:01						

Predicate Information (identified by operation id):

```
1 - access("S"."COURSE_NO"="C"."COURSE_NO")
2 - access("S"."INSTRUCTOR_ID"="I"."INSTRUCTOR_ID")
```

17 rows selected.

Generating Implicit Statement Results with DBMS_SQL

In older versions of Oracle, there were a few operations available in other database products such as Microsoft SQL Server that could not be done as elegantly in the Oracle platform. This created a challenge for companies that were migrating their applications to Oracle because they would have to make a great many changes to their stored procedures—even rewrite them—for use on the Oracle platform. Consider the ability to pass the results of a SQL statement out of a stored procedure. This can be done rather easily with the Transact SQL (T-SQL) procedural language of SQL Server, but in the Oracle platform it had to be done with a REF_CURSOR parameter. This is because T-SQL syntax permits implicit returns of a SQL result set from queries. A similar kind of functionality is now allowed in Oracle 12c through use of the DBMS_SQL package and the RETURN_RESULT procedure in that system package.

Using DBMS_SQL to Return a Result Set

The Oracle-supplied package DBMS_SQL includes an entity called a SQL cursor number. This PL/SQL integer can be passed as an IN or OUT parameter. You should use the DBMS_SQL package to run dynamic SQL when you don't know the details of the SELECT statement until it is run or you don't know which columns will be called in the SELECT statement until it is run.

In the following example, DBMS_SQL.RETURN_RESULT is used to return a result set without any OUT parameter.

For Example ch24.6.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PACKAGE Student_Instructor AS
PROCEDURE show_population
    (i_zip IN zipcode.zip%TYPE);
END Student_Instructor;
/

CREATE or REPLACE PACKAGE BODY Student_Instructor
AS
PROCEDURE show_population
    (i_zip IN zipcode.zip%TYPE)
AS
    student_list      SYS_REFCURSOR;
    instructor_list   SYS_REFCURSOR;
BEGIN
    OPEN student_list FOR
        SELECT 'Student' type, First_Name, Last_Name
        FROM student
        WHERE zip = i_zip;
    DBMS_SQL.RETURN_RESULT(student_list);
    OPEN instructor_list FOR
        SELECT 'Instructor' type, First_Name, Last_Name
        FROM instructor
        WHERE zip = i_zip;
    DBMS_SQL.RETURN_RESULT(instructor_list);
END show_population;
END Student_Instructor;
/
```

This script can be executed for the ZIP code 10025 as follows:

[Click here to view code image](#)

```
SQL> exec Student_Instructor.show_population('10025');
```

It produces the following result:

[Click here to view code image](#)

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

```
ResultSet #1
```

TYPE	FIRST_NAME	LAST_NAME
Student	Jerry	Abdou
Student	Nicole	Gillen
Student	Frank	Pace

```
ResultSet #2
```

TYPE	FIRST_NAME	LAST_NAME
Instructor	Tom	Wojick
Instructor	Nina	Schorin
Instructor	Todd	Smythe

Lab 24.2: Error Reporting with Oracle-Supplied Packages

After this lab, you will be able to

- [Use the DBMS.Utility Package for Error Reporting](#)
- [Use the UTL_Call_Stack Package for Error Reporting](#)

In [Chapters 8, 9, and 10](#), you explored various techniques for handling and reporting errors in your programs. The examples that you have seen so far are quite simple—that is, a single script that handles one or multiple exceptions. Oftentimes when working with applications, multiple programming units may be calling each other as well as passing control of the execution to a middle tier or a front end. In such circumstances, proper error reporting is important. Without it, both developers and users may encounter all sorts of problems.

In PL/SQL, two predefined packages may be used for this purpose: DBMS.Utility and UTL_Call_Stack.

Using the DBMS.Utility Package for Error Reporting

The DBMS.Utility package contains various utilitarian subprograms, some of which enhance error-reporting capabilities. These error-reporting functions are described in [Table 24.5](#).

Subprogram	Description
FORMAT_CALL_STACK	This function formats the execution call stack. Essentially it navigates through the code and leads you to the point at which the FORMAT_CALL_STACK function is called.
FORMAT_ERROR_BACKTRACE	This function formats the backtrace from the point where the error occurred to the exception handler where this error is handled. It provides the line numbers and the names of the subroutines in a manner that shows the path of the execution.
FORMAT_ERROR_STACK	This function formats the error stack. Basically, it provides the error number and error message associated with a particular exception.

Table 24.5 Error Reporting with the DBMS.Utility Package

FORMAT_CALL_STACK

As mentioned earlier, the FORMAT_CALL_STACK function formats and returns the current call stack up to 2000 bytes. It has the syntax shown in [Listing 24.1](#).

Listing 24.1 FORMAT_CALL_STACK

[Click here to view code image](#)

```
DBMS.Utility.FORMAT_CALL_STACK
RETURN VARCHAR2;
```

The next example illustrates the use of the FORMAT_CALL_STACK function. Note that it employs stored procedures.

For Example ch24_7.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE first
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_CALL_STACK);
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
    first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
    second;
END third;
/

BEGIN
    third;
END;
```

The preceding script creates three procedures: FIRST, SECOND, and THIRD. The FIRST procedure calls the FORMAT_CALL_STACK function, which then returns the call stack of the current execution. The SECOND procedure makes a call to the FIRST procedure, and the THIRD procedure makes a call to the SECOND procedure. Finally, the anonymous PL/SQL block at the end of the example makes a call to the THIRD procedure. When run, this example produces the following output:

[Click here to view code image](#)

```
-- PL/SQL Call Stack --
object      line      object
handle     number      name
0x104a93040      4  procedure STUDENT.FIRST
0xa06f8208      4  procedure STUDENT.SECOND
0x1045f1e68      4  procedure STUDENT.THIRD
0xa0259658      2  anonymous block
```

This call stack reveals the sequence of procedure invocations and should be read from the bottom up. First, the anonymous PL/SQL block calls THIRD, which in turn calls SECOND, which in turn calls FIRST. Finally, the FIRST procedure calls the FORMAT_CALL_STACK function.

FORMAT_ERROR_BACKTRACE

The FORMAT_ERROR_BACKTRACE function formats and returns the error backtrace (up to 2000 bytes) associated with the current error. It has the syntax shown in [Listing 24.2](#).

Listing 24.2 FORMAT_ERROR_BACKTRACE

[Click here to view code image](#)

```
DBMS_UTILITY.FORMAT_ERROR_BACKTRACE  
RETURN VARCHAR2;
```

Typically, the **FORMAT_ERROR_BACKTRACE** function is called inside the exception handler, as illustrated by the next example.

For Example *ch24_8.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE first  
IS  
    v_name VARCHAR2(30);  
BEGIN  
    DBMS_OUTPUT.PUT_LINE ('procedure FIRST');  
  
    SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)  
        INTO v_name  
        FROM student  
       WHERE student_id = 1000;  
END first;  
/  
  
CREATE OR REPLACE PROCEDURE second  
IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE ('procedure SECOND');  
    first;  
END second;  
/  
  
CREATE OR REPLACE PROCEDURE third  
IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE ('procedure THIRD');  
    second;  
END third;  
/  
BEGIN  
    third;  
EXCEPTION  
    WHEN OTHERS  
    THEN  
        DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);  
END;
```

Similar to the previous example, this script uses stored procedures to illustrate the behavior of the **FORMAT_ERROR_BACKTRACE** function. The **FIRST** procedure employs a **SELECT INTO** statement for the nonexistent student ID. The procedure itself does not have an exception handler; instead, the exception handler appears within the anonymous PL/SQL block. When run, this example produces the following output:

[Click here to view code image](#)

```
procedure THIRD  
procedure SECOND  
procedure FIRST
```

```
ORA-06512: at "STUDENT.FIRST", line 7
ORA-06512: at "STUDENT.SECOND", line 5
ORA-06512: at "STUDENT.THIRD", line 5
ORA-06512: at line 2
```

The first three lines of the output are produced by the DBMS_OUTPUT.PUT_LINE statements placed in each procedure. The last four lines of the output are produced by the FORMAT_ERROR_BACKTRACE function. Note how this error backtrace demonstrates the flow of the execution to the point where the exception occurred. Essentially, it tells us that the exception is found on line 7 in the procedure FIRST. As in the previous example, the output of this function should be read from the bottom up.

Only one item is missing in the output of this example—the exception itself. In essence, you are able to follow the execution path all the way to the precise line number where the exception occurred, but you do not know which exception has occurred. In this very simple example, it is easy to deduce that the problem is a NO_DATA_FOUND exception. In a more complex environment, however, you may be presented with much more intricate code or may not be as familiar with the table structures. In these circumstances, it is essential to know which exception was raised—and the FORMAT_ERROR_STACK function is able to answer this question.

FORMAT_ERROR_STACK

The FORMAT_ERROR_STACK function formats and returns the current error stack up to 2000 bytes. It has the syntax shown in [Listing 24.3](#).

Listing 24.3 FORMAT_ERROR_STACK

[Click here to view code image](#)

```
DBMS.Utility.Format_Error_Stack
RETURN VARCHAR2;
```

Like the FORMAT_ERROR_BACKTRACE function, the FORMAT_ERROR_STACK function is called inside the exception handler as well. This is illustrated by the modified version of the previous example.

For Example ch24_9.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE first
IS
    v_name VARCHAR2(30);
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
    SELECT RTRIM(first_name)||' '||RTRIM(last_name)
        INTO v_name
        FROM student
        WHERE student_id = 1000;
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
```

```

    first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
    second;
END third;
/


BEGIN
    third;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE ('Error Backtrace:');
        DBMS_OUTPUT.PUT_LINE ('-----');
        DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
        DBMS_OUTPUT.PUT_LINE ('Error Stack:');
        DBMS_OUTPUT.PUT_LINE ('-----');
        DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_STACK);
END;

```

When run, this version of the script produces the following output:

[Click here to view code image](#)

```

procedure THIRD
procedure SECOND
procedure FIRST
Error Backtrace:
-----
ORA-06512: at "STUDENT.FIRST", line 7
ORA-06512: at "STUDENT.SECOND", line 5
ORA-06512: at "STUDENT.THIRD", line 5
ORA-06512: at line 2
Error Stack:
-----
ORA-01403: no data found

```

This version of the output provides you with the flow of execution up to the point where the exception occurred. It also identifies the error number and error message associated with that exception.

Using the UTL_CALL_STACK Package for Error Reporting

The UTL_CALL_STACK package is a new built-in package introduced in Oracle 12c. It consists of a set of functions that provide various pieces of information on execution and error stacks, including subroutine and unit names and individual line numbers for dynamic depths. Some of these functions are described in [Table 24.6](#). For a complete list of the UTL_CALL_STACK subprograms, refer to the Oracle Database PL/SQL Packages and Types Reference available online.

Subprogram	Description
DYNAMIC_DEPTH	This function returns the number of subprograms in the current call stack. The dynamic depth of the currently executing subroutine is 1.
BACKTRACE_DEPTH	This function returns the number of items in the current backtrace. If there is no exception, the number of items in the backtrace remains 0.
BACKTRACE_LINE	This function returns the specific line number of the unit at the specified backtrace depth.
BACKTRACE_UNIT	This function returns the unit name at the specified backtrace depth.
ERROR_DEPTH	This function returns the number of errors in the current error stack.
ERROR_MSG	This function returns the error message associated with an error at the specified error depth.
ERROR_NUMBER	This function returns the error number associated with an error at the specified error depth.

Table 24.6 Error Reporting with the UTL_CALL_STACK Package

To enhance error reporting, the functions listed in [Table 24.6](#) are typically used in conjunction with other subroutines defined in the UTL_CALL_STACK package. These combinations are illustrated further by the examples in this lab.

DYNAMIC_DEPTH

The DYNAMIC_DEPTH function returns the number of subprograms in the current call stack. It has the syntax shown in [Listing 24.4](#).

Listing 24.4 DYNAMIC_DEPTH

```
UTL_CALL_STACK.DYNAMIC_DEPTH
RETURN PLS_INTEGER;
```

The use of this function is illustrated by the following example.

For Example ch24_10.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE first
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    first;
END second;
/
```

```

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    second;
END third;
/

BEGIN
    DBMS_OUTPUT.PUT_LINE ('anonymous block');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    third;
END;

```

This example creates three procedures: FIRST, SECOND, and THIRD. Each procedure and the anonymous PL/SQL block call the DYNAMIC_DEPTH function, which then returns the number of subprograms in the call stack of the current execution. When run, this example produces the following output:

```

anonymous block
dynamic depth: 1
procedure THIRD
dynamic depth: 2
procedure SECOND
dynamic depth: 3
procedure FIRST
dynamic depth: 4

```

This output illustrates the dynamic depth concept. The PL/SQL block is the currently executing subprogram; its dynamic depth is 1. Essentially, this is what has started the execution stack. When this block invokes the THIRD procedure, the dynamic depth of this procedure becomes 2. Basically, this is what was executed second in the call stack. Similarly, the dynamic depth of the SECOND procedure is 3, as it was executed third. Finally, the dynamic depth of the FIRST procedure is 4, as it was executed last.

Backtrace Depth, Unit, and Line Functions

The backtrace set of functions returns various backtrace data from the point where an exception was thrown to the point where the backtrace is examined. The syntax of the backtrace functions is shown in [Listing 24.5](#).

Listing 24.5 Backtrace Functions

[Click here to view code image](#)

```

UTL_CALL_STACK.BACKTRACE_DEPTH
RETURN PLS_INTEGER;

UTL_CALL_STACK.BACKTRACE_LINE (backtrace_depth IN PLS_INTEGER)
RETURN PLS_INTEGER;

UTL_CALL_STACK.BACKTRACE_UNIT (backtrace_depth IN PLS_INTEGER)
RETURN VARCHAR2;

```

The use of the backtrace functions is illustrated by the modified version of the earlier example. In this version, the **FIRST** procedure has been modified to cause a **VALUE_ERROR** exception, and the PL/SQL block has been extended with an exception-handling section. All changes are shown in bold.

For Example *ch24_11.sql*

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE first
IS
    v_string VARCHAR2(3);
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    v_string := 'ABCDEF';
END first;
/
CREATE OR REPLACE PROCEDURE second
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    second;
END third;
/
BEGIN
    DBMS_OUTPUT.PUT_LINE ('anonymous block');
    DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
    third;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE (CHR(10)||'Backtrace Stack: '||CHR(10)||RPAD('-', 15, '-'));
        DBMS_OUTPUT.PUT_LINE ('Backtrace Depth:
'||TO_CHAR(UTL_CALL_STACK.BACKTRACE_DEPTH));
        DBMS_OUTPUT.PUT_LINE ('Backtrace Line: ' ||
        TO_CHAR(UTL_CALL_STACK.BACKTRACE_LINE(UTL_CALL_STACK.BACKTRACE_DEPTH)));
        DBMS_OUTPUT.PUT_LINE ('Backtrace Unit: ' ||
        UTL_CALL_STACK.BACKTRACE_UNIT(UTL_CALL_STACK.BACKTRACE_DEPTH));
END;
```

Note how the value returned by the **BACKTRACE_DEPTH** function is used as an input

parameter to the BACKTRACE_LINE and BACKTRACE_UNIT functions. When run, this script produces the following output:

```
anonymous block
dynamic depth: 1
procedure THIRD
dynamic depth: 2
procedure SECOND
dynamic depth: 3
procedure FIRST
dynamic depth: 4

Backtrace Stack:
-----
Backtrace Depth: 4
Backtrace Line: 7
Backtrace Unit: STUDENT.FIRST
```

The backtrace stack reports that an exception was encountered in the backtrace depth 4, on line number 7 in the subroutine called FIRST in the STUDENT schema. This is very detailed backtrace output for such a simple example, yet it is still missing the exception itself. The set of error functions described next covers this exception-reporting gap.

Error Depth, Message, and Number Functions

Another set of error functions returns the error depth, message, and number of an error in the current stack. They have the syntax shown in [Listing 24.6](#).

Listing 24.6 Error Functions

[Click here to view code image](#)

```
UTL_CALL_STACK.ERROR_DEPTH
RETURN PLS_INTEGER;

UTL_CALL_STACK.ERROR_MSG (error_depth IN PLS_INTEGER)
RETURN VARCHAR2;

UTL_CALL_STACK.ERROR_NUMBER (error_depth IN PLS_INTEGER)
RETURN VARCHAR2;
```

Next consider how these functions may be utilized for error reporting. In this version of the script, the exception-handling section includes calls to these functions. All changes are shown in bold.

For Example ch24_12.sql

[Click here to view code image](#)

```
CREATE OR REPLACE PROCEDURE first
IS
  v_string VARCHAR2(3);
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  v_string := 'ABCDEF';
END first;
/
```

```

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  second;
END third;
/

BEGIN
  DBMS_OUTPUT.PUT_LINE ('anonymous block');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth:
'||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  third;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE (CHR(10)||'Backtrace Stack: '||CHR(10)||RPAD('-', 15, '-'));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Depth:
'||TO_CHAR(UTL_CALL_STACK.BACKTRACE_DEPTH));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Line: ' ||
TO_CHAR(UTL_CALL_STACK.BACKTRACE_LINE(UTL_CALL_STACK.BACKTRACE_DEPTH)));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Unit: ' ||
UTL_CALL_STACK.BACKTRACE_UNIT(UTL_CALL_STACK.BACKTRACE_DEPTH));
    DBMS_OUTPUT.PUT_LINE (CHR(10)||'Error Info: '||CHR(10)||RPAD('-', 15, '-'));
    DBMS_OUTPUT.PUT_LINE ('Error Depth:
'||TO_CHAR(UTL_CALL_STACK.ERROR_DEPTH));
    DBMS_OUTPUT.PUT_LINE ('Error Number: ' ||
TO_CHAR(UTL_CALL_STACK.ERROR_NUMBER (UTL_CALL_STACK.ERROR_DEPTH)));
    DBMS_OUTPUT.PUT_LINE ('Error Message: ' ||
UTL_CALL_STACK.ERROR_MSG(UTL_CALL_STACK.ERROR_DEPTH));
END;

```

When run, this example produces the following output:

[Click here to view code image](#)

```

anonymous block
dynamic depth: 1
procedure THIRD
dynamic depth: 2
procedure SECOND
dynamic depth: 3
procedure FIRST
dynamic depth: 4

```

```
Backtrace Stack:
```

```
-----
Backtrace Depth: 4
Backtrace Line: 7
Backtrace Unit: STUDENT.FIRST

Error Info:
-----
Error Depth: 1
Error Number: 6502
Error Message: PL/SQL: numeric or value error: character string buffer too
small
```

Note how the output now contains the error depth, number, and message. In a more complex environment, this type of trace data can provide invaluable insight for PL/SQL developers—insight that is essential in diagnosing and resolving problems in PL/SQL code efficiently.

This lab has covered only some of the functions of the `UTL_CALL_STACK` package. For additional information on how to utilize this package fully, refer to the Oracle Database PL/SQL Packages and Types Reference available online.

Summary

In this chapter, you learned about a variety of Oracle-supplied packages that can be used to extend the functionality of your programs. The strategy of accessing files on the operating system within a stored procedure by making use of `UTL_FILE` was reviewed. You also learned how to analyze SQL by making use of the explain plan generated by `DBMS_XPLAN`. In addition, you saw how to generate implicit statement results with `DBMS_SQL`. The chapter concluded with a discussion of the use of `DBMS.Utility` and `UTL_CALL_STACK` for error reporting.

25. Optimizing PL/SQL

In this chapter, you will learn about

- [PL/SQL Tuning Tools](#)
- [PL/SQL Optimization Levels](#)
- [Subprogram Inlining](#)

Oftentimes database developers need to improve the performance of their code when working in a complex development environment. The starting point of such exercise is usually the performance evaluation of the DML statements embedded in the PL/SQL code and their subsequent tuning. Once these statements have been improved, the tuning task is considered complete and the performance optimization of the PL/SQL code itself is usually overlooked.

In fact, Oracle provides a set of tools to help you identify performance bottlenecks and a wide variety of the optimization techniques specifically geared toward PL/SQL. For example, you have already seen how to minimize the number of context switches between PL/SQL and SQL engines and achieve better performance by employing bulk SQL and bulk binding. In this chapter, you will learn about the PL/SQL Profiler and Trace APIs, the PL/SQL Hierarchical Profiler tools, and the application of these tools to identify potential performance issues. In addition, you will learn about the PL/SQL performance optimizer and its optimization levels, and discover a new optimization technique called subprogram inlining.

Lab 25.1: PL/SQL Tuning Tools

After this lab, you will be able to

- Use the PL/SQL Profiler API
- Use the Trace API
- Use the PL/SQL Hierarchical Profiler

As mentioned previously, Oracle provides specific tools to help you diagnose performance issues in PL/SQL code: PL/SQL Profiler API, Trace API, and PL/SQL Hierarchical Profiler. All of these tools are implemented via Oracle-supplied packages, which makes them both readily available and fairly easy to install and use.

PL/SQL Profiler API

The PL/SQL Profiler API is implemented via the DBMS_PROFILER package. It computes how much time a PL/SQL program spends on executing each line of code and saves these computed times in database tables that may be queried later.

The following scripts located in the RDBMS/ADMIN directory are required to install

the PL/SQL Profiler API:

- PROFTAB.sql creates tables used by the profiler: PL/SQL_PROFILER_DATA, PLSQL_PROFILER_RUNS, and PLSQL_PROFILER_UNITS. This script should be executed by the STUDENT user.
- PROFLOAD.sql creates the DBMS_PROFILER package, and sets up synonyms and permissions for its usage. This script should be executed by the SYS user (connect as SYSDBA). The STUDENT user must be granted execute privileges on the DBMS_PROFILER package to run this script.

The main subprograms of the PL/SQL Profiler API are described in [Table 25.1](#). More detailed information on the DBMS_PROFILER routines, exceptions, and tables may be found in the Introduction to Oracle Supplied PL/SQL Packages & Types, which is available online as part of Oracle's documentation.

Subprogram Name	Description
START_PROFILER	This routine starts the profiler in the user's session. At this point, the profiler begins capturing various performance metrics.
STOP_PROFILER	This routine stops the profiler in the user's session. At this point, the profiler stops capturing any performance metrics.
PAUSE_PROFILER	This routine pauses the profiler operations and temporarily suspends collection of performance statistics. It may be helpful when collecting performance metrics for complex and large PL/SQL programs and when some portions of the code do not require performance tuning.
RESUME_PROFILER	This routine resumes the profiler operations and enables collection of performance statistics. It is used in conjunction with the PAUSE_PROFILER routine.
FLUSH_DATA	This routine writes captured performance data periodically to the database tables during program execution and while the profiler itself is still running. Similar to the PAUSE_PROFILER and RESUME_PROFILER routines, it is typically used when profiling complex and large PL/SQL programs.

Table 25.1 DBMS_PROFILER Main Subprograms

This lab introduces the PL/SQL Profiler API, whereas [Lab 25.1](#) demonstrates how this tool may be used to profile your code.

Trace API

The Trace API is implemented via the DBMS_TRACE package. Oftentimes this tool is used in conjunction with the PL/SQL Profiler API, as it provides an additional level of detail and context to the performance statistics captured by the profiler. The Trace API traces the order of the execution of the PL/SQL routines and saves this data in database tables that may be queried later.

The following scripts located in the RDBMS/ADMIN directory are required to install the Trace API:

- TRACETAB.sql creates tables used by the Trace API: PLSQL_TRACE_RUNS and PLSQL_TRACE_EVENTS. This script should be executed by the SYS user. The appropriate permissions (SELECT, INSERT, and DELETE) must be granted to the

STUDENT user to run this script. You might also find it helpful to create synonyms for these tables.

- The DBMS_TRACE package is usually installed as part of the default installation. However, the execute privileges may need to be granted to the STUDENT schema. This should be accomplished by the SYS user.

The main subprograms of the Trace API are described in [Table 25.2](#). More detailed information on the DBMS_TRACE routines, exceptions, and tables may be found in the Introduction to Oracle Supplied PL/SQL Packages & Types, which is available online as part of Oracle's documentation.

Subprogram Name	Description
SET_PLSQL_TRACE	This routine starts tracing in a particular session and begins collecting trace data. The type and volume of data collected are determined by the INTEGER value supplied to the TRACE_LEVEL parameter. These values are based in the DBMS_TRACE package constants (described in Table 25.3).
CLEAR_PLSQL_TRACE	This routine stops tracing in a particular session.
GET_PLSQL_TRACE_LEVEL	This routine gets the trace level set in a particular session.

Table 25.2 DBMS_TRACE Main Subprograms

As mentioned previously, the content and volume of the trace data are based on the trace level, which is supplied when the trace starts and is based on the DBMS_TRACE constants listed in [Table 25.3](#). A full and detailed description of the DBMS_TRACE constants may be found in the Introduction to Oracle Supplied PL/SQL Packages & Types, which is available online as part of Oracle's documentation.

Constant Name	Value	Description
TRACE_ALL_CALLS	1	Traces all calls to a particular PL/SQL routine
TRACE_ENABLED_CALLS	2	Traces calls to a particular PL/SQL routine only if it has been enabled for tracing
TRACE_ALL_EXCEPTIONS	4	Traces all raised exceptions
TRACE_ENABLED_EXCEPTION	8	Traces exceptions raised in a PL/SQL routine only if it has been enabled for tracing
TRACE_ALL_SQL	32	Traces all SQL statements
TRACE_ENABLED_SQL	64	Traces SQL statements in a particular PL/SQL routine only if it has been enabled for tracing
TRACE_PAUSE	4096	Pauses trace
TRACE_RESUME	8192	Resumes trace
TRACE_STOP	16384	Stops trace

Table 25.3 DBMS_TRACE Main Constants

Tracing in PL/SQL programs can be enabled at the session level via the ALTER SESSION command or for a particular stored PL/SQL routine via the ALTER command:

[Click here to view code image](#)

```
ALTER SESSION SET PLSQL_DEBUG = TRUE;
```

or

[Click here to view code image](#)

```
ALTER [PROCEDURE/FUNCTION/PACKAGE] ROUTINE_NAME  
COMPILE DEBUG [BODY (applicable to packages only)]
```

Consider the following example, which traces a simple PL/SQL procedure created specifically for this purpose:

For Example *ch25_1a.sql*

[Click here to view code image](#)

```
ALTER SESSION SET PLSQL_DEBUG = TRUE;  
  
-- Create test procedure to be traced  
CREATE OR REPLACE PROCEDURE TEST_TRACE  
AS  
    v_num1 NUMBER;  
    v_num2 NUMBER;  
    v_num3 NUMBER;  
    v_date DATE;  
BEGIN  
    FOR i IN 1..10  
    LOOP  
        v_num1 := 1;  
        v_num2 := i + i/2 + sqrt(i);  
        v_num3 := v_num1 + v_num2;  
  
        SELECT sysdate  
        INTO v_date  
        FROM DUAL;  
    END LOOP;  
END TEST_TRACE;  
/  
  
-- Trace TEST_TRACE procedure  
BEGIN  
    DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_CALLS);  
    TEST_TRACE;  
    DBMS_TRACE.CLEAR_PLSQL_TRACE;  
END;
```

In this script, the database session has been enabled for tracing. As a result, the **TEST_TRACE** procedure created in this session is compiled with the **DEBUG** option and, therefore, is available for tracing as well. The execution of the newly created procedure is then traced in the PL/SQL block. Note that the level of tracing has been set via the **DBMS_TRACE.TRACE_ALL_CALLS** constant.

Once the procedure is created and the PL/SQL block is executed, the trace data may be examined by querying the **PLSQL_TRACE_RUNS** and **PLSQL_TRACE_EVENTS** tables as shown here:

[Click here to view code image](#)

```
SELECT r.runid  
    ,e.event_seq  
    ,e.event_unit_owner  
    ,e.event_unit
```

```

,e.event_unit_kind
,e.proc_line
,e.event_comment
FROM plsql_trace_runs r
,plsql_trace_events e
WHERE r.runid = 1 -- this value must change based on the number of traces
run
AND r.runid = e.runid
ORDER BY r.runid, e.event_seq;

```

RUNID	EVENT	EVENT_UNIT	EVENT_UNIT_KIND	PROC_LINE	EVENT_COMMENT		
	SEQ	UNIT_OWNER					
	1	1				PL/SQL	
Trace	Tool started						
	1	2				Trace	
flags changed							
	1	3 SYS	DBMS_TRACE	PACKAGE BODY	75	Return	
from procedure call							
	1	4 SYS	DBMS_TRACE	PACKAGE BODY	81	Return	
from procedure call							
	1	5 SYS	DBMS_TRACE	PACKAGE BODY	3	Return	
from procedure call							
	1	6	<anonymous>	ANONYMOUS			
BLOCK		1	Procedure Call				
	1	7 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	8 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	9 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	10 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	11 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	12 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	13 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	14 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	15 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	16 STUDENT	TEST_TRACE	PROCEDURE		PL/SQL	
Internal Call							
	1	17 STUDENT	TEST_TRACE	PROCEDURE	4	Return	
from procedure call							
	1	18	<anonymous>	ANONYMOUS			
BLOCK		92	Procedure Call				
	1	19 SYS	DBMS_TRACE	PACKAGE			
BODY		69	Procedure Call				
	1	20 SYS	DBMS_TRACE	PACKAGE			
BODY		64	Procedure Call				
	1	21 SYS	DBMS_TRACE	PACKAGE			
BODY		12	Procedure Call				
	1	22 SYS	DBMS_TRACE	PACKAGE BODY	66	Return	
from procedure call							
	1	23 SYS	DBMS_TRACE	PACKAGE BODY	72	Return	

```

from procedure call
  1 24  SYS      DBMS_TRACE PACKAGE
BODY          21    Procedure Call
  1 25
trace stopped

```

PL/SQL

PL/SQL Hierarchical Profiler

The PL/SQL Hierarchical Profiler is implemented via the DBMS_HPROF package. It profiles the execution of PL/SQL applications and reports on the execution times for SQL and PL/SQL separately. The Hierarchical Profiler organizes the run-time data it collects based on subprogram calls. Similarly to the PL/SQL Profiler and Trace APIs, it stores the collected statistics in database tables that may be queried later.

The following scripts located in the RDBMS/ADMIN directory are required to install the PL/SQL Hierarchical Profiler API:

- DBMSHPTAB.sql creates the tables used by the Hierarchical Profiler: DBMSHP_RUNS, DBMSHP_FUNCTION_INFO, and DBMSHP_PARENT_CHILD_INFO. This script may be executed by the STUDENT user.
- The STUDENT user needs read and write privileges on the directory object and the directory itself to which the directory object is mapped. Note that these steps should be executed by the SYS user (connect as SYSDBA).
 - For example, if the file system includes the /plshprof/results directory, the following statement creates the directory object PLSHPROF_DIR and maps it to this directory:

[Click here to view code image](#)

```

CREATE DIRECTORY PLSHPOF_DIR AS '/plshprof/results';
GRANT READ, WRITE ON DIRECTORY PLSHPOF_DIR TO STUDENT;

```

- The DBMS_HPROF package is usually installed as part of the default installation. However, the execute privileges may need to be granted to the STUDENT schema. This should be accomplished by the SYS user (connect as SYSDBA).

The main subprograms of the PL/SQL Hierarchical Profiler are described in [Table 25.4](#). More detailed information on the DBMS_HPROF routines, exceptions, and tables may be found in the Introduction to Oracle Supplied PL/SQL Packages & Types, which is available online as part of Oracle's documentation.

Subprogram Name	Description
START_PROFILING	This routine starts the Hierarchical Profiler in the user's session. At this point, the profiler begins capturing various performance metrics.
STOP_PROFILING	This routine stops the Hierarchical Profiler in the user's session. At this point, the profiler stops capturing any performance metrics.
ANALYZE	This routine analyzes the Hierarchical Profiler data and writes it into the database tables.

Table 25.4 DBMS_HPROF Main Subprograms

Use of the PL/SQL Hierarchical Profiler is demonstrated in [Lab 25.3](#).

Lab 25.2: PL/SQL Optimization Levels

After this lab, you will be able to

- Understand PL/SQL Optimization Levels

In version 10g, Oracle introduced a new feature to PL/SQL compiler, called the performance optimizer. Essentially, the optimizer enables the PL/SQL compiler to reorganize source code to enhance its performance. The level of optimization that compiler applies to the code is controlled by the `PLSQL_OPTIMIZE_LEVEL` parameter, which may be set at the instance or session level. These levels of optimization are listed in the [Table 25.5](#).

Optimization Level	Description
Level 0	No optimization. This is equivalent to the versions of Oracle prior to 10g.
Level 1	Some optimization. This includes removal of unnecessary computations, exceptions, or assignments without rearranging of source code.
Level 2	Default optimization. This yields a more significant performance improvement as the code may be rewritten and/or relocated during the compilation process.
Level 3	Highest level of optimization added in version 11g. This uses automatic subprogram inlining.

Table 25.5 PL/SQL Optimization Levels

To illustrate how PL/SQL code performance is affected by different optimization levels, consider a simple example that executes a numeric FOR LOOP and performs some meaningless calculations. It is executed with `PLSQL_OPTIMIZE_LEVEL` set to 0, 1, and 2, respectively. In addition, it uses the PL/SQL Profiler so that you can get a closer look at how and where the performance actually occurs. Note that the line numbers listed in the example are provided for future reference and are not part of the actual PL/SQL code. The `SET TIMING ON` command enables Oracle to measure and display the execution time of the script. The `ALTER SESSION` command sets the `PL_SQL_OPTIMIZE` level variable to a specified value at the session level.

Watch Out!

Prior to running examples used in this lab, the `STUDENT` schema should be enabled to use the PL/SQL Profiler API as described in [Lab 25.1](#).

For Example `ch25_2a.sql`

[Click here to view code image](#)

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 0;

1 DECLARE
```

```

2   v_num1    NUMBER;
3   v_num2    NUMBER;
4   v_num3    NUMBER;
5   v_run_id BINARY_INTEGER; – run ID generated by the profiler
6 BEGIN
7   DBMS_PROFILER.START_PROFILER ('Optimizer level at 0');
8
9   FOR i IN 1..1000000
10  LOOP
11    v_num1 := 1;
12    v_num2 := i + i/2 + sqrt(i);
13    v_num3 := v_num1 + v_num2;
14  END LOOP;
15
16 DBMS_PROFILER.STOP_PROFILER();
17
18 SELECT runid
19   INTO v_run_id
20   FROM plsql_profiler_runs
21  WHERE run_comment = 'Optimizer level at 0';
22
23 DBMS_OUTPUT.PUT_LINE ('Optimizer level at 0, run ID - '||v_run_id);
24 END;

```

As mentioned previously, this script performs some worthless calculations and employs the PL/SQL Profiler API to collect runtime statistics. To analyze these run-time statistics, it defines the `v_run_id` variable to store the ID of the profiler run. This ID is selected from the `PLSQL_PROFILER_RUNS` table and displayed at the end of the script. The profiler runtime data collection starts with the `DBMS_PROFILER.START_PROFILER` procedure and ends with the `DBMS_PROFILER.STOP_PROFILER` procedure.

When run, the script produces the following output:

[Click here to view code image](#)

```

Elapsed: 00:00:03.317
Optimizer level at 0, run ID - 1

```

The first line of the output is shown in the Script Output window when this script is executed in the SQL Developer. For the second run, the `PLSQL_OPTIMIZE_LEVEL` is set to 1 and the script is modified as shown. All changes are highlighted in bold.

For Example `ch25_2b.sql`

[Click here to view code image](#)

```

SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

1 DECLARE
2   v_num1    NUMBER;
3   v_num2    NUMBER;
4   v_num3    NUMBER;
5   v_run_id BINARY_INTEGER; – run ID generated by the profiler
6 BEGIN
7   DBMS_PROFILER.START_PROFILER ('Optimizer level at 1');
8
9   FOR i IN 1.. 1000000
10  LOOP
11    v_num1 := 1;

```

```

12      v_num2 := i + i/2 + sqrt(i);
13      v_num3 := v_num1 + v_num2;
14  END LOOP;
15
16  DBMS_PROFILER.STOP_PROFILER();
17
18  SELECT runid
19    INTO v_run_id
20    FROM plsql_profiler_runs
21   WHERE run_comment = 'Optimizer level at 1';
22
23  DBMS_OUTPUT.PUT_LINE ('Optimizer level at 1, run ID - '||v_run_id);
24 END;

```

When run, this version produces the following output:

[Click here to view code image](#)

```

Elapsed: 00:00:03.103
Optimizer level at 1, run ID - 2

```

Note that there is a very negligible performance gain between the two runs.

Next, consider yet another version of the example where PLSQL_OPTIMIZE_LEVEL is set to 2 and the PL/SQL block is modified accordingly. Affected statements are highlighted in bold.

For Example *ch25_2c.sql*

[Click here to view code image](#)

```

SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1 DECLARE
2   v_num1  NUMBER;
3   v_num2  NUMBER;
4   v_num3  NUMBER;
5   v_run_id BINARY_INTEGER; -- run ID generated by the profiler
6 BEGIN
7   DBMS_PROFILER.START_PROFILER ('Optimizer level at 2');
8
9   FOR i IN 1.. 1000000
10  LOOP
11    v_num1 := 1;
12    v_num2 := i + i/2 + sqrt(i);
13    v_num3 := v_num1 + v_num2;
14  END LOOP;
15
16  DBMS_PROFILER.STOP_PROFILER();
17
18  SELECT runid
19    INTO v_run_id
20    FROM plsql_profiler_runs
21   WHERE run_comment = 'Optimizer level at 2';
22
23  DBMS_OUTPUT.PUT_LINE ('Optimizer level at 2, run ID - '||v_run_id);
24 END;

```

When run, this version produces the following output:

[Click here to view code image](#)

```
Elapsed: 00:00:02.562
Optimizer level at 2, run ID - 3
```

With the PLSQL_OPTIMIZE_LEVEL set to 2, the gain in performance is more noticeable.

What has happened since the optimization level was changed from 0 to 1 to 2? To answer this question, let's examine data generated by the PL/SQL Profiler:

[Click here to view code image](#)

```
SELECT r.runid, r.run_comment, d.line#, d.total_occur, d.total_time
  FROM plsql_profiler_runs    r
       ,plsql_profiler_data   d
       ,plsql_profiler_units  u
 WHERE r.runid = d.runid
   AND d.runid = u.runid
   AND d.unit_number = u.unit_number
   AND d.total_occur > 0
ORDER BY d.runid, d.line#;
```

RUNID	RUN_COMMENT	LINE#	TOTAL_OCCUR
--	-----	----	-----
1	Optimizer level at		
0	9 1000001	128784207	
1	Optimizer level at		
0	11 1000000	204515328	
1	Optimizer level at		
0	12 1000000	1928730621	
1	Optimizer level at		
0	13 1000000	235407659	
1	Optimizer level at		
0	14 1	0	
1	Optimizer level at		
0	16 1	13032	
2	Optimizer level at		
1	9 1000001	122832257	
2	Optimizer level at		
1	11 1000000	143920674	
2	Optimizer level at		
1	12 1000000	1820567385	
2	Optimizer level at		
1	13 1000000	181771219	
2	Optimizer level at		
1	14 1	0	
2	Optimizer level at		
1	16 1	12000	
3	Optimizer level at		
2	9 1000001	134848732	
3	Optimizer level at		
2	11 1000000	0	
3	Optimizer level at		
2	12 1000000	1583962321	
3	Optimizer level at		
2	13 1000000	188121402	
3	Optimizer level at		
2	16 1	10015	

The SELECT statement selects data from the PL/SQL profiler tables for each line of the code that was executed (`d.total_occur > 0`). Take a closer look at the data generated for line 11 (highlighted in bold). Line 11 corresponds to the assignment statement

```
11    v_num1 := 1;
```

in the body of the numeric FOR LOOP. Note that even though line 11 is executed 1 million times for run IDs 1 and 2 (optimization levels 0 and 1, respectively), there is a performance gain shown in the TOTAL_TIME column. Next, take a look at line 11 for run ID 3 (optimization level 2). Although the TOTAL_OCCUR column states that the assignment statement has executed 1 million times, the total time spent on this operation was 0. How is that possible?

Recall that setting the optimization level to 2 enables code relocation and rewriting. Thus it is possible that the assignment operation at line 11 was relocated outside the loop, or that variable `v_num1` was removed all together and its value was substituted on line 13. So, the original statement

[Click here to view code image](#)

```
13    v_num3 := v_num1 + v_num2;
```

could have become

```
13    v_num3 := 1 + v_num2;
```

Another important optimization technique introduced for optimization level 2 is implicit bulk fetches for static CURSOR FOR LOOPS with a limit of 100 records. Thus, if a piece of code fetches and processes records in the CURSOR FOR LOOP one at time, with level 2 optimization the records will be fetched in a bulk, 100 records at a time. This approach yields a significant performance improvement, as demonstrated by the following example.

For Example *ch25_3a.sql*

[Click here to view code image](#)

```
SET TIMING ON;

-- Create test table
CREATE TABLE TEST_TAB
  (col1 NUMBER);
/
-- Populate newly created table with random data
INSERT INTO TEST_TAB
SELECT ROUND(DBMS_RANDOM.VALUE (1, 99999999), 0)
  FROM dual
CONNECT by level < 100001;
COMMIT;

-- Collect statistics
EXEC DBMS_STATS.GATHER_TABLE_STATS (user, 'TEST_TAB');

-- Run the same code sample with different optimization levels
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

BEGIN
  FOR rec IN (SELECT col1 FROM test_tab)
```

```

LOOP
    null; — do nothing
END LOOP;
END;
/

ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;
BEGIN
    FOR REC IN (SELECT col1 FROM test_tab)
    LOOP
        NULL; — do nothing
    END LOOP;
END;
/

```

First, this script creates the TEST_TAB table. Then, it populates the TES_TAB table with some random numeric data and gathers table statistics via the DBMS_STATS.GATHER_TABLE_STATS procedure.

Did You Know?

That DBMS_STATS package is used to gather various statistics on database objects. These statistics may be gathered for one database object at a time, as in the preceding example, or for all objects in the database or schema.

Next, the script sets optimizer level to 1, and executes the CURSOR FOR LOOP against the TEST_TAB table. Note the usage of the NULL; statement in the body of the loop. Essentially, it means that nothing is done in the body of the loop. Finally, the script sets the optimizer level to 2 and executes the CURSOR FOR LOOP again.

When run, this example produces the following output:

```

table TEST_TAB created.
Elapsed: 00:00:00.060
100,000 rows inserted.
Elapsed: 00:00:01.595
committed.
Elapsed: 00:00:00.016
anonymous block completed
session SET altered.
Elapsed: 00:00:00.001
anonymous block completed
Elapsed: 00:00:00.767
session SET altered.
Elapsed: 00:00:00.002
anonymous block completed
Elapsed: 00:00:00.080

```

Take a closer look at the elapsed times highlighted in bold. The execution time went from .767 to .080. This is a significant improvement for such a simple script. Next, consider expanding on this example by creating a new table TEST_TAB1 and populating it in the CURSOR FOR LOOP. Much like the preceding example, this version is executed for optimizer levels 1 and 2.

For Example ch25_4a.sql

[Click here to view code image](#)

```
SET TIMING ON;

-- Create test table
CREATE TABLE test_tab1 (col1 NUMBER);
/
-- Run the same code sample with different optimization levels
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

BEGIN
  FOR rec IN (SELECT col1 FROM test_tab)
  LOOP
    INSERT INTO TEST_TAB1 VALUES (rec.col1); -- populate newly created
table
  END LOOP;
END;
/

ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

BEGIN
  FOR REC IN (SELECT col1 FROM test_tab)
  LOOP
    INSERT INTO TEST_TAB1 VALUES (rec.col1); -- populate newly created
table
  END LOOP;
END;
/
```

This example produces the following output:

```
table TEST_TAB1 created.
Elapsed: 00:00:00.059
session SET altered.
Elapsed: 00:00:00.001
anonymous block completed
Elapsed: 00:00:10.683
session SET altered.
Elapsed: 00:00:00.002
anonymous block completed
Elapsed: 00:00:09.668
```

As soon as the `INSERT` statement has been added to the body of the loop, the major performance gain between two optimization levels is lost. This is due to lack of implicit optimization for the DML statements. In this case, better performance is gained by changing the script and adding bulk SQL optimization techniques (covered in [Chapter 18](#)).

The last two examples demonstrate very clearly that while it is helpful to recognize that PL/SQL optimization may occur behind the scenes, it is not a good idea to rely exclusively on it. While sometimes the code you create may be optimized at the time of compilation, it is much better to go through the optimization exercise and change that code explicitly based on the performance findings.

Lab 25.3: Subprogram Inlining

After this lab, you will be able to

- Use Subprogram Inlining

In [Lab 25.2](#), you learned about different PL/SQL optimization levels. Specifically, you saw examples with optimization levels of 0, 1, and 2. In this lab, you will learn about the concept of subprogram inlining and discover how it is used with optimization level 3.

In the PL/SQL Language Reference, subprogram inlining concept is defined as follows: “Subprogram inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit).” Subprogram inlining may be enabled either by using the PRAGMA statement or by setting PLSQL_OPTIMIZE_LEVEL to 3 as shown in [Listing 25.1](#).

Listing 25.1 Enabling Subprogram Inlining

[Click here to view code image](#)

```
PRAGMA INLINE (subprogram_name, 'YES');
```

or

[Click here to view code image](#)

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;
```

When a PRAGMA INLINE statement is used, it should appear prior to each subprogram call. Recall that PLSQL_OPTIMIZE_LEVEL may be set to a particular value at the instance level as well. When PLSQL_OPTIMIZE_LEVEL is set to 3, subprogram inlining is done automatically.

The usage of subprogram inlining and the performance gains associated with it are best illustrated by examining some examples. In these scripts, the same PL/SQL code is executed twice. For both runs, the PL/SQL optimization level remains at 2, but subprogram inlining is enabled for the second run only. The line numbers in the examples are provided for future reference and are not part of the actual PL/SQL code.

Watch Out!

Prior to running the examples in this lab, the STUDENT schema should be extended to use the PL/SQL Hierarchical Profiler. The necessary steps are described in [Lab 25.1](#).

For Example ch25_5a.sql

[Click here to view code image](#)

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2      v_num      PLS_INTEGER;
3      v_run_id  BINARY_INTEGER; -- run ID generated by the profiler
```

```

4
5      FUNCTION test_func (num1 IN PLS_INTEGER
6                      ,num2 IN PLS_INTEGER)
7      RETURN PLS_INTEGER
8      IS
9      BEGIN
10         RETURN (num1 + num2);
11     END test_func;
12
13 BEGIN
14     DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15     FOR i IN 1..100000
16     LOOP
17         v_num := test_func (i-1, i);
18     END LOOP;
19     DBMS_HPROF.STOP_PROFILING;
20
21     -- Analyze profiler output and display its run ID
22     v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
23     DBMS_OUTPUT.PUT_LINE ('Inline pragma is not enabled, run ID -
' || v_run_id);
24 END;

```

This script defines a function `test_func` that returns the sum of two numbers. This function is then invoked in the body of the numeric `FOR LOOP`, where its result is assigned to the variable `v_num`. The execution of loop is profiled by the PL/SQL Hierarchical Profiler. The profiler writes its raw data into the file called `test.txt` located in the `/plshprof/results/` directory. Finally, the profiler output is analyzed and recorded in its tables, and the run ID is displayed on the screen for later reference.

When run, this version of the example produces the following output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is not enabled, run ID - 1
Elapsed: 00:00:00.602

```

Next, consider the modified version of the example in which subprogram inlining is enabled. Changes are highlighted in bold.

For Example `ch25_5b.sql`

[Click here to view code image](#)

```

SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2      v_num      PLS_INTEGER;
3      v_run_id  BINARY_INTEGER; -- run ID generated by the profiler
4
5      FUNCTION test_func (num1 IN PLS_INTEGER
6                          ,num2 IN PLS_INTEGER)
7      RETURN PLS_INTEGER
8      IS
9      BEGIN
10         RETURN (num1 + num2);
11     END test_func;

```

```

12
13 BEGIN
14   DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15   FOR i IN 1..100000
16   LOOP
17     -- Inline pragma is enabled for each function call
18     PRAGMA INLINE (test_func, 'YES');
19     v_num := test_func (i-1, i);
20   END LOOP;
21   DBMS_HPROF.STOP_PROFILING;
22
23   -- Analyze profiler output and display its run ID
24   v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
25   DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID -
'||v_run_id);
26 END;

```

When run, this version of the example produces the following output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is enabled, run ID - 2
Elapsed: 00:00:00.073

```

Note how much performance has been gained in the second run by adding the PRAGMA INLINE statement without changing the PL/SQL optimization level.

As mentioned previously, when the PL/SQL optimization level is set to 2, subprogram inlining must be enabled explicitly prior to each subprogram call. Hence, if the PRAGMA INLINE statement was placed outside the loop, subprogram inlining would not occur. This is demonstrated by the next example. Affected statements are shown in bold.

For Example *ch25_5c.sql*

[Click here to view code image](#)

```

SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2    v_num    PLS_INTEGER;
3    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
4
5    FUNCTION test_func (num1 IN PLS_INTEGER
6                      ,num2 IN PLS_INTEGER)
7    RETURN PLS_INTEGER
8    IS
9    BEGIN
10      RETURN (num1 + num2);
11    END test_func;
12
13 BEGIN
14   DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15
16   -- Inline pragma is moved outside the loop
17   PRAGMA INLINE (test_func, 'YES');
18   FOR i IN 1..100000
19   LOOP

```

```

20      v_num := test_func (i-1, i);
21  END LOOP;
22  DBMS_HPROF.STOP_PROFILING;
23
24  -- Analyze profiler output and display its run ID
25  v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
26  DBMS_OUTPUT.PUT_LINE
27    ('Inline pragma is enabled for a single call, run ID -
' ||v_run_id);
28 END;

```

When run, this version of the script produces the following output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is enabled for a single call, run ID - 3
Elapsed: 00:00:00.490

```

As you can see, we have lost almost all of the performance gain.

Since each example run was profiled and analyzed, let's see what information the PL/SQL Hierarchical Profiler has gathered:

[Click here to view code image](#)

```

SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info;

```

RUNID	FUNCTION	LINE#	CALLS	S_E_T
1	__anonymous_block.TEST_FUNC	5	100000	17461
1	STOP_PROFILING	63	1	0
2	STOP_PROFILING	63	1	0
3	__anonymous_block.TEST_FUNC	5	100000	18327
3	STOP_PROFILING	63	1	0

For run IDs 1 and 3, `test_func` was executed 100,000 times; in contrast, for run ID 2, there is no reference to `test_func`. In effect, at the time of compilation the code is revised to look like the following example (all changes are shown in bold):

For Example *ch25_5d.sql*

[Click here to view code image](#)

```

SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2      v_num      PLS_INTEGER;
3      v_run_id  BINARY_INTEGER; -- run ID generated by the profiler
4
5  BEGIN
6      DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
7      FOR i IN 1..100000
8      LOOP
9          v_num := i-1 + i; -- there is no reference to test_func
10     END LOOP;

```

```

11  DBMS_HPROF.STOP_PROFILING;
12
13  -- Analyze profiler output and display its run ID
14  v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
15  DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID -
16 ||v_run_id);
17 END;

```

As mentioned previously, setting the optimizer level to 3 ensures implicit subprogram inlining whenever possible. This is demonstrated by the next example (modified statements are shown in bold):

For Example *ch25_5e.sql*

[Click here to view code image](#)

```

SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;

1  DECLARE
2      v_num      PLS_INTEGER;
3      v_run_id  BINARY_INTEGER; -- run ID generated by the profiler
4
5      FUNCTION test_func (num1 IN PLS_INTEGER
6                          ,num2 IN PLS_INTEGER)
7          RETURN PLS_INTEGER
8      IS
9          BEGIN
10             RETURN (num1 + num2);
11         END test_func;
12
13 BEGIN
14     DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15
16 FOR i IN 1..100000
17 LOOP
18     v_num := test_func (i-1, i);
19 END LOOP;
20 DBMS_HPROF.STOP_PROFILING;
21
22 -- Analyze profiler output and display its run ID
23 v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
24 DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled implicitly, run ID -
25 ||v_run_id);
26 END;

```

In this version, PLSQL_OPTIMIZE_LEVEL has been set to 3 and the PL/SQL compiler is able to perform subprogram inlining explicitly. This is demonstrated by the script's output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is enabled implicitly, run ID - 4
Elapsed: 00:00:00.065

```

As you can see, based on the elapsed time, it seems that the compiler has performed subprogram inlining. This fact can be confirmed by examining the data produced by the PL/SQL Hierarchical Profiler:

[Click here to view code image](#)

```
SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 4;
RUNID      FUNCTION                      LINE#      CALLS      S_E_T      F_E_T
---  -----  --  --  ---  --
 4      STOP_PROFILING                  63          1          0          0
```

When the optimization level is set to 2 and subprogram inlining is specified for a particular function/procedure, this feature does not propagate to the procedures/functions that are being called from it. In other words, if subprogram inlining is enabled for procedure P1, and it references functions F1 and F2, these functions are not enabled for inlining. This concept is illustrated further by the next example.

For Example ch25_6a.sql

[Click here to view code image](#)

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2      v_run_id BINARY_INTEGER; -- run ID generated by the profiler
3
4      FUNCTION f1 (num1 IN PLS_INTEGER
5                      ,num2 IN PLS_INTEGER)
6      RETURN PLS_INTEGER
7      IS
8      BEGIN
9          RETURN (num1 + num2);
10     END f1;
11
12    FUNCTION f2 (str1 IN VARCHAR2
13                      ,str2 IN VARCHAR2)
14    RETURN VARCHAR2
15    IS
16    BEGIN
17        RETURN (str1||' '||str2);
18    END f2;
19
20    PROCEDURE p1 (num1 IN PLS_INTEGER
21                      ,num2 IN PLS_INTEGER
22                      ,str1 IN VARCHAR2
23                      ,str2 IN VARCHAR2)
24    IS
25        v_num NUMBER;
26        v_str VARCHAR2(100);
27    BEGIN
28        v_num := f1(num1, num2);
29        v_str := f2(str1, str2);
30    END p1;
31
32 BEGIN
33     DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
34     FOR i in 1..100000
35     LOOP
36         -- Inline pragma is enabled for each procedure call
37         PRAGMA INLINE (p1, 'YES');
```

```

38      p1 (i-1, i, to_char(i-1), to_char(i));
39  END LOOP;
40  DBMS_HPROF.STOP_PROFILING;
41
42  -- Analyze profiler output
43  v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
44  DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID -
' || v_run_id);
45 END;

```

In this example, there are two functions, **f1** and **f2**. These functions are called by the procedure **p1**, which in turn is called inside the numeric **FOR LOOP**. Note that subprogram inlining has been explicitly enabled for each procedure call. This example produces the following output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is enabled, run ID - 5
Elapsed: 00:00:01.179

```

The PL/SQL Hierarchical Profiler reports these runtime statistics:

[Click here to view code image](#)

```

SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 5;

RUNID   FUNCTION          LINE#    CALLS    S_E_T    F_E_T
--      -----    --  --  --  -----
5       __anonymous_block.F1      4      100000  20595  20595
5       __anonymous_block.F2     12      100000  42010  42010
5       STOP_PROFILING        63        1        0        0

```

To enable subprogram inlining for functions **f1** and **f2**, **PLSQL_OPTIMIZE_LEVEL** should be set to 3 or a **PRAGMA INLINE** statement should be executed prior to the functions' invocations. When the optimization level is set to 3, the script produces this output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is enabled, run ID - 6
Elapsed: 00:00:00.042

```

The PL/SQL Hierarchical Profiler reports these statistics:

[Click here to view code image](#)

```

SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 6;

RUNID   FUNCTION          LINE#    CALLS    S_E_T    F_E_T
--      -----    --  --  --  -----
6       STOP_PROFILING        63        1        0        0

```

When PL/SQL code contains embedded SQL statements, performance gains due to subprogram inlining become negligible. This is because oftentimes SQL statements are the main consumers of the execution time. Consider a modified version of the preceding example, where `SELECT INTO` statements have been added to the functions `f1` and `f2`. All changes are shown in bold.

For Example *ch25_6b.sql*

[Click here to view code image](#)

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;

1  DECLARE
2      v_run_id BINARY_INTEGER; – run ID generated by the profiler
3
4      FUNCTION f1 (num1 IN PLS_INTEGER
5                      ,num2 IN PLS_INTEGER)
6      RETURN PLS_INTEGER
7      IS
8          v_num PLS_INTEGER;
9      BEGIN
10         SELECT num1 + num2
11             INTO v_num
12             FROM dual;
13         RETURN v_num;
14     END f1;
15
16     FUNCTION f2 (str1 IN VARCHAR2
17                      ,str2 IN VARCHAR2)
18     RETURN VARCHAR2
19     IS
20         v_srt VARCHAR2(50);
21     BEGIN
22         SELECT str1||' '||str2
23             INTO v_srt
24             FROM dual;
25         RETURN (v_srt);
26     END f2;
27
28     PROCEDURE p1 (num1 IN PLS_INTEGER
29                      ,num2 IN PLS_INTEGER
30                      ,str1 IN VARCHAR2
31                      ,str2 IN VARCHAR2)
32     IS
33         v_num NUMBER;
34         v_str VARCHAR2(100);
35     BEGIN
36         v_num := f1(num1, num2);
37         v_str := f2(str1, str2);
38     END p1;
39
40 BEGIN
41     DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
42     FOR i in 1..100000
43     LOOP
44         p1 (i-1, i, to_char(i-1), to_char(i));
45     END LOOP;
46     DBMS_HPROF.STOP_PROFILING;
```

```

47
48  -- Analyze profiler output
49  v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
50  DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID -
' ||v_run_id);
51 END;

```

When run, the script produces the following output:

[Click here to view code image](#)

```

session SET altered.
Elapsed: 00:00:00.001
Inline pragma is enabled, run ID - 7
Elapsed: 00:00:06.156

```

Note how the elapsed time has increased from .042 to 6.156, even though the optimization level was set to 3 for both runs. The PL/SQL Hierarchical Profiler reports these statistics for run ID 7:

[Click here to view code image](#)

```

SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 7;

  RUNID   FUNCTION          LINE#    CALLS     S_E_T     F_E_T
  ---  -----  --  --  ---  -----
  7      __static_sql_exec_line    44  200000  4382821  4382821
  7      STOP_PROFILING        63      1        0        0

```

The PL/SQL optimizer was able to apply subprogram inlining, but performance was not gained due to the SQL statement that was executed on line 44. As expected, each SQL statement was executed 100,000 times, thereby increasing the overall execution time significantly. The results of this run also highlight that in this particular case, relocating the `SELECT INTO` statements from functions `f1` and `f2` inside the body of the loop would not have improved performance.

Summary

In this chapter, you learned about PL/SQL optimization levels and saw how the PL/SQL compiler optimizes code based on these levels. In addition, you explored the concept of subprogram inlining and considered in which instances it may be used to improve performance of PL/SQL code. Finally, you learned how to gather and interpret PL/SQL runtime statistics with the PL/SQL Profiler API and PL/SQL Hierarchical Profiler. Combining these optimization techniques with the optimization methods such as bulk SQL allows you to create robust and performant code.

A. PL/SQL Formatting Guide

This appendix summarizes some of the PL/SQL formatting guidelines used throughout this book. While formatting guidelines are not a required part of PL/SQL, they act as best practices that facilitate development of better-quality code, greater readability, and easier maintenance.

Case

PL/SQL, like SQL, is case insensitive. The general guidelines in regard to case are as follows:

- Use uppercase for keywords (e.g., BEGIN, EXCEPTION, END, IF-THEN-ELSE, LOOP, END LOOP), data types (e.g., VARCHAR2, NUMBER), built-in functions (e.g., LEAST, SUBSTR), and user-defined subroutines (e.g., procedures, functions, packages).
- Use lowercase for variable names as well as column and table names in SQL.

White Space

White space (extra lines and spaces) is as important in PL/SQL as it is in SQL. It is a major factor in improving readability. In other words, you can reveal the logical structure of the program by using appropriate indentation in your code. Here are some suggestions:

- Put spaces on both sides of an equality sign or comparison operator.
- Line up structure words on the left (e.g., DECLARE, BEGIN, EXCEPTION, and END; IF and END IF; LOOP and END LOOP). In addition, indent three spaces (use the spacebar, not the tab key) for structures within structures.
- Put blank lines between major sections to separate them from each other.
- Put different logical parts of the same structure on separate lines even if the structure is short. For example, IF and THEN are placed on one line, while ELSE and END IF are placed on separate lines.

Naming Conventions

To ensure against conflicts with keywords and column/table names, it is helpful to use the following prefixes:

- v_variable_name
- con_constant_name
- i_in_parameter_name, o_out_parameter_name,
io_in_out_parameter_name
- c_cursor_name or name_cur

- rc_reference_cursor_name
- r_record_name or name_rec
- FOR r_stud IN c_stud LOOP...
- FOR stud_rec IN stud_cur LOOP
- type_name or name_type (for user-defined types)
- t_table or name_tab (for PL/SQL tables)
- rec_record_name or name_rec (for record variables)
- e_exception_name (for user-defined exceptions)

The name of a package should be the name of the larger context of the actions performed by the procedures and functions contained within the package.

The name of a procedure should be the action description that is performed by the procedure. The name of a function should be the description of the return variable.

For Example

[Click here to view code image](#)

```
PACKAGE student_admin
  -- admin suffix may be used for administration.

  PROCEDURE remove_student (i_student_id IN student.studid%TYPE);

  FUNCTION student_enroll_count (i_student_id student.studid%TYPE)
  RETURN INTEGER;
```

Comments

Comments in PL/SQL are as important as they are in SQL. They should explain the main sections of the program and any major nontrivial logic steps.

Use single-line comments “--” instead of the multiline “/*” comments. While PL/SQL treats these comments in the same way, it will be easier for you to debug the code once it is complete because you cannot embed multiline comments within multiline comments. In other words, you can comment out portions of code that contain single-line comments, but you cannot comment out portions of code that contain multiline comments.

Other Suggestions

Here are a few additional small recommendations to assist you in making sure your PL/SQL code is neat and easy to follow.

- For SQL statements embedded in PL/SQL, use the same formatting guidelines to determine how the statements should appear in a block.
- Provide a comment header that explains the intent of the block and lists the creation date and the author’s name. Also include a line for each revision with the author’s name, date, and the description of the revision.

The following example shows the aforementioned suggestions. Notice that it also uses a monospaced font (Courier New) that makes the formatting easier. Proportionally spaced fonts can hide spaces and make lining up clauses difficult. Most text and programming editors by default use a monospaced font.

For Example

[Click here to view code image](#)

```
REM ****
REM * filename: coursediscount01.sql      version: 1
REM * purpose: To give discounts to courses that have at
REM *           least one section with an enrollment of more
REM *           than 10 students.
REM * args:    none
REM *
REM * created by: s.tashi      date: January 1, 2000
REM * modified by: y.sonam      date: February 1, 2000
REM * description: Fixed cursor, added indentation and
REM *               comments.
REM ****
DECLARE
  -- C_DISCOUNT_COURSE finds a list of courses that have
  -- at least one section with an enrollment of at least 10
  -- students.
  CURSOR c_discount_course IS
    SELECT DISTINCT course_no
      FROM section sect
      WHERE 10 <= (SELECT COUNT(*)
                     FROM enrollment enr
                     WHERE enr.section_id = sect.section_id
                   );
  -- discount rate for courses that cost more than $2000.00
  con_discount_2000 CONSTANT NUMBER := .90;

  -- discount rate for courses that cost between $1001.00
  -- and $2000.00
  con_discount_other CONSTANT NUMBER := .95;

  v_current_course_cost course.cost%TYPE;
  v_discount_all NUMBER;
  e_update_is_problematic EXCEPTION;
BEGIN
  -- For courses to be discounted, determine the current
  -- and new cost values
  FOR r_discount_course IN c_discount_course LOOP
    SELECT cost
      INTO v_current_course_cost
      FROM course
      WHERE course_no = r_discount_course.course_no;

    IF v_current_course_cost > 2000 THEN
      v_discount_all := con_discount_2000;
    ELSE
      IF v_current_course_cost > 1000 THEN
        v_discount_all := con_discount_other;
      ELSE
        v_discount_all := 1;
      END IF;
    END IF;
    BEGIN
      UPDATE course
        SET cost = v_discount_all * course.cost
      WHERE course_no = r_discount_course.course_no;
    EXCEPTION WHEN e_update_is_problematic THEN
      -- handle exception
    END;
  END LOOP;
END;
```

```
        END IF;
    END IF;

    BEGIN
        UPDATE course
            SET cost = cost * v_discount_all
            WHERE course_no = r_discount_course.course_no;
    EXCEPTION
        WHEN OTHERS THEN
            RAISE e_update_is_problematic;
    END;      -- end of sub-block to update record
END LOOP; -- end of main LOOP

COMMIT;
EXCEPTION
    WHEN e_update_is_problematic THEN
        -- Undo all transactions in this run of the program
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE
            ('There was a problem updating a course cost.');
    WHEN OTHERS THEN
        NULL;
END;
/
```

B. Student Database Schema

Table and Column Descriptions

This appendix lists all the tables from the STUDENT database schema that are used in the book. Each table is listed here with the database table name first; what follows are the columns in that table, an indicator of whether a null value is allowed, the column data type, and a description. The scripts to install this database can be found on the companion website for this book.

Column Name	Null	Type	Comments
COURSE_NO	NOT NULL	NUMBER (8, 0)	The unique course number
DESCRIPTION	NULL	VARCHAR2 (50)	The full name for this course
COST	NULL	NUMBER (9, 2)	The dollar amount charged for enrollment in this course
PREREQUISITE	NULL	NUMBER (8, 0)	The ID number of the course that must be taken as a prerequisite to this course
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—Indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

COURSE: Information for a Course

Column Name	Null	Type	Comments
SECTION_ID	NOT NULL	NUMBER(8, 0)	The unique ID for a section
COURSE_NO	NOT NULL	NUMBER(8, 0)	The course number for which this is a section
SECTION_NO	NOT NULL	NUMBER(3)	The individual section number within this course
START_DATE_TIME	NULL	DATE	The date and time at which this section meets
LOCATION	NULL	VARCHAR2(50)	The meeting room for the section
INSTRUCTOR_ID	NOT NULL	NUMBER(8, 0)	The ID number of the instructor who teaches this section
CAPACITY	NULL	NUMBER(3, 0)	The maximum number of students allowed in this section
CREATED_BY	NOT NULL	VARCHAR2(30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2(30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

SECTION: Information for an Individual Section (Class) of a Particular Course

Column Name	Null	Type	Comments
STUDENT_ID	NOT NULL	NUMBER(8, 0)	The unique ID for a student
SALUTATION	NULL	VARCHAR2(5)	This student's title (e.g., Ms., Mr., Dr.)
FIRST_NAME	NULL	VARCHAR2(25)	This student's first name
LAST_NAME	NOT NULL	VARCHAR2(25)	This student's last name
STREET_ADDRESS	NULL	VARCHAR2(50)	This student's street address
ZIP	NOT NULL	VARCHAR2(5)	The postal ZIP code for this student
PHONE	NULL	VARCHAR2(15)	The phone number for this student, including area code
EMPLOYER	NULL	VARCHAR2(50)	The name of the company where this student is employed
REGISTRATION_DATE	NOT NULL	DATE	The date this student registered in the program
CREATED_BY	NOT NULL	VARCHAR2(30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2(30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

STUDENT: Profile Information for a Student

Column Name	Null	Type	Comments
STUDENT_ID	NOT NULL	NUMBER (8, 0)	The ID for a student
SECTION_ID	NOT NULL	NUMBER (8, 0)	The ID for a section
ENROLL_DATE	NOT NULL	DATE	The date this student registered for this section
FINAL_GRADE	NULL	NUMBER (3, 0)	The final grade given to this student for all work in this section (class)
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

ENROLLMENT: Information for a Student Registered for a Particular Section of a Particular Course (Class)

Column Name	Null	Type	Comments
INSTRUCTOR_ID	NOT NULL	NUMBER (8)	The unique ID for an instructor
SALUTATION	NULL	VARCHAR2 (5)	This instructor's title (e.g., Mr., Ms., Dr., Rev.)
FIRST_NAME	NULL	VARCHAR2 (25)	This instructor's first name
LAST_NAME	NULL	VARCHAR2 (25)	This instructor's last name
STREET_ADDRESS	NULL	VARCHAR2 (50)	This instructor's street address
ZIP	NULL	VARCHAR2 (5)	The postal ZIP code for this instructor
PHONE	NULL	VARCHAR2 (15)	The phone number for this instructor, including area code
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

INSTRUCTOR: Profile Information for an Instructor

Column Name	Null	Type	Comments
ZIP	NOT NULL	VARCHAR2 (5)	The ZIP code number, unique for a city and state
CITY	NULL	VARCHAR2 (25)	The city name for this ZIP code
STATE	NULL	VARCHAR2 (2)	The postal abbreviation for the U.S. state
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

ZIPCODE: City, State, and ZIP Code Information

Column Name	Null	Type	Comments
GRADE_TYPE_CODE	NOT NULL	CHAR (2)	The unique code that identifies a category of grade (e.g., MT, HW)
DESCRIPTION	NOT NULL	VARCHAR2 (50)	The description for this code (e.g., Midterm, Homework)
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

GRADE_TYPE: Lookup Table of a Grade Type (Code) and its Description

Column Name	Null	Type	Comments
SECTION_ID	NOT NULL	NUMBER (8)	The ID for a section
GRADE_TYPE_CODE	NOT NULL	CHAR (2)	The code that identifies a category of grade
NUMBER_PER_SECTION	NOT NULL	NUMBER (3)	How many of these grade types can be used in this section (i.e., there may be three quizzes)
PERCENT_OF_FINAL_GRADE	NOT NULL	NUMBER (3)	The percentage this category of grade contributes to the final grade
DROP_LOWEST	NOT NULL	CHAR (1)	Is the lowest grade in this type removed when determining the final grade? (Y/N)
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

GRADE_TYPE_WEIGHT: Information on How the Final Grade for a Particular Section is Computed; For Example, the Midterm Constitutes 50%, the Quiz 10%, and the Final Examination 40% of the Final Grade

Column Name	Null	Type	Comments
STUDENT_ID	NOT NULL	NUMBER (8)	The ID for a student
SECTION_ID	NOT NULL	NUMBER (8)	The ID for a section
GRADE_TYPE_CODE	NOT NULL	CHAR (2)	The code that identifies a category of grade
GRADE_CODE_OCCURRENCE	NOT NULL	NUMBER (38)	The sequence number of one grade type for one section. For example, there could be multiple assignments numbered 1, 2, 3, etc.
NUMERIC_GRADE	NOT NULL	NUMBER (3)	Numeric grade value (e.g., 70, 75)
COMMENTS	NULL	VARCHAR2 (2000)	Instructor's comments on this grade
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

GRADE: The Individual Grades a Student Received for a Particular Section (Class)

Column Name	Null	Type	Comments
LETTER_GRADE	NOT NULL	VARCHAR (2)	The unique grade as a letter (A, A–, B, B+, etc.)
GRADE_POINT	NOT NULL	NUMBER (3, 2)	The number grade on a scale from 0 (F) to 4 (A)
MAX_GRADE	NOT NULL	NUMBER (3)	The highest grade number that corresponds to this letter grade
MIN_GRADE	NOT NULL	NUMBER (3)	The lowest grade number that corresponds to this letter grade
CREATED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates the user who inserted data
CREATED_DATE	NOT NULL	DATE	Audit column—indicates the date of data insertion
MODIFIED_BY	NOT NULL	VARCHAR2 (30)	Audit column—indicates who made the last update
MODIFIED_DATE	NOT NULL	DATE	Audit column—date of last update

GRADE_CONVERSION: Converts a Number Grade to a Letter Grade

The entity–relationship diagram for the tables in the student schema shown in [Figure B–1](#) shows the tables and their foreign key relationships using the standard crow's feet arrows.

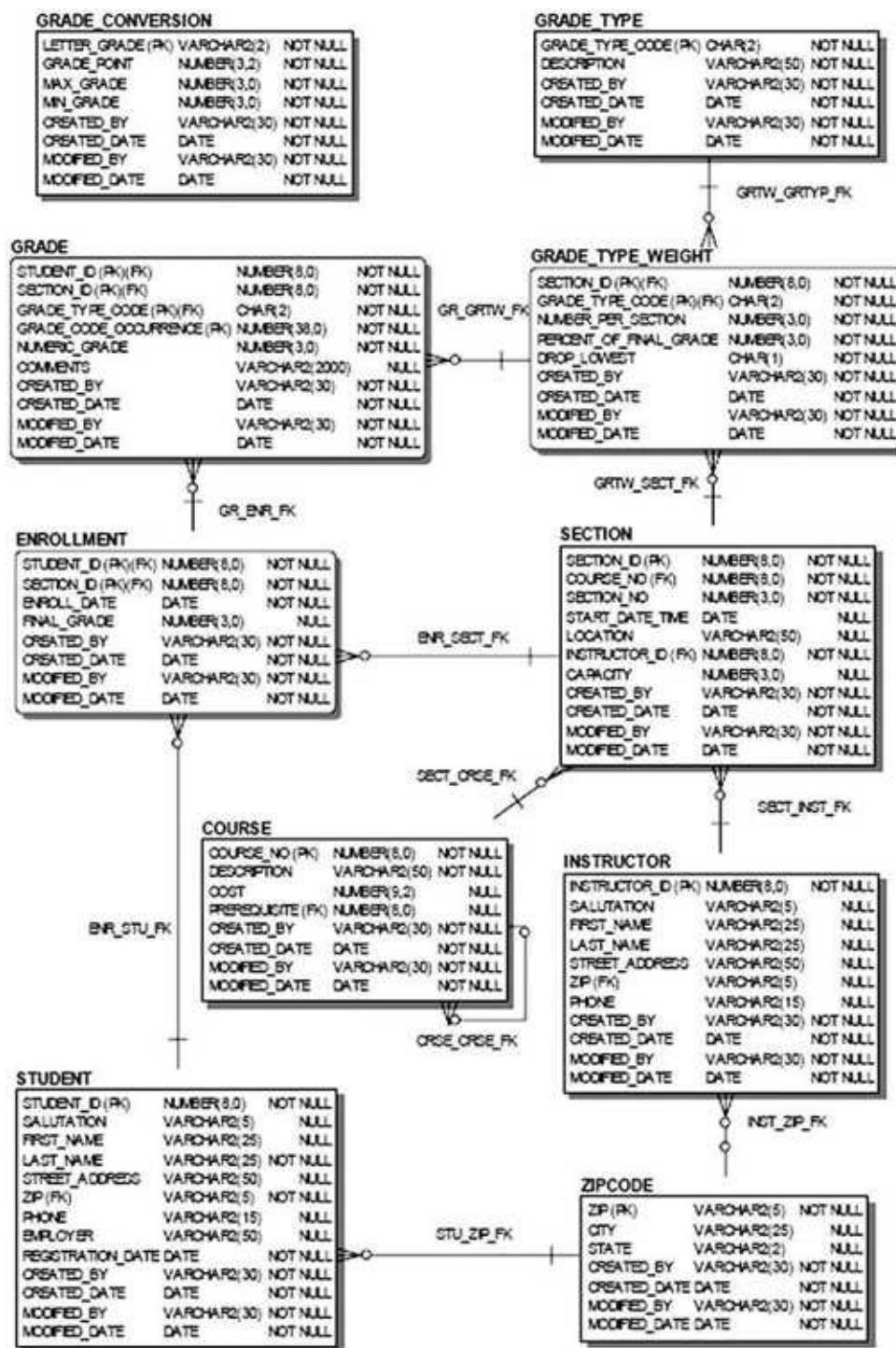


Figure B-1

Index

() (parentheses)

controlling order of operations, [38](#)

grouping for readability, [69](#), [252](#)

& (ampersand)

in substitution variable names, [20](#), [22](#), [25](#)

in variable names, [31](#)

:

(colon), in bind arguments, [260](#)

— (dashes), single-line comments, [29](#), [40](#)

/ (slash), block terminator, [16](#), [264](#)

:= (colon, equal sign), assignment operator, [37](#)

‘ ‘ (single quotes), enclosing substitution variables, [25](#)

/*...*/ (slash asterisk...), multiline comments, [29](#), [40](#)

&& (double ampersand), in substitution variable names, [20](#), [24](#), [25](#)

;(semicolon)

block terminator, [16–17](#)

SQL and PL/SQL statement terminator, [264–265](#)

variable terminator, [36–37](#)

A

ACCESSIBLE BY clause, [xxvii–xxviii](#)

Accessors

new for Oracle 12c, [xxvii–xxviii](#)

specifying, [xxvii–xxviii](#)

white lists, [xxvii–xxviii](#)

Actual parameters, [317–318](#)

AFTER triggers, [201–204](#)

ALL_DEPENDENCIES view, [376–377](#)

ALL_OBJECTS view, [374](#)

ALL_USER_OBJECTS view, [314–315](#)

ALL_USER_SOURCE view, [314–315](#)

ALTER SYSTEM command, [411](#)

ALTER TRIGGER command, [194](#)

Ampersand (&)

in substitution variable names, [20](#), [22](#), [25](#)

in variable names, [31](#)

ANALYZE routine, [437](#)

Anchored data types, [34](#)

Anonymous blocks. *See also* [Modular code](#); [Named blocks](#).

definition, [5](#)

description, [312](#)

executing, [8](#)

Application exception, profiling, [436](#)–[437](#)

Application processing tier, [3](#)

Architecture. *See also* [Blocks](#).

application processing tier, [3](#)

client-server, [5](#)

data management tier, [3](#)

Oracle server, [2](#)–[4](#)

overview, [2](#)–[5](#)

presentation tier, [3](#)

three-tier, [3](#)

Arithmetic operators, [38](#)

Arrays. *See* [Associative arrays](#); [Varrays](#).

Associative arrays

declaring, [227](#)

EXTEND method, [233](#)

LIMIT method, [238](#)

vs. nested tables and varrays, [239](#)–[240](#)

NO_DATA_FOUND exception, [228](#)–[229](#)

of objects, populating with data, [392](#)

populating, [227](#)

referencing individual elements, [227](#)–[228](#)

syntax, [226](#)
TRIM method, [233](#)
upper bounds, specifying, [238–239](#)
Attributes (data), object types, [386](#)
Autonomous transactions, triggers, [203–204](#)
AUTONOMOUS_TRANSACTION pragma, [204](#)

B

BACKTRACE_DEPTH function, [424](#), [426–427](#)

BACKTRACE_LINE function, [424](#), [426–427](#)

BACKTRACE_UNIT function, [424](#), [426–427](#)

Batch processing. See [Bulk SQL](#).

BEFORE triggers, [195–201](#)

BEGIN keyword, [7](#)

BEQUEATH CURRENT_USER clause, [xxxii](#)

BEQUEATH DEFINER clause, [xxxii](#)

Bind arguments

in CREATE TABLE statements, [263–264](#)

definition, [260](#)

passing run-time values to, [272](#)

Binding, definition, [9](#)

Binding collections with

CLOSE statements, [306–309](#)

EXECUTE IMMEDIATE statements, [299–305](#)

FETCH statements, [306–309](#)

OPEN-FOR statements, [306–309](#)

Blank lines, inserting in output, [242](#)

Blocks

; (semicolon), block terminator, [16](#)

anonymous, [5](#), [8](#)

binding, [9](#)

compilation errors, [7–8](#)

creating subroutines, [5](#)

declaration section, [6](#)
definition, [5](#)
displaying variable values. See [DBMS_OUTPUT.PUT_LINE statements](#).
error types, [7–8](#)
exception-handling section, [7–8](#)
executable section, [6–7](#)
executing, [8–9](#)
named, [5, 8–9](#)
nested, [5, 39–41](#)
runtime errors, [7–8](#)
sections, [6–8](#)
semantic checking, [9](#)
sequences in, [48–49](#)
syntax checking, [8–9](#)
terminating, [16, 264–265](#)
vs. transactions, [50, 54–55](#)
VALID vs. INVALID, [9](#)

Books and publications

Database Object-Relational Developer's Guide, [385](#)
Oracle Forms Developer: The Complete Video Course, [xxiii](#)
Oracle PL/SQL by Example, Fifth Edition, [xvii](#)
Oracle SQL by Example, [414](#)
Oracle Web Application Programming for PL/SQL Developers, [xxiii](#)

Boolean expressions, in WHILE loops, [101](#)

BROKEN procedure, [410](#)

Built-in exceptions, [126–132](#)

BULK COLLECT clause, [291–299](#)

BULK COLLECT INTO clause, [xxix](#)

BULK EXECUTE IMMEDIATE statements, [260](#)

BULK FETCH statements, [260](#)

Bulk SQL

BULK COLLECT clause, [291–299](#)

DELETE statements, in batches. *See* [FORALL statements](#).

fetching results, [291–299](#)

INSERT statements, in batches. *See* [FORALL statements](#).

limiting result sets, [292–293](#)

NO_DATA_FOUND exception, [292](#)

UPDATE statements, in batches. *See* [FORALL statements](#).

Bulk SQL, FORALL statements

description, [282–285](#)

error messages, displaying, [287–288](#)

exception handling, [285–288](#)

implicit loop counter, [283](#)

INDICES OF option, [283, 288](#)

looping, [283, 288–290](#)

SAVE EXCEPTIONS option, [285–288](#)

SQL%BULK_EXCEPTIONS attribute, [286–287](#)

VALUES OF option, [289–290](#)

C

Calling packages, [339–341](#)

CASE abbreviations. *See* [COALESCE function](#); [NULLIF function](#).

CASE expressions, [80–84](#)

Case sensitivity

formatting guide, [455](#)

passwords, [10](#)

PL/SQL, [29](#)

variables, [29](#)

CASE statements

Boolean results. *See* [Searched CASE statements](#).

vs. CASE expressions, [81–84](#)

description, [72–74](#)

searched CASE statements, [74–80](#)

CHANGE procedure, [410, 412](#)

CHAR data type, [35](#)

Character types, [28](#)

CLEAR_PLSQL_TRACE routine, [434–436](#)

Client-server architecture, [5](#)

CLOSE statements

binding collections with, [306–309](#)

closing cursors, [271–280](#)

Closing

cursor variables, [349](#)

cursors, [167–168, 170](#)

dynamic SQL cursors, [271–280](#)

explicit cursors, [162, 167–168, 172–173](#)

files, [407](#)

COALESCE function, [87–89](#). *See also* [NULLIF function](#).

Code generation, [9](#)

COLLECT INTO statements, [260](#)

Collection methods, [232–235](#)

Collections. *See also* [Tables](#).

counting elements, [232–235](#)

defined on user-defined records, [255–256](#)

definition, [225](#)

deleting elements, [233–235](#)

extending, [231](#)

multilevel, [240–242](#)

in nested records, [252–253](#)

NULL vs. empty, [232](#)

of object types, [391–394](#)

records, [253–256](#)

testing for elements, [232–235](#)

upper bounds, specifying, [238–239](#)

variable-size arrays. *See* [Varrays](#).

Collections, binding with

CLOSE statements, [306–309](#)

EXECUTE IMMEDIATE statements, [299–305](#)

FETCH statements, [306–309](#)

OPEN-FOR statements, [306–309](#)

Colon, equal sign (:=), assignment operator, [37](#)

Colon (:), in bind arguments, [260](#)

Columns

aliases, [175](#)

invisible, [xxxiii–xxxiv](#)

in a table, describing, [377–378](#)

Comments

formatting, [29, 456–459](#)

single-line vs. multiline, [29](#)

COMMIT statements

description, [49–52](#)

placing, [188, 314](#)

in triggers, [195](#)

Companion Website, URL for, [xviii](#)

Comparing objects

map methods, [400–401](#)

order methods, [401–404](#)

overview, [399–400](#)

Comparison operators, [38](#)

Compatibility, record types, [249–250](#)

Compilation errors, [7–8, 124–126](#)

Complex functions, creating, [328–329](#)

Complex nested cursors, [185–187](#)

Compound triggers

definition, [218](#)

firing order, [219](#)

resolving mutating table issues, [220–223](#)

restrictions, [219](#)

structure, [218](#)

Conditional control. See [CASE statements](#); [ELSIF statements](#); [IF statements](#).

Connecting to a database

SQL Developer, [10–11](#)

SQL*Plus, [13](#)

Connection name, SQL Developer, [10](#)

Constructor methods, [395–397](#)

Contiguous numbers, generating, [48](#)

CONTINUE statements, [111–115](#)

CONTINUE WHEN statements, [115–118](#)

COUNT method, [232–235](#)

Counting collection elements, [232–235](#)

CREATE reserved word, [192–193](#)

CREATE TABLE statements, [263–264](#)

CREATE TYPE statements, [229–230](#)

Creating

cursor variables, [345–346](#), [349–350](#)

error messages, [149–153](#)

event triggers on PDBs, [xxx](#)

nested tables, [229–230](#)

object types, [386–390](#)

procedures, [312–315](#)

triggers, [192–195](#), [197–201](#)

Creating functions

complex functions, [328–329](#)

stored functions, [322–325](#)

using a WITH clause, [329–330](#)

using the UDF pragma, [330–331](#)

Creating packages

information hiding, [335](#)

package body, [335–336](#), [337–339](#)

package specification, [335](#)

package variables, [367–368](#)

private elements, [341–344](#)

Creating user-defined functions with a

WITH clause, [xxxiv](#)

UDF pragma, [xxxiv–xxxv](#)

CREDENTIAL clause, [xxx–xxxi](#)

Currency conversion example, [334](#)

CURRVAL pseudocolumn, [48](#)

Cursor attributes, [170–174](#). *See also specific attributes.*

Cursor FOR loops, [175–177](#)

Cursor loops

closing a cursor, [167–168](#), [170](#)

explicit cursors, [165–168](#)

fetching rows in a cursor, [166–167](#)

opening a cursor, [165–166](#)

Cursor variables

closing, [349](#)

creating, [345–346](#), [349–350](#)

vs. cursors, [346](#)

definition, [345](#)

explicit, [345](#)

in packages, [347–348](#), [350–352](#)

processing, [346–347](#)

query results, printing automatically, [348](#)

rules for using, [353](#)

sharing result sets, [348–352](#)

strong (restrictive), [345–346](#)

weak (nonrestrictive), [345–346](#)

Cursor-based records

compatibility, [249–250](#)

creating, [163–165](#)

defining a collection on, [253–255](#)

definition, [163](#)

description, [244–246](#)

Cursors. *See also* [Dynamic SQL cursors](#).

column aliases, [175](#)

vs. cursor variables, [346](#)

definition, [159](#)

explicit, [160](#)

expressions in a select list, [175](#)

fetch status, getting, [170–174](#)

implicit, [160–161](#)

locking rows for update, [187–189](#)

most recently opened, [160](#)

number of records fetched, getting, [170–174](#)

number of rows updated, getting, [161](#)

open, detecting, [170–174](#)

parameterized, [183–185](#)

scope, [175](#)

select list, [175](#)

SQL, [160](#)

tips for using, [175](#)

types of, [159–165](#)

FOR UPDATE clause, [187–189](#)

FOR UPDATE OF clause, [189](#)

updating tables in a database, [187–190](#)

WHERE CURRENT OF clause, [189–190](#)

Cursors, explicit

associating with SELECT statements, [162](#)

closing, [162, 167–168, 172–173](#)

cursor-based records, [163–165](#)

declaring, [162–163, 172–173](#)

definition, [160](#)

fetching rows in a cursor, [162, 166–167, 170–174](#)

naming conventions, [162–163](#)

opening, [162](#), [165–166](#), [172–173](#)

processing, [165–168](#)

record types, [163–165](#)

records, [163–165](#)

table-based records, [163](#)

user-defined records, [168–170](#)

Cursors, nested

complex, [185–187](#)

looping through data, [177–181](#), [185–187](#)

processing, [177–181](#)

D

Dashes (—), single-line comments, [29](#), [40](#)

Data (attributes), object types, [386](#)

Data dictionary, examining stored code

ALL_DEPENDENCIES view, [376–377](#)

ALL_OBJECTS view, [374](#)

DBA_DEPENDENCIES view, [376–377](#)

DBA_OBJECTS view, [374](#)

debugging, [376](#)

dependencies, displaying, [376–377](#)

DESC command, [377–378](#)

describing columns in a table, [377–378](#)

displaying errors, [375–376](#)

identifying procedures, packages, and functions, [377–378](#)

modules with duplicate names. *See* [Overloading](#).

overloading modules, [378–382](#)

retrieving specified line numbers, [374–375](#)

SHO ERR command, [376](#)

USER_DEPENDENCIES view, [376–377](#)

USER_ERRORS view, [375–376](#)

USER_OBJECTS view, [374](#)

Data dictionary queries

ALL_USER_OBJECTS view, [314–315](#)

ALL_USER_SOURCE view, [314–315](#)

DBA_USER_OBJECTS view, [314–315](#)

DBA_USER_SOURCE view, [314–315](#)

displaying source code, [314–315](#)

object information, [314–315](#)

procedure information, [314–315](#)

USER_OBJECTS view, [314–315](#)

USER_SOURCE view, [314–315](#)

Data management tier, [3](#)

Data Manipulation Language (DML)

definition, [46](#)

and transaction control, [53–55](#)

Data types

based on database objects. *See Anchored data types.*

common, summary of, [35–36](#). *See also specific types.*

displaying maximum size, [xxx](#)

extended maximum size, [xxx](#)

for file handles, [407](#)

new for Oracle 12c, [xxx](#)

passing to procedures, [318](#)

Database Object-Relational Developer's Guide, [385](#)

Database triggers. *See Triggers.*

Databases

edition-based redefinition, [193](#)

erasing changes. *See ROLLBACK statements.*

saving changes. *See COMMIT statements.*

setting a save point. *See SAVEPOINT statements.*

STUDENT schema, [461–468](#)

used in this book, [461–468](#)

DATE data type, [36](#)

DBA_DEPENDENCIES view, [376–377](#)

DBA_OBJECTS view, [374](#)

DBA_USER_OBJECTS view, [314–315](#)

DBA_USER_SOURCE view, [314–315](#)

DBMS_HPROF package, [436–437](#)

DBMSHPTAB.sql script, [437](#)

DBMS_JOB package, [410–412](#)

DBMS_OUTPUT.PUT_LINE statements, [18–19](#), [21](#)

DBMS_PROFILER package, [432–433](#)

DBMS_SQL package, [417–418](#)

DBMS_TRACE package, [433–436](#)

DBMS.Utility package, [419–424](#)

DBMS_XPLAN package, [414–417](#)

Debugging

new for Oracle 12c, [xxxvii](#)

stored code, [376](#)

Declaration section, [6](#)

DECLARE keyword, [6](#)

Declaring

associative arrays, [227](#)

explicit cursors, [162–163](#), [172–173](#)

variables, [36–39](#)

varrays, [236–238](#)

exceptions, [137–141](#)

Definer rights (DR) subprogram, [xxvi–xxvii](#)

DELETE method

deleting collection elements, [233–235](#)

deleting varray elements, [239](#)

DELETE statements. *See also* [DML \(Data Manipulation Language\)](#).

batch processing. *See* [FORALL statements](#).

with BULK COLLECT clause, [295](#)

Deleting

collection elements, [233–235](#)

statements, [295](#)
varray elements, [239](#)

Delimiters, [29](#)

Dependencies, displaying, [376–377](#)

DESC command, [377–378](#)

Development environment. *See* [PL/SQL Scripts](#); [SQL Developer](#); [SQL*Plus](#).

DIRECTORY objects, defining LIBRARY objects as, [xxx–xxxi](#)

DISABLE option, [194](#)

Disabling substitution variable verification, [23](#)

Disconnecting from a database

- SQL Developer, [11–12](#)
- SQL*Plus, [13](#)

Displaying

- code dependencies, [376–377](#)
- code errors, [375–376](#)
- data type maximum size, [xxx](#)
- data type size, [xxx](#)
- error messages, [287–288](#)
- errors, [375–376](#)
- invalid procedures, [315](#)
- passwords, [13](#)
- procedures, [314–315](#)
- source code, [314–315](#)
- stored code dependencies, [376–377](#)
- variable values. *See* [DBMS_OUTPUT.PUT_LINE statements](#).

DML (Data Manipulation Language)

- definition, [46](#)
- and transaction control, [53–55](#)

DML statements. *See also* [DELETE statements](#); [INSERT statements](#); [UPDATE statements](#).

- in blocks, [47–49](#)
- as triggering events, [47–49](#)

Double ampersand (`&&`), in substitution variable names, [20](#), [24](#), [25](#)

DR (definer rights) subprogram, [xxvi–xxvii](#)

Duplicate names. *See* [Overloading](#).

DUP_VALUE_ON_INDEX exception, [129](#)

Dynamic SELECT statements, [259](#)

Dynamic SQL, optimizing, [260](#)

Dynamic SQL cursors. *See also* [Cursors](#).

closing, [271–280](#)

fetching from, [271–280](#)

opening, [271–280](#)

passing run-time values to bind arguments, [272](#)

Dynamic SQL statements

CLOSE, [271–280](#)

example, [260](#)

FETCH, [271–280](#)

multirow queries, [271–280](#)

OPEN-FOR, [271–280](#)

passing NULLS to, [265–266](#)

single-row queries, [261–271](#)

terminating, [264](#)

Dynamic SQL statements, EXECUTE IMMEDIATE

avoiding ORA errors, [262–271](#)

binding collections, [299–305](#)

description, [260–261](#)

RETURNING INTO clause, [261–262](#)

USING clause, [261–262](#)

DYNAMIC_DEPTH function, [424–426](#)

E

EDITIONABLE property, [xxxiv](#), [193](#)

Edition-based redefinition, [193](#)

ELSIF statements, [63–67](#). *See also* [IF statements](#).

Empty vs. NULL, [232](#)

ENABLE option, [194](#)

Encapsulation, [386](#)

Erasing database changes. *See also* [ROLLBACK statements](#).

Error handling. *See also* [Error messages](#).

compilation errors, [7–8](#), [124–126](#)

runtime errors, [7–8](#), [124–126](#), [141–147](#). *See also* [Exception propagation](#); [Exceptions](#).

Error isolation, SQL*Plus, [314](#)

Error messages. *See also* [Error handling](#).

creating, [149–153](#)

displaying, [287–288](#)

getting, [155–158](#), [424](#), [428–429](#)

names, associating with numbers, [153–155](#)

references to line numbers and keywords, [126](#)

Error numbers, getting, [155–158](#), [424](#), [428–429](#)

Error reporting

DBMS_UTILITY package, [419–424](#)

UTL_CALL_STACK package, [424–429](#)

Error types, [7–8](#)

ERROR_DEPTH function, [424](#), [428–429](#)

error_message parameter, [150](#)

ERROR_MSG function, [424](#), [428–429](#)

ERROR_NUMBER function, [424](#), [428–429](#)

error_number parameter, [150](#)

Errors, displaying, [375–376](#)

Event triggers, creating on PDBs, [xxx](#)

Exception handling. *See also* [User-defined exceptions](#).

built-in, [126–132](#)

EXCEPTION keyword, [8](#)

EXCEPTION_INIT pragma, [153–155](#)

file location not valid, [408](#)

filename not valid, [408](#)

FORALL statements, [285–288](#)

INTERNAL_ERROR, [408](#)

invalid file handle, [408](#)
invalid mode, [408](#)
invalid operation, [408](#)
INVALID_FILEHANDLE, [408](#)
INVALID_MODE, [408](#)
INVALID_OPERATION, [408](#)
INVALID_PATH, [408](#)
predefined, [128–129](#). See also [OTHERS exception](#); specific exceptions.
raising implicitly, [127](#)
read error, [408](#)
READ_ERROR, [408](#)
re-raising, [146–148](#)
scope, [133–137](#)
unspecified PL/SQL error, [408](#)
UTL_FILE, [408](#)
write error, [408](#)
WRITE_ERROR, [408](#)

EXCEPTION keyword, [8](#)
Exception propagation, [141–147](#)
Exception-handling section, [7–8](#)
EXCEPTION_INIT pragma, [153–155](#)
Exceptions, raising
 explicitly, [144–145](#)
 implicitly, [127](#)
 re-raising, [147](#)
 user-defined, [138](#)
Executable section, [6–7](#)
EXECUTE IMMEDIATE statements
 avoiding ORA errors, [262–271](#)
 binding collections with, [299–305](#)
 description, [260–261](#)
 RETURNING INTO clause, [261–262](#)

USING clause, [261–262](#)

Executing blocks

overview, [8–9](#)

SQL Developer, [14–16](#)

Executing queries

SQL Developer, [14](#)

SQL*Plus, [15](#)

Execution times

baseline, computing, [432–433](#)

for SQL and PL/SQL, separating, [436–437](#)

EXISTS method, [232–235](#)

EXIT statements, [93–97](#)

EXIT WHEN statements, [97–98](#)

Explain plan, generating, [414–417](#)

Explicit cursor variables, [345](#)

Expressions

() (parentheses), controlling order of operations, [38](#)

CASE expressions, [80–84](#)

comparing. *See* [COALESCE function](#); [NULLIF function](#).

in a cursor select lists, [175](#)

operands, [38](#)

operators, [38–39](#). *See also* specific operators.

EXTEND method, [231, 232–235](#)

Extending collections, [232–235](#)

Extending packages

with additional procedures, [353–366](#)

final_grade function, [355–366](#)

manage_grades package specification, [354–356](#)

median_grade function, [362–365](#)

F

FCLOSE function, [407](#)

FCLOSE_ALL procedure, [407](#)

FETCH command, [166–167](#)

FETCH FIRST clause, [xxviii–xxix](#)

FETCH statements, [271–280](#), [306–309](#)

Fetch status, getting, [170–174](#)

Fetching records

from dynamic SQL cursors, [271–280](#)

results in bulk SQL, [291–299](#)

rows in a cursor, [166–167](#)

FFLUSH procedure, [407](#)

File handle invalid, exception, [408](#)

File location not valid exception, [408](#)

Filename not valid, exception, [408](#)

Files, accessing within PL/SQL, [406–410](#)

FILE_TYPE data type, [407](#)

Firing order, compound triggers, [219](#)

Firing triggers, [192](#), [194](#)

FIRST method, [233–235](#)

Flushing the data buffer, [407](#)

FLUSH_PROFILER routine, [433](#)

FOLLOW option, [194](#)

FOPEN function, [407](#)

FOR loops. *See* [Numeric FOR loops](#).

FOR reserved word, [104](#)

FOR UPDATE clause, [187–189](#)

FOR UPDATE OF clause, [189](#)

FORALL statements

description, [282–285](#)

error messages, displaying, [287–288](#)

exception handling, [285–288](#)

implicit loop counter, [283](#)

improving performance, [260](#)

INDICES OF option, [283](#), [288](#)

looping, [283](#), [288–290](#)

SAVE EXCEPTIONS option, [285–288](#)

SQL%BULK_EXCEPTIONS attribute, [286–287](#)

VALUES OF option, [289–290](#)

Formal parameters, [317–318](#)

FORMAT_CALL_STACK function, [419–421](#)

FORMAT_ERROR_BACKTRACE function, [419](#), [421–422](#)

FORMAT_ERROR_STACK function, [419](#), [422–424](#)

Formatting guide

case sensitivity, [455](#)

comments, [456–459](#)

naming conventions, [456–457](#)

white space, [455–456](#)

Formatting guide, for readability by humans

dynamic SQL statements, [275](#)

EXCEPTION_INIT pragma, [155](#)

formatting IF statements, [66–67](#)

formatting SELECT statements, [275](#)

grouping with parentheses, [69](#), [252](#)

inserting blank lines, [242](#)

inserting blank spaces, [275](#)

labels on nested blocks, [39–40](#)

labels on nested loops, [120](#)

WORK keyword, [51–52](#)

%FOUND attribute, [170–174](#)

Functions. *See also* [Modular code](#).

collections of. *See* [Packages](#).

final_grade function, [355–366](#)

identifying, [377–378](#)

invoking in SQL statements, [327–328](#)

IR (invoker rights), [xxvi–xxvii](#)

median_grade function, [362–365](#)

optimizing execution, [329–331](#)
vs. procedures, [322](#)
syntax, [322–327](#)
user-defined. *See* [User-defined functions](#).
uses for, [325–327](#)

Functions, creating
complex functions, [328–329](#)
stored functions, [322–325](#)
using a WITH clause, [329–330](#)
using the UDF pragma, [330–331](#)

G

GET_LINE procedure, [407](#)
GET_NEXT_RESULT procedure, [xxxii](#)
GET_PLSQL_TRACE_LEVEL routine, [434–436](#)
Getting records. *See* [Fetching records](#).

Grouping transactions, [49](#)

H

Help, Oracle online, [193](#)
Hierarchical Profiler, [436–437](#)

I

Identifiers, [29](#), [33–34](#). *See also* [Variables](#).
IF statements. *See also* [ELSIF statements](#).
description, [58](#)
formatting for readability, [66–67](#)
inner, [67](#)
logical operators, [68–70](#)
nested, [67–70](#)
outer, [67](#)

IF-THEN statements

description, [58–60](#)

inner IF, [67](#)

IF-THEN-ELSE statements

description, [60–63](#)

outer IF, [60–63](#)

Implicit cursors, [160–161](#)

Implicit statement results, [xxxii–xxxii](#)

Implicit statement results, generating, [417–418](#)

IN option, [105–107](#)

IN OUT parameter, [316–317](#)

IN parameter, [315–319](#)

Index-by tables. *See* [Associative arrays](#).

INDICES OF option, [283, 288](#)

Infinite loops

definition, [93](#)

simple, [95](#)

WHILE, [100](#)

Information hiding, [335](#)

INHERIT ANY PRIVILEGES clause, [xxxii–xxxiii](#)

INHERIT PRIVILEGES clause, [xxxii–xxxiii](#)

Initializing

nested tables, [230–232](#)

object attributes, [389–390](#)

packages, [367–368](#)

Initializing variables

with an assignment operator, [36–39](#)

with CASE expressions, [83–84](#)

to a null value, [32](#)

with SELECT INTO statements, [44–47, 83–84](#)

Inner IF statements, [67](#)

INSERT statements. *See also* [DML \(Data Manipulation Language\)](#).

batch processing. *See* [FORALL statements](#).

with BULK COLLECT clause, [295](#)

Instantiating packages, [366](#)

INSTEAD OF triggers, [206–211](#)

INTERNAL_ERROR exception, [408](#)

Interpreted mode code generation, [9](#)

INTERVAL parameter, [411](#)

INTERVAL procedure, [410](#)

Invalid

file handle exception, [408](#)

mode exception, [408](#)

operation exception, [408](#)

procedures, [315](#)

INVALID blocks vs. VALID, [9](#)

INVALID_FILEHANDLE exception, [408](#)

INVALID_MODE exception, [408](#)

INVALID_OPERATION exception, [408](#)

INVALID_PATH exception, [408](#)

Invisible columns, [xxxiii–xxxiv](#)

IR (invoker rights) unit

creating views, [xxxii](#)

new for Oracle 12c, [xxvi–xxvii](#), [xxxii–xxxiii](#)

permissions, [xxxii–xxxiii](#)

%ISOPEN attribute, [170–174](#)

IS_OPEN function, [407](#)

Iterative control. See [CONTINUE statements](#); [Loops](#).

J

JOB parameter, [411](#)

Job queue

changing items in the queue, [410](#)

changing job intervals, [410](#)

DBMS_JOB package, [410–412](#)

disabling jobs, [410](#), [412](#)

examining, [412](#)

flagging jobs as broken, [412](#)
forcing a job to run, [410](#), [412](#)
job numbers, assigning, [411](#)
removing jobs from, [410](#), [412](#)
scheduling the next run date, [410](#)
submitting jobs, [410](#), [411–412](#)

K

keep_errors parameter, [150](#)

L

Labels on

nested blocks, [39–40](#)
nested loops, [120](#)

Language components

anchored data types, [34](#)
character types, [28](#)
comments, [29](#)
delimiters, [29](#)
identifiers, [29](#), [33–34](#). *See also* [Variables](#).
lexical units, [28–29](#)
literals, [29](#)
reserved words, [29](#), [32–33](#)
variables, [29–32](#), [36–39](#). *See also* [Identifiers](#); [Substitution variables](#).

LAST method, [233–235](#)

Lexical units, [28–29](#)

LIBRARY objects, defining as DIRECTORY objects, [xxx–xxxi](#)

LIMIT method, [238](#), [292–293](#)

Limiting result sets, bulk SQL, [292–293](#)

Line terminators, inserting, [408](#)

Literals

definition, [29](#)
in expressions, [38](#)

LOB data type, [36](#)

Locking rows for update, [187–189](#)

Logical operators, [39](#), [68–70](#)

LOGIN_DENIED exception, [128](#)

LONG data type, [36](#)

LONG RAW data type, [36](#)

Loop labels, [120–122](#)

LOOP reserved word, [92](#)

Looping

FORALL statements, [283](#), [288–290](#)

INDICES OF option, [283](#), [288](#)

VALUES OF option, [289–290](#)

Loops, nested, [118–120](#). *See also* [Nested cursors](#).

Loops, numeric FOR

description, [104–105](#)

IN option, [105–107](#)

premature termination, [108–109](#)

REVERSE option, [107–108](#)

Loops, simple

description, [92–93](#)

EXIT statements, [93–97](#)

EXIT WHEN statements, [97–98](#)

infinite, [93](#), [95](#)

inner loops, [119](#)

RETURN statements, [96](#)

terminating, [93–98](#)

Loops, WHILE

Boolean expressions as test conditions, [101](#)

description, [98–101](#)

infinite, [100](#)

outer loops, [119](#)

premature termination, [101–103](#)

M

Map methods, [400–401](#)

MAX_STRING_SIZE parameter

displaying data type size, [xxx](#)

Member methods, [398](#)

Methods (functions and procedures), [386](#)

Modes

code generation, [9](#)

invalid, exception, [408](#)

procedure parameters, [317–318](#)

Modular code

anonymous blocks, [312](#)

benefits of, [312](#)

block structure, [312](#)

definition, [311](#)

types of, [312](#). *See also* specific types.

Multilevel collections, [240–242](#)

Multirow queries, [271–280](#)

Mutating table errors, [214](#)

Mutating tables

definition, [214](#)

resolving issues, [215–223](#)

N

Named blocks, [5, 8–9](#). *See also* [Anonymous blocks](#).

Named notation, procedure parameters, [318–319](#)

Naming conventions

explicit cursors, [162–163](#)

formatting guide, [456–457](#)

variables, [29–30](#)

Native code, [9](#)

Native dynamic SQL. *See* [Dynamic SQL](#).

Native mode code generation, [9](#)

Nested

blocks, [5, 39–41](#)

collections in object types, [393](#)

cursors, [177–181](#)

IF statements, [67–70](#)

loops, [118–120](#)

records, [250–253](#)

varrays, [240–242](#)

Nested cursors

complex, [185–187](#)

looping through data, [177–181, 185–187](#)

processing, [177–181](#)

Nested tables

vs. associative arrays and varrays, [239–240](#)

creating, [229–230](#)

initializing, [230–232](#)

LIMIT method, [238](#)

populating with the BULK COLLECT clause, [292](#)

upper bounds, specifying, [238–239](#)

New features, summary of, [xxv–xxvi](#). *See also specific features.*

:NEW pseudorecords, [196–199](#)

NEW_LINE function, [408](#)

NEXT DATE procedure, [410](#)

NEXT method, [233–235](#)

NEXT_DATE parameter, [411](#)

NEXTVAL pseudocolumn, [48](#)

NO_DATA_FOUND exception, [128](#)

associative arrays, [228–229](#)

bulk SQL, [292](#)

NONEDITABLE property, [xxxiv, 193](#)

Nonrestrictive (weak) cursor variables, [345–346](#)

NO_PARSE parameter, [411](#)

Not null, constraining variables to, [32](#)

%NOTFOUND attribute, [170–174](#)

Null condition, IF-THEN-ELSE statements, [61–63](#)

Null values

assigning to expressions in NULLIF functions, [86–87](#)

variables, [32](#)

NULL vs. empty, [232](#)

NULLIF function, [84–87](#). *See also* [COALESCE function](#).

NULLS, passing to dynamic SQL statements, [265–266](#)

NUMBER data type, [35](#)

Numeric FOR loops

in cursors, [175–177](#)

description, [104–105](#)

IN option, [105–107](#)

premature termination, [108–109](#)

REVERSE option, [107–108](#)

NVACHAR2 data type, [xxx](#)

O

Object attributes, initializing, [389–390](#)

Object instances. *See* [Objects](#).

Object specification, [388](#)

Object type methods

comparing objects, [399–404](#)

constructor, [395–397](#)

definition, [395](#)

functions and procedures, [386](#)

member, [398](#)

parameter, [395](#)

SELF parameter, [395, 397, 398, 401](#)

static, [398–399](#)

Object types

attributes (data), [386](#)
with collections, [391–394](#)
components of, [386](#)
creating, [386–390](#)
encapsulation, [386](#)
methods (functions and procedures), [386](#)
nesting collections in, [393](#)

Objects

associative arrays, populating with data, [392](#)
comparing, [399–404](#)
getting information about, [314–315](#)
initial value, [389](#)
schema, editionable vs. noneditionable, [xxxiv](#)
:OLD pseudorecords, [196–199](#)

Open cursors, testing for, [170–174](#)

Open files

testing for, [407](#)
writing to, [408](#)

OPEN-FOR statements

binding collections with, [306–309](#)
opening cursors, [271–280](#)

Opening

dynamic SQL cursors, [271–280](#)
explicit cursors, [162](#), [165–166](#), [172–173](#)
files, [407](#)

Operands

definition, [38](#)
in expressions, [38](#)

Operation invalid, exception, [408](#)

Operators

definition, [38](#)
in expressions, [38](#)

precedence, [39](#)

Optimization levels

examples of, [439–444](#)

performance optimizer, [438](#)

PLSQL_OPTIMIZE_LEVEL parameter, [438](#)

summary of, [438](#)

Optimizing

dynamic SQL, [260](#)

function execution, [329–331](#)

Optimizing PL/SQL, tuning tools

ANALYZE routine, [437](#)

CLEAR_PLSQL_TRACE routine, [434–436](#)

computing execution time baseline, [432–433](#)

DBMS_HPROF package, [436–437](#)

DBMSHPTAB.sql script, [437](#)

DBMS_PROFILER package, [432–433](#)

DBMS_TRACE package, [433–436](#)

FLUSH_PROFILER routine, [433](#)

GET_PLSQL_TRACE_LEVEL routine, [434–436](#)

Hierarchical Profiler, [436–437](#)

PAUSE_PROFILER routine, [433](#)

Profiler API, [432–433](#)

profiling execution of applications, [436–437](#)

PROFLOAD.sql script, [432–433](#)

PROFTAB.sql script, [432–433](#)

RESUME_PROFILER routine, [433](#)

separating execution times for SQL and PL/SQL, [436–437](#)

SET_PLSQL_TRACE routine, [434–436](#)

START_PROFILER routine, [432–433](#)

START_PROFILING routine, [437](#)

STOP_PROFILER routine, [432–433](#)

STOP_PROFILING routine, [437](#)

Trace API, [433–436](#)

TRACE_ALL_CALLS constant, [434–436](#)

TRACE_ALL_EXCEPTIONS constant, [434–436](#)

TRACE_ALL_SQL constant, [434–436](#)

TRACE_ENABLED_CALLS constant, [434–436](#)

TRACE_ENABLED_EXCEPTION constant, [434–436](#)

TRACE_ENABLED_SQL constant, [434–436](#)

TRACE_PAUSE constant, [434–436](#)

TRACE_RESUME constant, [434–436](#)

TRACE_STOP constant, [434–436](#)

TRACETAB.sql script, [433–436](#)

tracing order of execution, [433–436](#)

ORA errors, avoiding, [262–271](#)

Oracle Forms Developer: The Complete Video Course, [xxiii](#)

Oracle online help, [193](#)

Oracle PL/SQL by Example, Fifth Edition, [xvii](#)

Oracle sequences. *See* [Sequences](#).

Oracle server, [2–4](#)

Oracle SQL by Example, [414](#)

Oracle SQL Developer. *See* [SQL Developer](#).

Oracle Web Application Programming for PL/SQL Developers, [xxiii](#)

Oracle-supplied packages

accessing files within PL/SQL, [406–410](#)

DBMS_JOB, [410–412](#)

DBMS_SQL, [417–418](#)

DBMS_XPLAN, [414–417](#)

explain plan, generating, [414–417](#)

implicit statement results, generating, [417–418](#)

scheduling jobs, [410–413](#)

text file capabilities, [406–410](#)

UTL_FILE, [406–410](#)

Oracle-supplied packages, error reporting

DBMS.Utility package, [419–424](#)
UTL_Call_Stack package, [424–429](#)
Order methods, [401–404](#)
Order of execution, tracing, [433–436](#)
OTHERS exception, [131](#), [155–156](#). *See also* [SQLCODE function](#); [SQLERRM function](#).
OUT parameter, [315–319](#)
Outer IF statements, [67](#)
Overloading
 construction methods, [397](#)
 modules, [378–382](#)

P

Packages. *See also* [Modular code](#).
 benefits of, [334](#)
 currency conversion example, [334](#)
 definition, [333](#)
 granting roles to, [xxix–xxx](#)
 identifying, [377–378](#)
 initialization, [367–368](#)
 instantiation, [366](#)
 manage_grades package specification, [354–356](#)
 referencing packaged elements, [336–337](#). *See also* [Cursor variables](#).
 serialization, [368–371](#)
 stored, calling, [339–341](#)
 supplied by Oracle. *See* [Oracle-supplied packages](#).

Packages, creating
 information hiding, [335](#)
 package body, [335–336](#), [337–339](#)
 package specification, [335](#)
 package variables, [367–368](#)
 private elements, [341–344](#)
Packages, extending
 with additional procedures, [353–366](#)

`final_grade` function, [355–366](#)

`manage_grades` package specification, [354–356](#)

`median_grade` function, [362–365](#)

Parameterized cursors, [183–185](#)

Parameters, passing to procedures

actual parameters, [317–318](#)

data types, [318](#)

default values, [318–319](#)

formal parameters, [317–318](#)

modes, [317–318](#)

named notation, [318–319](#)

OUT parameter, [315–319](#)

IN OUT parameter, [316–317](#)

IN parameter, [315–319](#)

positional notation, [318–319](#)

Parentheses ()

controlling order of operations, [38](#)

grouping for readability, [69, 252](#)

Parse trees, [8](#)

Passing

data types to procedures, [318](#)

NULLS to dynamic SQL statements, [265–266](#)

run-time values to bind arguments, [272](#)

Passing parameters to procedures

actual parameters, [317–318](#)

data types, [318](#)

default values, [318–319](#)

formal parameters, [317–318](#)

modes, [317–318](#)

named notation, [318–319](#)

OUT parameter, [315–319](#)

IN OUT parameter, [316–317](#)

IN parameter, [315–319](#)

positional notation, [318–319](#)

Passwords

SQL Developer, case sensitivity, [10](#)

SQL*Plus, displaying, [13](#)

PAUSE_PROFILER routine, [433](#)

P-code, [9](#)

PDBs (pluggable databases), [xxx](#)

Performance. *See* [Optimizing](#).

Performance optimizer, [438](#). *See also* [Optimizing PL/SQL](#).

PL/SQL Scripts, [14–16](#)

PL/SQL statements, [44](#). *See also* [SQL statements](#); specific statements.

PLSQL_CODE_TYPE parameter, [9](#)

PLSQL_DEBUG parameter, [xxxvii](#)

\$\$PLSQL_LINE directive, [xxxvi–xxxvii](#)

PL/SQL-only data types, [xxvi–xxvii](#)

PLSQL_OPTIMIZE_LEVEL parameter, [438](#)

\$\$PLSQL_UNIT directive, [xxxvi–xxxvii](#)

\$\$PLSQL_UNIT_OWNER directive, [xxxvi–xxxvii](#)

\$\$PLSQL_UNIT_TYPE directive, [xxxvi–xxxvii](#)

Populating associative arrays, [227](#)

Positional notation, procedure parameters, [318–319](#)

PRAGMA INLINE statement, [445](#)

Pragmas, definition, [153](#)

PRECEDES option, [194](#)

Predefined exceptions, [128–129](#)

Predefined inquiry directives, new for Oracle 12c, [xxxvi–xxxvii](#)

Presentation tier, [3](#)

Primary key values, generating. *See* [Sequences](#).

Printing query results automatically, [348](#)

PRIOR method, [233–235](#)

Privileges for creating views, [207](#)

Procedures. *See also* [Modular code](#).

collections of. *See* [Packages](#).

creating, [312–315](#)

vs. functions, [322](#)

getting information about, [314–315](#)

identifying, [377–378](#)

invalid, recompiling, [315](#)

Procedures, displaying

 data dictionary queries, [314–315](#)

 invalid, recompiling, [315](#)

 invalid vs. valid, [315](#)

 red X, [315](#)

 with SQL Developer, [315](#)

Procedures, passing parameters

 actual parameters, [317–318](#)

 data types, [318](#)

 default values, [318–319](#)

 formal parameters, [317–318](#)

 modes, [317–318](#)

 named notation, [318–319](#)

 OUT parameter, [315–319](#)

 IN OUT parameter, [316–317](#)

 IN parameter, [315–319](#)

 positional notation, [318–319](#)

Profiler API, [432–433](#)

PROFLOAD.sql script, [432–433](#)

PROFTAB.sql script, [432–433](#)

PROGRAM_ERROR exception, [128](#)

PUT procedure, [408](#)

PUTF procedure, [408](#)

PUT_LINE procedure, [408](#)

Q

Queries. See [SQL queries](#).

Query results

printing automatically, [348](#)

sharing. See [Cursor variables](#).

R

RAISE statements

in conjunction with IF statements, [140](#)

raising exceptions explicitly, [144–145](#)

raising user-defined exceptions, [138](#)

re-raising exceptions, [147](#)

RAISE_APPLICATION_ERROR procedure, [149–153](#)

Raising exceptions

explicitly, [144–145](#)

implicitly, [127](#)

re-raising exceptions, [147](#)

user-defined, [138](#)

RAW data type, [xxx](#), [36](#)

Read error, exception, [408](#)

Readability (by humans)

dynamic SQL statements, [275](#)

EXCEPTION_INIT pragma, [155](#)

formatting IF statements, [66–67](#)

formatting SELECT statements, [275](#)

grouping with parentheses, [69](#), [252](#)

inserting blank lines, [242](#)

inserting blank spaces, [275](#)

labels on nested blocks, [39–40](#)

labels on nested loops, [120](#)

WORK keyword, [51–52](#)

READ_ERROR exception, [408](#)

Reading

records from a database. See [Fetching records](#).

text from an open file, [407](#)

Record types

compatibility, [249–250](#)

cursor based, [244–246](#), [249–250](#), [253–255](#)

explicit cursors, [163–165](#)

table based, [244–246](#), [249–250](#)

user defined, [246–250](#), [255–256](#)

Records

collections of, [253–256](#)

compatibility, [248–250](#)

cursor-based, [163–165](#)

enclosing, [250](#)

explicit cursors, [163–165](#)

nested, [250–253](#)

reading. *See* [Fetching records](#).

table-based, [163–165](#)

testing values of, [244](#)

user-defined, [168–170](#)

Red X on displayed procedures, [315](#)

REF CURSOR data type, [345–346](#). *See also* [Cursor variables](#).

REMOVE procedure, [410](#), [412](#)

REPLACE reserved word, [192–193](#)

Re-raising exceptions, [146–148](#)

Reserved words, [29](#), [32–33](#)

Restricted mode, turning on/off, [411](#)

Restrictive (strong) cursor variables, [345–346](#)

Result sets, sharing. *See* [Cursor variables](#).

Result-caching, IR (invoker rights) functions, [xxvi–xxvii](#)

RESUME_PROFILER routine, [433](#)

RETURN statements, [96](#)

RETURNING clause, with BULK COLLECT clause, [295](#)

RETURNING INTO clause, [261–262](#)

RETURN_RESULT procedure, [xxxi–xxxii](#)
REVERSE option, [107–108](#)
Roles, granting to PL/SQL packages and standalone subprograms, [xxix–xxx](#)
ROLLBACK statements, [49–51](#), [52](#), [195](#)
%ROWCOUNT attribute, [170–174](#)
Row-level triggers, [194](#), [205–206](#)
Rows, locking for update, [187–189](#)
%ROWTYPE attribute, [163–165](#), [244–246](#)
RUN procedure, [410](#), [412](#)
Runtime errors. *See also* [Error handling](#); [Exceptions](#).
 vs. compilation errors, [124–126](#)
 in a declaration section, [142–143](#). *See also* [Exception propagation](#).
 definition, [7–8](#)
 error handling, [141–147](#)
 in an exception-handling section, [143–144](#). *See also* [Exception propagation](#).

S

SAVE EXCEPTIONS option, [285–288](#)
SAVEPOINT statements
 breaking down large PL/SQL statements, [44](#)
 setting a save point, [49–51](#), [52–53](#)
 in triggers, [195](#)
Saving database changes. *See* [COMMIT statements](#).
Scheduling jobs, [410–413](#)
Scope
 cursors, [175](#)
 exceptions, [133–137](#)
 labels, [39–41](#)
 nested blocks, [39–41](#)
 variables, [39](#)
Searched CASE statements
 vs. CASE statements, [76–80](#)
 description, [74–80](#)

Sections of blocks, [6–8](#)

SELECT INTO statements, [44–47](#)

Select list, cursors, [175](#)

SELECT statements

dynamic, [259](#). *See also* [Dynamic SQL](#).

formatting for readability, [275](#)

returning no rows, [47](#)

returning too many rows, [47](#)

static, [259](#)

SELF parameter, [395](#), [397](#), [398](#), [401](#)

Semantic checking, [9](#)

Semicolon (;

block terminator, [16–17](#)

dynamic SQL statement terminator, [264–265](#)

variable terminator, [36–37](#)

Sequences

accessing, [48](#)

in blocks, [48–49](#)

of contiguous numbers, [48](#)

definition, [47](#)

drawing numbers from, [48](#)

incrementing, [48](#)

uses for, [47](#)

Serialized packages, [368–371](#)

SERIALLY_REUSEABLE pragma, [368–371](#)

SET_PLSQL_TRACE routine, [434–436](#)

Setting a save point. *See* [SAVEPOINT statements](#).

SHO ERR command, [376](#)

SID, default, [10](#)

Simple loops

description, [92–93](#)

EXIT statements, [93–97](#)

EXIT WHEN statements, [97–98](#)

infinite, [95](#)

inner loops, [119](#)

RETURN statements, [96](#)

terminating, [93–98](#)

Single quotes (' '), enclosing substitution variables, [25](#)

Single-row queries, [261–271](#)

Slash (/), block terminator, [16, 264](#)

Slash asterisk... /*...*/, multiline comments, [29, 40](#)

Source code, displaying, [314–315](#)

SQL cursors, [160](#)

SQL Developer

connecting to a database, [10–11](#)

connection name, [10](#)

default SID, [10](#)

definition, [9](#)

disabling substitution variable verification, [23](#)

disconnecting from a database, [11–12](#)

displaying procedures, [315](#)

executing a block, [14–16](#)

executing a query, [14](#)

getting started with, [10–11](#)

launching, [10](#)

password, [10](#)

substitution variables, [19–25](#)

user input at runtime. See [Substitution variables](#).

user name, [10](#)

SQL queries

implicit statement results, [xxxii–xxxii](#)

multirow, [271–280](#)

new for Oracle 12c, [xxxii–xxxii](#)

single-row, [261–271](#)

SQL statements. *See also* [PL/SQL statements](#).

; (semicolon), statement terminator, [15](#)

vs. PL/SQL, [14](#)

SQL%BULK_EXCEPTIONS attribute, [286–287](#)

SQLCODE function, [155–158](#). *See also* [OTHERS exception](#); [SQLERRM function](#).

SQLERRM function, [155–158](#). *See also* [OTHERS exception](#); [SQLCODE function](#).

SQL*Plus

/ (slash), block terminator, [16](#)

; (semicolon), block terminator, [16–17](#)

accessing, [11, 13](#)

connecting to a database, [13](#)

definition, [9](#)

disabling substitution variable verification, [23](#)

disconnecting from a database, [13](#)

error isolation, [314](#)

executing a query, [15](#)

getting started with, [11–13](#)

password, [13](#)

substitution variables, [19–25](#)

sqlplus command, [13](#)

START_PROFILER routine, [432–433](#)

START_PROFILING routine, [437](#)

Statement-level triggers, [194, 205–206](#)

Statements. *See* [PL/SQL statements](#).

Static methods, [398–399](#)

Static SELECT statements, [259](#)

STOP_PROFILER routine, [432–433](#)

STOP_PROFILING routine, [437](#)

Stored code, examining

ALL_DEPENDENCIES view, [376–377](#)

ALL_OBJECTS view, [374](#)

with the data dictionary, [374–378](#)

DBA_DEPENDENCIES view, [376–377](#)
DBA_OBJECTS view, [374](#)
debugging, [376](#)
dependencies, displaying, [376–377](#)
DESC command, [377–378](#)
describing columns in a table, [377–378](#)
displaying errors, [375–376](#)
identifying procedures, packages, and functions, [377–378](#)
overloading modules, [378–382](#)
retrieving specified line numbers, [374–375](#)
SHO ERR command, [376](#)
USER_DEPENDENCIES view, [376–377](#)
USER_ERRORS view, [375–376](#)
USER_OBJECTS view, [374](#)
Stored functions, creating, [322–325](#)
Stored packages, calling, [339–341](#)
Stored queries. *See* [Views](#).
String operators, [39](#)
Strong (restrictive) cursor variables, [345–346](#)
STUDENT database schema, [461–468](#)
SUBMIT procedure, [410](#)
Submitting jobs, [410, 411–412](#). *See also* [Job queue](#).
Subprogram inlining, [445–453](#)
Subprograms, granting roles to, [xxix–xxx](#)
Substitution variables. *See also* [Variables](#).
‘ ‘ (single quotes), enclosing in, [25](#)
& (ampersand), name prefix, [20, 22, 25](#)
&& (double ampersand), name prefix, [20, 24, 25](#)
disabling, [25](#)
disabling verification, [23](#)
name prefix character, changing, [25](#)
overview, [19–25](#)

Syntax checking, [8–9](#)

Syntax errors. *See* [Compilation errors](#).

T

Table-based records

compatibility, [249–250](#)

creating, [163–165](#)

definition, [163](#)

description, [244–246](#)

Tables

mutating, [213–223](#)

PL/SQL, [226](#). *See also* [Associative arrays](#); [Nested tables](#).

Tables, nested

vs. associative arrays and varrays, [239–240](#)

creating, [229–230](#)

initializing, [230–232](#)

LIMIT method, [238](#)

upper bounds, specifying, [238–239](#)

Text file capabilities, [406–410](#)

Three-tier architecture, [3](#)

TOO_MANY_ROWS exception, [128](#)

Trace API, [433–436](#)

TRACE_ALL_CALLS constant, [434–436](#)

TRACE_ALL_EXCEPTIONS constant, [434–436](#)

TRACE_ALL_SQL constant, [434–436](#)

TRACE_ENABLED_CALLS constant, [434–436](#)

TRACE_ENABLED_EXCEPTION constant, [434–436](#)

TRACE_ENABLED_SQL constant, [434–436](#)

TRACE_PAUSE constant, [434–436](#)

TRACE_RESUME constant, [434–436](#)

TRACE_STOP constant, [434–436](#)

TRACETAB.sql script, [433–436](#)

Tracing order of execution, [433–436](#)

Transaction control

and DML, [53–55](#)

erasing changes. *See* [ROLLBACK statements](#).

saving changes. *See* [COMMIT statements](#).

setting a save point. *See* [SAVEPOINT statements](#).

Transactional control statements, from triggers, [195](#)

Transactions

vs. blocks, [50, 54–55](#)

breaking down large statements, [44](#)

definition, [43](#)

grouping, [49](#)

Triggering events, [192](#)

Triggers. *See also* [Modular code](#).

AFTER, [201–204](#)

autonomous transactions, [203–204](#)

BEFORE, [195–201](#)

compound, [217–223](#)

creating, [192–195, 197–201](#)

defined on views, [206–211](#)

definition, [192](#)

in dropped tables, [195](#)

enabling/disabling, [194](#)

event, [xxx](#)

firing, [192](#)

firing order, specifying, [194](#)

INSTEAD OF clause, [206–211](#)

issuing transactional control statements, [195](#)

mutating table errors, [214–223](#)

:NEW pseudorecords, [196–199](#)

:OLD pseudorecords, [196–199](#)

restrictions, [195](#)

row-level, [194, 205–206](#)

statement-level, [194](#), [205–206](#)

types of, [205–211](#)

uses for, [195](#)

TRIM method, [233–235](#)

Tuning PL/SQL. *See* [Optimizing PL/SQL, tuning tools](#).

TYPE statements, [247–248](#)

U

UDF pragma

creating functions, [330–331](#)

creating user-defined functions, [xxxiv–xxxv](#)

Undoing database changes. *See* [ROLLBACK statements](#).

Unique numbers, generating, [47–49](#)

UPDATE statements. *See also* [DML \(Data Manipulation Language\)](#).

batch processing. *See* [FORALL statements](#).

with BULK COLLECT clause, [295](#)

Updating tables in a database, [187–190](#). *See also* [UPDATE statements](#).

User name, SQL Developer, [10](#)

User-defined exceptions

declaring, [137](#)

description, [137–141](#)

raising explicitly, [138–139](#)

unhandled, [145](#)

User-defined functions

creating with a UDF pragma, [xxxiv–xxxv](#)

creating with a WITH clause, [xxxiv](#)

running under SQL, [xxxiv–xxxv](#)

User-defined records

compatibility, [249–250](#)

defining a collection on, [255–256](#)

description, [168–170](#), [246–249](#)

USER_DEPENDENCIES view, [376–377](#)

USER_ERRORS view, [375–376](#)

USER_OBJECTS view, [314–315](#), [374](#)

USER_SOURCE view, [314–315](#)

USING clause, [261–262](#)

UTL_CALL_STACK package, [424–429](#)

UTL_FILE package, [406–410](#)

V

VALID blocks vs. INVALID, [9](#)

VALUE_ERROR exception, [129](#)

VALUES OF option, [289–290](#)

VARCHAR2 data type, [xxx](#), [35](#)

Variables. *See also* [Identifiers](#); [Substitution variables](#).

; (semicolon), variable terminator, [36–37](#)

case sensitivity, [29](#)

constraining to not null, [32](#)

declaring, [36–39](#)

displaying values. *See* [DBMS_OUTPUT.PUT_LINE statements](#).

in expressions, [38](#)

with identical names, [121–122](#)

naming conventions, [29–30](#)

null values, [32](#)

overview, [29–32](#)

scope, [39](#)

visibility, [40](#)

Variables, initializing

with an assignment operator, [36–39](#)

with CASE expressions, [83–84](#)

to a null value, [32](#)

with SELECT INTO statements, [44–47](#), [83–84](#)

Varrays

declaring, [236–238](#)

definition, [235–236](#)

nested, [240–242](#)

vs. nested tables and associative arrays, [239–240](#)

upper bounds, setting, [238–239](#)

View queries, [208](#). *See also* [SELECT statements](#).

Views, creating

BEQUEATH CURRENT_USER clause, [xxxii](#)

BEQUEATH DEFINER clause, [xxxii](#)

as an IR (invoker rights) unit, [xxxii](#)

new for Oracle 12c, [xxxii](#)

privileges for, [207](#)

Views, triggers defined on, [206–211](#)

Visibility of variables, [40](#)

W

Weak (nonrestrictive) cursor variables, [345–346](#)

Website, companion to this book. *See* [Companion Website](#).

WHAT parameter, [411](#)

WHERE CURRENT OF clause, [189–190](#)

WHILE loops

Boolean expressions as test conditions, [101](#)

description, [98–101](#)

infinite, [100](#)

outer loops, [119](#)

premature termination, [101–103](#)

WHILE reserved word, [99](#)

White space, formatting guide, [455–456](#)

WITH clause

creating functions, [329–330](#)

creating user-defined functions, [xxxiv](#)

WORK keyword, for readability, [51–52](#)

Write error, exception, [408](#)

WRITE_ERROR exception, [408](#)

Z

ZERO_DIVIDE exception, [128](#)

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the leading brands, authors, and contributors from the tech community.

▲ Addison-Wesley Cisco Press IBM
Press Microsoft Press

PEARSON IT CERTIFICATION PRENTICE HALL DUE SAMS VMWARE PRESS

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? InformIT has a solution.

- Learn about new releases and special promotions by subscribing to a wide variety of monthly newsletters. Visit informit.com/newsletters.
- FREE Podcasts from experts at informit.com/podcasts.
- Read the latest author articles and sample chapters at informit.com/articles.
- Access thousands of books and videos in the Safari Books Online digital library. safari.informit.com.
- Get Advice and tips from expert blogs at informit.com/blogs.

Visit informit.com to find out all the ways you can access the hottest technology content.

Are you part of the **IT** crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube and more! Visit informit.com/socialconnect.





REGISTER



THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to informit.com/register to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

Code Snippets

```
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;
```

```
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

```
-- 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
```

```
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
```

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

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

```
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;
/
```

```
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
```

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

```
-- 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
```

```
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 Garcia
Mr. Gerry Tripp
Mr. Rommel Frost
Mr. Roger Snow
Ms. Z.A. Scrittore
Mr. Joseph Yourish
```

```
CREATE OR REPLACE FUNCTION format_name (p_salutation IN VARCHAR2
                                         ,p_first_name IN VARCHAR2
                                         ,p_last_name  IN VARCHAR2)
RETURN VARCHAR2
AS
  PRAGMA UDP;
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. Scrittoreale
Mr. Joseph Yourish

```
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_DIRECTIVES 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
```

```
DECLARE
  Declaration statements
BEGIN
  Executable statements
EXCEPTION
  Exception-handling statements
END;
```

```
DECLARE  
  v_first_name VARCHAR2(35);  
  v_last_name  VARCHAR2(35);
```

```
BEGIN
  SELECT first_name, last_name
  INTO v_first_name, v_last_name
  FROM student
  WHERE student_id = 123;

  DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
END;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('This is a test')
END;
```

```
BEGIN
    SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM student
    WHERE student_id = 123;
    DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);

EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
END;
```

```
DECLARE
  v_first_name VARCHAR2(35);
  v_last_name  VARCHAR2(35);
BEGIN
  SELECT first_name, last_name
    INTO v_first_name, v_last_name
   FROM student
  WHERE student_id = 123;

  DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);

EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
END;
```

```
DECLARE
  v_first_name VARCHAR2(35);
  v_last_name  VARCHAR2(35);
BEGIN
  SELECT first_name, last_name
    INTO v_first_name, v_last_name
   FROM student
  WHERE student_id = 123;

  DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no student with student id 123');
END;
```

```
SET SERVEROUTPUT ON SIZE 5000;
```

```
DECLARE
  v_student_id NUMBER := &sv_student_id;
  v_first_name VARCHAR2(35);
  v_last_name  VARCHAR2(35);
BEGIN
  SELECT first_name, last_name
    INTO v_first_name, v_last_name
   FROM student
  WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('Student name: '||v_first_name||' '||v_last_name);

EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Today is'||'&v_day');
  DBMS_OUTPUT.PUT_LINE ('Tomorrow will be'||'&v_day');
END;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Today is'||'&sv_day');
  DBMS_OUTPUT.PUT_LINE ('Tomorrow will be'||'&sv_day');
END;
```

```
DECLARE  
  v_course_no VARCHAR2(5) := '&sv_course_no';
```

```
Syntax : <variable-name> <data type> [optional default -assignment]
```

```
SET SERVEROUTPUT ON;
DECLARE
    first&last_names  VARCHAR2(30);
BEGIN
    first&last_names := 'TEST NAME';
    DBMS_OUTPUT.PUT_LINE(first&last_names);
END;
```

```
Enter value for last_names: Ben
old 2:    first&last_names VARCHAR2(30);
new 2:    firstBen VARCHAR2(30);
Enter value for last_names: Ben
old 4:    first&last_names := 'TEST NAME';
new 4:    firstBen := 'TEST NAME';
Enter value for last_names: Ben
old 5:    DBMS_OUTPUT.PUT_LINE(first&last_names);
new 5:    DBMS_OUTPUT.PUT_LINE(firstBen);
TEST NAME
PL/SQL procedure successfully completed.
```

```
SET SERVEROUTPUT ON
DECLARE
  v_name VARCHAR2(30);
  v_dob DATE;
  v_us_citizen BOOLEAN;
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_name || 'born on' || v_dob);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
    exception VARCHAR2(15);
BEGIN
    exception := 'This is a test';
    DBMS_OUTPUT.PUT_LINE(exception);
END;
```

```
exception VARCHAR2(15);

ORA-06550: line 2, column 4:
PLS-00103: Encountered the symbol "EXCEPTION" when expecting one of the following:

begin function pragma procedure subtype type <an identifier>
<a double-quoted delimited-identifier> current cursor delete
exists prior.../
```

```
SET SERVEROUTPUT ON;
DECLARE
  v_var1 VARCHAR2(20);
  v_var2 VARCHAR2(6);
  v_var3 NUMBER(5,3);
BEGIN
  v_var1 := 'string literal';
  v_var2 := '12.345';
  v_var3 := 12.345;
  DBMS_OUTPUT.PUT_LINE('v_var1: '||v_var1);
  DBMS_OUTPUT.PUT_LINE('v_var2: '||v_var2);
  DBMS_OUTPUT.PUT_LINE('v_var3: '||v_var3);
END;
```

```
v_var1: string literal
v_var2: 12.345
v_var3: 12.345
PL/SQL procedure successfully completed.
```

```
SET SERVEROUTPUT ON;
DECLARE
  v_var1 NUMBER(2) := 123;
  v_var2 NUMBER(3) := 123;
  v_var3 NUMBER(5,3) := 123456.123;
BEGIN
  DBMS_OUTPUT.PUT_LINE('v_var1: '||v_var1);
  DBMS_OUTPUT.PUT_LINE('v_var2: '||v_var2);
  DBMS_OUTPUT.PUT_LINE('v_var3: '||v_var3);
END;
```

```
ORA-06512: at line 2 ORA-06502: PL/SQL: numeric or value error: number precision too large  
ORA-06512: at line 2
```

Syntax: <variable_name> <type attribute>%TYPE

```
SET SERVEROUTPUT ON
DECLARE
  v_name student.first_name%TYPE;
  v_grade grade.numeric_grade%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE(NVL(v_name, 'No Name ') ||
    ' has grade of '||NVL(v_grade, 0));
END;
```

No Name has grade of 0
PL/SQL procedure successfully completed.

```
SET SERVEROUTPUT ON
DECLARE
  v_cookies_amt NUMBER := 2;
  v_calories_per_cookie CONSTANT NUMBER := 300;
BEGIN
  DBMS_OUTPUT.PUT_LINE('I ate ' || v_cookies_amt ||
    ' cookies with ' || v_cookies_amt *
    v_calories_per_cookie || ' calories.');
  v_cookies_amt := 3;
  DBMS_OUTPUT.PUT_LINE('I really ate ' ||
    v_cookies_amt
    || ' cookies with ' || v_cookies_amt *
    v_calories_per_cookie || ' calories.');
  v_cookies_amt := v_cookies_amt + 5;
  DBMS_OUTPUT.PUT_LINE('The truth is, I actually ate '
    || v_cookies_amt || ' cookies with ' ||
    v_cookies_amt * v_calories_per_cookie
    || ' calories.');
END;
```

```
I ate 2 cookies with 600 calories.  
I really ate 3 cookies with 900 calories.  
The truth is, I actually ate 8 cookies with  
2400 calories.  
PL/SQL procedure successfully completed.
```

```
SET SERVEROUTPUT ON
DECLARE
  v_lname VARCHAR2(30);
  v_regdate DATE;
  v_pctincr CONSTANT NUMBER(4,2) := 1.50;
  v_counter NUMBER := 0;
  v_new_cost course.cost%TYPE;
  v_YorN BOOLEAN := TRUE;
BEGIN
  v_counter := ((v_counter + 5)*2) / 2;
  v_new_cost := (v_new_cost * v_counter)/4;
  --
  v_counter := COALESCE(v_counter, 0) + 1;
  v_new_cost := 800 * v_pctincr;
  --
  DBMS_OUTPUT.PUT_LINE(V_COUNTER);
  DBMS_OUTPUT.PUT_LINE(V_NEW_COST);
END;
```

6
1800
PL/SQL procedure successfully completed.

Arithmetic (** , * , / , + , -)
Comparison(=, <>, !=, <, >, <=, >=, LIKE, IN, BETWEEN, IS NULL, IS
NOT NULL, NOT IN)
Logical (AND, OR, NOT)
String (||, LIKE)
Expressions
Operator Precedence
** , NOT
+, - (arithmetic identity and negation), *, /, , - , ||, =, <>, !=,
<=, >=, <, >, LIKE, BETWEEN, IN, IS NULL
AND-logical conjunction
OR-logical inclusion

```
BEGIN -- outer block
    BEGIN -- inner block
        ...
    END; -- end of inner block
END; -- end of outer block
```

```
set serveroutput on
<< find_stu_num >>
BEGIN
    DBMS_OUTPUT.PUT_LINE('The procedure
        find_stu_num has been executed.');
END find_stu_num;
```

```
SET SERVEROUTPUT ON
<< outer_block >>
DECLARE
    v_test NUMBER := 123;
BEGIN
    DBMS_OUTPUT.PUT_LINE
        ('Outer Block, v_test: '||v_test);
<< inner_block >>
DECLARE
    v_test NUMBER := 456;
BEGIN
    DBMS_OUTPUT.PUT_LINE
        ('Inner Block, v_test: '||v_test);
    DBMS_OUTPUT.PUT_LINE
        ('Inner Block, outer_block.v_test: '||
            Outer_block.v_test);
END inner_block;
END outer_block;
```

```
Outer Block, v_test: 123
Inner Block, v_test: 456
Inner Block, outer_block.v_test: 123
```

```
SET SERVEROUTPUT ON
DECLARE
  e_show_exception_scope EXCEPTION;
  v_student_id      NUMBER := 123;
BEGIN
  DBMS_OUTPUT.PUT_LINE('outer student id is '
    ||v_student_id);
  DECLARE
    v_student_id  VARCHAR2(8) := 125;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('inner student id is '
      ||v_student_id);
    RAISE e_show_exception_scope;
  END;
EXCEPTION
  WHEN e_show_exception_scope
  THEN
    DBMS_OUTPUT.PUT_LINE('When am I displayed?');
    DBMS_OUTPUT.PUT_LINE('outer student id is '
      ||v_student_id);
END;
```

```
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;
```

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

The average cost of a course in the CTA program

is \$1,198.33

PL/SQL procedure successfully completed.

```
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;
```

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

```
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;
```

```
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;
```

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

```
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;
```

```
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;
```

```
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;
```

ROLLBACK [WORK] to SAVEPOINT name;

```
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;
```

```
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;
```

```
DECLARE
  v_num1 NUMBER := 5;
  v_num2 NUMBER := 3;
  v_temp NUMBER;
BEGIN
  -- if v_num1 is greater than v_num2 rearrange their values
  IF v_num1 > v_num2
  THEN
    v_temp := v_num1;
    v_num1 := v_num2;
    v_num2 := v_temp;
  END IF;

  -- display the values of v_num1 and v_num2
  DBMS_OUTPUT.PUT_LINE ('v_num1 = '||v_num1);
  DBMS_OUTPUT.PUT_LINE ('v_num2 = '||v_num2);
END;
```

```
DECLARE
    v_num NUMBER := &sv_user_num;
BEGIN
    -- test if the number provided by the user is even
    IF MOD(v_num,2) = 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
    ELSE
        DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END IF;
END;
```

```
DECLARE
  v_num1 NUMBER := 0;
  v_num2 NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
  IF v_num1 = v_num2
  THEN
    DBMS_OUTPUT.PUT_LINE ('v_num1 = v_num2');
  END IF;
  DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

```
DECLARE
  v_num1 NUMBER := 0;
  v_num2 NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
  IF v_num1 = v_num2
  THEN
    DBMS_OUTPUT.PUT_LINE ('v_num1 = v_num2');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('v_num1 != v_num2');
  END IF;
  DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

```
DECLARE
  v_num NUMBER := &sv_num;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
  IF v_num < 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
  ELSIF v_num = 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is equal to zero');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
  END IF;
  DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

ORA-06550: line 13, column 4:
PLS-00103: Encountered the symbol ";" when expecting one of the following: if

```
IF v_num >= 0
THEN
  DBMS_OUTPUT.PUT_LINE ('v_num is greater than 0');
ELSIF v_num <= 10
THEN
  DBMS_OUTPUT.PUT_LINE ('v_num is less than 10');
ELSE
  DBMS_OUTPUT.PUT_LINE ('v_num is less than ? or greater than ?');
END IF;
```

```
DECLARE
  v_num NUMBER := &sv_num;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Before IF statement...');
  IF v_num < 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is a negative number');
  ELSIF v_num > 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is a positive number');
  END IF;
  DBMS_OUTPUT.PUT_LINE ('After IF statement...');
END;
```

```
IF x = y THEN v_txt := 'YES'; ELSE v_txt := 'NO'; END IF;
```

```
DECLARE
  v_num1  NUMBER := &sv_num1;
  v_num2  NUMBER := &sv_num2;
  v_total NUMBER;
BEGIN
  IF v_num1 > v_num2
  THEN
    DBMS_OUTPUT.PUT_LINE ('IF part of the outer IF');
    v_total := v_num1 - v_num2;
  ELSE
    DBMS_OUTPUT.PUT_LINE ('ELSE part of the outer IF');
    v_total := v_num1 + v_num2;

    IF v_total < 0
    THEN
      DBMS_OUTPUT.PUT_LINE ('Inner IF');
      v_total := v_total * (-1);
    END IF;
  END IF;
  DBMS_OUTPUT.PUT_LINE ('v_total = '||v_total);
END;
```

```
DECLARE
  v_letter CHAR(1) := '&sv_letter';
BEGIN
  IF (v_letter >= 'A' AND v_letter <= 'Z') OR
    (v_letter >= 'a' AND v_letter <= 'z')
  THEN
    DBMS_OUTPUT.PUT_LINE ('This is a letter');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('This is not a letter');
    IF v_letter BETWEEN '0' and '9'
    THEN
      DBMS_OUTPUT.PUT_LINE ('This is a number');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('This is not a number');
    END IF;
  END IF;
END;
```

```
(v_letter >= 'A' AND v_letter <= 'Z') OR  
(v_letter >= 'a' AND v_letter <= 'z')
```

```
(v_letter >= 'A' AND v_letter <= 'Z')
```

```
(v_letter >= 'a' AND v_letter <= 'z')
```

```
DECLARE
  v_var1 PLS_INTEGER := 100;
  v_var2 PLS_INTEGER := 200;
  v_var3 PLS_INTEGER := 300;
  v_var4 PLS_INTEGER := 400;
BEGIN
  IF v_var1 >= 100
  THEN
    IF v_var2 >= 200
    THEN
      IF v_var3 >= 300
      THEN
        IF v_var4 >= 400
        THEN
          DBMS_OUTPUT.PUT_LINE
            ('v_var1 = '|v_var1||', v_var2 = '|v_var2||
             ', v_var3 = '|v_var3||', v_var4 = '|v_var4);
        END IF;
      END IF;
    END IF;
  END IF;
END;
```

```
IF v_var1 >= 100 AND v_var2 >= 200 and v_var3 >= 300 AND v_var4 >= 400  
THEN  
    ...  
END IF;
```

```
CASE SELECTOR
WHEN EXPRESSION 1 THEN STATEMENT 1;
WHEN EXPRESSION 2 THEN STATEMENT 2;
-- 
WHEN EXPRESSION N THEN STATEMENT N;
ELSE STATEMENT N+1;
END CASE;
```

```
DECLARE
  v_num NUMBER := &sv_user_num;
BEGIN
  -- test if the number provided by the user is even
  IF MOD(v_num,2) = 0
  THEN
    DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
  ELSE
    DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END IF;
END;
```

```
DECLARE
  v_num      NUMBER := &sv_user_num;
  v_num_flag NUMBER;
BEGIN
  v_num_flag := MOD(v_num,2);

  -- test if the number provided by the user is even
CASE v_num_flag
  WHEN 0
    THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
  ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
END CASE;
END;
```

```
CASE
WHEN SEARCH CONDITION 1 THEN STATEMENT 1;
WHEN SEARCH CONDITION 2 THEN STATEMENT 2;
-- 
WHEN SEARCH CONDITION N THEN STATEMENT N;
ELSE STATEMENT N+1;
END CASE;
```

```
DECLARE
  v_num NUMBER := &sv_user_num;
BEGIN
  -- test if the number provided by the user is even
  CASE
    WHEN MOD(v_num,2) = 0
    THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
    ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END CASE;
END;
```

```
DECLARE
  v_num      NUMBER := &sv_num;
  v_num_flag NUMBER;
BEGIN
  CASE v_num_flag
    WHEN MOD(v_num, 2) = 0
    THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
    ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END CASE;
END;
```

```
ORA-06550: line 5, column 9:  
PLS-00615: type mismatch found at 'V_NUM_FLAG' between CASE operand and WHEN operands  
ORA-06550: line 5, column 4:  
PL/SQL: Statement ignored
```

```
DECLARE
  v_num      NUMBER := &sv_num;
  v_num_flag Boolean;
BEGIN
  CASE v_num_flag
    WHEN MOD(v_num, 2) = 0
    THEN
      DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
    ELSE
      DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
  END CASE;
END;
```

```
DECLARE
  v_final_grade  NUMBER := &sv_final_grade;
  v_letter_grade CHAR(1);
BEGIN
  CASE
    WHEN v_final_grade >= 60
    THEN
      v_letter_grade := 'D';
    WHEN v_final_grade >= 70
    THEN
      v_letter_grade := 'C';
    WHEN v_final_grade >= 80
    THEN
      v_letter_grade := 'B';
    WHEN v_final_grade >= 90
    THEN
      v_letter_grade := 'A';
    ELSE
      v_letter_grade := 'F';
  END CASE;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Final grade is: ||v_final_grade);
  DBMS_OUTPUT.PUT_LINE ('Letter grade is: ||v_letter_grade);
END;
```

```
DECLARE
  v_final_grade  NUMBER := &sv_final_grade;
  v_letter_grade CHAR(1);
BEGIN
  CASE
    WHEN v_final_grade >= 90
    THEN
      v_letter_grade := 'A';
    WHEN v_final_grade >= 80
    THEN
      v_letter_grade := 'B';
    WHEN v_final_grade >= 70
    THEN
      v_letter_grade := 'C';
    WHEN v_final_grade >= 60
    THEN
      v_letter_grade := 'D';
    ELSE
      v_letter_grade := 'F';
  END CASE;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Final grade is: ||v_final_grade);
  DBMS_OUTPUT.PUT_LINE ('Letter grade is: ||v_letter_grade);
END;
```

```
DECLARE
    v_num      NUMBER := &sv_user_num;
    v_num_flag NUMBER;
BEGIN
    v_num_flag := MOD(v_num,2);

    -- test if the number provided by the user is even
    CASE v_num_flag
        WHEN 0
        THEN
            DBMS_OUTPUT.PUT_LINE (v_num||' is even number');
        ELSE
            DBMS_OUTPUT.PUT_LINE (v_num||' is odd number');
    END CASE;
END;
```

```
DECLARE
  v_num      NUMBER := &sv_user_num;
  v_num_flag NUMBER;
  v_result   VARCHAR2(30);
BEGIN
  v_num_flag := MOD(v_num,2);

  -- test if the number provided by the user is even
  v_result := CASE v_num_flag
    WHEN 0
    THEN
      v_num||' is even number'
    ELSE
      v_num||' is odd number'
  END;
  DBMS_OUTPUT.PUT_LINE (v_result);
END;
```

```
DECLARE
  v_num      NUMBER := &sv_user_num;
  v_result VARCHAR2(30);
BEGIN
  -- test if the number provided by the user is even
  v_result := CASE
    WHEN MOD(v_num,2) = 0
    THEN
      v_num||' is even number'
    ELSE
      v_num||' is odd number'
    END;
  DBMS_OUTPUT.PUT_LINE (v_result);
END;
```

```
DECLARE
  v_course_no    NUMBER;
  v_description VARCHAR2(50);
  v_prereq       VARCHAR2(35);
BEGIN
  SELECT course_no
    ,description
    ,CASE
      WHEN prerequisite IS NULL
      THEN
        'No prerequisite course required'
      ELSE
        TO_CHAR(prerequisite)
      END prerequisite
  INTO v_course_no
    ,v_description
    ,v_prereq
  FROM course
 WHERE course_no = 20;

DBMS_OUTPUT.PUT_LINE ('Course:     '||v_course_no);
DBMS_OUTPUT.PUT_LINE ('Description: '||v_description);
DBMS_OUTPUT.PUT_LINE ('Prerequisite: '||v_prereq);
END;
```

Course: 20

Description: Intro to Information Systems

Prerequisite: No prerequisite course required

```
CASE
  WHEN prerequisite IS NULL
  THEN
    'No prerequisite course required'
  ELSE
    TO_CHAR(prerequisite)
END
```

```
ORA-06550: line 13, column 17:  
PL/SQL: ORA-00932: inconsistent datatypes: expected CHAR got NUMBER  
ORA-06550: line 6, column 4:  
PL/SQL: SQL Statement ignored
```

`NULLIF (EXPRESSION 1, EXPRESSION 2)`

```
CASE
WHEN EXPRESSION 1 = EXPRESSION 2 THEN NULL
ELSE EXPRESSION 1
END
```

```
DECLARE
  v_num      NUMBER := &sv_user_num;
  v_remainder NUMBER;
BEGIN
  -- calculate the remainder and if it is zero return NULL
  v_remainder := NULLIF(MOD(v_num, 2),0);
  DBMS_OUTPUT.PUT_LINE ('v_remainder: '||v_remainder);
END;
```

```
DECLARE
  v_remainder NUMBER;
BEGIN
  -- calculate the remainder and if it is zero return NULL
  v_remainder := NULLIF(NULL,0);
  DBMS_OUTPUT.PUT_LINE ('v_remainder: '||v_remainder);
END;
```

```
v_remainder := NULLIF(NULL,0); *  
ERROR at line 5:  
ORA-06550: line 5, column 26:  
PLS-00619: the first operand in the NULLIF expression must not be NULL  
ORA-06550: line 5, column 4:  
PL/SQL: Statement ignored
```

COALESCE (*EXPRESSION 1*, *EXPRESSION 2*, ..., *EXPRESSION N*)

```
NVL(EXPRESSION 1  
    ,NVL(EXPRESSION 2  
        ,NVL(EXPRESSION 3,...)  
    )  
)
```

COALESCE (*EXPRESSION 1*, *EXPRESSION 2*)

```
CASE
WHEN EXPRESSION 1 IS NOT NULL
THEN
  EXPRESSION 1
ELSE
  EXPRESSION 2
END
```

COALESCE (*EXPRESSION 1*, *EXPRESSION 2*, ..., *EXPRESSION N*)

```
CASE
  WHEN EXPRESSION 1 IS NOT NULL
  THEN
    EXPRESSION 1
  ELSE
    COALESCE (EXPRESSION 2, ..., EXPRESSION N)
END
```

```
CASE
WHEN EXPRESSION 1 IS NOT NULL
THEN
    EXPRESSION 1
WHEN EXPRESSION 2 IS NOT NULL
THEN
    EXPRESSION 2
...
ELSE
    EXPRESSION N
END
```

```
SELECT e.student_id
      ,e.section_id
      ,e.final_grade
      ,g.numeric_grade
      ,COALESCE(e.final_grade, g.numeric_grade, 0) grade
  FROM enrollment e
     ,grade g
 WHERE e.student_id = g.student_id
   AND e.section_id = g.section_id
   AND e.student_id = 102
   AND g.grade_type_code = 'FI';
```

STUDENT_ID	SECTION_ID	FINAL_GRADE	NUMERIC_GRADE	GRADE
102	86	(null)	85	85
102	89	92	92	92

```
NVL(e.final_grade, NVL(g.numeric_grade, 0))  
CASE  
  WHEN e.final_grade IS NOT NULL  
  THEN  
    e.final_grade  
  ELSE  
    COALESCE(g.numeric_grade, 0)  
END
```

```
CASE
WHEN e.final_grade IS NOT NULL
THEN
    e.final_grade
WHEN g.numeric_grade IS NOT NULL
THEN
    g.numeric_grade
ELSE
    0
END
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    -- if exit condition yields TRUE exit the loop
    IF v_counter = 5
    THEN
      EXIT;
    END IF;
  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;

    -- if exit condition yields TRUE exit the loop
    IF v_counter = 5
    THEN
      EXIT;
    END IF;

    DBMS_OUTPUT.PUT_LINE ('v_counter = ' || v_counter);
  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');

END;
```

```
DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
```

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Line 1');
    RETURN;
    DBMS_OUTPUT.PUT_LINE ('Line 2');
END;
```

```
DECLARE
    v_counter NUMBER := 0;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
        EXIT;
    END LOOP;
END;
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    -- if exit condition yields TRUE exit the loop
    EXIT WHEN v_counter = 5;
  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

```
DECLARE
  v_counter NUMBER := 5;
BEGIN
  WHILE v_counter < 5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    -- decrement the value of v_counter by one
    v_counter := v_counter - 1;
  END LOOP;
END;
```

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter < 5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    -- decrement the value of v_counter by one
    v_counter := v_counter - 1;
  END LOOP;
END;
```

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter < 5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    -- increment the value of v_counter by one
    v_counter := v_counter + 1;
  END LOOP;
END;
```

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter <= 5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);

    IF v_counter = 2
    THEN
      EXIT;
    END IF;

    v_counter := v_counter + 1;
  END LOOP;
END;
```

```
DECLARE
  v_counter NUMBER := 1;
BEGIN
  WHILE v_counter <= 2
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    v_counter := v_counter + 1;

    IF v_counter = 5
    THEN
      EXIT;
    END IF;
  END LOOP;
END;
```

```
FOR loop_counter IN [REVERSE] lower_limit..upper_limit
LOOP
  STATEMENT 1;
  STATEMENT 2;
  ...
  STATEMENT N;
END LOOP;
```

```
BEGIN
  FOR v_counter IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
END;
```

```
BEGIN
  FOR v_counter IN 1..5
  LOOP
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE ('v_counter = '|| v_counter);
  END LOOP;
END;
```

```
ORA-06550: line 4, column 7:  
PLS-00363: expression 'V_COUNTER' cannot be used as an assignment target  
ORA-06550: line 4, column 7:  
PL/SQL: Statement ignored
```

```
BEGIN
  FOR v_counter IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('Counter outside the loop is '||v_counter);
END;
```

```
('Counter outside the loop is'||v_counter);
*
ORA-06550: line 6, column 58:
PLS-00201: identifier 'V_COUNTER' must be declared
ORA-06550: line 6, column 4:
PL/SQL: Statement ignored
```

```
BEGIN
  FOR v_counter IN REVERSE 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
  END LOOP;
END;
```

```
FOR loop_counter IN lower_limit..upper_limit
LOOP
    STATEMENT 1;
    STATEMENT 2;
    IF EXIT CONDITION THEN
        EXIT;
    END IF;
END LOOP;
STATEMENT 3;
```

```
POR loop_counter IN lower_limit..upper_limit
LOOP
  STATEMENT 1;
  STATEMENT 2;
  EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;
```

```
BEGIN
  FOR v_counter IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    EXIT WHEN v_counter = 3;
  END LOOP;
END;
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE
      ('Before continue condition, v_counter = '||v_counter);

    -- if continue condition yields TRUE pass control to the first
    -- executable statement of the loop
    IF v_counter < 3
    THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('After continue condition, v_counter = '||v_counter);

    -- if exit condition yields TRUE exit the loop
    IF v_counter = 5
    THEN
      EXIT;
    END IF;

  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');
END;
```

```
Before continue condition, v_counter = 1
Before continue condition, v_counter = 2
Before continue condition, v_counter = 3
After continue condition, v_counter = 3
Before continue condition, v_counter = 4
After continue condition, v_counter = 4
Before continue condition, v_counter = 5
After continue condition, v_counter = 5
Done...
```

```
Before continue condition, v_counter = 1  
Before continue condition, v_counter = 2
```

```
Before continue condition, v_counter = 3
After continue condition,  v_counter = 3
Before continue condition, v_counter = 4
After continue condition,  v_counter = 4
Before continue condition, v_counter = 5
After continue condition,  v_counter = 5
```

```
DECLARE
  v_counter NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter = '||v_counter);
    CONTINUE;

    v_counter := v_counter + 1;
    EXIT WHEN v_counter = 5;
  END LOOP;
END;
```

```
LOOP
  STATEMENT 1;
  STATEMENT 2;
  CONTINUE WHEN CONTINUE CONDITION;

  EXIT WHEN EXIT CONDITION;
END LOOP;
STATEMENT 3;
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE
      ('Before continue condition, v_counter = ' || v_counter);

    -- if continue condition yields TRUE pass control to the first
    -- executable statement of the loop
    CONTINUE WHEN v_counter < 3;

    DBMS_OUTPUT.PUT_LINE
      ('After continue condition, v_counter = ' || v_counter);

    -- if exit condition yields TRUE exit the loop
    IF v_counter = 5
    THEN
      EXIT;
    END IF;

  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');

END;
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE
      ('Before continue condition, v_counter = '||v_counter);

    -- if continue condition yields TRUE pass control to the first
    -- executable statement of the loop
    CONTINUE WHEN v_counter > 3;

    DBMS_OUTPUT.PUT_LINE
      ('After continue condition, v_counter = '||v_counter);

    -- if exit condition yields TRUE exit the loop
    IF v_counter = 5
    THEN
      EXIT;
    END IF;

  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');

END;
```

```
DECLARE
  v_counter BINARY_INTEGER := 0;
BEGIN
  LOOP
    -- increment loop counter by one
    v_counter := v_counter + 1;

    -- if exit condition yields TRUE exit the loop
    IF v_counter = 5
    THEN
      EXIT;
    END IF;

    DBMS_OUTPUT.PUT_LINE
      ('Before continue condition, v_counter = ' || v_counter);

    -- if continue condition yields TRUE pass control to the first
    -- executable statement of the loop
    CONTINUE WHEN v_counter > 3;

    DBMS_OUTPUT.PUT_LINE
      ('After continue condition, v_counter = ' || v_counter);
  END LOOP;
  -- control resumes here
  DBMS_OUTPUT.PUT_LINE ('Done...');

END;
```

```
Before continue condition, v_counter = 1
After continue condition, v_counter = 1
Before continue condition, v_counter = 2
After continue condition, v_counter = 2
Before continue condition, v_counter = 3
After continue condition, v_counter = 3
Before continue condition, v_counter = 4
Done...
```

```
DECLARE
  v_counter1 BINARY_INTEGER := 0;
  v_counter2 BINARY_INTEGER;
BEGIN
  WHILE v_counter1 < 3
  LOOP
    DBMS_OUTPUT.PUT_LINE ('v_counter1: '||v_counter1);
    v_counter2 := 0;
    LOOP
      DBMS_OUTPUT.PUT_LINE (' v_counter2: '||v_counter2);
      v_counter2 := v_counter2 + 1;
      EXIT WHEN v_counter2 >= 2;
    END LOOP;
    v_counter1 := v_counter1 + 1;
  END LOOP;
END;
```

```
<<label_name>>
FOR loop_counter IN lower_limit..upper_limit
LOOP
    STATEMENT 1;
    ...
    STATEMENT N;
END LOOP label_name;
```

```
BEGIN
  <<outer_loop>>
  FOR i IN 1..3
  LOOP
    DBMS_OUTPUT.PUT_LINE ('i = '||i);
    <<inner_loop>>
    FOR j IN 1..2
    LOOP
      DBMS_OUTPUT.PUT_LINE ('j = '||j);
    END LOOP inner_loop;
  END LOOP outer_loop;
END;
```

```
BEGIN
  <<outer>>
  FOR v_counter IN 1..3
  LOOP
    <<inner>>
    FOR v_counter IN 1..2
    LOOP
      DBMS_OUTPUT.PUT_LINE ('outer.v_counter'||outer.v_counter);
      DBMS_OUTPUT.PUT_LINE ('inner.v_counter'||inner.v_counter);
    END LOOP inner;
  END LOOP outer;
END;
```

```
BEGIN
  <<outer>>
  FOR v_counter IN 1..3
  LOOP
    DBMS_OUTPUT.PUT_LINE ('outer.v_counter' || v_counter);
  <<inner>>
  FOR v_counter IN 1..2
  LOOP
    DBMS_OUTPUT.PUT_LINE (' outer.v_counter' || v_counter);
    DBMS_OUTPUT.PUT_LINE (' inner.v_counter' || v_counter);
  END LOOP inner;
END LOOP outer;
END;
```

```
DECLARE
  v_num1  INTEGER := &sv_num1;
  v_num2  INTEGER := &sv_num2;
  v_result NUMBER;
BEGIN
  v_result = v_num1 / v_num2;
  DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
END;
```

```
ORA-06550: line 6, column 13:
PLS-00103: Encountered the symbol "=" when expecting one of the following:
:= . ( @ % ;
The symbol ":=" was inserted before "=" to continue.
```

```
ORA-01476: divisor is equal to zero
ORA-06512: at line 6
01476. 00000 - "divisor is equal to zero"
```

```
EXCEPTION
WHEN EXCEPTION_NAME
THEN
  ERROR PROCESSING STATEMENTS;
```

```
DECLARE
  v_num1  INTEGER := &sv_num1;
  v_num2  INTEGER := &sv_num2;
  v_result NUMBER;
BEGIN
  v_result := v_num1 / v_num2;
  DBMS_OUTPUT.PUT_LINE ('v_result: '||v_result);
EXCEPTION
  WHEN ZERO_DIVIDE
  THEN
    DBMS_OUTPUT.PUT_LINE ('A number cannot be divided by zero.');
END;
```

A number cannot be divided by zero.

```
DECLARE
  ...
BEGIN
  EXECUTABLE STATEMENTS;
EXCEPTION
  WHEN EXCEPTION_NAME
  THEN
    ERROR PROCESSING STATEMENTS;
END;
```

```
DECLARE
  v_student_name VARCHAR2(50);
BEGIN
  SELECT first_name||' '||last_name
    INTO v_student_name
   FROM student
  WHERE student_id = 101;

  DBMS_OUTPUT.PUT_LINE ('Student name is'||v_student_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

ORA-01476: divisor is equal to zero

```
DECLARE
    v_student_id NUMBER      := &sv_student_id;
    v_enrolled   VARCHAR2(3) := 'NO';
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Check if the student is enrolled');
    SELECT 'YES'
        INTO v_enrolled
        FROM enrollment
        WHERE student_id = v_student_id;

    DBMS_OUTPUT.PUT_LINE ('The student is enrolled into one course');
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        DBMS_OUTPUT.PUT_LINE ('The student is not enrolled');

    WHEN TOO_MANY_ROWS
    THEN
        DBMS_OUTPUT.PUT_LINE ('The student is enrolled in multiple courses');
END;
```

Check if the student is enrolled
The student is enrolled in multiple courses

Check if the student is enrolled
The student is enrolled into one course

Check if the student is enrolled
The student is not enrolled

```
DECLARE
  v_instructor_id  NUMBER := &sv_instructor_id;
  v_instructor_name VARCHAR2(50);
BEGIN
  SELECT first_name||' '||last_name
    INTO v_instructor_name
   FROM instructor
  WHERE instructor_id = v_instructor_id;

  DBMS_OUTPUT.PUT_LINE ('Instructor name is'||v_instructor_name);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

```
DECLARE
  v_student_id NUMBER := &sv_student_id;
  v_name        VARCHAR2(30);
BEGIN
  SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
    INTO v_name
   FROM student
  WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('Student name is '||v_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

```
<<outer_block>>
DECLARE
  v_student_id NUMBER := &sv_student_id;
  v_name      VARCHAR2(30);
  v_total     NUMBER(1);

BEGIN
  SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
    INTO v_name
   FROM student
  WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('Student name is'||v_name);

<<inner_block>>
BEGIN
  SELECT COUNT(*)
    INTO v_total
   FROM enrollment
  WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('Student is registered for '||v_total|||' course(s)');
EXCEPTION
  WHEN VALUE_ERROR OR INVALID_NUMBER
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
  END;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such student');
  END;
```

```
<<outer_block>>
DECLARE
  v_student_id NUMBER := &sv_student_id;
  v_name      VARCHAR2(30);
  v_registered CHAR;

BEGIN
  SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
    INTO v_name
   FROM student
  WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('Student name is'||v_name);

<<inner_block>>
BEGIN
  SELECT 'Y'
    INTO v_registered
   FROM enrollment
  WHERE student_id = v_student_id;
  DBMS_OUTPUT.PUT_LINE ('Student is registered');
EXCEPTION
  WHEN VALUE_ERROR OR INVALID_NUMBER
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;

EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE ('There is no such student');
END;
```

Student name is Salewa Lindeman
There is no such student

```
DECLARE
  e_invalid_id EXCEPTION;
BEGIN
  --
EXCEPTION
  WHEN e_invalid_id
  THEN
    DBMS_OUTPUT.PUT_LINE ('An ID cannot be negative');
END;
```

```
DECLARE
    exception_name EXCEPTION;
BEGIN
    --
    IF CONDITION
        THEN
            RAISE exception_name;
    END IF;
    --
EXCEPTION
    WHEN exception_name
        THEN
            ERROR-PROCESSING STATEMENTS;
END;
```

```
DECLARE
  v_student_id      STUDENT.STUDENT_ID%TYPE := &sv_student_id;
  v_total_courses   NUMBER;
  e_invalid_id      EXCEPTION;
BEGIN
  IF v_student_id < 0
  THEN
    RAISE e_invalid_id;
  END IF;

  SELECT COUNT(*)
    INTO v_total_courses
    FROM enrollment
   WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('The student is registered for '||v_total_courses||' courses');
  DBMS_OUTPUT.PUT_LINE ('No exception has been raised');

EXCEPTION
  WHEN e_invalid_id
  THEN
    DBMS_OUTPUT.PUT_LINE ('An ID cannot be negative');
END;
```

The student is registered for 2 courses
No exception has been raised

```
DECLARE
  e_test_exception EXCEPTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Exception has not been raised');
  RAISE e_test_exception;
  DBMS_OUTPUT.PUT_LINE ('Exception has been raised');
EXCEPTION
  WHEN e_test_exception
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

Exception has not been raised
An error has occurred

```
<<outer_block>>
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Outer block');
  <<inner_block>>
DECLARE
  e_my_exception EXCEPTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Inner block');
EXCEPTION
  WHEN e_my_exception
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;

IF 10 > &sv_number
THEN
  RAISE e_my_exception;
END IF;
END;
```

```
ORA-06550: line 19, column 13:  
PLS-00201: identifier 'E_MY_EXCEPTION' must be declared  
ORA-06550: line 19, column 7:  
PL/SQL: Statement ignored
```

PLS-00201: identifier 'E_MY_EXCEPTION' must be declared

```
DECLARE
  v_test_var CHAR(3) := 'ABCDE';
BEGIN
  DBMS_OUTPUT.PUT_LINE ('This is a test');
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small  
ORA-06512: at line 2
```

```
<<outer_block>>
BEGIN
  <<inner_block>>
  DECLARE
    v_test_var CHAR(3):= 'ABCDE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('This is a test');
  EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
      DBMS_OUTPUT.PUT_LINE ('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the program');
END;
```

An error has occurred in the program

```
DECLARE
  v_test_var CHAR(3) := 'ABC';
BEGIN
  v_test_var := '1234';
  DBMS_OUTPUT.PUT_LINE ('v_test_var: '||v_test_var);
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    v_test_var := 'ABCD';
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 9
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
```

```
<<outer_block>>
BEGIN
  <<inner_block>>
  DECLARE
    v_test_var CHAR(3) := 'ABC';
  BEGIN
    v_test_var := '1234';
    DBMS_OUTPUT.PUT_LINE ('v_test_var: '||v_test_var);
  EXCEPTION
    WHEN INVALID_NUMBER OR VALUE_ERROR
    THEN
      v_test_var := 'ABCD';
      DBMS_OUTPUT.PUT_LINE ('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN INVALID_NUMBER OR VALUE_ERROR
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the program');
END;
```

An error has occurred in the program

```
<<outer_block>>
DECLARE
  e_exception1 EXCEPTION;
  e_exception2 EXCEPTION;
BEGIN
  <<inner_block>>
  BEGIN
    RAISE e_exception1;
  EXCEPTION
    WHEN e_exception1
    THEN
      RAISE e_exception2;
    WHEN e_exception2
    THEN
      DBMS_OUTPUT.PUT_LINE ('An error has occurred in the inner block');
  END;
EXCEPTION
  WHEN e_exception2
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred in the program');
END;
```

An error has occurred in the program

```
DECLARE
  e_exception1 EXCEPTION;
BEGIN
  RAISE e_exception1;
END;

ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 4
```

```
<<outer_block>>
DECLARE
  e_exception EXCEPTION;
BEGIN
  <<inner_block>>
  BEGIN
    RAISE e_exception;
  EXCEPTION
    WHEN e_exception
    THEN
      RAISE;
  END;
EXCEPTION
  WHEN e_exception
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

```
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 7
```

```
RAISE_APPLICATION_ERROR (error_number, error_message);
```

```
RAISE_APPLICATION_ERROR (error_number, error_message, keep_errors);
```

```
DECLARE
  v_student_id      STUDENT.STUDENT_ID%TYPE := &sv_student_id;
  v_total_courses  NUMBER;
  e_invalid_id     EXCEPTION;
BEGIN
  IF v_student_id < 0
  THEN
    RAISE e_invalid_id;
  END IF;
  SELECT COUNT(*)
    INTO v_total_courses
    FROM enrollment
   WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('The student is registered for'||v_total_courses||' courses');
  DBMS_OUTPUT.PUT_LINE ('No exception has been raised');
EXCEPTION
  WHEN e_invalid_id
  THEN
    DBMS_OUTPUT.PUT_LINE ('An ID cannot be negative');
END;
```

```
DECLARE
  v_student_id      STUDENT.STUDENT_ID%TYPE := &sv_student_id;
  v_total_courses NUMBER;
BEGIN
  IF v_student_id < 0
  THEN
    RAISE_APPLICATION_ERROR (-20000, 'An ID cannot be negative');
  END IF;

  SELECT COUNT(*)
    INTO v_total_courses
    FROM enrollment
   WHERE student_id = v_student_id;

  DBMS_OUTPUT.PUT_LINE ('The student is registered for '|| v_total_courses|| ' courses');
END;
```

ORA-20000: An ID cannot be negative
ORA-06512: at line 7

```
DECLARE
  v_student_id STUDENT.STUDENT_ID%TYPE := &sv_student_id;
  v_name        VARCHAR2(50);
BEGIN
  SELECT first_name||' '||last_name
    INTO v_name
   FROM student
  WHERE student_id = v_student_id;
  DBMS_OUTPUT.PUT_LINE (v_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RAISE_APPLICATION_ERROR (-20001, 'This ID is invalid');
END;
```

ORA-20001: This ID is invalid
ORA-06512: at line 13

```
DECLARE
exception_name EXCEPTION;
PRAGMA EXCEPTION_INIT (exception_name, error_code);
```

```
DECLARE
  v_zip ZIPCODE.ZIP%TYPE := '&sv_zip';
BEGIN
  DELETE FROM zipcode
  WHERE zip = v_zip;
  DBMS_OUTPUT.PUT_LINE ('Zip'||v_zip||' has been deleted');
  COMMIT;
END;
```

ORA-02292: integrity constraint (STUDENT.STU_ZIP_FK)violated - child record found
ORA-06512: at line 4

```
DECLARE
  v_zip          ZIPCODE.ZIP%TYPE := '&sv_zip';
  e_child_exists EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_child_exists, -2292);
BEGIN
  DELETE FROM zipcode
  WHERE zip = v_zip;

  DBMS_OUTPUT.PUT_LINE ('Zip '||v_zip||' has been deleted');
  COMMIT;
EXCEPTION
  WHEN e_child_exists
  THEN
    DBMS_OUTPUT.PUT_LINE ('Delete students for this ZIP code first');
END;
```

Delete students for this zipcode first

```
DECLARE
  v_zip  VARCHAR2(5) := '&sv_zip';
  v_city VARCHAR2(15);
  v_state CHAR(2);
BEGIN
  SELECT city, state
    INTO v_city, v_state
   FROM zipcode
  WHERE zip = v_zip;

  DBMS_OUTPUT.PUT_LINE (v_city||', '||v_state);

EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE ('An error has occurred');
END;
```

```
DECLARE
  v_zip      VARCHAR2(5) := '&sv_zip';
  v_city     VARCHAR2(15);
  v_state    CHAR(2);
  v_err_code NUMBER;
  v_err_msg  VARCHAR2(200);
BEGIN
  SELECT city, state
    INTO v_city, v_state
    FROM zipcode
   WHERE zip = v_zip;

  DBMS_OUTPUT.PUT_LINE (v_city||', '||v_state);

EXCEPTION
  WHEN OTHERS
  THEN
    v_err_code := SQLCODE;
    v_err_msg  := SUBSTR(SQLERRM, 1, 200);
    DBMS_OUTPUT.PUT_LINE ('Error code: '||v_err_code);
    DBMS_OUTPUT.PUT_LINE ('Error message: '||v_err_msg);
END;
```

Error code: -6502

Error message: ORA-06502: PL/SQL: numeric or value error: character string buffer too small

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Error code: '||SQLCODE);
  DBMS_OUTPUT.PUT_LINE ('Error message1: '||SQLERRM(SQLCODE));
  DBMS_OUTPUT.PUT_LINE ('Error message2: '||SQLERRM(100));
  DBMS_OUTPUT.PUT_LINE ('Error message3: '||SQLERRM(200));
  DBMS_OUTPUT.PUT_LINE ('Error message4: '||SQLERRM(-20000));
END;
```

```
Error code: 0
Error message1: ORA-0000: normal, successful completion
Error message2: ORA-01403: no data found
Error message3: -200: non-ORACLE exception
Error message4: ORA-20000:
```

```
SET SERVEROUTPUT ON
BEGIN
    UPDATE student
        SET first_name = 'B'
        WHERE first_name LIKE 'B%';
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
  v_first_name VARCHAR2(35);
  v_last_name VARCHAR2(35);
BEGIN
  SELECT first_name, last_name
    INTO v_first_name, v_last_name
    FROM student
   WHERE student_id = 123;
  DBMS_OUTPUT.PUT_LINE ('Student name: ' ||
    v_first_name||' '||v_last_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
      ('There is no student with student ID 123');
END;
```

```
CURSOR c_cursor_name IS select statement
```

```
DECLARE
  CURSOR C_MyCursor IS
    SELECT *
      FROM zipcode
     WHERE state = 'NY';
--<code would continue here with opening, fetching,
 and closing of the cursor>
```

```
<record_name> <table_name or cursor_name>%ROWTYPE
```

```
SET SERVEROUTPUT ON
DECLARE
  vr_student student%ROWTYPE;
BEGIN
  SELECT *
    INTO vr_student
    FROM student
   WHERE student_id = 156;
  DBMS_OUTPUT.PUT_LINE (vr_student.first_name||' '
    ||vr_student.last_name||' has an ID of 156');
EXCEPTION
  WHEN no_data_found
  THEN
    RAISE_APPLICATION_ERROR(-2001,'The Student ' ||
      'is not in the database');
END;
```

```
DECLARE
  vr_zip ZIPCODE%ROWTYPE;
  vr_instructor INSTRUCTOR%ROWTYPE;
```

```
SET SERVEROUTPUT ON;
DECLARE
  vr_zip ZIPCODE%ROWTYPE;
BEGIN
  SELECT *
    INTO vr_zip
    FROM zipcode
   WHERE rownum < 2;
  DBMS_OUTPUT.PUT_LINE('City: '||vr_zip.city);
  DBMS_OUTPUT.PUT_LINE('State: '||vr_zip.state);
  DBMS_OUTPUT.PUT_LINE('Zip: '||vr_zip.zip);
END;
```

City: Santurce

State: PR

Zip: 00914

PL/SQL procedure successfully completed.

```
DECLARE
  CURSOR c_student_name IS
    SELECT first_name, last_name
      FROM student;
  vr_student_name c_student_name%ROWTYPE;
```

```
DECLARE
  CURSOR c_student is
    SELECT first_name||' '||Last_name name
      FROM student;
  vr_student c_student%ROWTYPE;
```

```
DECLARE
  CURSOR c_student is
    SELECT first_name||' '||Last_name name
      FROM student;
  vr_student c_student%ROWTYPE;
BEGIN
  OPEN c_student;
```

```
FETCH cursor_name INTO PL/SQL variables;
```

```
FETCH cursor_name INTO PL/SQL record;
```

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR c_student_name IS
    SELECT first_name, last_name
      FROM student
     WHERE rownum <= 5;
  vr_student_name c_student_name%ROWTYPE;
BEGIN
  OPEN c_student_name;
  LOOP
    FETCH c_student_name INTO vr_student_name;
    EXIT WHEN c_student_name%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Student name: ' ||
      vr_student_name.first_name
      || ' ' || vr_student_name.last_name);
  END LOOP;
  CLOSE c_student_name;
END;
```

Student name: Austin V. Cadet
Student name: Frank M. Orent
Student name: Yvonne Winnicki
Student name: Mike Madej
Student name: Paula Valentine
PL/SQL procedure successfully completed.

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR c_student_name IS
    SELECT first_name, last_name
      FROM student
     WHERE rownum <= 5;
  vr_student_name c_student_name%ROWTYPE;
BEGIN
  OPEN c_student_name;
  LOOP
    FETCH c_student_name INTO vr_student_name;
    EXIT WHEN c_student_name%NOTFOUND;
  END LOOP;
  CLOSE c_student_name;
  DBMS_OUTPUT.PUT_LINE('Student name: ' ||
    vr_student_name.first_name||' '
    ||vr_student_name.last_name);
END;
```

```
type type_name IS RECORD
  (field_name 1 DATATYPE 1,
   field_name 2 DATATYPE 2,
   ...
   field_name N DATATYPE N);
record_name TYPE_NAME%ROWTYPE;
```

```
SET SERVEROUTPUT ON;
DECLARE
  -- declare user-defined type
  TYPE instructor_info IS RECORD
    (instructor_id instructor.instructor_id%TYPE,
     first_name instructor.first_name%TYPE,
     last_name instructor.last_name%TYPE,
     sections NUMBER(1));
  -- declare a record based on the type defined above
  rv_instructor instructor_info;
```

```
SET SERVEROUTPUT ON;
DECLARE
  TYPE instructor_info IS RECORD
    (first_name instructor.first_name%TYPE,
     last_name instructor.last_name%TYPE,
     sections NUMBER);
  rv_instructor instructor_info;
BEGIN
  SELECT RTRIM(i.first_name),
         RTRIM(i.last_name), COUNT(*)
    INTO rv_instructor
   FROM instructor i, section s
  WHERE i.instructor_id = s.instructor_id
    AND i.instructor_id = 102
  GROUP BY i.first_name, i.last_name;
  DBMS_OUTPUT.PUT_LINE('Instructor, ' ||
    rv_instructor.first_name ||
    ' || ' || rv_instructor.last_name ||
    ', teaches ' || rv_instructor.sections ||
    ' section(s)');
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
      ('There is no such instructor');
END;
```

Instructor, Tom Wojick, teaches 9 section(s)
PL/SQL procedure successfully completed.

```
    EXIT WHEN c_student%NOTFOUND;
END LOOP;
CLOSE c_student;
EXCEPTION
WHEN OTHERS
THEN
  IF c_student%ISOPEN
  THEN
    CLOSE c_student;
  END IF;
END;
```

```
SET SERVEROUTPUT ON;
DECLARE
  v_city zipcode.city%type;
BEGIN
  SELECT city
    Into v_city
   from zipcode
  Where zip = 07002;
  IF SQL%ROWCOUNT = 1
  THEN
    DBMS_OUTPUT.PUT_LINE(v_city ||' has a ' ||
      'ZIP code of 07002');
  ELSIF SQL%ROWCOUNT = 0
  THEN
    DBMS_OUTPUT.PUT_LINE('The ZIP code 07002 is ' ||
      ' not in the database');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Stop harassing me');
  END IF;
END;
```

Bayonne has a ZIP code of 07002
PL/SQL procedure successfully completed.

```
1> DECLARE
2>   v_sid    student.student_id%TYPE;
3>   CURSOR c_student IS
4>     SELECT student_id
5>       FROM student
6>      WHERE student_id < 110;
7> BEGIN
8>   OPEN c_student;
9>   LOOP
10>    FETCH c_student INTO v_sid;
11>    EXIT WHEN c_student%NOTFOUND;
12>    DBMS_OUTPUT.PUT_LINE('STUDENT ID : '||v_sid);
13> END LOOP;
14> CLOSE c_student;
15> EXCEPTION
16> WHEN OTHERS
17> THEN
18>   IF c_student%ISOPEN
19>     THEN
20>       CLOSE c_student;
21>     END IF;
22> END;
```

```
SET SERVEROUTPUT ON
DECLARE
  v_sid    student.student_id%TYPE;
  CURSOR c_student IS
    SELECT student_id
      FROM student
     WHERE student_id < 110;
BEGIN
  OPEN c_student;
  LOOP
    FETCH c_student INTO v_sid;
    IF c_student%FOUND THEN
      DBMS_OUTPUT.PUT_LINE
        ('Just FETCHED row '
         || TO_CHAR(c_student%ROWCOUNT) ||
         ' Student ID: '||v_sid);
    ELSE
      EXIT;
    END IF;
  END LOOP;
  CLOSE c_student;
EXCEPTION
  WHEN OTHERS
  THEN
    IF c_student%ISOPEN
    THEN
      CLOSE c_student;
    END IF;
END;
```

```
SET SERVEROUTPUT ON
DECLARE
  CURSOR c_student_enroll IS
    SELECT s.student_id, first_name, last_name,
           COUNT(*) enroll,
           (CASE
             WHEN count(*) = 1 Then ' class.'
             WHEN count(*) is null then
               ' no classes.'
             ELSE ' classes.'
           END) class
      FROM student s, enrollment e
     WHERE s.student_id = e.student_id
       AND s.student_id <110
   GROUP BY s.student_id, first_name, last_name;
  r_student_enroll  c_student_enroll%ROWTYPE;
BEGIN
  OPEN c_student_enroll;
  LOOP
    FETCH c_student_enroll INTO r_student_enroll;
    EXIT WHEN c_student_enroll%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Student INFO: ID ' ||
      r_student_enroll.student_id||' is ' ||
      r_student_enroll.first_name|| ' ' ||
      r_student_enroll.last_name||'
      is enrolled in '||r_student_enroll.enroll|||
      r_student_enroll.class);
  END LOOP;
  CLOSE c_student_enroll;
EXCEPTION
  WHEN OTHERS
  THEN
    IF c_student_enroll %ISOPEN
    THEN
      CLOSE c_student_enroll;
    END IF;
END;
```

```
create table table_log  
  (description VARCHAR2(250));
```

```
DECLARE
  CURSOR c_student IS
    SELECT student_id, last_name, first_name
      FROM student
     WHERE student_id < 110;
BEGIN
  FOR r_student IN c_student
  LOOP
    INSERT INTO table_log
      VALUES(r_student.last_name);
  END LOOP;
END;
SELECT * from table_log;
```

```
DECLARE
  CURSOR c_group_discount IS
    SELECT DISTINCT s.course_no
      FROM section s, enrollment e
     WHERE s.section_id = e.section_id
   GROUP BY s.course_no, e.section_id, s.section_id
  HAVING COUNT(*)>=8;
BEGIN
  FOR r_group_discount IN c_group_discount  LOOP
    UPDATE course
      SET cost = cost * .95
    WHERE course_no = r_group_discount.course_no;
  END LOOP;
  COMMIT;
END;
```

```
SET SERVEROUTPUT ON
1  DECLARE
2    v_zip zipcode.zip%TYPE;
3    v_student_flag CHAR;
4    CURSOR c_zip IS
5      SELECT zip, city, state
6        FROM zipcode
7       WHERE state = 'CT';
8    CURSOR c_student IS
9      SELECT first_name, last_name
10        FROM student
11       WHERE zip = v_zip;
12  BEGIN
13    FOR r_zip IN c_zip
14    LOOP
15      v_student_flag := 'N';
16      v_zip := r_zip.zip;
17      DBMS_OUTPUT.PUT_LINE(CHR(10));
18      DBMS_OUTPUT.PUT_LINE('Students living in ' ||
19                           r_zip.city);
20      FOR r_student in c_student
21      LOOP
22        DBMS_OUTPUT.PUT_LINE(
23          r_student.first_name ||
24          ' ' || r_student.last_name);
25        v_student_flag := 'Y';
26      END LOOP;
27      IF v_student_flag = 'N'
28      THEN
29        DBMS_OUTPUT.PUT_LINE
30          ('No students for this ZIP code');
31      END IF;
32    END LOOP;
33  END;
```

```
SET SERVEROUTPUT ON
DECLARE
    v_sid student.student_id%TYPE;
    CURSOR c_student IS
        SELECT student_id, first_name, last_name
        FROM student
        WHERE student_id < 110;
    CURSOR c_course IS
        SELECT c.course_no, c.description
        FROM course c, section s, enrollment e
        WHERE c.course_no = s.course_no
        AND s.section_id = e.section_id
        AND e.student_id = v_sid;
BEGIN
    FOR r_student IN c_student
    LOOP
        v_sid := r_student.student_id;
        DBMS_OUTPUT.PUT_LINE(chr(10));
        DBMS_OUTPUT.PUT_LINE(' The Student ' ||
            r_student.student_id||' '|||
            r_student.first_name||' '|||
            r_student.last_name);
        DBMS_OUTPUT.PUT_LINE(' is enrolled in the ' ||
            'following courses: ');
        FOR r_course IN c_course
        LOOP
            DBMS_OUTPUT.PUT_LINE(r_course.course_no|||
                ' '|||r_course.description);
        END LOOP;
    END LOOP;
END;
```

```
SET SERVEROUTPUT ON
DECLARE
  v_amount course.cost%TYPE;
  v_instructor_id instructor.instructor_id%TYPE;
  CURSOR c_inst IS
    SELECT first_name, last_name, instructor_id
      FROM instructor;
  CURSOR c_cost IS
    SELECT c.cost
      FROM course c, section s, enrollment e
     WHERE s.instructor_id = v_instructor_id
       AND c.course_no = s.course_no
       AND s.section_id = e.section_id;
BEGIN
  FOR r_inst IN c_inst
  LOOP
    v_instructor_id := r_inst.instructor_id;
    v_amount := 0;
    DBMS_OUTPUT.PUT_LINE(
      'Amount generated by instructor ' ||
      r_inst.first_name||' '||r_inst.last_name
      ||' is');
    FOR r_cost IN c_cost
    LOOP
      v_amount := v_amount + NVL(r_cost.cost, 0);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE
      ('-'||TO_CHAR(v_amount,'$999,999'));
  END LOOP;
END;
```

Amount generated by instructor Fernand Hanks is
\$49,110
Amount generated by instructor Tom Wojick is
\$24,582
Amount generated by instructor Nina Schorin is
\$43,319
Amount generated by instructor Gary Pertez is
\$29,317
Amount generated by instructor Anita Morris is
\$18,662
Amount generated by instructor Todd Smythe is
\$21,092
Amount generated by instructor Marilyn Frantzen is
\$34,311
Amount generated by instructor Charles Lowry is
\$37,512
Amount generated by instructor Rick Chow is
\$0
Amount generated by instructor Irene Willig is
\$0

```
CURSOR c_zip (p_state IN zipcode.state%TYPE) IS
  SELECT zip, city, state
    FROM zipcode
   WHERE state = p_state;
```

```
DECLARE
  CURSOR c_zip (p_state IN zipcode.state%TYPE) IS
    SELECT zip, city, state
      FROM zipcode
     WHERE state = p_state
BEGIN
  FOR r_zip IN c_zip('NJ')
  LOOP
    DBMS_OUTPUT.PUT_LINE(r_zip.city|||
      ' '||r_zip.zip');
  END LOOP;
END;
```

```
SET SERVEROUTPUT ON
1  DECLARE
2    CURSOR c_student IS
3      SELECT first_name, last_name, student_id
4        FROM student
5       WHERE last_name LIKE 'J%';
6  CURSOR c_course
7    (i_student_id IN student.student_id%TYPE)
8  IS
9    SELECT c.description, s.section_id sec_id
10      FROM course c, section s, enrollment e
11     WHERE e.student_id = i_student_id
12     AND c.course_no = s.course_no
13     AND s.section_id = e.section_id;
14 CURSOR c_grade(i_section_id IN section.section_id%TYPE,
15                  i_student_id IN student.student_id%TYPE)
16
17    IS
18    SELECT gt.description grd_desc,
19          TO_CHAR
20            (AVG(g.numeric_grade), '999.99') num_grd
21    FROM enrollment e,
22          grade g, grade_type gt
23   WHERE e.section_id = i_section_id
24     AND e.student_id = g.student_id
25     AND e.student_id = i_student_id
26     AND e.section_id = g.section_id
27     AND g.grade_type_code = gt.grade_type_code
28   GROUP BY gt.description ;
29 BEGIN
30   FOR r_student IN c_student
31   LOOP
32     DBMS_OUTPUT.PUT_LINE(CHR(10));
33     DBMS_OUTPUT.PUT_LINE(r_student.first_name ||
34                           ' '||r_student.last_name);
35   FOR r_course IN c_course(r_student.student_id)
36   LOOP
37     DBMS_OUTPUT.PUT_LINE ('Grades for course :'|||
38                           r_course.description);
39   FOR r_grade IN c_grade(r_course.sec_id,
40                         r_student.student_id)
41   LOOP
42     DBMS_OUTPUT.PUT_LINE(r_grade.num_grd|||
43                           ' '||r_grade.grd_desc);
44   END LOOP;
45   END LOOP;
46 END;
```

```
DECLARE
  CURSOR c_course IS
    SELECT course_no, cost
      FROM course FOR UPDATE;
BEGIN
  FOR r_course IN c_course
  LOOP
    IF r_course.cost < 2500
    THEN
      UPDATE course
        SET cost = r_course.cost + 10
          WHERE course_no = r_course.course_no;
    END IF;
  END LOOP;
END;
```

```
DECLARE
  CURSOR c_grade(
    i_student_id IN enrollment.student_id%TYPE,
    i_section_id IN enrollment.section_id%TYPE)
  IS
    SELECT final_grade
      FROM enrollment
     WHERE student_id = i_student_id
       AND section_id = i_section_id
     FOR UPDATE;
CURSOR c_enrollment IS
  SELECT e.student_id, e.section_id
    FROM enrollment e, section s
   WHERE s.course_no = 135
     AND e.section_id = s.section_id;
BEGIN
  FOR r_enroll IN c_enrollment
  LOOP
    FOR r_grade IN c_grade(r_enroll.student_id,
                           r_enroll.section_id)
    LOOP
      UPDATE enrollment
        SET final_grade = 90
       WHERE student_id = r_enroll.student_id
         AND section_id = r_enroll.section_id;
    END LOOP;
  END LOOP;
END;
```

```
DECLARE
  CURSOR c_stud_zip IS
    SELECT s.student_id, z.city
      FROM student s, zipcode z
     WHERE z.city = 'Brooklyn'
       AND s.zip = z.zip
    FOR UPDATE OF phone;
BEGIN
  FOR r_stud_zip IN c_stud_zip
  LOOP
    UPDATE student
      SET phone = '718'||SUBSTR(phone,4)
     WHERE student_id = r_stud_zip.student_id;
  END LOOP;
END;
```

```
DECLARE
  CURSOR c_stud_zip IS
    SELECT s.student_id, z.city
      FROM student s, zipcode z
     WHERE z.city = 'Brooklyn'
       AND s.zip = z.zip
    FOR UPDATE OF phone;
BEGIN
  FOR r_stud_zip IN c_stud_zip
  LOOP
    DBMS_OUTPUT.PUT_LINE(r_stud_zip.student_id);
    UPDATE student
      SET phone = '718'||SUBSTR(phone,4)
     WHERE CURRENT OF c_stud_zip;
  END LOOP;
END;
```

```
CREATE [OR REPLACE] [EDITIONABLE|NONEDITIONABLE] TRIGGER trigger_name
{BEFORE|AFTER} triggering_event ON table_name
[FOR EACH ROW]
[FOLLOWS|PRECEDES another_trigger]
[ENABLE/DISABLE]
[WHEN condition]
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

ORA-04095: trigger 'STUDENT_BI' already exists on another table, cannot replace it

```
ALTER TRIGGER trigger_name DISABLE;
```

```
ALTER TRIGGER trigger_name ENABLE;
```

```
CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON STUDENT
FOR EACH ROW
BEGIN
    :NEW.student_id      := STUDENT_ID_SEQ.NEXTVAL;
    :NEW.created_by     := USER;
    :NEW.created_date   := SYSDATE;
    :NEW.modified_by    := USER;
    :NEW.modified_date  := SYSDATE;
END;
```

```
CREATE OR REPLACE TRIGGER student_bi
--  
DECLARE
  v_student_id STUDENT.STUDENT_ID%TYPE;
BEGIN
  SELECT STUDENT_ID_SEQ.NEXTVAL
    INTO v_student_id
   FROM dual;
--  
END;
```

```
INSERT INTO STUDENT
(student_id, first_name, last_name, zip, registration_date,
created_by, created_date, modified_by, modified_date)
VALUES
(STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '00914', SYSDATE,
USER, SYSDATE, USER, SYSDATE);
```

```
INSERT INTO STUDENT
(first_name, last_name, zip, registration_date)
VALUES
('John', 'Smith', '00914', SYSDATE);
```

```
CREATE OR REPLACE TRIGGER instructor_aud
AFTER UPDATE OR DELETE ON INSTRUCTOR
DECLARE
    v_trans_type VARCHAR2(10);
BEGIN
    v_trans_type := CASE
        WHEN UPDATING THEN 'UPDATE'
        WHEN DELETING THEN 'DELETE'
    END;
    INSERT INTO audit_trail
    (TABLE_NAME, TRANSACTION_NAME, TRANSACTION_USER, TRANSACTION_DATE)
    VALUES
    ('INSTRUCTOR', v_trans_type, USER, SYSDATE);
END;
```

```
SELECT *
  FROM audit_trail;

TABLE_NAME TRANSACTION_NAME TRANSACTION_USER TRANSACTION_DATE
----- -----
INSTRUCTOR   UPDATE          STUDENT           05/07/2014
```

```
ROLLBACK;

SELECT *
  FROM audit_trail;

TABLE_NAME  TRANSACTION_NAME  TRANSACTION_USER  TRANSACTION_DATE
-----  -----  -----  -----
```

```
DECLARE  
PRAGMA AUTONOMOUS_TRANSACTION;
```

```
CREATE OR REPLACE TRIGGER instructor_aud
AFTER UPDATE OR DELETE ON INSTRUCTOR
DECLARE
    v_trans_type VARCHAR2(10);
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    v_trans_type := CASE
        WHEN UPDATING THEN 'UPDATE'
        WHEN DELETING THEN 'DELETE'
    END;
    INSERT INTO audit_trail
    (TABLE_NAME, TRANSACTION_NAME, TRANSACTION_USER, TRANSACTION_DATE)
    VALUES
    ('INSTRUCTOR', v_trans_type, USER, SYSDATE);
    COMMIT;
END;
```

```
CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
```

```
--
```

```
CREATE OR REPLACE TRIGGER enrollment_ad  
AFTER DELETE ON ENROLLMENT
```

```
--
```

```
CREATE OR REPLACE TRIGGER instructor_biud
BEFORE INSERT OR UPDATE OR DELETE ON INSTRUCTOR
DECLARE
  v_day VARCHAR2(10);
BEGIN
  v_day := RTRIM(TO_CHAR(SYSDATE, 'DAY'));
  IF v_day LIKE ('S%')
  THEN
    RAISE_APPLICATION_ERROR (-20000, 'A table cannot be modified during off hours');
  END IF;
END;
```

```
update INSTRUCTOR
*
ERROR at line 1:
ORA-20000: A table cannot be modified during off hours
ORA-06512: at "STUDENT.INSTRUCTOR_BIUD", line 8
ORA-04088: error during execution of trigger 'STUDENT.INSTRUCTOR_BIUD'
```

```
CREATE VIEW course_cost
AS
  SELECT course_no, description, cost
    FROM course;
```

```
SELECT *  
  FROM course_cost  
 WHERE course_no = 450;  
  
COURSE_NO      DESCRIPTION          COST  
-----  
450           DB Programming in Java    2000  
SELECT course_no, cost  
  FROM course  
 WHERE course_no = 450;  
  
COURSE_NO      COST  
-----  
450           2000
```

```
CREATE VIEW instructor_summary_view
AS
  SELECT i.instructor_id, COUNT(s.section_id) total_courses
    FROM instructor i
    LEFT OUTER JOIN section s
      ON (i.instructor_id = s.instructor_id)
   GROUP BY i.instructor_id;
```

```
DELETE FROM instructor_summary_view  
WHERE instructor_id = 109;
```

ORA-01732: data manipulation operation not legal on this view
01732. 00000 - "data manipulation operation not legal on this view"

```
CREATE OR REPLACE TRIGGER instructor_summary_del
INSTEAD OF DELETE ON instructor_summary_view
FOR EACH ROW
BEGIN
    DELETE FROM instructor
    WHERE instructor_id = :OLD.INSTRUCTOR_ID;
END;
```

```
DELETE FROM instructor_summary_view  
WHERE instructor_id = 109;
```

```
1 row deleted.
```

```
DELETE FROM instructor_summary_view  
WHERE instructor_id = 101;
```

```
ORA-02292: integrity constraint (STUDENT.SECT_INST_FK) violated - child record found
ORA-06512: at "STUDENT.INSTRUCTOR_SUMMARY_DEL", line 2
ORA-04088: error during execution of trigger 'STUDENT.INSTRUCTOR_SUMMARY_DEL'
```

```
CREATE OR REPLACE TRIGGER instructor_summary_del
INSTEAD OF DELETE ON instructor_summary_view
FOR EACH ROW
BEGIN
    DELETE FROM section
    WHERE instructor_id = :OLD.INSTRUCTOR_ID;
    DELETE FROM instructor
    WHERE instructor_id = :OLD.INSTRUCTOR_ID;
END;
```

```
DELETE FROM instructor_summary_view  
WHERE instructor_id = 101;
```

```
ORA-02292: integrity constraint (STUDENT.GRTW_SECT_FK) violated - child record found  
ORA-06512: at "STUDENT.INSTRUCTOR_SUMMARY_DEL", line 2  
ORA-04088: error during execution of trigger 'STUDENT.INSTRUCTOR_SUMMARY_DEL'
```

```
CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
DECLARE
    v_total NUMBER;
    v_name  VARCHAR2(30);
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM section -- SECTION is MUTATING
        WHERE instructor_id = :NEW.instructor_id;

    -- check if the current instructor is overbooked
    IF v_total >= 10
    THEN
        SELECT first_name||' '||last_name
            INTO v_name
            FROM instructor
            WHERE instructor_id = :NEW.instructor_id;

        RAISE_APPLICATION_ERROR (-20000, 'Instructor, ''||v_name||'', is overbooked');
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        RAISE_APPLICATION_ERROR (-20001, 'This is not a valid instructor');
END;
```

```
ORA-04091: table STUDENT.SECTION is mutating, trigger/function may not see it
ORA-06512: at "STUDENT.SECTION_BIU", line 5
ORA-04088: error during execution of trigger 'STUDENT.SECTION_BIU'
```

```
SELECT COUNT(*)
  INTO v_total
  FROM section
 WHERE instructor_id = :NEW.INSTRUCTOR_ID;
```

```
CREATE OR REPLACE PACKAGE instructor_adm
AS
  g_instructor_id    instructor.instructor_id$TYPE;
  g_instructor_name varchar2(50);
END;
```

```
CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
BEGIN
  IF :NEW.instructor_id IS NOT NULL
  THEN
    BEGIN
      -- Assign new instructor ID to the global variable
      instructor_adm.g_instructor_id := :NEW.INSTRUCTOR_ID;

      SELECT first_name||' '||last_name
        INTO instructor_adm.g_instructor_name
        FROM instructor
       WHERE instructor_id = instructor_adm.g_instructor_id;

      EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
          RAISE_APPLICATION_ERROR (-20001, 'This is not a valid instructor');
        END;
    END IF;
END;
```

```
CREATE OR REPLACE TRIGGER section_aiu
AFTER INSERT OR UPDATE ON section
DECLARE
    v_total INTEGER;
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM section
       WHERE instructor_id = instructor_adm.g_instructor_id;
    -- check if the current instructor is overbooked
    IF v_total >= 10
    THEN
        RAISE_APPLICATION_ERROR
            (-20000, 'Instructor,'||instructor_adm.g_instructor_name||', is overbooked');
    END IF;
END;
```

ORA-20000: Instructor, Fernand Hanks, is overbooked
ORA-06512: at "STUDENT.SECTION_AIU", line 12
ORA-04088: error during execution of trigger 'STUDENT.SECTION_AIU'

```
CREATE [OR REPLACE] TRIGGER trigger_name
triggering_event ON table_name
COMPOUND TRIGGER
```

Declaration Statements

```
BEFORE STATEMENT IS
BEGIN
```

Executable statements

```
END BEFORE STATEMENT;
```

```
BEFORE EACH ROW IS
BEGIN
```

Executable statements

```
END BEFORE EACH ROW;
```

```
AFTER EACH ROW IS
BEGIN
```

Executable statements

```
END AFTER EACH ROW;
```

```
AFTER STATEMENT IS
BEGIN
```

Executable statements

```
END AFTER STATEMENT;
```

```
END;
```

```
CREATE OR REPLACE TRIGGER student_compound
FOR INSERT ON STUDENT
COMPOUND TRIGGER

-- Declaration section
v_day  VARCHAR2(10);

BEFORE STATEMENT IS
BEGIN
  v_day := RTRIM(TO_CHAR(SYSDATE, 'DAY'));
  IF v_day LIKE ('S%')
  THEN
    RAISE_APPLICATION_ERROR
      (-20000, 'A table cannot be modified during off hours');
  END IF;
END BEFORE STATEMENT;

BEFORE EACH ROW IS
BEGIN
  :NEW.student_id    := STUDENT_ID_SEQ.NEXTVAL;
  :NEW.created_by   := USER;
  :NEW.created_date := SYSDATE;
  :NEW.modified_by  := USER;
  :NEW.modified_date := SYSDATE;
END BEFORE EACH ROW;

END;
```

PLS-00363: expression 'NEW.CREATED_BY' cannot be used as an assignment target
PLS-00679: trigger binds not allowed in before/after statement section
PL/SQL: Statement ignored

```
CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
DECLARE
    v_total NUMBER;
    v_name  VARCHAR2(30);
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM section -- SECTION is MUTATING
        WHERE instructor_id = :NEW.instructor_id;

    -- check if the current instructor is overbooked
    IF v_total >= 10
    THEN
        SELECT first_name||' '||last_name
            INTO v_name
            FROM instructor
            WHERE instructor_id = :NEW.instructor_id;

        RAISE_APPLICATION_ERROR (-20000, 'Instructor, ''||v_name||'', is overbooked');
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
        RAISE_APPLICATION_ERROR
            (-20001, 'This is not a valid instructor');
END;
```

```
CREATE OR REPLACE PACKAGE instructor_adm
AS g_instructor_id  instructor.instructor_id%TYPE;
g_instructor_name varchar2(50);
END;
```

```
CREATE OR REPLACE TRIGGER section_biu
BEFORE INSERT OR UPDATE ON section
FOR EACH ROW
BEGIN
  IF :NEW.instructor_id IS NOT NULL
  THEN
    BEGIN
      instructor_adm.g_instructor_id := :NEW.INSTRUCTOR_ID;

      SELECT first_name||' '||last_name
        INTO instructor_adm.g_instructor_name
       FROM instructor
      WHERE instructor_id = instructor_adm.g_instructor_id;
    EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
        RAISE_APPLICATION_ERROR (-20001, 'This is not a valid instructor');
    END;
  END IF;
END;
```

```
CREATE OR REPLACE TRIGGER section_aiu
AFTER INSERT OR UPDATE ON section
DECLARE
    v_total INTEGER;
BEGIN
    SELECT COUNT(*)
        INTO v_total
        FROM section
       WHERE instructor_id = instructor_adm.v_instructor_id;

    -- check if the current instructor is overbooked
    IF v_total >= 10 THEN
        RAISE_APPLICATION_ERROR
            (-20000, 'Instructor,'||instructor_adm.v_instructor_name||
             ', is overbooked');
    END IF;
END;
```

```
CREATE OR REPLACE TRIGGER section_compound
FOR INSERT OR UPDATE ON SECTION
COMPOUND TRIGGER

    -- Declaration Section
    v_instructor_id    INSTRUCTOR.INSTRUCTOR_ID%TYPE;
    v_instructor_name VARCHAR2(50);
    v_total            INTEGER;

BEFORE EACH ROW IS
BEGIN
    IF :NEW.instructor_id IS NOT NULL
    THEN
        BEGIN
            v_instructor_id := :NEW.instructor_id;

            SELECT first_name||' '||last_name
              INTO v_instructor_name
             FROM instructor
            WHERE instructor_id = v_instructor_id;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR
            (-20001, 'This is not a valid instructor');
    END;
END IF;
END BEFORE EACH ROW;

AFTER STATEMENT IS
BEGIN
    SELECT COUNT(*)
      INTO v_total
     FROM section
    WHERE instructor_id = v_instructor_id;

    -- check if the current instructor is overbooked
    IF v_total >= 10
    THEN
        RAISE_APPLICATION_ERROR
            (-20000, 'Instructor,'||v_instructor_name||', is overbooked');
    END IF;
END AFTER STATEMENT;

END;
```

ORA-20000: Instructor, Fernand Hanks, is overbooked
ORA-06512: at "STUDENT.SECTION_COMPOUND", line 38
ORA-04088: error during execution of trigger 'STUDENT.SECTION_COMPOUND'

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
  INDEX BY index_type;
table_name TYPE_NAME;
```

```
DECLARE
  TYPE last_name_type IS TABLE OF student.last_name%TYPE
    INDEX BY PLS_INTEGER;
  last_name_tab last_name_type;
```

```
DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;
  TYPE last_name_type IS TABLE OF student.last_name%TYPE
    INDEX BY PLS_INTEGER;
  last_name_tab last_name_type;

  v_index PLS_INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_tab(v_index) := name_rec.last_name;
    DBMS_OUTPUT.PUT_LINE ('last_name('||v_index||') : '||last_name_tab(v_index));
  END LOOP;
END;
```

```
DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;

  TYPE last_name_type IS TABLE OF student.last_name%TYPE
    INDEX BY PLS_INTEGER;
  last_name_tab last_name_type;

  v_index PLS_INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_tab(v_index) := name_rec.last_name;
    DBMS_OUTPUT.PUT_LINE ('last_name('|| v_index ||') : '||last_name_tab(v_index));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('last_name(10) : '||last_name_tab(10));
END;
```

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
table_name TYPE_NAME;
```

```
DECLARE
  TYPE last_name_type IS TABLE OF student.last_name%TYPE;
  last_name_tab last_name_type;
```

```
CREATE OR REPLACE TYPE last_name_type AS TABLE OF VARCHAR2(30);
/
CREATE OR REPLACE TYPE last_name_table AS TABLE OF last_name_type;
/
```

```
DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;

  TYPE last_name_type IS TABLE OF student.last_name%TYPE;
  last_name_tab last_name_type;

  v_index PLS_INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_tab(v_index) := name_rec.last_name;
    DBMS_OUTPUT.PUT_LINE ('last_name('|| v_index ||'): '||last_name_tab(v_index));
  END LOOP;
END;
```

ORA-06531: Reference to uninitialized collection
ORA-06512: at line 15

```
last_name_tab := last_name_type('Rosenzweig', 'Rakhimov');
```

```
last_name_tab := last_name_type();
```

```
DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;

  TYPE last_name_type IS TABLE OF student.last_name%TYPE;
  last_name_tab last_name_type := last_name_type();

  v_index PLS_INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_tab.EXTEND;
    last_name_tab(v_index) := name_rec.last_name;

    DBMS_OUTPUT.PUT_LINE ('last_name('||v_index||'): '||last_name_tab(v_index));
  END LOOP;
END;
```

```
DECLARE
  TYPE integer_type IS TABLE OF INTEGER;
  integer_tab integer_type;

  v_index PLS_INTEGER := 1;
BEGIN
  DBMS_OUTPUT.PUT_LINE (integer_tab(v_index));
END;

ORA-06531: Reference to uninitialized collection
ORA-06512: at line 7
```

```
DECLARE
  TYPE integer_type IS TABLE OF INTEGER;
  integer_tab integer_type := integer_type();
  v_index PLS_INTEGER := 1;
BEGIN
  DBMS_OUTPUT.PUT_LINE (integer_tab(v_index));
END;
```

```
ORA-06533: Subscript beyond count
ORA-06512: at line 7
```

```

DECLARE
  TYPE index_by_type IS TABLE OF NUMBER
    INDEX BY PLS_INTEGER;
  index_by_table index_by_type;

  TYPE nested_type IS TABLE OF NUMBER;
  nested_table nested_type := nested_type(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

BEGIN
  -- Populate associative array
  FOR i IN 1..10
  LOOP
    index_by_table(i) := i;
  END LOOP;

  -- Check if the associative array has third element
  IF index_by_table.EXISTS(3)
  THEN
    DBMS_OUTPUT.PUT_LINE ('index_by_table(3) = '||index_by_table(3));
  END IF;

  -- Delete 10th element from associative array
  index_by_table.DELETE(10);
  -- Delete 10th element from nested table
  nested_table.DELETE(10);
  -- Delete elements 1 through 3 from nested table
  nested_table.DELETE(1,3);

  -- Get element counts for associative array and nested table
  DBMS_OUTPUT.PUT_LINE ('index_by_table.COUNT = '||index_by_table.COUNT);
  DBMS_OUTPUT.PUT_LINE ('nested_table.COUNT      = '||nested_table.COUNT);
  -- Get first and last indexes of the associative array
  -- and nested table
  DBMS_OUTPUT.PUT_LINE ('index_by_table.FIRST = '||index_by_table.FIRST);
  DBMS_OUTPUT.PUT_LINE ('index_by_table.LAST   = '||index_by_table.LAST);
  DBMS_OUTPUT.PUT_LINE ('nested_table.FIRST  = '||nested_table.FIRST);
  DBMS_OUTPUT.PUT_LINE ('nested_table.LAST   = '||nested_table.LAST);

  -- Get indexes that precede and succeed 2nd indexes of the associative array
  -- and nested table
  DBMS_OUTPUT.PUT_LINE ('index_by_table.PRIOR(2) = '||index_by_table.PRIOR(2));
  DBMS_OUTPUT.PUT_LINE ('index_by_table.NEXT(2)  = '||index_by_table.NEXT(2));
  DBMS_OUTPUT.PUT_LINE ('nested_table.PRIOR(2) = '||nested_table.PRIOR(2));
  DBMS_OUTPUT.PUT_LINE ('nested_table.NEXT(2)  = '||nested_table.NEXT(2));

  -- Delete last two elements of the nested table
  nested_table.TRIM(2);
  -- Delete last element of the nested table
  nested_table.TRIM;

  -- Get last index of the nested table
  DBMS_OUTPUT.PUT_LINE ('nested_table.LAST = '||nested_table.LAST);
END;

```

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit) OF element_type [NOT NULL];
varray_name TYPE_NAME;
```

```
DECLARE
  TYPE last_name_type IS VARRAY(10) OF student.last_name%TYPE;
  last_name_varray last_name_type;
```

```
DECLARE
  CURSOR name_cur IS
    SELECT last_name
      FROM student
     WHERE rownum < 10;

  TYPE last_name_type IS VARRAY(10) OF student.last_name%TYPE;
  last_name_varray last_name_type := last_name_type();

  v_index PLS_INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    last_name_varray.EXTEND;
    last_name_varray(v_index) := name_rec.last_name;

    DBMS_OUTPUT.PUT_LINE ('last_name('||v_index||'): '||last_name_varray(v_index));
  END LOOP;
END;
```

```
DECLARE
  TYPE varray_type IS VARRAY(10) OF NUMBER;
  varray varray_type := varray_type(1, 2, 3, 4, 5, 6);

BEGIN
  DBMS_OUTPUT.PUT_LINE ('varray.COUNT = '||varray.COUNT);
  DBMS_OUTPUT.PUT_LINE ('varray.LIMIT = '||varray.LIMIT);

  DBMS_OUTPUT.PUT_LINE ('varray.FIRST = '||varray.FIRST);
  DBMS_OUTPUT.PUT_LINE ('varray.LAST = '||varray.LAST);
  -- Append two copies of the 4th element to the varray
  varray.EXTEND(2, 4);
  DBMS_OUTPUT.PUT_LINE ('varray.LAST = '||varray.LAST);
  DBMS_OUTPUT.PUT_LINE ('varray('' || varray.LAST||') = '||varray(varray.LAST));

  -- Trim last two elements
  varray.TRIM(2);
  DBMS_OUTPUT.PUT_LINE ('varray.LAST = '||varray.LAST);
END;
```

```
DECLARE
  TYPE varray_type IS VARRAY(3) OF CHAR(1);
  varray varray_type := varray_type('A', 'B', 'C');
BEGIN
  varray.DELETE(3);
END;
```

```
ORA-06550: line 6, column 4:
PLS-00306: wrong number or types of arguments in call to 'DELETE'
ORA-06550: line 6, column 4:
PL/SQL: Statement ignored
```

```
varray_name(subscript of the outer varray) (subscript of the inner varray)
```

```
DECLARE
  TYPE varray_type1 IS VARRAY(4) OF INTEGER;
  TYPE varray_type2 IS VARRAY(3) OF varray_type1;

  varray1 varray_type1 := varray_type1(2, 4, 6, 8);
  varray2 varray_type2 := varray_type2(varray1);

BEGIN
  DBMS_OUTPUT.PUT_LINE ('Varray of integers');
  FOR i IN 1..4
  LOOP
    DBMS_OUTPUT.PUT_LINE ('varray1'||i||': '||varray1(i));
  END LOOP;

  varray2.EXTEND;
  varray2(2) := varray_type1(1, 3, 5, 7);

  DBMS_OUTPUT.PUT_LINE (chr(10)||'Varray of varrays of integers');
  FOR i IN 1..2
  LOOP
    FOR j IN 1..4
    LOOP
      DBMS_OUTPUT.PUT_LINE ('varray2'||i||'('||j||'): '||varray2(i)(j));
    END LOOP;
  END LOOP;
END;
```

```
varray2(2) := varray_type1(1, 3, 5, 7);
```

```
varray1(2) := varray_type1(1, 3, 5, 7);  
varray2(2) := varray_type2(varray1);
```

```
DECLARE
    course_rec course%ROWTYPE;
BEGIN
    SELECT *
        INTO course_rec
        FROM course
       WHERE course_no = 25;

    DBMS_OUTPUT.PUT_LINE ('Course No: '||course_rec.course_no);
    DBMS_OUTPUT.PUT_LINE ('Course Description: '||course_rec.description);
    DBMS_OUTPUT.PUT_LINE ('Prerequisite: '||course_rec.prerequisite);
END;
```

Course No: 25

Course Description: Intro to Programming

Prerequisite: 140

```
IF course_rec IS NULL THEN ...
IF course_rec1 = course_rec2 THEN ...
```

```
DECLARE
  CURSOR student_cur IS
    SELECT first_name, last_name, registration_date
      FROM student
     WHERE rownum <= 4;

  student_rec student_cur%ROWTYPE;
BEGIN
  OPEN student_cur;
  LOOP
    FETCH student_cur INTO student_rec;
    EXIT WHEN student_cur%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE
      ('Name: '||student_rec.first_name||' '||student_rec.last_name);
    DBMS_OUTPUT.PUT_LINE
      ('Registration Date: '||to_char(student_rec.registration_date, 'MM/DD/YYYY'));
  END LOOP;
END;
```

```
DECLARE
    student_rec student_cur%ROWTYPE;

CURSOR student_cur IS
    SELECT first_name, last_name, registration_date
    FROM student
    WHERE rownum <= 4;

BEGIN
    OPEN student_cur;
    LOOP
        FETCH student_cur INTO student_rec;
        EXIT WHEN student_cur%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE
            ('Name: '||student_rec.first_name||' '||student_rec.last_name);
        DBMS_OUTPUT.PUT_LINE
            ('Registration Date: '|| to_char(student_rec.registration_date, 'MM/DD/YYYY'));
    END LOOP;
END;
```

```
ORA-06550: line 2, column 16:  
PLS-00320: the declaration of the type of this expression is incomplete or malformed  
ORA-06550: line 2, column 16:  
PL/SQL: Item ignored  
ORA-06550: line 12, column 30:  
PLS-00320: the declaration of the type of this expression is incomplete or malformed  
ORA-06550: line 12, column 7:  
PL/SQL: SQL Statement ignored  
ORA-06550: line 16, column 21:  
PLS-00320: the declaration of the type of this expression is incomplete or malformed  
ORA-06550: line 15, column 7:  
PL/SQL: Statement ignored  
ORA-06550: line 18, column 40:  
PLS-00320: the declaration of the type of this expression is incomplete or malformed  
ORA-06550: line 17, column 7:  
PL/SQL: Statement ignored
```

```
TYPE type_name IS RECORD
  (field_name1 datatype1 [NOT NULL] [ := DEFAULT EXPRESSION],
   field_name2 datatype2 [NOT NULL] [ := DEFAULT EXPRESSION],
   ...
   field_nameN datatypeN [NOT NULL] [ := DEFAULT EXPRESSION]);
record_name TYPE_NAME;
```

```
DECLARE
  TYPE time_rec_type IS RECORD
    (curr_date DATE,
     curr_day  VARCHAR2(12),
     curr_time VARCHAR2(8) := '00:00:00');

  time_rec TIME_REC_TYPE;
BEGIN
  SELECT sysdate
    INTO time_rec.curr_date
   FROM dual;

  time_rec.curr_day := TO_CHAR(time_rec.curr_date, 'DAY');
  time_rec.curr_time := TO_CHAR(time_rec.curr_date, 'HH24:MI:SS');

  DBMS_OUTPUT.PUT_LINE ('Date: '||to_char(time_rec.curr_date, 'MM/DD/YYYY HH24:MI:SS'));
  DBMS_OUTPUT.PUT_LINE ('Day:  '||time_rec.curr_day);
  DBMS_OUTPUT.PUT_LINE ('Time: '||time_rec.curr_time);
END;
```

```
DECLARE
  TYPE sample_type IS RECORD
    (field1 NUMBER(3),
     field2 VARCHAR2(3) NOT NULL);
  sample_rec sample_type;

BEGIN
  sample_rec.field1 := 10;
  sample_rec.field2 := 'ABC';

  DBMS_OUTPUT.PUT_LINE ('sample_rec.field1 = '||sample_rec.field1);
  DBMS_OUTPUT.PUT_LINE ('sample_rec.field2 = '||sample_rec.field2);
END;
```

ORA-06550: line 4, column 8:
PLS-00218: a variable declared NOT NULL must have an initialization assignment

```
DECLARE
  TYPE sample_type IS RECORD
    (field1 NUMBER(3),
     field2 VARCHAR2(3) NOT NULL := 'ABC'); -- initialize a NOT NULL field
  sample_rec sample_type;

BEGIN
  sample_rec.field1 := 10;
  DBMS_OUTPUT.PUT_LINE ('sample_rec.field1 = '||sample_rec.field1);
  DBMS_OUTPUT.PUT_LINE ('sample_rec.field2 = '||sample_rec.field2);
END;
```

```
DECLARE
  TYPE name_type1 IS RECORD
    (first_name VARCHAR2(15),
     last_name  VARCHAR2(30));
  TYPE name_type2 IS RECORD
    (first_name VARCHAR2(15),
     last_name  VARCHAR2(30));

  name_rec1 name_type1;
  name_rec2 name_type2;
BEGIN
  name_rec1.first_name := 'John';
  name_rec1.last_name := 'Smith';
  name_rec2 := name_rec1; -- illegal assignment
END;
```

```
name_rec2 := name_rec1; -- illegal assignment
```

ORA-06550: line 15, column 17:
PLS-00382: expression is of wrong type
ORA-06550: line 15, column 4:
PL/SQL: Statement ignored

```
DECLARE
  TYPE name_type1 IS RECORD
    (first_name VARCHAR2(15),
     last_name  VARCHAR2(30));
  name_rec1 name_type1;
  name_rec2 name_type1;
BEGIN
  name_rec1.first_name := 'John';
  name_rec1.last_name := 'Smith';
  name_rec2 := name_rec1; -- no longer illegal assignment
END;
```

```
DECLARE
  CURSOR course_cur IS
    SELECT *
      FROM course
     WHERE rownum < 2;

  TYPE course_type IS RECORD
    (course_no      NUMBER(38)
     ,description   VARCHAR2(50)
     ,cost          NUMBER(9,2)
     ,prerequisite  NUMBER(8)
     ,created_by    VARCHAR2(30)
     ,created_date  DATE
     ,modified_by   VARCHAR2(30)
     ,modified_date DATE);

  course_rec1 course%ROWTYPE;      -- table-based record
  course_rec2 course_cur%ROWTYPE; -- cursor-based record
  course_rec3 course_type;       -- user-defined record

BEGIN
  -- Populate table-based record
  SELECT *
    INTO course_rec1
    FROM course
   WHERE course_no = 10;

  -- Populate cursor-based record
  OPEN course_cur;
  LOOP
    FETCH course_cur INTO course_rec2;
    EXIT WHEN course_cur%NOTFOUND;
  END LOOP;

  -- Assign COURSE_REC2 to COURSE_REC1 and COURSE_REC3
  course_rec1 := course_rec2;
  course_rec3 := course_rec2;

  DBMS_OUTPUT.PUT_LINE (course_rec1.course_no||' - '||course_rec1.description);
  DBMS_OUTPUT.PUT_LINE (course_rec2.course_no||' - '||course_rec2.description);
  DBMS_OUTPUT.PUT_LINE (course_rec3.course_no||' - '||course_rec3.description);
END;
```

```
DECLARE
  TYPE name_type IS RECORD
    (first_name VARCHAR2(15),
     last_name  VARCHAR2(30));
  TYPE person_type IS RECORD
    (name   name_type,
     street VARCHAR2(50),
     city   VARCHAR2(25),
     state  VARCHAR2(2),
     zip    VARCHAR2(5));
  person_rec person_type;
BEGIN
  SELECT first_name, last_name, street_address, city, state, zip
    INTO person_rec.name.first_name, person_rec.name.last_name,
         person_rec.street, person_rec.city, person_rec.state,
         person_rec.zip
   FROM student
  JOIN zipcode USING (zip)
 WHERE rownum < 2;
  DBMS_OUTPUT.PUT_LINE ('Name: ' ||
    person_rec.name.first_name||' '||person_rec.name.last_name);
  DBMS_OUTPUT.PUT_LINE ('Street: '||person_rec.street);
  DBMS_OUTPUT.PUT_LINE ('City:  '||person_rec.city);
  DBMS_OUTPUT.PUT_LINE ('State: '||person_rec.state);
  DBMS_OUTPUT.PUT_LINE ('Zip:    '||person_rec.zip);
END;
```

```
enclosing_record.(nested_record or nested_collection).field_name
```

```
DECLARE
  TYPE last_name_type IS TABLE OF student.last_name%TYPE
    INDEX BY PLS_INTEGER;

  TYPE zip_info_type IS RECORD
    (zip      VARCHAR2(5),
     last_name_tab last_name_type);

  CURSOR name_cur (p_zip VARCHAR2) IS
    SELECT last_name
      FROM student
     WHERE zip = p_zip;

  zip_info_rec zip_info_type;
  v_zip          VARCHAR2(5) := '&sv_zip';
  v_index        PLS_INTEGER := 0;
BEGIN
  zip_info_rec.zip := v_zip;
  DBMS_OUTPUT.PUT_LINE ('ZIP: '||zip_info_rec.zip);

  FOR name_rec IN name_cur (v_zip)
  LOOP
    v_index := v_index + 1;
    zip_info_rec.last_name_tab(v_index) := name_rec.last_name;
    DBMS_OUTPUT.PUT_LINE
      ('Names('||v_index||'): '||zip_info_rec.last_name_tab(v_index));
  END LOOP;
END;
```

```
DECLARE
  CURSOR name_cur IS
    SELECT first_name, last_name
      FROM student
     WHERE ROWNUM <= 4;

  TYPE name_type IS TABLE OF name_cur%ROWTYPE
    INDEX BY PLS_INTEGER;

  name_tab name_type;
  v_index  INTEGER := 0;

BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;

    name_tab(v_index).first_name := name_rec.first_name;
    name_tab(v_index).last_name  := name_rec.last_name;

    DBMS_OUTPUT.PUT_LINE('First Name('||v_index ||'): ' ||
      name_tab(v_index).first_name);
    DBMS_OUTPUT.PUT_LINE('Last Name('||v_index ||'): ' ||
      name_tab(v_index).last_name);
  END LOOP;
END;
```

```
collection_name(index).record_field_name1  
collection_name(index).record_field_name2  
-  
collection_name(index).record_field_nameN
```

```
DECLARE
  CURSOR name_cur IS
    SELECT first_name, last_name
      FROM student
     WHERE ROWNUM <= 4;

  TYPE name_type IS TABLE OF name_cur%ROWTYPE;

  name_tab name_type := name_type();
  v_index INTEGER := 0;
BEGIN
  FOR name_rec IN name_cur
  LOOP
    v_index := v_index + 1;
    name_tab.EXTEND;

    name_tab(v_index).first_name := name_rec.first_name;
    name_tab(v_index).last_name := name_rec.last_name;

    DBMS_OUTPUT.PUT_LINE('First Name('||v_index||'): ' ||
      name_tab(v_index).first_name);
    DBMS_OUTPUT.PUT_LINE('Last Name('||v_index||'): ' ||
      name_tab(v_index).last_name);
  END LOOP;
END;
```

```
DECLARE
  CURSOR enroll_cur IS
    SELECT first_name, last_name, COUNT(*) total
      FROM student
     JOIN enrollment USING (student_id)
   GROUP BY first_name, last_name;
  TYPE enroll_rec_type IS RECORD
    (first_name  VARCHAR2(15),
     last_name   VARCHAR2(30),
     enrollments INTEGER);

  TYPE enroll_array_type IS TABLE OF enroll_rec_type
    INDEX BY PLS_INTEGER;

  enroll_tab enroll_array_type;
  v_index   INTEGER := 0;
BEGIN
  FOR enroll_rec IN enroll_cur
  LOOP
    v_index := v_index + 1;

    enroll_tab(v_index).first_name  := enroll_rec.first_name;
    enroll_tab(v_index).last_name   := enroll_rec.last_name;
    enroll_tab(v_index).enrollments := enroll_rec.total;

    IF v_index <= 4
    THEN
      DBMS_OUTPUT.PUT_LINE('First Name('||v_index||'): ' ||
                           enroll_tab(v_index).first_name);
      DBMS_OUTPUT.PUT_LINE('Last Name('||v_index||'): ' ||
                           enroll_tab(v_index).last_name);
      DBMS_OUTPUT.PUT_LINE('Enrollments('||v_index||'): ' ||
                           enroll_tab(v_index).enrollments);
      DBMS_OUTPUT.PUT_LINE ('-----');
    END IF;
  END LOOP;
END;
```

```
'SELECT first_name, last_name FROM student  
WHERE student_id = :student_id'
```

```
'SELECT first_name, last_name
FROM student WHERE
student_id = :student_id'
'SELECT first_name, last_name
FROM student WHERE student_id = :id'
```

```
EXECUTE IMMEDIATE dynamic_SQL_string
[INTO defined_variable1, defined_variable2, ...]
[USING [IN | OUT | IN OUT] bind_argument1, bind_argument2,
...] [{RETURNING | RETURN} field1, field2,
... INTO bind_argument1, bind_argument2, ...]
```

```
DECLARE
    sql_stmt VARCHAR2(100);
    plsql_block VARCHAR2(300);
    v_zip VARCHAR2(5) := '11106';
    v_total_students NUMBER;
    v_new_zip VARCHAR2(5);
    v_student_id NUMBER := 151;
BEGIN
    -- Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ' ||
        'AS SELECT * FROM student WHERE zip = '||v_zip;
    EXECUTE IMMEDIATE sql_stmt;

    -- Select total number of records from MY_STUDENT table
    -- and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM my_student'
    INTO v_total_students;
    DBMS_OUTPUT.PUT_LINE ('Students added: '||v_total_students);

    -- Select current date and display it on the screen
    plsql_block := 'DECLARE
        v_date DATE;
        BEGIN
            SELECT SYSDATE INTO v_date FROM DUAL;
            DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_date,
                ''DD-MON-YYYY''));
        END;';
    EXECUTE IMMEDIATE plsql_block;

    -- Update record in MY_STUDENT table
    sql_stmt := 'UPDATE my_student SET zip = 11105 WHERE student_id =
        :1 ' ||
        'RETURNING zip INTO :2';
    EXECUTE IMMEDIATE sql_stmt USING v_student_id RETURNING INTO
        v_new_zip;
    DBMS_OUTPUT.PUT_LINE ('New zip code: '||v_new_zip);
END;
```

Students added: 4
22-JUN-2003
New zip code: 11105

PL/SQL procedure successfully completed.

```
DECLARE
    sql_stmt VARCHAR2(100);
    v_zip VARCHAR2(5) := '11106';
    v_total_students NUMBER;
BEGIN
    -- Drop table MY_STUDENT
    EXECUTE IMMEDIATE 'DROP TABLE my_student';
    -- Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ' ||
        'AS SELECT * FROM student ' ||
        'WHERE zip = :zip';
    EXECUTE IMMEDIATE sql_stmt USING v_zip;

    -- Select total number of records from MY_STUDENT table
    -- and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM my_student'
    INTO v_total_students;

    DBMS_OUTPUT.PUT_LINE ('Students added: '|| v_total_students);
END;
```

```
DECLARE
*
ERROR at line 1:
ORA-01027: bind variables not allowed for data definition operations
ORA-06512: at line 12
```

```
DECLARE
  sql_stmt  VARCHAR2(100);
  v_zip    VARCHAR2(5) := '11106';
  v_total_students NUMBER;
BEGIN
  -- Create table MY_STUDENT
  sql_stmt := 'CREATE TABLE my_student ' ||
    'AS SELECT * FROM student ' || 'WHERE zip =' || v_zip;
  EXECUTE IMMEDIATE sql_stmt;
  -- Select total number of records from MY_STUDENT table
  -- and display results on the screen
  EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM :my_table'
  INTO v_total_students
  USING 'my_student';
  DBMS_OUTPUT.PUT_LINE ('Students added: '|| v_total_students);
END;
```

```
DECLARE
*
ERROR at line 1:
ORA-00903: invalid table name
ORA-06512: at line 13
```

```
EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM '||my_table  
INTO v_total_students;
```

```
DECLARE
    sql_stmt VARCHAR2(100);
    v_zip VARCHAR2(5) := '11106';
    v_total_students NUMBER;
BEGIN
    -- Create table MY_STUDENT
    sql_stmt := 'CREATE TABLE my_student ' ||
        'AS SELECT * FROM student '|| 'WHERE zip = '||v_zip;
    EXECUTE IMMEDIATE sql_stmt;

    -- Select total number of records from MY_STUDENT table
    -- and display results on the screen
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM my_student;'
    INTO v_total_students;
    DBMS_OUTPUT.PUT_LINE ('Students added: '|| v_total_students);
END;
```

```
DECLARE
  plsql_block VARCHAR2(300);
BEGIN
  -- Select current date and display it on the screen
  plsql_block := 'DECLARE
    v_date DATE;
  BEGIN
    SELECT SYSDATE INTO v_date FROM DUAL;
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_date,
                                   ''DD-MON-YYYY''));
  END;
  /';
  EXECUTE IMMEDIATE plsql_block;
END;
```

```
DECLARE
*
ERROR at line 1:
ORA-06550: line 1, column 133:
PLS-00103: Encountered the symbol "/" The symbol /* was ignored.
ORA-06512: at line 12
```

```
DECLARE
    sql_stmt VARCHAR2(100);
BEGIN
    sql_stmt := 'UPDATE course' ||
        '    SET prerequisite = :some_value';
    EXECUTE IMMEDIATE sql_stmt
    USING NULL;
END;
```

```
USING NULL;
*
ERROR at line 7:
ORA-06550: line 7, column 10:
PLS-00457: expressions have to be of SQL types
ORA-06550: line 6, column 4:
PL/SQL: Statement ignored
```

```
DECLARE
    sql_stmt VARCHAR2(100);
    v_null VARCHAR2(1);
BEGIN
    sql_stmt := 'UPDATE course'|||
        '      SET prerequisite = :some_value';
    EXECUTE IMMEDIATE sql_stmt
    USING v_null;
END;
```

```
SET SERVEROUTPUT ON
DECLARE
  sql_stmt VARCHAR2(200);
  v_student_id NUMBER := &sv_student_id;
  v_first_name VARCHAR2(25);
  v_last_name VARCHAR2(25);
BEGIN
  sql_stmt := 'SELECT first_name, last_name' ||
    '   FROM student' ||
    ' WHERE student_id = :1';
  EXECUTE IMMEDIATE sql_stmt
  INTO v_first_name, v_last_name
  USING v_student_id;
  DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
  DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
END;
```

```
Enter value for sv_student_id: 105
old  3:      v_student_id NUMBER := &sv_student_id;
new  3:      v_student_id NUMBER := 105;
First Name: Angel
Last Name: Moskowitz
PL/SQL procedure successfully completed
```

```
SET SERVEROUTPUT ON
DECLARE
    sql_stmt VARCHAR2(200);
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
    v_street VARCHAR2(50);
    v_city VARCHAR2(25);
    v_state VARCHAR2(2);
    v_zip VARCHAR2(5);
BEGIN
    sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
               '           ,b.city, b.state, b.zip' ||
               '      FROM student a, zipcode b' ||
               '     WHERE a.zip = b.zip' ||
               '       AND student_id = :1';
    EXECUTE IMMEDIATE sql_stmt
    INTO v_first_name, v_last_name, v_street, v_city, v_state, v_zip
    USING v_student_id;
    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
    DBMS_OUTPUT.PUT_LINE ('Street:      '||v_street);
    DBMS_OUTPUT.PUT_LINE ('City:        '||v_city);
    DBMS_OUTPUT.PUT_LINE ('State:       '||v_state);
    DBMS_OUTPUT.PUT_LINE ('Zip Code:    '||v_zip);
END;
```

```
Enter value for sv_student_id: 105
old  3:      v_student_id NUMBER := &sv_student_id;
new  3:      v_student_id NUMBER := 105;
First Name: Angel
Last Name: Moskowitz
Street:    320 John St.
City:      Ft. Lee
State:     NJ
Zip Code:  07024
```

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

```
SET SERVEROUTPUT ON
DECLARE
    sql_stmt VARCHAR2(200);
    v_student_id NUMBER := &sv_student_id;
    v_first_name VARCHAR2(25);
    v_last_name VARCHAR2(25);
    v_street VARCHAR2(50);
    v_city VARCHAR2(25);
    v_state VARCHAR2(2);
    v_zip VARCHAR2(5);
BEGIN
    sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
               '           ,b.city, b.state, b.zip' ||
               '      FROM student a, zipcode b' ||
               '     WHERE a.zip = b.zip' ||
               '       AND student_id = :1';
    EXECUTE IMMEDIATE sql_stmt;
    -- variables v_state and v_zip are misplaced
    INTO v_first_name, v_last_name, v_street, v_city, v_zip, v_state
    USING v_student_id;
    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
    DBMS_OUTPUT.PUT_LINE ('Street: '||v_street);
    DBMS_OUTPUT.PUT_LINE ('City: '||v_city);
    DBMS_OUTPUT.PUT_LINE ('State: '||v_state);
    DBMS_OUTPUT.PUT_LINE ('Zip Code: '||v_zip);
END;
```

```
Enter value for sv_student_id: 105
old  3:      v_student_id NUMBER := &sv_student_id;
new  3:      v_student_id NUMBER := 105;

DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 16
```

```
SET SERVEROUTPUT ON
DECLARE
  sql_stmt VARCHAR2(200);
  v_table_name VARCHAR2(20) := '&sv_table_name';
  v_id NUMBER := &sv_id;
  v_first_name VARCHAR2(25);
  v_last_name VARCHAR2(25);
  v_street VARCHAR2(50);
  v_city VARCHAR2(25);
  v_state VARCHAR2(2);
  v_zip VARCHAR2(5);
BEGIN
  sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address' ||
              ' ,b.city, b.state, b.zip' ||
              ' FROM'||v_table_name||' a, zipcode b' ||
              ' WHERE a.zip = b.zip' ||
              ' AND'||v_table_name||'_id = :1';
  EXECUTE IMMEDIATE sql_stmt
  -INTO v_first_name, v_last_name, v_street, v_city, v_state, v_zip
  USING v_id;
  DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
  DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
  DBMS_OUTPUT.PUT_LINE ('Street: '||v_street);
  DBMS_OUTPUT.PUT_LINE ('City: '||v_city);
  DBMS_OUTPUT.PUT_LINE ('State: '||v_state);
  DBMS_OUTPUT.PUT_LINE ('Zip Code: '||v_zip);
END;
```

```
sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address'||  
          ' ,b.city, b.state, b.zip'  
          ' FROM student a, zipcode b'  
          ' WHERE a.zip = b.zip'  
          ' AND student_id = :1';
```

```
sql_stmt := 'SELECT a.first_name, a.last_name, a.street_address'||  
          ' ,b.city, b.state, b.zip'||  
          ' FROM'||v_table_name||' a, zipcode b'||  
          ' WHERE a.zip = b.zip'||  
          ' AND'||v_table_name||'_id = :1';
```

```
Enter value for sv_table_name: student
old  3:      v_table_name VARCHAR2(20) := '&sv_table_name';
new  3:      v_table_name VARCHAR2(20) := 'student';
Enter value for sv_id: 105
old  4:      v_id NUMBER := &sv_id;
new  4:      v_id NUMBER := 105;
First Name: Angel
Last Name: Moskowitz
Street:     320 John St.
City:       Ft. Lee
State:      NJ
Zip Code:   07024
PL/SQL procedure successfully completed.
```

```
Enter value for sv_table_name: instructor
old  3:      v_table_name VARCHAR2(20) := '&sv_table_name';
new  3:      v_table_name VARCHAR2(20) := 'instructor';
Enter value for sv_id: 105
old  4:      v_id NUMBER := &sv_id;
new  4:      v_id NUMBER := 105;
First Name: Anita
Last Name: Morris
Street:      34 Maiden Lane
City:        New York
State:       NY
Zip Code:    10015
PL/SQL procedure successfully completed.
```

```
OPEN cursor_variable FOR dynamic_SQL_string  
[USING bind_argument1, bind_argument2, ...]
```

```
DECLARE
  TYPE student_cur_type IS REF CURSOR;
  student_cur student_cur_type;
  v_zip VARCHAR2(5) := '&sv_zip';
  v_first_name VARCHAR2(25);
  v_last_name VARCHAR2(25);
BEGIN
  OPEN student_cur FOR
    'SELECT first_name, last_name FROM student '||'WHERE zip = :1'
  USING v_zip;
  ...

```

```
FETCH cursor_variable  
INTO defined_variable1, defined_variable2, ...  
EXIT WHEN cursor_variable%NOTFOUND;
```

```
DECLARE
  TYPE student_cur_type IS REF CURSOR;
  student_cur student_cur_type;
  v_zip VARCHAR2(5) := '&sv_zip';
  v_first_name VARCHAR2(25);
  v_last_name VARCHAR2(25);
BEGIN
  OPEN student_cur FOR
    'SELECT first_name, last_name FROM student '||'WHERE zip = :1'
  USING v_zip;

  LOOP
    FETCH student_cur INTO v_first_name, v_last_name;
    EXIT WHEN student_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
  END LOOP;
  ...

```

```
DECLARE
  TYPE student_cur_type IS REF CURSOR;
  student_cur student_cur_type;
  v_zip VARCHAR2(5) := '&sv_zip';
  v_first_name VARCHAR2(25);
  v_last_name VARCHAR2(25);

BEGIN
  OPEN student_cur FOR
    'SELECT first_name, last_name FROM student '||'WHERE zip = :1'
  USING v_zip;
  LOOP
    FETCH student_cur INTO v_first_name, v_last_name;
    EXIT WHEN student_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE ('First Name: '||v_first_name);
    DBMS_OUTPUT.PUT_LINE ('Last Name: '||v_last_name);
  END LOOP;
  CLOSE student_cur;
EXCEPTION
  WHEN OTHERS THEN
    IF student_cur%ISOPEN THEN
      CLOSE student_cur;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
END;
```

```
Enter value for sv_zip: 11236
old  5:      v_zip VARCHAR2(5) := '&sv_zip';
new  5:      v_zip VARCHAR2(5) := '11236';
First Name: Derrick
Last Name: Baltazar
First Name: Michael
Last Name: Lefbowitz
First Name: Bridget
Last Name: Hagel
```

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

```
SET SERVEROUTPUT ON
DECLARE
  TYPE zip_cur_type IS REF CURSOR;
  zip_cur zip_cur_type;
  sql_stmt VARCHAR2(500);
  v_zip VARCHAR2(5);
  v_total NUMBER;
  v_count NUMBER;
BEGIN
  sql_stmt := 'SELECT zip, COUNT(*) total' ||
              ' FROM student ' ||
              'GROUP BY zip';
  v_count := 0;
  OPEN zip_cur FOR sql_stmt;
  LOOP
    FETCH zip_cur INTO v_zip, v_total;
    EXIT WHEN zip_cur%NOTFOUND;
    -- Limit the number of lines printed on the
    -- screen to 10
    v_count := v_count + 1;
    IF v_count <= 10 THEN
      DBMS_OUTPUT.PUT_LINE ('Zip code: '||v_zip||
                            ' Total: '||v_total);
    END IF;
  END LOOP;
  CLOSE zip_cur;
EXCEPTION
  WHEN OTHERS THEN
    IF zip_cur%ISOPEN THEN
      CLOSE zip_cur;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
END;
```

```
sql_stmt := 'SELECT zip, COUNT(*) total'||  
          ' FROM student '||  
          'GROUP BY zip';
```

```
SELECT zip, COUNT(*) total FROM student GROUP BY zip
```

```
SELECT zip, COUNT(*) total FROM student GROUP BY zip
```

ERROR: ORA-00933: SQL command not properly ended

PL/SQL procedure successfully completed.

```
Zip code: 01247 Total: 1
Zip code: 02124 Total: 1
Zip code: 02155 Total: 1
Zip code: 02189 Total: 1
Zip code: 02563 Total: 1
Zip code: 06483 Total: 1
Zip code: 06605 Total: 1
Zip code: 06798 Total: 1
Zip code: 06820 Total: 3
Zip code: 06830 Total: 3
PL/SQL procedure successfully completed.
```

```
SET SERVEROUTPUT ON
DECLARE
  TYPE zip_cur_type IS REF CURSOR;
  zip_cur zip_cur_type;
  v_table_name VARCHAR2(20) := '&sv_table_name';
  sql_stmt VARCHAR2(500);
  v_zip VARCHAR2(5);
  v_total NUMBER;
  v_count NUMBER;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Totals from |||v_table_name|||
                        ' table');

  sql_stmt := 'SELECT zip, COUNT(*) total'|||
              ' FROM |||v_table_name|||' '|||
              ' GROUP BY zip';

  v_count := 0;
  OPEN zip_cur FOR sql_stmt;
  LOOP
    FETCH zip_cur INTO v_zip, v_total;
    EXIT WHEN zip_cur%NOTFOUND;
    -- Limit the number of lines printed on the
    -- screen to 10
    v_count := v_count + 1;
    IF v_count <= 10 THEN
      DBMS_OUTPUT.PUT_LINE ('Zip code: |||v_zip|||
                            ' Total: |||v_total|||');
    END IF;
  END LOOP;
  CLOSE zip_cur;
EXCEPTION
  WHEN OTHERS THEN
    IF zip_cur%ISOPEN THEN
      CLOSE zip_cur;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('ERROR: ||| SUBSTR(SQLERRM, 1, 200))');
END;
```

```
sql_stmt := 'SELECT zip, COUNT(*) total'||  
          ' FROM'|||v_table_name|||' '|  
          'GROUP BY zip';
```

```
Enter value for sv_table_name: student
old  5:      v_table_name VARCHAR2(20) := '&sv_table_name';
new  5:      v_table_name VARCHAR2(20) := 'student';
Totals from student table
Zip code: 01247 Total: 1
Zip code: 02124 Total: 1
Zip code: 02155 Total: 1
Zip code: 02189 Total: 1
Zip code: 02563 Total: 1
Zip code: 06483 Total: 1
Zip code: 06605 Total: 1
Zip code: 06798 Total: 1
Zip code: 06820 Total: 3
Zip code: 06830 Total: 3
```

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

```
Enter value for sv_table_name: instructor
old  5:      v_table_name VARCHAR2(20) := '&sv_table_name';
new  5:      v_table_name VARCHAR2(20) := 'instructor';
Totals from instructor table
Zip code: 10005 Total: 1
Zip code: 10015 Total: 3
Zip code: 10025 Total: 4
Zip code: 10035 Total: 1

PL/SQL procedure successfully completed.
```

```

SET SERVEROUTPUT ON
DECLARE
  TYPE zip_cur_type IS REF CURSOR;
  zip_cur zip_cur_type;

  TYPE zip_rec_type IS RECORD
    (zip VARCHAR2(5),
     total NUMBER);
  zip_rec zip_rec_type;

  v_table_name VARCHAR2(20) := '&sv_table_name';
  sql_stmt VARCHAR2(500);
  v_count NUMBER;

BEGIN
  DBMS_OUTPUT.PUT_LINE ('Totals from'||v_table_name||
                        ' table');
  sql_stmt := 'SELECT zip, COUNT(*) total'|||
              ' FROM'|||v_table_name|||' '||
              'GROUP BY zip';

  v_count := 0;
  OPEN zip_cur FOR sql_stmt;
  LOOP
    FETCH zip_cur INTO zip_rec;
    EXIT WHEN zip_cur%NOTFOUND;

    -- Limit the number of lines printed on the
    -- screen to 10
    v_count := v_count + 1;
    IF v_count <= 10 THEN
      DBMS_OUTPUT.PUT_LINE ('Zip code: '||zip_rec.zip||
                            ' Total: '||zip_rec.total);
    END IF;
  END LOOP;
  CLOSE zip_cur;
EXCEPTION
  WHEN OTHERS THEN
    IF zip_cur%ISOPEN THEN
      CLOSE zip_cur;
    END IF;
    DBMS_OUTPUT.PUT_LINE ('ERROR: '|| SUBSTR(SQLERRM, 1, 200));
END;

```

```
Enter value for sv_table_name: student
old 10:      v_table_name VARCHAR2(20) := '&sv_table_name';
new 10:      v_table_name VARCHAR2(20) := 'student';
Totals from student table
Zip code: 01247 Total: 1
Zip code: 02124 Total: 1
Zip code: 02155 Total: 1
Zip code: 02189 Total: 1
Zip code: 02563 Total: 1
Zip code: 06483 Total: 1
Zip code: 06605 Total: 1
Zip code: 06798 Total: 1
Zip code: 06820 Total: 3
Zip code: 06830 Total: 3
```

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

```
Enter value for sv_table_name: instructor
old 10:      v_table_name VARCHAR2(20) := '&sv_table_name';
new 10:      v_table_name VARCHAR2(20) := 'instructor';
Totals from instructor table
Zip code: 10005 Total: 1
Zip code: 10015 Total: 3
Zip code: 10025 Total: 4
Zip code: 10035 Total: 1

PL/SQL procedure successfully completed.
```

```
FORALL loop_counter IN bounds_clause
  SQL_STATEMENT [SAVE EXCEPTIONS];
```

```
lower_limit..upper_limit  
INDICES OF collection_name BETWEEN lower_limit..upper_limit  
VALUES OF collection_name
```

```
CREATE TABLE test
  (row_num NUMBER
 ,row_text VARCHAR2(10));

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER      INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;
  v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Populate TEST table
  FORALL i IN 1..10
    INSERT INTO test (row_num, row_text)
    VALUES (row_num_tab(i), row_text_tab(i));

  COMMIT;
  -- Check how many rows were inserted in the TEST table
  -- display it on the screen
  SELECT COUNT(*)
    INTO v_rows
   FROM TEST;

  DBMS_OUTPUT.PUT_LINE ('There are '||v_rows||' rows in the TEST table');
END;
```

There are 10 rows in the TEST table

```
TRUNCATE TABLE test;

DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

    row_num_tab row_num_type;
    row_text_tab row_text_type;

    v_start_time INTEGER;
    v_end_time   INTEGER;
BEGIN
    -- Populate collections
    FOR i IN 1..100
    LOOP
        row_num_tab(i) := i;
        row_text_tab(i) := 'row '||i;
    END LOOP;

    -- Record start time
    v_start_time := DBMS_UTILITY.GET_TIME;

    -- Insert first 100 rows
    FOR i IN 1..100
    LOOP
        INSERT INTO test (row_num, row_text)
        VALUES (row_num_tab(i), row_text_tab(i));
    END LOOP;

    -- Record end time
    v_end_time := DBMS_UTILITY.GET_TIME;

    -- Calculate and display elapsed time
    DBMS_OUTPUT.PUT_LINE ('Duration of the FOR LOOP: ' ||
        (v_end_time - v_start_time));

    -- Record start time
    v_start_time := DBMS_UTILITY.GET_TIME;

    -- Insert second 100 rows
    FORALL i IN 1..100
        INSERT INTO test (row_num, row_text)
        VALUES (row_num_tab(i), row_text_tab(i));

    -- Record end time
    v_end_time := DBMS_UTILITY.GET_TIME;

    -- Calculate and display elapsed time
    DBMS_OUTPUT.PUT_LINE ('Duration of the FORALL statement: ' ||
        (v_end_time - v_start_time));

    COMMIT;
END;
```

Duration of the FOR LOOP: 1

Duration of the FORALL statement: 0

```

TRUNCATE TABLE TEST;

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(11) INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;

  -- Define user-defined exception and associated Oracle
  -- error number with it
  errors EXCEPTION;
  PRAGMA EXCEPTION_INIT(errors, -24381);

  v_rows NUMBER;
BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row'||i;
  END LOOP;

  -- Modify 1, 5, and 7 elements of the V_ROW_TEXT collection
  -- These rows will cause exceptions in the FORALL statement
  row_text_tab(1) := RPAD(row_text_tab(1), 11, ' ');
  row_text_tab(5) := RPAD(row_text_tab(5), 11, ' ');
  row_text_tab(7) := RPAD(row_text_tab(7), 11, ' ');

  -- Populate TEST table
  FORALL i IN 1..10 SAVE EXCEPTIONS
    INSERT INTO test (row_num, row_text)
    VALUES (row_num_tab(i), row_text_tab(i));
  COMMIT;

EXCEPTION
  WHEN errors
  THEN
    -- Display total number of records inserted in the TEST table
    SELECT count(*)
    INTO v_rows
    FROM test;
    DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' records in the TEST table');

    -- Display total number of exceptions encountered
    DBMS_OUTPUT.PUT_LINE ('There were'||SQL%BULK_EXCEPTIONS.COUNT||' exceptions');

    -- Display detailed exception information
    FOR i in 1.. SQL%BULK_EXCEPTIONS.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE ('Record'|||
        SQL%BULK_EXCEPTIONS(i).error_index||' caused error'||i||':'|||
        SQL%BULK_EXCEPTIONS(i).error_code||' '|||
        SQLERRM(-SQL%BULK_EXCEPTIONS(i).error_code));
    END LOOP;
END;

```

```
SQL%BULK_EXCEPTIONS(i).error_index
```

```
SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE)
```

There are 7 records in the TEST table

There were 3 exceptions

Record 1 caused error 1: 12899 ORA-12899: value too large for column (actual: , maximum:)

Record 5 caused error 2: 12899 ORA-12899: value too large for column (actual: , maximum:)

Record 7 caused error 3: 12899 ORA-12899: value too large for column (actual: , maximum:)

```
TRUNCATE TABLE TEST;

DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER      INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

    row_num_tab row_num_type;
    row_text_tab row_text_type;

    v_rows NUMBER;
BEGIN
    -- Populate collections
    FOR i IN 1..10
    LOOP
        row_num_tab(i) := i;
        row_text_tab(i) := 'row '||i;
    END LOOP;

    -- Delete 1, 5, and 7 elements of collections
    row_num_tab.DELETE(1); row_text_tab.DELETE(1);
    row_num_tab.DELETE(5); row_text_tab.DELETE(5);
    row_num_tab.DELETE(7); row_text_tab.DELETE(7);

    -- Populate TEST table
    FORALL i IN INDICES OF row_num_tab
        INSERT INTO test (row_num, row_text)
        VALUES (row_num_tab(i), row_text_tab(i));
    COMMIT;

    SELECT COUNT(*)
        INTO v_rows
        FROM test;

    DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' rows in the TEST table');
END;
```

There are 7 rows in the TEST table

```

CREATE TABLE TEST_EXC
  (row_num NUMBER
   ,row_text VARCHAR2(50));

TRUNCATE TABLE TEST;

DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER           INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(11)    INDEX BY PLS_INTEGER;
  TYPE exc_ind_type IS TABLE OF PLS_INTEGER      INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;
  exc_ind_tab exc_ind_type;

  -- Define user-defined exception and associated Oracle
  -- error number with it
  errors EXCEPTION;
  PRAGMA EXCEPTION_INIT(errors, -24381);

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row'||i;
  END LOOP;

  -- Modify 1, 5, and 7 elements of the ROW_TEXT_TAB collection
  -- These rows will cause exceptions in the FORALL statement
  row_text_tab(1) := RPAD(row_text_tab(1), 11, ' ');
  row_text_tab(5) := RPAD(row_text_tab(5), 11, ' ');
  row_text_tab(7) := RPAD(row_text_tab(7), 11, ' ');

  -- Populate TEST table
  FORALL i IN 1..10 SAVE EXCEPTIONS
    INSERT INTO test (row_num, row_text)
    VALUES (row_num_tab(i), row_text_tab(i));
  COMMIT;

EXCEPTION
  WHEN errors
  THEN
    -- Populate EXC_IND_TAB collection to be used in the VALUES OF
    -- clause
    FOR i in 1..SQL%BULK_EXCEPTIONS.COUNT
    LOOP
      exc_ind_tab(i) := SQL%BULK_EXCEPTIONS(i).error_index;
    END LOOP;
    -- Insert records that caused exceptions in the TEST_EXC table
    FORALL i in VALUES OF exc_ind_tab
      INSERT INTO test_exc (row_num, row_text)
      VALUES (row_num_tab(i), row_text_tab(i));
    COMMIT;
END;

```

```
select *
  from test;

  ROW_NUM      ROW_TEXT
-----
    2          row 2
    3          row 3
    4          row 4
    6          row 6
    8          row 8
    9          row 9
   10         row 10
```

```
select *
  from test_exc;

  ROW_NUM      ROW_TEXT
-----
    1          row 1
    5          row 5
    7          row 7
```

```
DECLARE
  CURSOR student_cur IS
    SELECT student_id, first_name, last_name
      FROM student;
BEGIN
  FOR rec IN student_cur
  LOOP
    DBMS_OUTPUT.PUT_LINE ('student_id: '||rec.student_id);
    DBMS_OUTPUT.PUT_LINE ('first_name: '||rec.first_name);
    DBMS_OUTPUT.PUT_LINE ('last_name: '||rec.last_name);
  END LOOP;
END;
```

```
DECLARE
  -- Define collection type and variables to be used by the
  -- BULK COLLECT clause
  TYPE student_id_type IS TABLE OF student.student_id%TYPE;
  TYPE first_name_type IS TABLE OF student.first_name%TYPE;
  TYPE last_name_type IS TABLE OF student.last_name%TYPE;
  student_id_tab student_id_type;
  first_name_tab first_name_type;
  last_name_tab last_name_type;

BEGIN
  -- Fetch all student data at once via BULK COLLECT clause
  SELECT student_id, first_name, last_name
    BULK COLLECT INTO student_id_tab, first_name_tab, last_name_tab
      FROM student;

  FOR i IN student_id_tab.FIRST..student_id_tab.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE ('student_id: '||student_id_tab(i));
    DBMS_OUTPUT.PUT_LINE ('first_name: '||first_name_tab(i));
    DBMS_OUTPUT.PUT_LINE ('last_name: '||last_name_tab(i));
  END LOOP;
END;
```

```
DECLARE
  CURSOR student_cur IS
    SELECT student_id, first_name, last_name
      FROM student;
  -- Define collection type and variables to be used by the
  -- BULK COLLECT clause
  TYPE student_id_type IS TABLE OF student.student_id%TYPE;
  TYPE first_name_type IS TABLE OF student.first_name%TYPE;
  TYPE last_name_type  IS TABLE OF student.last_name%TYPE;

  student_id_tab student_id_type;
  first_name_tab first_name_type;
  last_name_tab  last_name_type;

  -- Define variable to be used by the LIMIT clause
  v_limit PLS_INTEGER := 50;

BEGIN
  OPEN student_cur;
  LOOP
    -- Fetch 50 rows at once
    FETCH student_cur
      BULK COLLECT INTO student_id_tab, first_name_tab, last_name_tab
      LIMIT v_limit;

    EXIT WHEN student_id_tab.COUNT = 0;

    FOR i IN student_id_tab.FIRST..student_id_tab.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE ('student_id: '||student_id_tab(i));
      DBMS_OUTPUT.PUT_LINE ('first_name: '||first_name_tab(i));
      DBMS_OUTPUT.PUT_LINE ('last_name:  '||last_name_tab(i));
    END LOOP;
  END LOOP;
  CLOSE student_cur;
END;
```

```
DECLARE
  CURSOR student_cur IS
    SELECT student_id, first_name, last_name
      FROM student;

  -- Define record type
  TYPE student_rec IS RECORD
    (student_id student.student_id%TYPE,
     first_name student.first_name%TYPE,
     last_name student.last_name%TYPE);

  -- Define collection type
  TYPE student_type IS TABLE OF student_rec;

  -- Define collection variable
  student_tab student_type;

  -- Define variable to be used by the LIMIT clause
  v_limit PLS_INTEGER := 50;

BEGIN
  OPEN student_cur;
  LOOP
    -- Fetch 50 rows at once
    FETCH student_cur BULK COLLECT INTO student_tab LIMIT v_limit;

    EXIT WHEN student_tab.COUNT = 0;

    FOR i IN student_tab.FIRST..student_tab.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE ('student_id: '||student_tab(i).student_id);
      DBMS_OUTPUT.PUT_LINE ('first_name: '||student_tab(i).first_name);
      DBMS_OUTPUT.PUT_LINE ('last_name: '||student_tab(i).last_name);
    END LOOP;
  END LOOP;
  CLOSE student_cur;
END;
```

```
DECLARE
  -- Define collection types and variables
  TYPE row_num_type IS TABLE OF NUMBER      INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

  row_num_tab row_num_type;
  row_text_tab row_text_type;

BEGIN
  DELETE FROM test
  RETURNING row_num, row_text
  BULK COLLECT INTO row_num_tab, row_text_tab;

  DBMS_OUTPUT.PUT_LINE ('Deleted '||SQL%ROWCOUNT||' rows:');

  FOR i IN row_num_tab.FIRST..row_num_tab.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
    ('row_num = '||row_num_tab(i)||' row_text = '||row_text_tab(i));
  END LOOP;

  COMMIT;
END;
```

```
Deleted 7 rows:  
row_num = 2 row_text = row 2  
row_num = 3 row_text = row 3  
row_num = 4 row_text = row 4  
row_num = 6 row_text = row 6  
row_num = 8 row_text = row 8  
row_num = 9 row_text = row 9  
row_num = 10 row_text = row 10
```

```
DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

    row_num_tab row_num_type;
    row_text_tab row_text_type;

BEGIN
    SELECT row_num, row_text
        BULK COLLECT INTO row_num_tab, row_text_tab
        FROM test;

    FOR i IN row_num_tab.FIRST..row_num_tab.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE
            ('row_num = '||row_num_tab(i)||' row_text = '||row_text_tab(i));
    END LOOP;
END;
```

```
ORA-06502: PL/SQL: numeric or value error  
ORA-06512: at line 14
```

```
FOR i IN row_num_tab.FIRST..row_num_tab.LAST
```

```
DECLARE
    -- Define collection types and variables
    TYPE row_num_type IS TABLE OF NUMBER           INDEX BY PLS_INTEGER;
    TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;
    row_num_tab row_num_type;
    row_text_tab row_text_type;

BEGIN
    SELECT row_num, row_text
    BULK COLLECT INTO row_num_tab, row_text_tab
    FROM test;

    IF row_num_tab.COUNT != 0
    THEN
        FOR i IN row_num_tab.FIRST..row_num_tab.LAST
        LOOP
            DBMS_OUTPUT.PUT_LINE
                ('row_num = '||row_num_tab(i)||' row_text = '||row_text_tab(i));
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE ('row_num_tab.COUNT = '||row_num_tab.COUNT);
        DBMS_OUTPUT.PUT_LINE ('row_text_tab.COUNT = '||row_text_tab.COUNT);
    END IF;
END;
```

```
CREATE TABLE my_zipcode AS
SELECT *
  FROM zipcode
 WHERE 1 = 2;

DECLARE
  -- Declare collection types
  TYPE string_type IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
  TYPE date_type   IS TABLE OF DATE           INDEX BY PLS_INTEGER;

  -- Declare collection variables to be used by the FORALL statement
  zip_tab      string_type;
  city_tab     string_type;
  state_tab    string_type;
  cr_by_tab    string_type;
  cr_date_tab  date_type;
  mod_by_tab   string_type;
  mod_date_tab date_type;

  v_rows INTEGER := 0;
BEGIN
  -- Populate individual collections
  SELECT *
    BULK COLLECT INTO zip_tab, city_tab, state_tab, cr_by_tab,
      cr_date_tab, mod_by_tab, mod_date_tab
   FROM zipcode
  WHERE state = 'CT';

  -- Populate MY_ZIPCODE table
  FORALL i in 1..zip_tab.COUNT
    INSERT INTO my_zipcode
      (zip, city, state, created_by, created_date, modified_by,
       modified_date)
    VALUES
      (zip_tab(i), city_tab(i), state_tab(i), cr_by_tab(i),
       cr_date_tab(i), mod_by_tab(i), mod_date_tab(i));
  COMMIT;

  -- Check how many records were added to MY_ZIPCODE table
  SELECT COUNT(*)
    INTO v_rows
   FROM my_zipcode;

  DBMS_OUTPUT.PUT_LINE (v_rows||' records were added to MY_ZIPCODE table');
END;
```

19 records were added to MY_ZIPCODE table

```
DECLARE
  -- Declare collection types
  TYPE string_type IS TABLE OF VARCHAR2(100);

  -- Declare collection variables to be used by the FORALL statement
  -- and BULK COLLECT clause
  zip_codes string_type := string_type ('06401', '06455', '06483', '06520', '06605');
  zip_tab   string_type;
  city_tab  string_type;

  v_rows INTEGER := 0;
BEGIN
  -- Delete some records from MY_ZIPCODE table
  FORALL i in zip_codes.FIRST..zip_codes.LAST
    DELETE FROM my_zipcode
      WHERE zip = zip_codes(i)
    RETURNING zip, city
    BULK COLLECT INTO zip_tab, city_tab;
  COMMIT;

  DBMS_OUTPUT.PUT_LINE ('The following records were deleted from MY_ZIPCODE table:');
  FOR i in zip_tab.FIRST..zip_tab.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Zip code '||zip_tab(i)||', city '||city_tab(i));
  END LOOP;
END;
```

The following records were deleted from MY_ZIPCODE table:
Zip code 06401, city Ansonia
Zip code 06455, city Middlefield
Zip code 06483, city Oxford
Zip code 06520, city New Haven
Zip code 06605, city Bridgeport

```
CREATE OR REPLACE PACKAGE test_adm_pkg
AS
  -- Define collection types
  TYPE row_num_type IS TABLE OF NUMBER          INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

  -- Define procedures
  PROCEDURE populate_test (row_num_tab ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE);

  PROCEDURE update_test (row_num_tab ROW_NUM_TYPE
                         ,row_num_type ROW_TEXT_TYPE);

  PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE);
END test_adm_pkg;
/

CREATE OR REPLACE PACKAGE BODY test_adm_pkg
AS
  PROCEDURE populate_test (row_num_tab ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE)
  IS
  BEGIN
    FORALL i IN 1..10
      INSERT INTO test (row_num, row_text)
      VALUES (row_num_tab(i), row_num_type(i));
  END populate_test;

  PROCEDURE update_test (row_num_tab ROW_NUM_TYPE
                         ,row_num_type ROW_TEXT_TYPE)
  IS
  BEGIN
    FORALL i IN 1..10
      UPDATE test
        SET row_text = row_num_type(i)
       WHERE row_num = row_num_tab(i);
  END update_test;

  PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE)
  IS
  BEGIN
    FORALL i IN 1..10
      DELETE from test
        WHERE row_num = row_num_tab(i);
  END delete_test;

END test_adm_pkg;
/
```

```
DECLARE
  row_num_tab test_adm_pkg.row_num_type;
  row_text_tab test_adm_pkg.row_text_type;

  v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Delete previously added data from the TEST table
  test_adm_pkg.delete_test (row_num_tab);

  -- Populate TEST table
  test_adm_pkg.populate_test (row_num_tab, row_text_tab);
  COMMIT;

  -- Check how many rows were inserted in the TEST table
  -- and display this number on the screen
  SELECT COUNT(*)
    INTO v_rows
   FROM TEST;

  DBMS_OUTPUT.PUT_LINE ('There are '||v_rows||' rows in the TEST table');
END;
```

There are 10 rows in the TEST table

```
DECLARE
  row_num_tab test_adm_pkg.row_num_type;
  row_text_tab test_adm_pkg.row_text_type;

  v_dyn_sql VARCHAR2(1000);
  v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Delete previously added data from the TEST table
  v_dyn_sql := 'begin test_adm_pkg.delete_test (:row_num_tab); end;';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab;

  -- Populate TEST table
  v_dyn_sql := 'begin test_adm_pkg.populate_test (:row_num_tab, :row_text_tab); end;';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab, row_text_tab;
  COMMIT;

  -- Check how many rows were inserted in the TEST table
  -- display it on the screen
  SELECT COUNT(*)
    INTO v_rows
   FROM TEST;

  DBMS_OUTPUT.PUT_LINE ('There are '||v_rows||' rows in the TEST table');
END;
```

```
-- Delete previously added data from the TEST table  
test_adm_pkg.delete_test (row_num_tab);  
  
-- Populate TEST table  
test_adm_pkg.populate_test (row_num_tab, row_text_tab);
```

```
-- Delete previously added data from the TEST table
v_dyn_sql := 'begin test_adm_pkg.delete_test (:row_num_tab); end;';
EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab;

-- Populate TEST table
v_dyn_sql := 'begin test_adm_pkg.populate_test (:row_num_tab, :row_text_tab); end;';
EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab, row_text_tab;
```

```
DECLARE
  row_num_tab test_adm_pkg.row_num_type;
  row_text_tab test_adm_pkg.row_text_type;

  v_dyn_sql VARCHAR2(1000);
  v_rows NUMBER;

BEGIN
  -- Populate collections
  FOR i IN 1..10
  LOOP
    row_num_tab(i) := i;
    row_text_tab(i) := 'row '||i;
  END LOOP;

  -- Delete previously added data from the TEST table
  v_dyn_sql := 'test_adm_pkg.delete_test (:row_num_tab);';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab;

  -- Populate TEST table
  v_dyn_sql := 'test_adm_pkg.populate_test (:row_num_tab, :row_text_tab);';
  EXECUTE IMMEDIATE v_dyn_sql USING row_num_tab, row_text_tab;
  COMMIT;

  -- Check how many rows were inserted in the TEST table
  -- display it on the screen
  SELECT COUNT(*)
    INTO v_rows
   FROM TEST;

  DBMS_OUTPUT.PUT_LINE ('There are'||v_rows||' rows in the TEST table');
END;
```

```
ORA-00900: invalid SQL statement
ORA-06512: at line 18
```

```

CREATE OR REPLACE PACKAGE test_adm_pkg
AS
  -- Define collection types
  TYPE row_num_type IS TABLE OF NUMBER           INDEX BY PLS_INTEGER;
  TYPE row_text_type IS TABLE OF VARCHAR2(10) INDEX BY PLS_INTEGER;

  -- Define record type
  TYPE rec_type IS RECORD
    (row_num NUMBER
     ,row_text VARCHAR2(10));

  -- Define procedures
  PROCEDURE populate_test (row_num_tab ROW_NUM_TYPE
                           ,row_num_type ROW_TEXT_TYPE);

  PROCEDURE update_test (row_num_tab ROW_NUM_TYPE
                         ,row_num_type ROW_TEXT_TYPE);

  PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE);

  PROCEDURE populate_test_rec (row_num_val IN NUMBER
                               ,test_rec OUT REC_TYPE);
END test_adm_pkg;
/

```

CREATE OR REPLACE PACKAGE BODY test_adm_pkg

- AS
- PROCEDURE populate_test (row_num_tab ROW_NUM_TYPE
 ,row_num_type ROW_TEXT_TYPE);
- IS
- BEGIN
- FORALL i IN 1..10
 INSERT INTO test (row_num, row_text)
 VALUES (row_num_tab(i), row_num_type(i));
- END populate_test;

```

  PROCEDURE update_test (row_num_tab ROW_NUM_TYPE
                        ,row_num_type ROW_TEXT_TYPE);
  IS
  BEGIN
    FORALL i IN 1..10
      UPDATE test
        SET row_text = row_num_type(i)
        WHERE row_num = row_num_tab(i);
  END update_test;

  PROCEDURE delete_test (row_num_tab ROW_NUM_TYPE)
  IS
  BEGIN
    FORALL i IN 1..10
      DELETE from test
        WHERE row_num = row_num_tab(i);
  END delete_test;

  PROCEDURE populate_test_rec (row_num_val IN NUMBER
                               ,test_rec OUT REC_TYPE)
  IS
  BEGIN
    SELECT *
      INTO test_rec
      FROM test
      WHERE row_num = row_num_val;
  END populate_test_rec;

END test_adm_pkg;
/

```

```
DECLARE
  test_rec test_adm_pkg.rec_type;

  v_dyn_sql VARCHAR2(1000);

BEGIN
  -- Select record from the TEST table
  v_dyn_sql := 'begin test_adm_pkg.populate_test_rec (:val, :rec); end;';
  EXECUTE IMMEDIATE v_dyn_sql USING IN 10, OUT test_rec;
  COMMIT;

  -- Display newly selected record
  DBMS_OUTPUT.PUT_LINE ('test_rec.row_num = '||test_rec.row_num);
  DBMS_OUTPUT.PUT_LINE ('test_rec.row_text = '||test_rec.row_text);
END;
```

```
test_rec.row_num = 10  
test_rec.row_text = row 10
```

```
DECLARE
  TYPE student_cur_typ IS REF CURSOR;
  student_cur student_cur_typ;
  student_rec student%ROWTYPE;
  v_zip_code student.zip%TYPE := '06820';

BEGIN
  OPEN student_cur
    FOR 'SELECT * FROM student WHERE zip = :my_zip' USING v_zip_code;

  LOOP
    FETCH student_cur INTO student_rec;
    EXIT WHEN student_cur%NOTFOUND;

    -- Display student ID, first and last names
    DBMS_OUTPUT.PUT_LINE ('student_rec.student_id = '||student_rec.student_id);
    DBMS_OUTPUT.PUT_LINE ('student_rec.first_name = '||student_rec.first_name);
    DBMS_OUTPUT.PUT_LINE ('student_rec.last_name = '||student_rec.last_name);
  END LOOP;
  CLOSE student_cur;
END;
```

```
student_rec.student_id = 240
student_rec.first_name = Z.A.
student_rec.last_name = Scrittoriale
student_rec.student_id = 326
student_rec.first_name = Piotr
student_rec.last_name = Padel
student_rec.student_id = 360
student_rec.first_name = Calvin
student_rec.last_name = Kiraly
```

```
CREATE OR REPLACE PACKAGE student_adm_pkg
AS
  -- Define collection type
  TYPE student_tab_type IS TABLE OF student%ROWTYPE INDEX BY PLS_INTEGER;

  -- Define procedures
  PROCEDURE populate_student_tab (zip_code      IN VARCHAR2
                                  ,student_tab OUT student_tab_type);

  PROCEDURE display_student_info (student_rec student%ROWTYPE);

END student_adm_pkg;
/

CREATE OR REPLACE PACKAGE BODY student_adm_pkg
AS
  PROCEDURE populate_student_tab (zip_code      IN VARCHAR2
                                  ,student_tab OUT student_tab_type)

  IS
  BEGIN
    SELECT *
      BULK COLLECT INTO student_tab
        FROM student
       WHERE zip = zip_code;
  END populate_student_tab;

  PROCEDURE display_student_info (student_rec student%ROWTYPE)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('student_rec.zip = '||student_rec.zip);
    DBMS_OUTPUT.PUT_LINE ('student_rec.student_id = '||student_rec.student_id);
    DBMS_OUTPUT.PUT_LINE ('student_rec.first_name = '||student_rec.first_name);
    DBMS_OUTPUT.PUT_LINE ('student_rec.last_name = '||student_rec.last_name);
  END display_student_info;

END student_adm_pkg;
/
```

```
DECLARE
  TYPE student_cur_typ IS REF CURSOR;
  student_cur student_cur_typ;

  -- Collection and record variables
  student_tab student_adm_pkg.student_tab_type;
  student_rec student%ROWTYPE;

BEGIN
  -- Populate collection of records
  student_adm_pkg.populate_student_tab ('06820', student_tab);

  OPEN student_cur
    FOR 'SELECT * FROM TABLE(:my_table)' USING student_tab;

  LOOP
    FETCH student_cur INTO student_rec;
    EXIT WHEN student_cur%NOTFOUND;

    student_adm_pkg.display_student_info (student_rec);
  END LOOP;
  CLOSE student_cur;
END;
```

```
student_rec.zip      = 06820
student_rec.student_id = 240
student_rec.first_name = Z.A.
student_rec.last_name  = Scrittore
student_rec.zip      = 06820
student_rec.student_id = 326
student_rec.first_name = Piotr
student_rec.last_name  = Padel
student_rec.zip      = 06820
student_rec.student_id = 360
student_rec.first_name = Calvin
student_rec.last_name  = Kiraly
```

```
CREATE OR REPLACE PROCEDURE name
  [(parameter[, parameter, ...])]

AS
  [local declarations]
BEGIN
  executable statements
[EXCEPTION
  exception handlers]
END [name];
```

```
CREATE OR REPLACE PROCEDURE Discount
AS
  CURSOR c_group_discount
  IS
    SELECT distinct s.course_no, c.description
      FROM section s, enrollment e, course c
     WHERE s.section_id = e.section_id
       AND c.course_no = s.course_no
   GROUP BY s.course_no, c.description,
            e.section_id, s.section_id
  HAVING COUNT(*) >=8;
BEGIN
  FOR r_group_discount IN c_group_discount
  LOOP
    UPDATE course
      SET cost = cost * .95
     WHERE course_no = r_group_discount.course_no;
    DBMS_OUTPUT.PUT_LINE
      ('A 5% discount has been given to ' ||
       r_group_discount.course_no||' ' ||
       r_group_discount.description
      );
  END LOOP;
END;
```

5% discount has been given to 25 Adv. Word Perfect
..... (through each course with an enrollment over 8)
PL/SQL procedure successfully completed.

```
L start_line_number end_line_number
```

```
SELECT object_name, object_type, status  
  FROM user_objects  
 WHERE object_name = 'DISCOUNT';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
DISCOUNT	PROCEDURE	VALID

```
SELECT TO_CHAR(line, 99) || '>', text
  FROM user_source
 WHERE name = 'DISCOUNT'
```

```
alter procedure procedure_name compile
```

```
formal_parameter_name => argument_value
```

```
CREATE OR REPLACE PROCEDURE find_sname
  (i_student_id IN NUMBER,
   o_first_name OUT VARCHAR2,
   o_last_name OUT VARCHAR2
  )
AS
BEGIN
  SELECT first_name, last_name
    INTO o_first_name, o_last_name
    FROM student
   WHERE student_id = i_student_id;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('Error in finding student_id:
  '||i_student_id);
END find_sname;
```

```
DECLARE
  v_local_first_name student.first_name%TYPE;
  v_local_last_name student.last_name%TYPE;
BEGIN
  find_sname
    (145, v_local_first_name, v_local_last_name);
  DBMS_OUTPUT.PUT_LINE
    ('Student 145 is: '||v_local_first_name||
     ' '|| v_local_last_name||'.'
    );
END;
```

```
CREATE [OR REPLACE] FUNCTION function_name
  (parameter list)
    RETURN datatype
IS
BEGIN
  <body>
  RETURN (return_value);
END;
```

```
CREATE OR REPLACE FUNCTION show_description
  (i_course_no course.course_no%TYPE)
RETURN varchar2
AS
  v_description varchar2(50);
BEGIN
  SELECT description
    INTO v_description
   FROM course
  WHERE course_no = i_course_no;
  RETURN v_description;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RETURN('The Course is not in the database');
  WHEN OTHERS
  THEN
    RETURN('Error in running show_description');
END;
```

FUNCTION SHOW_DESCRIPTION compiled

```
CREATE OR REPLACE FUNCTION id_is_good
  (i_student_id IN NUMBER)
  RETURN BOOLEAN
AS
  v_id_cnt NUMBER;
BEGIN
  SELECT COUNT(*)
    INTO v_id_cnt
   FROM student
  WHERE student_id = i_student_id;
  RETURN 1 = v_id_cnt;
EXCEPTION
  WHEN OTHERS
  THEN
    RETURN FALSE;
END id_is_good;
```

```
SET SERVEROUTPUT ON
DECLARE
  v_description VARCHAR2(50);
BEGIN
  v_description := show_description(&sv_cnumber);
  DBMS_OUTPUT.PUT_LINE(v_description);
END;
```

```
old 4: v_descript := show_description();
new 4: v_descript := show_description(350);
Java Developer II
PL/SQL procedure successfully completed.
```

```
DECLARE
  V_id number;
BEGIN
  V_id := &id;
  IF id_is_good(v_id)
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Student ID: '||v_id||' is a valid.');
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Student ID: '||v_id||' is not valid.');
  END IF;
END;
```

```
SELECT course_no, show_description(course_no)
  FROM course;
```

```
SELECT course_no, description  
FROM course;
```

```
SELECT UPPER('bill') FROM DUAL;
```

```
CREATE OR REPLACE FUNCTION new_instructor_id
  RETURN instructor.instructor_id%TYPE
AS
  v_new_instid instructor.instructor_id%TYPE;
BEGIN
  SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
    INTO v_new_instid
    FROM dual;
  RETURN v_new_instid;
EXCEPTION
  WHEN OTHERS
  THEN
    DECLARE
      v_sqlerrm VARCHAR2(250)
      := SUBSTR(SQLERRM,1,250);
    BEGIN
      RAISE_APPLICATION_ERROR(-20003,
        'Error in    instructor_id: '||v_sqlerrm);
    END;
END new_instructor_id;
```

```
WITH
  FUNCTION show_descript
  (i_course_no course.course_no%TYPE)
RETURN varchar2
AS
  v_description varchar2(50);
BEGIN
  SELECT description
    INTO v_description
   FROM course
  WHERE course_no = i_course_no;
  RETURN v_description;
END;
SELECT course_no, show_descript(course_no), cost
  FROM COURSE
```

```
CREATE OR REPLACE FUNCTION show_description
  (i_course_no course.course_no%TYPE)
RETURN varchar2
AS
  pragma UDF;
  v_description varchar2(50);
BEGIN
  SELECT description
    INTO v_description
   FROM course
  WHERE course_no = i_course_no;
  RETURN v_description;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    RETURN('The Course is not in the database');
  WHEN OTHERS
  THEN
    RETURN('Error in running show_description');
END;
```

```
PACKAGE package_name
IS
[ declarations of variables and types ]
[ specifications of cursors ]
[ specifications of modules ]
END [ package_name ];
```

```
PACKAGE BODY package_name
IS
  [ declarations of variables and types ]
  [ specification and SELECT statement of cursors ]
  [ specification and body of modules ]
  [ BEGIN
  executable statements ]
  [ EXCEPTION
  exception handlers ]
END [ package_name ];
```

```
1 CREATE OR REPLACE PACKAGE manage_students
2 AS
3   PROCEDURE find_sname
4     (i_student_id IN student.student_id%TYPE,
5      o_first_name OUT student.first_name%TYPE,
6      o_last_name OUT student.last_name%TYPE
7    );
8   FUNCTION id_is_good
9     (i_student_id IN student.student_id%TYPE)
10  RETURN BOOLEAN;
11 END manage_students;
```

```
SET SERVEROUTPUT ON
DECLARE
  v_first_name student.first_name%TYPE;
  v_last_name student.last_name%TYPE;
BEGIN
  manage_students.find_sname
  (125, v_first_name, v_last_name);
  DBMS_OUTPUT.PUT_LINE(v_first_name||' '||v_last_name);
END;
```

```
ERROR at line 1:  
ORA-04068: existing state of packages has been discarded  
ORA-04067: not executed, package body  
    "STUDENT.MANAGE_STUDENTS" does not exist  
ORA-06508: PL/SQL: could not find program  
          unit being called  
ORA-06512: at line 5
```

```
CREATE OR REPLACE PACKAGE school_api AS
  PROCEDURE discount_cost;
  FUNCTION new_instructor_id
    RETURN instructor.instructor_id%TYPE;
END school_api;
```

```
1 CREATE OR REPLACE PACKAGE BODY manage_students
2 AS
3     PROCEDURE find_sname
4         (i_student_id IN student.student_id%TYPE,
5          o_first_name OUT student.first_name%TYPE,
6          o_last_name OUT student.last_name%TYPE
7      )
8     IS
9        v_student_id student.student_id%TYPE;
10    BEGIN
11        SELECT first_name, last_name
12            INTO o_first_name, o_last_name
13            FROM student
14            WHERE student_id = i_student_id;
15    EXCEPTION
16        WHEN OTHERS
17        THEN
18            DBMS_OUTPUT.PUT_LINE
19            ('Error in finding student_id: '||v_student_id);
20    END find_sname;
21    FUNCTION id_is_good
22        (i_student_id IN student.student_id%TYPE)
23        RETURN BOOLEAN
24    IS
25        v_id_cnt number;
26    BEGIN
27        SELECT COUNT(*)
28            INTO v_id_cnt
29            FROM student
30            WHERE student_id = i_student_id;
31        RETURN 1 = v_id_cnt;
32    EXCEPTION
33        WHEN OTHERS
34        THEN
35            RETURN FALSE;
36    END id_is_good;
37    END manage_students;
```

```
1 CREATE OR REPLACE PACKAGE BODY school_api AS
2     PROCEDURE discount_cost
3     IS
4         CURSOR c_group_discount
5         IS
6             SELECT distinct s.course_no, c.description
7                 FROM section s, enrollment e, course c
8                 WHERE s.section_id = e.section_id
9                 GROUP BY s.course_no, c.description,
10                         e.section_id, s.section_id
11                 HAVING COUNT(*) >=8;
12 BEGIN
13     FOR r_group_discount IN c_group_discount
14     LOOP
15         UPDATE course
16             SET cost = cost * .95
17             WHERE course_no = r_group_discount.course_no;
18             DBMS_OUTPUT.PUT_LINE
19                 ('A 5% discount has been given to'
20                  ||r_group_discount.course_no||'
21                  ||r_group_discount.description);
22     END LOOP;
23     END discount_cost;
24     FUNCTION new_instructor_id
25     RETURN instructor.instructor_id%TYPE
26     IS
27         v_new_instid instructor.instructor_id%TYPE;
28     BEGIN
29         SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
30             INTO v_new_instid
31             FROM dual;
32         RETURN v_new_instid;
33     EXCEPTION
34         WHEN OTHERS
35         THEN
36             DECLARE
37                 v_sqlerrm VARCHAR2(250) :=
38                     SUBSTR(SQLERRM,1,250);
39             BEGIN
40                 RAISE_APPLICATION_ERROR(-20003,
41                     'Error in instructor_id: '||v_sqlerrm);
42             END;
43     END new_instructor_id;
44 END school_api;
```

```
SET SERVEROUTPUT ON
DECLARE
    v_first_name student.first_name%TYPE;
    v_last_name student.last_name%TYPE;
BEGIN
    IF manage_students.id_is_good(&&v_id)
    THEN
        manage_students.find_sname(&&v_id, v_first_name,
                                    v_last_name);
        DBMS_OUTPUT.PUT_LINE('Student No. '||&&v_id||' is '
                            ||v_last_name||', '||v_first_name);
    ELSE
        DBMS_OUTPUT.PUT_LINE
        ('Student ID: '||&&v_id||' is not in the database.');
    END IF;
END;
```

```
old:DECLARE
  v_first_name student.first_name%TYPE;
  v_last_name student.last_name%TYPE;
BEGIN
  IF manage_students.id_is_good(&&v_id)
  THEN
    manage_students.find_sname(&&v_id, v_first_name,
      v_last_name);
    DBMS_OUTPUT.PUT_LINE('Student No. '||&&v_id||' is '
      ||v_last_name||', '||v_first_name);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Student ID: '||&&v_id||' is not in the database.');
  END IF;
END;
new:DECLARE
  v_first_name student.first_name%TYPE;
  v_last_name student.last_name%TYPE;
BEGIN
  IF manage_students.id_is_good(145)
  THEN
    manage_students.find_sname(145, v_first_name,
      v_last_name);
    DBMS_OUTPUT.PUT_LINE('Student No. '||145||' is '
      ||v_last_name||', '||v_first_name);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Student ID: '||145||' is not in the database.');
  END IF;
END;
anonymous block completed
Student No. 145 is Lefkowitz, Paul
```

```
SET SERVEROUTPUT ON
DECLARE
  V_instructor_id instructor.instructor_id%TYPE;
BEGIN
  School_api.Discount_Cost;
  v_instructor_id := school_api.new_instructor_id;
  DBMS_OUTPUT.PUT_LINE
    ('The new id is: '||v_instructor_id);
END;
```

```
11 PROCEDURE display_student_count;
12 END manage_students;
```

```
37  FUNCTION student_count_priv
38    RETURN NUMBER
39  IS
40    v_count NUMBER;
41  BEGIN
42    select count(*)
43    into v_count
44    from student;
45    return v_count;
46  EXCEPTION
47    WHEN OTHERS
48      THEN
49      return(0);
50  END student_count_priv;
51  PROCEDURE display_student_count
52  IS
53    v_count NUMBER;
54  BEGIN
55    v_count := student_count_priv;
56    DBMS_OUTPUT.PUT_LINE
57    ('There are '|v_count||' students.');
58  END display_student_count;
59 END manage_students;
```

```
DECLARE
  V_count NUMBER;
BEGIN
  V_count := Manage_students.student_count_priv;
  DBMS_OUTPUT.PUT_LINE(v_count);
END;
```

```
ERROR at line 1:  
ORA-06550: line 4, column 31:  
PLS-00302: component 'STUDENT_COUNT_PRIV' must be declared  
ORA-06550: line 4, column 3:  
PL/SQL: Statement ignored
```

```
SET SERVEROUTPUT ON
Execute manage_students.display_student_count;
```

PLS-00323: subprogram or cursor 'procedure_name' is
declared in a package specification and must be
defined in the package body

```
CREATE OR REPLACE PACKAGE manage_students
AS
  PROCEDURE find_sname
    (i_student_id IN student.student_id%TYPE,
     o_first_name OUT student.first_name%TYPE,
     o_last_name OUT student.last_name%TYPE
    );
  FUNCTION id_is_good
    (i_student_id IN student.student_id%TYPE)
    RETURN BOOLEAN;
  PROCEDURE display_student_count;
END manage_students;
```

```

CREATE OR REPLACE PACKAGE BODY manage_students
AS
    PROCEDURE find_sname
        (i_student_id IN student.student_id%TYPE,
         o_first_name OUT student.first_name%TYPE,
         o_last_name OUT student.last_name%TYPE
        )
    IS
        v_student_id student.student_id%TYPE;
    BEGIN
        SELECT first_name, last_name
          INTO o_first_name, o_last_name
         FROM student
        WHERE student_id = i_student_id;
    EXCEPTION
        WHEN OTHERS
            THEN
                DBMS_OUTPUT.PUT_LINE
                    ('Error in finding student_id: '||v_student_id);
    END find_sname;

    FUNCTION id_is_good
        (i_student_id IN student.student_id%TYPE)
        RETURN BOOLEAN
    IS

        v_id_cnt number;
    BEGIN
        SELECT COUNT(*)
          INTO v_id_cnt
         FROM student
        WHERE student_id = i_student_id;
        RETURN 1 = v_id_cnt;
    EXCEPTION
        WHEN OTHERS
            THEN
                RETURN FALSE;
        END id_is_good;
    FUNCTION student_count_priv
        RETURN NUMBER
    IS
        v_count NUMBER;
    BEGIN
        select count(*)
        into v_count
        from student;
        return v_count;
    EXCEPTION
        WHEN OTHERS
            THEN
                return(0);
    END student_count_priv;
    PROCEDURE display_student_count
        is

```

```
v_count NUMBER;
BEGIN
  v_count := student_count_priv;
  DBMS_OUTPUT.PUT_LINE
    ('There are '||v_count||' students.');
END display_student_count;
FUNCTION get_course_descript_private
  (i_course_no course.course_no%TYPE)
  RETURN course.description%TYPE
IS
  v_course_descript course.description%TYPE;
BEGIN
  SELECT description
    INTO v_course_descript
    FROM course
   WHERE course_no = i_course_no;
  RETURN v_course_descript;
EXCEPTION
  WHEN OTHERS
  THEN
    RETURN NULL;
END get_course_descript_private;
END manage_students;
```

```
TYPE inst_city_type IS RECORD
(first_name instructor.first_name%TYPE;
last_name   instructor.last_name%TYPE;
city        zipcode.city%TYPE;
state       zipcode.state%TYPE)
```

```
TYPE ref_type_name is REF CURSOR [RETURN return_type];
```

```
TYPE inst_city_cur IS REF CURSOR RETURN inst_city_type;
```

```

CREATE OR REPLACE PACKAGE course_pkg AS
  TYPE course_rec_t typ IS RECORD
    (first_name student.first_name%TYPE,
     last_name student.last_name%TYPE,
     course_no course.course_no%TYPE,
     description course.description%TYPE,
     section_no section.section_no%TYPE
    );
  TYPE course_cur IS REF CURSOR RETURN course_rec_typ;
  PROCEDURE get_course_list
    (p_student_id NUMBER ,
     p_instructor_id NUMBER ,
     course_list_cv IN OUT course_cur);
END course_pkg;
/

CREATE OR REPLACE PACKAGE BODY course_pkg AS
  PROCEDURE get_course_list
    (p_student_id NUMBER ,
     p_instructor_id NUMBER ,
     course_list_cv IN OUT course_cur)
  IS
  BEGIN
    IF p_student_id IS NULL AND p_instructor_id
      IS NULL THEN
      OPEN course_list_cv FOR
        SELECT 'Please choose a student-' First_name,
              'instructor combination' Last_name,
              NULL course_no,
              NULL description,
              NULL section_no
        FROM dual;

      ELSIF p_student_id IS NULL THEN
      OPEN course_list_cv FOR
        SELECT s.first_name first_name,
               s.last_name last_name,
               c.course_no course_no,
               c.description description,
               se.section_no section_no
        FROM instructor i, student s,
             section se, course c, enrollment e
        WHERE i.instructor_id = p_instructor_id
          AND i.instructor_id = se.instructor_id
          AND se.course_no = c.course_no
          AND e.student_id = s.student_id
          AND e.section_id = se.section_id
        ORDER BY c.course_no, se.section_no;
      ELSIF p_instructor_id IS NULL THEN
      OPEN course_list_cv FOR
        SELECT i.first_name first_name,
               i.last_name last_name,
               c.course_no course_no,
               c.description description,
               se.section_no section_no
        FROM instructor i, student s,
             section se, course c, enrollment e
        WHERE s.student_id = p_student_id
          AND i.instructor_id = se.instructor_id
          AND se.course_no = c.course_no
          AND e.student_id = s.student_id
          AND e.section_id = se.section_id
        ORDER BY c.course_no, se.section_no;
      END IF;
    END get_course_list;
  END course_pkg;

```

```
VARIABLE course_cv REF CURSOR
```

```
exec course_pkg.get_course_list(102, NULL, :course_cv);
```

```
SQL> print course_cv
```

FIRST_NAME	LAST_NAME	COURSE_NO	DESCRIPTION	SECTION_NO
Charles	Lowry	25	Intro to Programming	2
Nina	Schorin	25	Intro to Programming	5

```
SQL> exec course_pkg.get_course_list(NULL, 102, :course_cv);
PL/SQL procedure successfully completed.
```

```
SQL> print course_cv
```

FIRST_NAME	LAST_NAME	COURSE_NO DESCRIPTION	SECTION_NO
Jeff	Runyan	10 Technology Concepts	2
Dawn	Dennis	25 Intro to Programming	4
May	Jodoin	25 Intro to Programming	4
Jim	Joas	25 Intro to Programming	4
Arun	Griffen	25 Intro to Programming	4
Alfred	Hutheesing	25 Intro to Programming	4
Lula	Oates	100 Hands-On Windows	1
Regina	Bose	100 Hands-On Windows	1
Jenny	Goldsmith	100 Hands-On Windows	1
Roger	Snow	100 Hands-On Windows	1
Rommel	Frost	100 Hands-On Windows	1
Debra	Boyce	100 Hands-On Windows	1
Janet	Jung	120 Intro to Java Programming	4
John	Smith	124 Advanced Java Programming	1
Charles	Caro	124 Advanced Java Programming	1
Sharon	Thompson	124 Advanced Java Programming	1
Evan	Fielding	124 Advanced Java Programming	1
Ronald	Tangaribuan	124 Advanced Java Programming	1
N	Kuehn	146 Java for C/C++ Programmers	2
Derrick	Baltazar	146 Java for C/C++ Programmers	2
Angela	Torres	240 Intro to the Basic Language	2

```
SQL> exec course_pkg.get_course_list(NULL, NULL, :course_cv);
PL/SQL procedure successfully completed.
```

```
SQL> print course_cv
```

FIRST_NAME	LAST_NAME	C DESCRIPTION	S

Please choose a student- instructor combination			

```

CREATE OR REPLACE PACKAGE student_info_pkg AS
  TYPE student_details IS REF CURSOR;
  PROCEDURE get_student_info
    (p_student_id NUMBER ,
     p_choice      NUMBER ,
     details_cv IN OUT student_details);
END student_info_pkg;
/
CREATE OR REPLACE PACKAGE BODY student_info_pkg AS
  PROCEDURE get_student_info
    (p_student_id NUMBER ,
     p_choice      NUMBER ,
     details_cv IN OUT student_details)
  IS
  BEGIN
    IF p_choice = 1 THEN
      OPEN details_cv FOR
        SELECT s.first_name   first_name,
               s.last_name    last_name,
               s.street_address address,
               z.city         city,
               z.state        state,
               z.zip          zip
          FROM student s, zipcode z
         WHERE s.student_id = p_student_id
           AND z.zip = s.zip;
    ELSIF p_choice = 2 THEN
      OPEN details_cv FOR
        SELECT c.course_no course_no,
               c.description      description,
               se.section_no       section_no,
               s.first_name first_name,
               s.last_name  last_name
          FROM student s, section se,
               course c, enrollment e
         WHERE se.course_no = c.course_no
           AND e.student_id = s.student_id
           AND e.section_id = se.section_id
           AND se.section_id in (SELECT e.section_id
                                  FROM student s,
                                       enrollment e
                                 WHERE s.student_id =
                                       p_student_id
                                       AND s.student_id =
                                         e.student_id)
        ORDER BY c.course_no;
    ELSIF p_choice = 3 THEN
      OPEN details_cv FOR
        SELECT i.first_name   first_name,
               i.last_name    last_name,
               c.course_no    course_no,
               c.description   description,
               se.section_no   section_no
          FROM instructor i, student s,
               section se, course c, enrollment e
         WHERE s.student_id = p_student_id
           AND i.instructor_id = se.instructor_id
           AND se.course_no = c.course_no
           AND e.student_id = s.student_id
           AND e.section_id = se.section_id
        ORDER BY c.course_no, se.section_no;
    END IF;
  END get_student_info;
END student_info_pkg;

```

```
VARIABLE student_cv REF CURSOR
```

```
SQL> execute student_info_pkg.GET_STUDENT_INFO  
(102, 1, :student_cv);
```

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

```
SQL> print student_cv
FIRST_ LAST_NAM ADDRESS          CITY          ST ZIP
-----
Fred   Crocitto 101-09 120th St. Richmond Hill NY 11419

SQL> execute student_info_pkg.GET_STUDENT_INFO
      (102, 2, :student_cv);
PL/SQL procedure successfully completed.
```

```
SQL> print student_cv
COURSE_NO DESCRIPTION      SECTION_NO FIRST_NAME LAST_NAME
-----
25 Intro to Programming    2 Fred        Crocitto
25 Intro to Programming    2 Judy        Sethi
5 Intro to Programming     2 Jenny       Goldsmith
25 Intro to Programming     2 Barbara      Robichaud
25 Intro to Programming     2 Jeffrey     Citron
25 Intro to Programming     2 George      Kocka
25 Intro to Programming     5 Fred        Crocitto
25 Intro to Programming     5 Hazel       Lasseter
25 Intro to Programming     5 James       Miller
25 Intro to Programming     5 Regina      Gates
25 Intro to Programming     5 Arlyne      Sheppard
25 Intro to Programming     5 Thomas      Edwards
25 Intro to Programming     5 Sylvia      Perrin
25 Intro to Programming     5 M.         Diokno
25 Intro to Programming     5 Edgar       Moffat
25 Intro to Programming     5 Bessie      Heedles
25 Intro to Programming     5 Walter      Boremann
25 Intro to Programming     5 Lorrane     Velasco
```

```
SQL> execute student_info_pkg.GET_STUDENT_INFO
      (214, 3, :student_cv);
PL/SQL procedure successfully completed.
```

```
SQL> print student_cv
FIRST_NAME LAST_NAME      COURSE_NO DESCRIPTION      SECTION_NO
-----
Marilyn Frantzen        120 Intro to Java Programming  1
Fernand Hanks           122 Intermediate Java Programming 5
Gary   Pertez           130 Intro to Unix            2
Marilyn Frantzen        145 Internet Protocols        1
```

```
SELECT GRADE_TYPE_CODE,
       NUMBER_PER_SECTION,
       PERCENT_OF_FINAL_GRADE,
       DROP_LOWEST
  FROM grade_Type_weight
 WHERE section_id = 106
   AND section_id IN (SELECT section_id
                      FROM grade
                     WHERE student_id = 145)
```

```
CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
-- Cursor to loop through all grade types for a given section
CURSOR c_grade_type
  (pc_section_id  section.section_id%TYPE,
   PC_student_ID  student.student_id%TYPE)
IS
SELECT GRADE_TYPE_CODE,
       NUMBER_PER_SECTION,
       PERCENT_OF_FINAL_GRADE,
       DROP_LOWEST
  FROM grade_Type_weight
 WHERE section_id = pc_section_id
   AND section_id IN (SELECT section_id
                        FROM grade
                       WHERE student_id = pc_student_id);
END MANAGE_GRADES;
```

```
CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
  -- Cursor to loop through all grade types for a given section.
  CURSOR c_grade_type
    (pc_section_id section.section_id%TYPE,
     PC_student_ID student.student_id%TYPE)
  IS
    SELECT GRADE_TYPE_CODE,
           NUMBER_PER_SECTION,
           PERCENT_OF_FINAL_GRADE,
           DROP_LOWEST
      FROM grade_Type_weight
     WHERE section_id = pc_section_id
       AND section_id IN (SELECT section_id
                           FROM grade
                          WHERE student_id = pc_student_id);
  -- Cursor to loop through all grades for a given student
  -- in a given section.
  CURSOR c_grades
    (p_grade_type_code
     grade_Type_weight.grade_type_code%TYPE,
     pc_student_id student.student_id%TYPE,
     pc_section_id section.section_id%TYPE) IS
    SELECT grade_type_code,grade_code_occurrence,
           numeric_grade
      FROM grade
     WHERE student_id = pc_student_id
       AND section_id = pc_section_id
       AND grade_type_code = p_grade_type_code;
END MANAGE_GRADES;
```

```
CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
    -- Cursor to loop through all grade types for a given section.
    CURSOR c_grade_type
        (pc_section_id section.section_id%TYPE,
         PC_student_ID student.student_id%TYPE)
        IS
            SELECT GRADE_TYPE_CODE,
                   NUMBER_PER_SECTION,
                   PERCENT_OF_FINAL_GRADE,
                   DROP_LOWEST
              FROM grade_Type_weight
             WHERE section_id = pc_section_id
               AND section_id IN (SELECT section_id
                                   FROM grade
                                  WHERE student_id = pc_student_id);
    -- Cursor to loop through all grades for a given student
    -- in a given section.
    CURSOR c_grades
        (p_grade_type_code
         grade_Type_weight.grade_type_code%TYPE,
         pc_student_id student.student_id%TYPE,
         pc_section_id section.section_id%TYPE) IS
            SELECT grade_type_code,grade_code_occurrence,
                   numeric_grade
              FROM grade
             WHERE student_id = pc_student_id
               AND section_id = pc_section_id
               AND grade_type_code = p_grade_type_code;
    -- Function to calculate a student's final grade
    -- in one section
    Procedure final_grade
        (P_student_id   IN student.student_id%type,
         P_section_id   IN section.section_id%TYPE,
         P_Final_grade OUT enrollment.final_grade%TYPE,
         P_Exit_Code     OUT CHAR);
END MANAGE_GRADES;
```

```
CREATE OR REPLACE PACKAGE BODY MANAGE_GRADES AS
  Procedure final_grade
    (P_student_id    IN student.student_id%TYPE,
     P_section_id   IN section.section_id%TYPE,
     P_Final_grade  OUT enrollment.final_grade%TYPE,
     P_Exit_Code     OUT CHAR)
  IS
    v_student_id          student.student_id%TYPE;
    v_section_id          section.section_id%TYPE;
    v_grade_type_code     grade_type_weight.grade_type_code%TYPE;
    v_grade_percent       NUMBER;
    v_final_grade         NUMBER;
    v_grade_count         NUMBER;
    v_lowest_grade        NUMBER;
    v_exit_code           CHAR(1) := 'S';
    v_no_rows1             CHAR(1) := 'N';
    v_no_rows2             CHAR(1) := 'N';
    e_no_grade             EXCEPTION;
  BEGIN
    NULL;
  END;
END MANAGE_GRADES;
```

```

CREATE OR REPLACE PACKAGE BODY MANAGE_GRADES AS
  Procedure final_grade
    (P_student_id    IN student.student_id%TYPE,
     P_section_id    IN section.section_id%TYPE,
     P_Final_grade   OUT enrollment.final_grade%TYPE,
     P_Exit_Code      OUT CHAR)
  IS
    v_student_id          student.student_id%TYPE;
    v_section_id          section.section_id%TYPE;
    v_grade_type_code     grade_type_weight.grade_type_code%TYPE;
    v_grade_percent        NUMBER;
    v_final_grade         NUMBER;
    v_grade_count          NUMBER;
    v_lowest_grade         NUMBER;
    v_exit_code            CHAR(1) := 'S';
    v_no_rows1             CHAR(1) := 'N';
    v_no_rows2             CHAR(1) := 'N';
    e_no_grade              EXCEPTION;
  BEGIN
    v_section_id := p_section_id;
    v_student_id := p_student_id;
    -- Start loop of grade types for the section.
    FOR r_grade IN c_grade_type(v_section_id, v_student_id)
    LOOP
    -- Since cursor is open it has a result
    -- set; change indicator.
    v_no_rows1 := 'Y';

    -- To hold the number of grades per section,
    -- reset to 0 before detailed cursor loops.
    v_grade_count := 0;
    v_grade_type_code := r_grade.GRADE_TYPE_CODE;
    -- Variable to hold the lowest grade.
    -- 500 will not be the lowest grade.
    v_lowest_grade := 500;
    -- Determine what to multiply a grade by to
    -- compute final grade; must take into consideration
    -- if the drop lowest grade indicator is Y.
    SELECT (r_grade.percent_of_final_grade /
           DECODE(r_grade.drop_lowest, 'Y',
                  (r_grade.number_per_section - 1),
                  r_grade.number_per_section
           )) * 0.01
    INTO v_grade_percent
    FROM dual;
    -- Open cursor of detailed grade for a student in a
    -- given section.
    FOR r_detail IN c_grades(v_grade_type_code,
                             v_student_id, v_section_id) LOOP
    -- Since cursor is open it has a result
    -- set; change indicator.
    v_no_rows2 := 'Y';
    v_grade_count := v_grade_count + 1;
    -- Handle the situation where there are more
    -- entries for grades of a given grade type
    -- than there should be for that section.
    IF v_grade_count > r_grade.number_per_section THEN
      v_exit_code := 'T';
      raise e_no_grade;
    END IF;
  END;

```

```

-- If drop lowest flag is Y determine which is lowest
-- grade to drop.
    IF r_grade.drop_lowest = 'Y' THEN
        IF nvl(v_lowest_grade, 0) >=
            r_detail.numeric_grade
        THEN
            v_lowest_grade := r_detail.numeric_grade;
        END IF;
    END IF;
-- Increment the final grade with percentage of current
-- grade in the detail loop.
    v_final_grade := nvl(v_final_grade, 0) +
        (r_detail.numeric_grade * v_grade_percent);
END LOOP;
-- Once detailed loop is finished, if the number of grades
-- for a given student for a given grade type and section
-- is less than the required amount, raise an exception.
    IF v_grade_count < r_grade.NUMBER_PER_SECTION THEN
        v_exit_code := 'I';
        raise e_no_grade;
    END IF;
-- If the drop lowest flag was Y, then you need to take
-- the lowest grade out of the final grade; it was not
-- known when it was added which was the lowest grade
-- to drop until all grades were examined.
    IF r_grade.drop_lowest = 'Y' THEN
        v_final_grade := nvl(v_final_grade, 0) -
            (v_lowest_grade * v_grade_percent);
    END IF;
END LOOP;
-- If either cursor had no rows then there is an error.
IF v_no_rows1 = 'N' OR v_no_rows2 = 'N' THEN
    v_exit_code := 'N';
    raise e_no_grade;
END IF;

P_final_grade := v_final_grade;
P_exit_code   := v_exit_code;
EXCEPTION
    WHEN e_no_grade THEN
        P_final_grade := null;
        P_exit_code   := v_exit_code;
    WHEN OTHERS THEN
        P_final_grade := null;
        P_exit_code   := 'E';
END final_grade;
END MANAGE_GRADES;

```

```
SQL> desc manage_grades
PROCEDURE FINAL_GRADE
```

Argument Name	Type	In/Out Default?
P_STUDENT_ID	NUMBER(8)	IN
P_SECTION_ID	NUMBER(8)	IN
P_FINAL_GRADE	NUMBER(3)	OUT
P_EXIT_CODE	CHAR	OUT

```
SET SERVEROUTPUT ON

DECLARE
  v_student_id    student.student_id%TYPE := &sv_student_id;
  v_section_id    section.section_id%TYPE := &sv_section_id;
  v_final_grade   enrollment.final_grade%TYPE;
  v_exit_code     CHAR;
BEGIN
  manage_grades.final_grade(v_student_id, v_section_id,
    v_final_grade, v_exit_code);
  DBMS_OUTPUT.PUT_LINE('The Final Grade is'||v_final_grade);
  DBMS_OUTPUT.PUT_LINE('The Exit Code is'||v_exit_code);
END;
```

```
Enter value for sv_student_id: 102
old 2: v_student_id student.student_id%TYPE := &sv_student_id;
new 2: v_student_id student.student_id%TYPE := 102;
Enter value for sv_section_id: 86
old 3: v_section_id section.section_id%TYPE := &sv_section_id;
new 3: v_section_id section.section_id%TYPE := 86;
The Final Grade is 89
The Exit Code is 0
PL/SQL procedure successfully completed.
```

```

CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
  -- Cursor to loop through all grade types for a given section.
  CURSOR c_grade_type
    (pc_section_id section.section_id%TYPE,
     PC_student_ID student.student_id%TYPE)
  IS
    SELECT GRADE_TYPE_CODE,
           NUMBER_PER_SECTION,
           PERCENT_OF_FINAL_GRADE,
           DROP_LOWEST
      FROM grade_Type_weight
     WHERE section_id = pc_section_id
       AND section_id IN (SELECT section_id
                           FROM grade
                          WHERE student_id = pc_student_id);
  -- Cursor to loop through all grades for a given student
  -- in a given section.
  CURSOR c_grades
    (p_grade_type_code
     grade_Type_weight.grade_type_code%TYPE,
     pc_student_id student.student_id%TYPE,
     pc_section_id section.section_id%TYPE) IS
    SELECT grade_type_code,grade_code_occurrence,
           numeric_grade
      FROM grade
     WHERE student_id = pc_student_id
       AND section_id = pc_section_id
       AND grade_type_code = p_grade_type_code;
  -- Function to calculate a student's final grade
  -- in one section.
  Procedure final_grade
    (P_student_id   IN student.student_id%type,
     P_section_id   IN section.section_id%TYPE,
     P_Final_grade OUT enrollment.final_grade%TYPE,
     P_Exit_Code    OUT CHAR);

  -----
  -- Function to calculate the median grade .
  FUNCTION median_grade
    (p_course_number section.course_no%TYPE,
     p_section_number section.section_no%TYPE,
     p_grade_type grade.grade_type_code%TYPE)
  RETURN grade.numeric_grade%TYPE;
  CURSOR c_work_grade
    (p_course_no section.course_no%TYPE,
     p_section_no section.section_no%TYPE,
     p_grade_type_code grade.grade_type_code%TYPE
    ) IS
    SELECT distinct numeric_grade
      FROM grade
     WHERE section_id = (SELECT section_id
                           FROM section
                          WHERE course_no= p_course_no
                            AND section_no = p_section_no)
           AND grade_type_code = p_grade_type_code
    ORDER BY numeric_grade;
  TYPE t_grade_type IS TABLE OF c_work_grade%ROWTYPE
    INDEX BY BINARY_INTEGER;
  t_grade t_grade_type;
END MANAGE_GRADES;

```

```

CREATE OR REPLACE PACKAGE MANAGE_GRADES AS
CREATE OR REPLACE PACKAGE BODY MANAGE_GRADES AS
    Procedure final_grade
        (P_student_id IN student.student_id%TYPE,
         P_section_id IN section.section_id%TYPE,
         P_Final_grade OUT enrollment.final_grade%TYPE,
         P_Exit_Code OUT CHAR)
    IS
        v_student_id          student.student_id%TYPE;
        v_section_id          section.section_id%TYPE;
        v_grade_type_code     grade_type_weight.grade_type_code%TYPE;
        v_grade_percent       NUMBER;
        v_final_grade         NUMBER;
        v_grade_count         NUMBER;
        v_lowest_grade        NUMBER;
        v_exit_code           CHAR(1) := 'S';
        -- Next two variables are used to calculate whether a cursor
        -- has no result set.
        v_no_rows1             CHAR(1) := 'N';
        v_no_rows2             CHAR(1) := 'N';
        e_no_grade             EXCEPTION;
    BEGIN
        v_section_id := p_section_id;
        v_student_id := p_student_id;
        -- Start loop of grade types for the section.
        FOR r_grade IN c_grade_type(v_section_id, v_student_id)
        LOOP
            -- Since cursor is open it has a result
            -- set; change indicator.
            v_no_rows1 := 'Y';
            -- To hold the number of grades per section,
            -- reset to 0 before detailed cursor loops.
            v_grade_count := 0;
            v_grade_type_code := r_grade.GRADE_TYPE_CODE;
            -- Variable to hold the lowest grade.
            -- 500 will not be the lowest grade.
            v_lowest_grade := 500;
            -- Determine what to multiply a grade by to
            -- compute final grade; must take into consideration
            -- if the drop lowest grade indicator is Y.
            SELECT (r_grade.percent_of_final_grade /
                    DECODE(r_grade.drop_lowest, 'Y',
                           (r_grade.number_per_section - 1),
                           r_grade.number_per_section
                           )) * 0.01
            INTO v_grade_percent
            FROM dual;
            -- Open cursor of detailed grade for a student in a
            -- given section.
            FOR r_detail IN c_grades(v_grade_type_code,
                                      v_student_id, v_section_id) LOOP
                -- Since cursor is open it has a result
                -- set; change indicator.
                v_no_rows2 := 'Y';
                v_grade_count := v_grade_count + 1;
                -- Handle the situation where there are more
                -- entries for grades of a given grade type
                -- than there should be for that section.
                IF v_grade_count > r_grade.number_per_section THEN
                    v_exit_code := 'T';
                    raise e_no_grade;
                END IF;
            END LOOP;
        END LOOP;
    END final_grade;
END MANAGE_GRADES;

```

```

-- If drop lowest flag is Y determine which is lowest
-- grade to drop.
    IF r_grade.drop_lowest = 'Y' THEN
        IF nvl(v_lowest_grade, 0) >=
            r_detail.numeric_grade
        THEN
            v_lowest_grade := r_detail.numeric_grade;
        END IF;
    END IF;
-- Increment the final grade with percentage of current
-- grade in the detail loop.
    v_final_grade := nvl(v_final_grade, 0) +
        (r_detail.numeric_grade * v_grade_percent);
    END LOOP;
-- Once detailed loop is finished, if the number of grades
-- for a given student for a given grade type and section
-- is less than the required amount, raise an exception.
    IF v_grade_count < r_grade.NUMBER_PER_SECTION THEN
        v_exit_code := 'I';
        raise e_no_grade;
    END IF;
-- If the drop lowest flag was Y then you need to take
-- the lowest grade out of the final grade. It was not
-- known when it was added which was the lowest grade
-- to drop until all grades were examined.
    IF r_grade.drop_lowest = 'Y' THEN
        v_final_grade := nvl(v_final_grade, 0) -
            (v_lowest_grade * v_grade_percent);
    END IF;
END LOOP;
-- If either cursor had no rows then there is an error.
IF v_no_rows1 = 'N' OR v_no_rows2 = 'N' THEN
    v_exit_code := 'N';
    raise e_no_grade;
END IF;

P_final_grade := v_final_grade;
P_exit_code := v_exit_code;
EXCEPTION
    WHEN e_no_grade THEN
        P_final_grade := null;
        P_exit_code := v_exit_code;
    WHEN OTHERS THEN
        P_final_grade := null;
        P_exit_code := 'E';
END final_grade;

FUNCTION median_grade
    (p_course_number section.course_no%TYPE,
    p_section_number section.section_no%TYPE,
    p_grade_type grade.grade_type_code%TYPE)
RETURN grade.numeric_grade%TYPE
IS
BEGIN
    FOR r_work_grade
        IN c_work_grade(p_course_number, p_section_number, p_grade_type)
    LOOP
        t_grade(NVL(t_grade.COUNT, 0) + 1).numeric_grade := r_work_grade.numeric_grade;
    END LOOP;
    IF t_grade.COUNT = 0
    THEN
        RETURN NULL;
    ELSE
        IF MOD(t_grade.COUNT, 2) = 0
        THEN
            -- There is an even number of work grades. Find the middle
            -- two and average them.
            RETURN (t_grade(t_grade.COUNT / 2).numeric_grade +
                t_grade((t_grade.COUNT / 2) + 1).numeric_grade
            ) / 2;
        ELSE

```

```
-- There is an odd number of grades. Return the one in the middle.  
RETURN t_grade(TRUNC(t_grade.COUNT / 2, 0) + 1).numeric_grade;  
END IF;  
END IF;  
EXCEPTION  
WHEN OTHERS  
THEN  
RETURN NULL;  
END median_grade;  
END MANAGE_GRADES;
```

```
SELECT COURSE_NO,
       COURSE_NAME,
       SECTION_NO,
       GRADE_TYPE,
       manage_grades.median_grade
          (COURSE_NO,
           SECTION_NO,
           GRADE_TYPE)
       median_grade
FROM
(SELECT DISTINCT
       C.COURSE_NO      COURSE_NO,
       C.DESCRIPTION    COURSE_NAME,
       S.SECTION_NO     SECTION_NO,
       G.GRADE_TYPE_CODE GRADE_TYPE
  FROM SECTION S, COURSE C, ENROLLMENT E, GRADE G
 WHERE C.course_no = s.course_no
   AND s.section_id = e.section_id
   AND e.student_id = g.student_id
   AND c.course_no = 25
   AND s.section_no between 1 and 2
 ORDER BY 1, 4, 3) grade_source
```

COURSE_NO	COURSE_NAME	SECTION_NO	GRADE_TYPE	MEDIAN_GRADE
25	Intro to Programming	1	FI	98
25	Intro to Programming	2	FI	71
25	Intro to Programming	1	HM	76
25	Intro to Programming	2	HM	83
25	Intro to Programming	1	MT	86
25	Intro to Programming	2	MT	89
25	Intro to Programming	1	PA	91
25	Intro to Programming	2	PA	97
25	Intro to Programming	1	QZ	71
25	Intro to Programming	2	QZ	78

10 rows selected.

```
CREATE OR REPLACE PACKAGE school_api AS
  v_current_date DATE;
  PROCEDURE Discount_Cost;
  FUNCTION new_instructor_id
    RETURN instructor.instructor_id%TYPE;
END school_api;
```

```

CREATE OR REPLACE PACKAGE BODY school_api AS
  PROCEDURE discount_cost
  IS
    CURSOR c_group_discount
    IS
      SELECT distinct s.course_no, c.description
        FROM section s, enrollment e, course c
       WHERE s.section_id = e.section_id
     GROUP BY s.course_no, c.description,
              e.section_id, s.section_id
    HAVING COUNT(*) >=8;
  BEGIN
    FOR r_group_discount IN c_group_discount
    LOOP
      UPDATE course
        SET cost = cost * .95
      WHERE course_no = r_group_discount.course_no;
      DBMS_OUTPUT.PUT_LINE
        ('A 5% discount has been given to'
         || r_group_discount.course_no ||
         || r_group_discount.description);
    END LOOP;
  END discount_cost;
  FUNCTION new_instructor_id
    RETURN instructor.instructor_id%TYPE
  IS
    v_new_instid instructor.instructor_id%TYPE;
  BEGIN
    SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
      INTO v_new_instid
      FROM dual;
    RETURN v_new_instid;
  EXCEPTION
    WHEN OTHERS
    THEN
      DECLARE
        v_sqlerrm VARCHAR2(250) :=
          SUBSTR(SQLERRM,1,250);
      BEGIN
        RAISE_APPLICATION_ERROR(-20003,
          'Error in  instructor_id: '||v_sqlerrm);
      END;
  END new_instructor_id;
  BEGIN
    SELECT trunc(sysdate, 'DD')
      INTO v_current_date
      FROM dual;
  END school_api;

```

```
CREATE OR REPLACE PACKAGE show_date
IS
  PRAGMA SERIALLY_REUSABLE;
  the_date DATE := SYSDATE + 4;
  PROCEDURE display_DATE;
  PROCEDURE set_date;
END show_date;
/
CREATE OR REPLACE PACKAGE BODY show_date
IS
  PRAGMA SERIALLY_REUSABLE;
  PROCEDURE display_DATE IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('The date is ' || show_date.the_date);
  END;
  -- Initialize package state
  PROCEDURE set_date IS
  BEGIN
    show_date.the_date := sysdate;
  END;
END show_date;
```

```
begin
  -- initialize and print the package variable
  show_date.display_DATE;
  -- change the value of the variable the_date
  show_date.set_date;
  -- Display the new value of variable the_date
  show_date.display_DATE;
end;
/
begin
  show_date.display_DATE;
end;
/
```

```
SELECT OBJECT_TYPE, OBJECT_NAME, STATUS
FROM   USER_OBJECTS
WHERE  OBJECT_TYPE IN
      ('FUNCTION', 'PROCEDURE', 'PACKAGE',
       'PACKAGE_BODY')
ORDER BY OBJECT_TYPE;
```

```
CREATE OR REPLACE FUNCTION scode_at_line
  (i_name_in IN VARCHAR2,
   i_line_in IN INTEGER := 1,
   i_type_in IN VARCHAR2 := NULL)
RETURN VARCHAR2
IS
  CURSOR scode_cur IS
    SELECT text
      FROM user_source
     WHERE name = UPPER (i_name_in)
       AND (type = UPPER (i_type_in)
            OR i_type_in IS NULL)
       AND line = i_line_in;
  scode_rec scode_cur%ROWTYPE;
BEGIN
  OPEN scode_cur;
  FETCH scode_cur INTO scode_rec;
  IF scode_cur%NOTFOUND
    THEN
      CLOSE scode_cur;
      RETURN NULL;
    ELSE
      CLOSE scode_cur;
      RETURN scode_rec.text;
    END IF;
END;
```

Name	Null?	Type
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(12)
SEQUENCE	NOT NULL	NUMBER
LINE	NOT NULL	NUMBER
POSITION	NOT NULL	NUMBER
TEXT	NOT NULL	VARCHAR2(2000)

```
CREATE OR REPLACE PROCEDURE FORCE_ERROR
as
BEGIN
  SELECT course_no
  INTO v_temp
  FROM course;
END;
```

```
Errors for PROCEDURE FORCE_ERROR:  
LINE/COL ERROR
```

```
-----  
4/4      PL/SQL: SQL Statement ignored  
5/9      PLS-00201: identifier 'V_TEMP' must be declared  
6/4      PL/SQL: ORA-00904: : invalid identifier
```

```
SELECT line||'/'||position "LINE/COL", TEXT "ERROR"
FROM user_errors
WHERE name = 'FORCE_ERROR'
```

Name	Null?	Type
NAME	NOT NULL	VARCHAR2 (30)
TYPE		VARCHAR2 (12)
REFERENCED_OWNER		VARCHAR2 (30)
REFERENCED_NAME	NOT NULL	VARCHAR2 (30)
REFERENCED_TYPE		VARCHAR2 (12)
REFERENCED_LINK_NAME		VARCHAR2 (30)

```
CREATE OR REPLACE PACKAGE school_api AS
  v_current_date DATE;
  PROCEDURE Discount_Cost;
  FUNCTION new_instructor_id
  RETURN instructor.instructor_id%TYPE;
  FUNCTION total_cost_for_student
    (i_student_id IN student.student_id%TYPE)
  RETURN course.cost%TYPE;
  PRAGMA RESTRICT_REFERENCES
    (total_cost_for_student, WNDS, WNPS, RNPS);
  PROCEDURE get_student_info
    (i_student_id IN student.student_id%TYPE,
     o_last_name OUT student.last_name%TYPE,
     o_first_name OUT student.first_name%TYPE,
     o_zip      OUT student.zip%TYPE,
     o_return_code OUT NUMBER);
  PROCEDURE get_student_info
    (i_last_name  IN student.last_name%TYPE,
     i_first_name IN student.first_name%TYPE,
     o_student_id OUT student.student_id%TYPE,
     o_zip        OUT student.zip%TYPE,
     o_return_code OUT NUMBER);
END school_api;
```

```

CREATE OR REPLACE PACKAGE BODY school_api AS
  PROCEDURE discount_cost
  IS
    CURSOR c_group_discount
    IS
      SELECT distinct s.course_no, c.description
        FROM section s, enrollment e, course c
       WHERE s.section_id = e.section_id
      GROUP BY s.course_no, c.description,
               e.section_id, s.section_id
     HAVING COUNT(*) >=8;
  BEGIN
    FOR r_group_discount IN c_group_discount
    LOOP
      UPDATE course
        SET cost = cost * .95
       WHERE course_no = r_group_discount.course_no;
      DBMS_OUTPUT.PUT_LINE
        ('A 5% discount has been given to'
         ||r_group_discount.course_no||'
         ||r_group_discount.description);
    END LOOP;
  END discount_cost;
  FUNCTION new_instructor_id
    RETURN instructor.instructor_id%TYPE
  IS
    v_new_instid instructor.instructor_id%TYPE;
  BEGIN
    SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
      INTO v_new_instid
      FROM dual;
    RETURN v_new_instid;
  EXCEPTION
    WHEN OTHERS
    THEN
      DECLARE
        v_sqlerrm VARCHAR2(250) :=
          SUBSTR(SQLERRM,1,250);
      BEGIN
        RAISE_APPLICATION_ERROR(-20003,
          'Error in instructor_id: '||v_sqlerrm);
      END;
  END new_instructor_id;
  FUNCTION total_cost_for_student
    (i_student_id IN student.student_id%TYPE)
    RETURN course.cost%TYPE
  IS
    v_cost course.cost%TYPE;
  BEGIN
    SELECT sum(cost)
      INTO v_cost
      FROM course c, section s, enrollment e
     WHERE c.course_no = s.course_no
       AND e.section_id = s.section_id
       AND e.student_id = i_student_id;
    RETURN v_cost;
  EXCEPTION
    WHEN OTHERS THEN
      RETURN NULL;
  END total_cost_for_student;

```

```

PROCEDURE get_student_info
  (i_student_id IN student.student_id%TYPE,
   o_last_name   OUT student.last_name%TYPE,
   o_first_name  OUT student.first_name%TYPE,
   o_zip         OUT student.zip%TYPE,
   o_return_code OUT NUMBER)
IS
BEGIN
  SELECT last_name, first_name, zip
    INTO o_last_name, o_first_name, o_zip
    FROM student
   WHERE student.student_id = i_student_id;
  o_return_code := 0;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Student ID is not valid.');
    o_return_code := -100;
    o_last_name := NULL;
    o_first_name := NULL;
    o_zip := NULL;
WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Error in procedure get_student_info');
END get_student_info;
PROCEDURE get_student_info
  (i_last_name  IN student.last_name%TYPE,
   i_first_name IN student.first_name%TYPE,
   o_student_id OUT student.student_id%TYPE,
   o_zip        OUT student.zip%TYPE,
   o_return_code OUT NUMBER)
IS
BEGIN
  SELECT student_id, zip
    INTO o_student_id, o_zip
    FROM student
   WHERE UPPER(last_name) = UPPER(i_last_name)
     AND UPPER(first_name) = UPPER(i_first_name);
  o_return_code := 0;
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Student name is not valid.');
    o_return_code := -100;
    o_student_id := NULL;
    o_zip := NULL;
WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Error in procedure get_student_info');
END get_student_info;
BEGIN
  SELECT TRUNC(sysdate, 'DD')
    INTO v_current_date
    FROM dual;
END school_api;

```

```
DECLARE
  v_student_ID  student.student_id%TYPE;
  v_last_name   student.last_name%TYPE;
  v_first_name  student.first_name%TYPE;
  v_zip         student.zip%TYPE;
  v_return_code NUMBER;
BEGIN
  school_api.get_student_info
    (&p_id, v_last_name, v_first_name,
     v_zip,v_return_code);
  IF v_return_code = 0
  THEN
    DBMS_OUTPUT.PUT_LINE
      ('Student with ID '||&p_id||' is '||v_first_name
       ||' '||v_last_name
      );
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('The ID '||&p_id||'is not in the database'
      );
  END IF;
  school_api.get_student_info
    (&&p_last_name , &&p_first_name, v_student_id,
     v_zip , v_return_code);
  IF v_return_code = 0
  THEN
    DBMS_OUTPUT.PUT_LINE
      (&&p_first_name||' '|| &&p_last_name|||
       ' has an ID of '||v_student_id      );
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (&&p_first_name||' '|| &&p_last_name|||
       'is not in the database'
      );
  END IF;
END;
```

```
Enter value for p_id: 149
Enter value for p_last_name: 'Prochaska'
Enter value for p_first_name: 'Judith'
```

```
PROCEDURE calc_total (reg_in IN CHAR);  
PROCEDURE calc_total (reg_in IN VARCHAR2);
```

```
DECLARE
  PROCEDURE calc (comp_id_IN IN NUMBER)
  IS
    BEGIN ... END;
  PROCEDURE calc
  (comp_id_IN IN company.comp_id%TYPE)
  IS
    BEGIN ... END;
```

PLS-00307: too many declarations of '<program>' match this call

```
CREATE [OR REPLACE] TYPE type_name AS OBJECT
(attribute_name1 attribute_type,
attribute_name2 attribute_type,
...
attribute_nameN attribute_type,
[method1 specification],
[method2 specification],
...
[methodN specification]);
[CREATE [OR REPLACE] TYPE BODY type_name AS
method1 body;
method2 body;
...
methodN body;]
END;
```

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
(zip          VARCHAR2(5)
,city         VARCHAR2(25)
,state        VARCHAR2(2)
,created_by   VARCHAR2(30)
,created_date DATE
,modified_by  VARCHAR2(30)
,modified_date DATE);
```

```
DECLARE
    zip_obj zipcode_obj_type;
BEGIN
    SELECT zipcode_obj_type(zip, city, state, null, null, null, null)
    INTO zip_obj
    FROM zipcode
    WHERE zip = '06883';

    DBMS_OUTPUT.PUT_LINE ('Zip:    '||zip_obj.zip);
    DBMS_OUTPUT.PUT_LINE ('City:   '||zip_obj.city);
    DBMS_OUTPUT.PUT_LINE ('State:  '||zip_obj.state);
END;
```

```
DECLARE
    zip_obj zipcode_obj_type;
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Object instance has not been initialized');

    IF zip_obj IS NULL
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj instance is null');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('zip_obj instance is not null');
    END IF;

    IF zip_obj.zip IS NULL
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj.zip is null');
    END IF;

    -- Initialize zip_obj_instance
    zip_obj := zipcode_obj_type(null, null, null, null, null, null);

    DBMS_OUTPUT.PUT_LINE ('Object instance has been initialized');

    IF zip_obj IS NULL
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj instance is null');
    ELSE
        DBMS_OUTPUT.PUT_LINE ('zip_obj instance is not null');
    END IF;

    IF zip_obj.zip IS NULL
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj.zip is null');
    END IF;
END;
```

```
Object instance has not been initialized  
zip_obj instance is null  
zip_obj.zip is null  
Object instance has been initialized  
zip_obj instance is not null  
zip_obj.zip is null
```

```
DECLARE
  zip_obj zipcode_obj_type;
BEGIN
  zip_obj.zip := '12345';
END;
```

```
ORA-06530: Reference to uninitialized composite
ORA-06512: at line 4
```

```
DECLARE
  TYPE zip_type IS TABLE OF zipcode_obj_type INDEX BY PLS_INTEGER;
  zip_tab zip_type;
BEGIN
  SELECT zipcode_obj_type(zip, city, state, null, null, null, null)
    BULK COLLECT INTO zip_tab
    FROM zipcode
   WHERE rownum <= 5;

  IF zip_tab.COUNT > 0
  THEN
    FOR i in 1..zip_tab.count
    LOOP
      DBMS_OUTPUT.PUT_LINE ('Zip:    '||zip_tab(i).zip);
      DBMS_OUTPUT.PUT_LINE ('City:   '||zip_tab(i).city);
      DBMS_OUTPUT.PUT_LINE ('State:  '||zip_tab(i).state);
      DBMS_OUTPUT.PUT_LINE ('-----');
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Collection of objects is empty');
  END IF;
END;
```

```
CREATE OR REPLACE TYPE zip_tab_type AS TABLE OF zipcode_obj_type;
/
DECLARE
    zip_tab zip_tab_type := zip_tab_type();
    v_zip   VARCHAR2(5);
    v_city  VARCHAR2(20);
    v_state VARCHAR2(2);
BEGIN
    SELECT zipcode_obj_type(zip, city, state, null, null, null)
      BULK COLLECT INTO zip_tab
        FROM zipcode
       WHERE rownum <= 5;

    SELECT zip, city, state
      INTO v_zip, v_city, v_state
      FROM TABLE(zip_tab)
     WHERE rownum < 2;

    DBMS_OUTPUT.PUT_LINE ('Zip:    '||v_zip);
    DBMS_OUTPUT.PUT_LINE ('City:   '||v_city);
    DBMS_OUTPUT.PUT_LINE ('State:  '||v_state);
END;
```

```
CREATE OR REPLACE TYPE city_tab_type AS TABLE OF VARCHAR2(25);
/
CREATE OR REPLACE TYPE zip_tab_type AS TABLE OF VARCHAR2(5);
/
CREATE OR REPLACE TYPE state_obj_type AS OBJECT
  (state VARCHAR2(2)
   ,city  city_tab_type
   ,zip   zip_tab_type);
/
```

```
DECLARE
    city_tab  city_tab_type;
    zip_tab   zip_tab_type;

    state_obj state_obj_type := state_obj_type(null, city_tab_type(), zip_tab_type());

BEGIN
    SELECT city, zip
        BULK COLLECT INTO city_tab, zip_tab
        FROM zipcode
       WHERE state = 'NY'
         AND rownum <= 5;

    state_obj := state_obj_type ('NY', city_tab, zip_tab);

    DBMS_OUTPUT.PUT_LINE ('State: '||state_obj.state);
    DBMS_OUTPUT.PUT_LINE ('-----');

    IF state_obj.city.COUNT > 0
    THEN
        FOR i in state_obj.city.FIRST..state_obj.city.LAST
        LOOP
            DBMS_OUTPUT.PUT_LINE ('City: '||state_obj.city(i));
            DBMS_OUTPUT.PUT_LINE ('Zip:  '||state_obj.zip(i));
        END LOOP;
    END IF;
END;
```

```
state_obj state_obj_type := state_obj_type(null, city_tab_type(), zip_tab_type());
```

```
state_obj := state_obj_type ('NY', city_tab, zip_tab);
```

```
zip_obj1 := ZIPCODE_OBJ_TYPE('00914', 'Santurce', 'PR', USER, SYSDATE, USER, SYSDATE);
```

```
zip_obj2 := ZIPCODE_OBJ_TYPE(NULL, NULL, NULL, NULL, NULL, NULL NULL);
```



```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
  (zip          VARCHAR2(5)
   ,city         VARCHAR2(25)
   ,state        VARCHAR2(2)
   ,created_by   VARCHAR2(30)
   ,created_date DATE
   ,modified_by  VARCHAR2(30)
   ,modified_date DATE

  ,CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY zipcode_obj_type
                                         ,zip      VARCHAR2)
    RETURN SELF AS RESULT
  ,CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY zipcode_obj_type
                                         ,zip      VARCHAR2
                                         ,city     VARCHAR2
                                         ,state    VARCHAR2)
    RETURN SELF AS RESULT)
/
CREATE OR REPLACE TYPE BODY zipcode_obj_type AS

  CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY zipcode_obj_type
                                         ,zip      VARCHAR2)
    RETURN SELF AS RESULT
  IS
  BEGIN
    SELF.zip := zip;
    SELECT city, state
      INTO SELF.city, SELF.state
      FROM zipcode
     WHERE zip = SELF.zip;

    RETURN;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN;
  END;

  CONSTRUCTOR FUNCTION zipcode_obj_type (SELF IN OUT NOCOPY zipcode_obj_type
                                         ,zip      VARCHAR2
                                         ,city     VARCHAR2
                                         ,state    VARCHAR2)
    RETURN SELF AS RESULT
  IS
  BEGIN
    SELF.zip  := zip;
    SELF.city := city;
    SELF.state := state;

    RETURN;
  END;
END;
/
```

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
  ...
  , MEMBER PROCEDURE get_zipcode_info (out_zip    OUT VARCHAR2
                                      ,out_city   OUT VARCHAR2
                                      ,out_state  OUT VARCHAR2));
/
CREATE OR REPLACE TYPE BODY zipcode_obj_type AS
  ...
  MEMBER PROCEDURE get_zipcode_info (out_zip    OUT VARCHAR2
                                      ,out_city   OUT VARCHAR2
                                      ,out_state  OUT VARCHAR2)
IS
BEGIN
  out_zip  := SELF.zip;
  out_city := SELF.city;
  out_state := SELF.state;
END;
END;
/
```

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
  ...
  ,STATIC PROCEDURE display_zipcode_info (in_zip_obj IN zipcode_obj_type));
/

CREATE OR REPLACE TYPE BODY zipcode_obj_type AS

  ...

  STATIC PROCEDURE display_zipcode_info (in_zip_obj IN zipcode_obj_type)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('Zip: ' ||in_zip_obj.zip);
    DBMS_OUTPUT.PUT_LINE ('City: ' ||in_zip_obj.city);
    DBMS_OUTPUT.PUT_LINE ('State: '||in_zip_obj.state);
  END;
END;
/
```

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
  ...
  ,MAP MEMBER FUNCTION zipcode RETURN VARCHAR2);
/
CREATE OR REPLACE TYPE BODY zipcode_obj_type AS
  ...
  MAP MEMBER FUNCTION zipcode RETURN VARCHAR2
  IS
    BEGIN
      RETURN (zip);
    END;
  END;
/
```

```
zip_obj1.zipcode() > zip_obj2.zipcode()
```

```
DECLARE
    zip_obj1 zipcode_obj_type;
    zip_obj2 zipcode_obj_type;
BEGIN
    -- Initialize object instances with user-defined constructor methods
    zip_obj1 := zipcode_obj_type (zip    => '12345'
                                ,city   => 'Some City'
                                ,state  => 'AB');

    zip_obj2 := zipcode_obj_type (zip => '48104');

    -- Compare object instances via map methods
    IF zip_obj1 > zip_obj2
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is greater than zip_obj2');
    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('zip_obj1 is not greater than zip_obj2');
    END IF;
END;
```

v_zip_obj1 is not greater than v_zip_obj2

PLS-00154: An object type may have only 1 MAP or 1 ORDER method.

```
CREATE OR REPLACE TYPE zipcode_obj_type AS OBJECT
  ...
  ,ORDER MEMBER FUNCTION zipcode (zip_obj zipcode_obj_type) RETURN INTEGER);
/

CREATE OR REPLACE TYPE BODY zipcode_obj_type AS

  ...
  ORDER MEMBER FUNCTION zipcode (zip_obj zipcode_obj_type) RETURN INTEGER
  IS
  BEGIN
    IF      zip < zip_obj.zip THEN RETURN -1;
    ELSIF zip = zip_obj.zip THEN RETURN  0;
    ELSIF zip > zip_obj.zip THEN RETURN  1;
    END IF;
  END;
END;
/
```

```
DECLARE
    zip_obj1 zipcode_obj_type;
    zip_obj2 zipcode_obj_type;

    v_result INTEGER;
BEGIN
    -- Initialize object instances with user-defined constructor methods
    zip_obj1 := zipcode_obj_type ('12345', 'Some City', 'AB');
    zip_obj2 := zipcode_obj_type ('48104');

    -- Compare objects instances via ORDER method
    v_result := zip_obj1.zipcode(zip_obj2);
    DBMS_OUTPUT.PUT_LINE ('The result of comparison is'||v_result);

    IF v_result = 1
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is greater than zip_obj2');

    ELSIF v_result = 0
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is equal to zip_obj2');

    ELSIF v_result = -1
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj1 is less than zip_obj2');
    END IF;
END;
```

```
v_result := zip_obj1.zipcode(zip_obj2);
```

The result of comparison is -1
zip_obj1 is less than zip_obj2

```
DECLARE
    zip_obj1 zipcode_obj_type;
    zip_obj2 zipcode_obj_type;

    v_result INTEGER;
BEGIN
    -- Initialize object instances with user-defined constructor methods
    zip_obj1 := zipcode_obj_type ('12345', 'Some City', 'AB');
    zip_obj2 := zipcode_obj_type ('48104');

    -- Compare objects instances via ORDER method
    v_result := zip_obj2.zipcode(zip_obj1);
    DBMS_OUTPUT.PUT_LINE ('The result of comparison is'||v_result);
    IF v_result = 1
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj2 is greater than zip_obj1');

    ELSIF v_result = 0
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj2 is equal to zip_obj1');

    ELSIF v_result = -1
    THEN
        DBMS_OUTPUT.PUT_LINE ('zip_obj2 is less than zip_obj1');
    END IF;
END;
```

```
The result of comparison is 1  
zip_obj2 is greater than zip_obj1
```

```
GRANT SELECT ON sys.v$session TO student;
ch24_1.sql"
```

```
CREATE OR REPLACE PACKAGE BODY school_api AS
CREATE OR REPLACE PROCEDURE LOG_USER_COUNT
(PI_DIRECTORY IN VARCHAR2,
 PI_FILE_NAME IN VARCHAR2)
AS
  V_File_handle UTL_FILE.FILE_TYPE;
  V_user_count number;
BEGIN
  SELECT count(*)
  INTO V_user_count
  FROM v$session
  WHERE username is not null;

  V_File_handle := 
    UTL_FILE.FOPEN(PI_DIRECTORY, PI_FILE_NAME, 'A');
  UTL_FILE.NEW_LINE(V_File_handle);
  UTL_FILE.PUT_LINE(V_File_handle , '----- User log -----');
  UTL_FILE.NEW_LINE(V_File_handle);
  UTL_FILE.PUT_LINE(V_File_handle , 'on'|| 
    TO_CHAR(SYSDATE, 'MM/DD/YY HH24:MI'));
  UTL_FILE.PUT_LINE(V_File_handle ,
    'Number of users logged on: '|| V_user_count);
  UTL_FILE.PUT_LINE(V_File_handle , '----- End log -----');
  UTL_FILE.NEW_LINE(V_File_handle);
  UTL_FILE.FCLOSE(V_File_handle);

EXCEPTION
  WHEN UTL_FILE.INVALID_FILENAME THEN
    DBMS_OUTPUT.PUT_LINE('File is invalid');
  WHEN UTL_FILE.WRITE_ERROR THEN
    -DBMS_OUTPUT.PUT_LINE('Oracle is not able to write to file');
END;
```

```
SQL> exec LOG_USER_COUNT('C:\working\', 'USER.LOG');
```

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

```
CREATE OR REPLACE PROCEDURE READ_FILE
(PI_DIRECTORY IN VARCHAR2,
 PI_FILE_NAME IN VARCHAR2)
AS
  V_File_Handle UTL_FILE.FILE_TYPE;
  V_File_Line    VARCHAR2(1024);
BEGIN
  V_File_Handle := UTL_FILE.FOPEN(PI_DIRECTORY, PI_FILE_NAME, 'R');
  LOOP
    UTL_FILE.GET_LINE( V_File_Handle , V_File_Line );
    DBMS_OUTPUT.PUT_LINE(V_File_Line);
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND
    THEN UTL_FILE.FCLOSE( V_File_Handle );
END;
```

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;  
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

```
DECLARE
  V_JOB_NO NUMBER;
BEGIN
  DBMS_JOB.SUBMIT( JOB      => v_job_no,
                    WHAT     -=> 'LOG_USER_COUNT
                               (''C:\WORKING\'', ''USER.LOG''),',
                    NEXT_DATE => SYSDATE,
                    INTERVAL  => 'SYSDATE + 1/4 ');
  Commit;
  DBMS_OUTPUT.PUT_LINE(v_job_no);
END;
```

```
GRANT SELECT on DBA_JOBS to STUDENT;
```

```
SELECT JOB, NEXT_DATE, NEXT_SEC, BROKEN, WHAT  
FROM    DBA_JOBS;
```

```
JOB NEXT_DATE NEXT_SEC B WHAT
-----
1 05-JUL-03 16:56:30 N LOG_USER_COUNT('D:\WORKING', 'USER.LOG');
```

```
-- execute job number 1
exec dbms_job.run(1);

-- remove job number 1 from the job queue
exec dbms_job.remove(1);

-- change job number 1 to run immediately and then every hour of
-- the day
exec DBMS_JOB.CHANGE(1, null, SYSDATE, 'SYSDATE + 1/24 '');
```

```
-- set job 1 to be broken  
exec dbms_job.BROKEN(1, TRUE);  
  
-- set job 1 not to be broken  
exec dbms_job.BROKEN(1, FALSE);
```

```
CREATE or REPLACE procedure DELETE_ENROLL
AS
  CURSOR C_NO_GRADES is
    SELECT st.student_id, se.section_id
      FROM student st,
           enrollment e,
           section se
     WHERE st.student_id = e.student_id
       AND e.section_id = se.section_id
       AND se.start_date_time < ADD_MONTHS(SYSDATE, -1)
       AND NOT EXISTS (SELECT g.student_id, g.section_id
                         FROM grade g
                        WHERE g.student_id = st.student_id
                          AND g.section_id = se.section_id);
BEGIN
  FOR R in C_NO_GRADES LOOP
    DELETE enrollment
      WHERE section_id = r.section_id
        AND student_id = r.student_id;
  END LOOP;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

```
SQL> VARIABLE V_JOB NUMBER
SQL> EXEC DBMS_JOB.SUBMIT(:v_job, 'DELETE_ENROLL;', SYSDATE,
  'ADD_MONTHS(SYSDATE, 1)');
PL/SQL procedure successfully completed
SQL> commit;

Commit complete.

SQL> print v_job
      V_JOB
-----
      2
```

```
create table PLAN_TABLE (
    statement_id          varchar2(30),
    plan_id               number,
    timestamp             date,
    remarks               varchar2(4000),
    operation              varchar2(30),
    options                varchar2(255),
    object_node            varchar2(128),
    object_owner           varchar2(30),
    object_name            varchar2(30),
    object_alias            varchar2(65),
    object_instance         numeric,
    object_type             varchar2(30),
    optimizer               varchar2(255),
    search_columns          number,
    id                     numeric,
    parent_id              numeric,
    depth                  numeric,
    position                numeric,
    cost                   numeric,
    cardinality            numeric,
    bytes                  numeric,
    other_tag               varchar2(255),
    partition_start          varchar2(255),
    partition_stop           varchar2(255),
    partition_id             numeric,
    other                   long,
    distribution             varchar2(30),
    cpu_cost                numeric,
    io_cost                 numeric,
    temp_space               numeric,
    access_predicates        varchar2(4000),
    filter_predicates        varchar2(4000),
    projection               varchar2(4000),
    time                    numeric,
    qblock_name              varchar2(30),
    other_xml                clob
);

```

```
SQL> CONN sys/password AS SYSDBA
Connected
SQL> @$ORACLE_HOME/rdbms/admin/utlxplan.sql
SQL> GRANT ALL ON sys.plan_table TO public;
SQL> CREATE PUBLIC SYNONYM plan_table FOR sys.plan_table;
```

```
SQL> SET AUTOTRACE TRACE EXPLAIN
  1  SELECT s.course_no,
  2        c.description,
  3        i.first_name,
  4        i.last_name,
  5        s.section_no,
  6        TO_CHAR(-s.start_date_time,'Mon-DD-YYYY HH:MIAM'),
  7        s.location
  8  FROM section s,
  9       course c,
10      instructor i
11 WHERE s.course_no      = c.course_no
12* AND   s.instructor_id= i.instructor_id

Execution Plan
-----
 0  SELECT STATEMENT Optimizer=CHOOSE (Cost=9 Card=78 Bytes=4368)
 1  0  HASH JOIN (Cost=9 Card=78 Bytes=4368)
 2  1  HASH JOIN (Cost=6 Card=78 Bytes=2574)
 3  2   TABLE ACCESS (FULL) OF 'INSTRUCTOR' (Cost=3 Card=10 Bytes=140)
 4  2   TABLE ACCESS (FULL) OF 'SECTION' (Cost=3 Card=78 Bytes=1482)
 5  1   TABLE ACCESS (FULL) OF 'COURSE' (Cost=3 Card=30 Bytes=690)
```

```
SQL> explain plan for
  2  SELECT s.course_no,
  3        c.description,
  4        i.first_name,
  5        i.last_name,
  6        s.section_no,
  7        TO_CHAR(s.start_date_time,'Mon-DD-YYYY HH:MIAM'),
  8        s.location
  9  FROM section s,
 10      course c,
 11      instructor i
12  WHERE s.course_no    = c.course_no
13  AND   s.instructor_id= i.instructor_id;
```

Explained.

```
select rtrim ( lpad ( ' ', 2*level ) || 
               rtrim ( operation ) || ' ' || 
               rtrim ( options ) || ' ' || 
               object_name || ' ' || 
               partition_start || ' ' || 
               partition_stop || ' ' || 
               to_char ( partition_id ) || 
               ) the_query_plan
from plan_table
connect by prior id = parent_id
start with id = 0;
```

THE_QUERY_PLAN

```
-----  
SELECT STATEMENT  
HASH JOIN  
  HASH JOIN  
    TABLE ACCESS BY INDEX ROWID SECTION  
      INDEX FULL SCAN SECT_INST_FK_I  
    SORT JOIN  
      TABLE ACCESS FULL INSTRUCTOR  
    TABLE ACCESS FULL COURSE
```

```
SQL> explain plan for
  2  SELECT s.course_no,
  3        c.description,
  4        i.first_name,
  5        i.last_name,
  6        s.section_no,
  7        TO_CHAR(s.start_date_time,'Mon-DD-YYYY HH:MIAM'),
  8        s.location
  9  FROM  section s,
 10        course c,
 11        instructor i
12 WHERE s.course_no      = c.course_no
13 AND   s.instructor_id= i.instructor_id;
```

Explained.

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

PLAN_TABLE_OUTPUT

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		78	4368	9 (34)	*00:00:01
*	1	HASH JOIN	78	4368	9 (34)	00:00:01
*	2	HASH JOIN	78	2574	6 (34)	00:00:01
3	TABLE ACCESS FULL	INSTRUCTOR	10	140	3 (34)	00:00:01
4	TABLE ACCESS FULL	SECTION	78	1482	3 (34)	00:00:01
5	TABLE ACCESS FULL	COURSE	30	690	3 (34)	00:00:01

Predicate Information (identified by operation id):

```
1 - access("S"."COURSE_NO"="C"."COURSE_NO")
2 - access("S"."INSTRUCTOR_ID"="I"."INSTRUCTOR_ID")
```

17 rows selected.

```
CREATE OR REPLACE PACKAGE Student_Instructor AS
PROCEDURE show_population
    (i_zip IN zipcode.zip%TYPE);
END Student_Instructor;
/

CREATE or REPLACE PACKAGE BODY Student_Instructor
AS
PROCEDURE show_population
    (i_zip IN zipcode.zip%TYPE)
AS
student_list      SYS_REFCURSOR;
instructor_list   SYS_REFCURSOR;
BEGIN
OPEN student_list FOR
    SELECT 'Student' type, First_Name, Last_Name
        FROM student
       WHERE zip = i_zip;
    DBMS_SQL.RETURN_RESULT(student_list);
OPEN instructor_list FOR
    SELECT 'Instructor' type, First_Name, Last_Name
        FROM instructor
       WHERE zip = i_zip;
    DBMS_SQL.RETURN_RESULT(instructor_list);
END show_population;
END Student_Instructor;
/
```

```
SQL> exec Student_Instructor.show_population('10025');
```

PL/SQL procedure successfully completed.

ResultSet #1

TYPE	FIRST_NAME	LAST_NAME
Student	Jerry	Abdou
Student	Nicole	Gillen
Student	Frank	Pace

ResultSet #2

TYPE	FIRST_NAME	LAST_NAME
Instructor	Tom	Wojick
Instructor	Nina	Schorin
Instructor	Todd	Smythe
Instructor	Charles	Lowry

```
DBMS.Utility.Format_Call_Stack  
RETURN VARCHAR2;
```

```
CREATE OR REPLACE PROCEDURE first
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_CALL_STACK);
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
  first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
  second;
END third;
/

BEGIN
  third;
END;
```

```
----- PL/SQL Call Stack -----  
object    line    object  
handle   number  name  
0x104a93040      4  procedure STUDENT.FIRST  
0xa06f8208       4  procedure STUDENT.SECOND  
0x1045f1e68      4  procedure STUDENT.THIRD  
0xa0259658       2  anonymous block
```

```
DBMS.Utility.Format_Error_Backtrace  
RETURN VARCHAR2;
```

```
CREATE OR REPLACE PROCEDURE first
IS
  v_name VARCHAR2(30);
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure FIRST');

  SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
    INTO v_name
   FROM student
  WHERE student_id = 1000;
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
  first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
  second;
END third;
/
BEGIN
  third;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
END;
```

```
procedure THIRD
procedure SECOND
procedure FIRST
ORA-06512: at "STUDENT.FIRST", line 7
ORA-06512: at "STUDENT.SECOND", line 5
ORA-06512: at "STUDENT.THIRD", line 5
ORA-06512: at line 2
```

```
DBMS.Utility.Format_Error_Stack  
RETURN VARCHAR2;
```

```
CREATE OR REPLACE PROCEDURE first
IS
    v_name VARCHAR2(30);
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
    SELECT RTRIM(first_name) || ' ' || RTRIM(last_name)
        INTO v_name
        FROM student
       WHERE student_id = 1000;
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
    first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
    second;
END third;
/

BEGIN
    third;
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE ('Error Backtrace:');
        DBMS_OUTPUT.PUT_LINE ('-----');
        DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_BACKTRACE);
        DBMS_OUTPUT.PUT_LINE ('Error Stack:');
        DBMS_OUTPUT.PUT_LINE ('-----');
        DBMS_OUTPUT.PUT_LINE (DBMS_UTILITY.FORMAT_ERROR_STACK);
END;
```

```
procedure THIRD
procedure SECOND
procedure FIRST
Error Backtrace:
-----
ORA-06512: at "STUDENT.FIRST", line 7
ORA-06512: at "STUDENT.SECOND", line 5
ORA-06512: at "STUDENT.THIRD", line 5
ORA-06512: at line 2
Error Stack:
-----
ORA-01403: no data found
```

```
CREATE OR REPLACE PROCEDURE first
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  second;
END third;
/

BEGIN
  DBMS_OUTPUT.PUT_LINE ('anonymous block');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  third;
END;
```

```
UTL_CALL_STACK.BACKTRACE_DEPTH
RETURN PLS_INTEGER;

UTL_CALL_STACK.BACKTRACE_LINE (backtrace_depth IN PLS_INTEGER)
RETURN PLS_INTEGER;

UTL_CALL_STACK.BACKTRACE_UNIT (backtrace_depth IN PLS_INTEGER)
RETURN VARCHAR2;
```

```
CREATE OR REPLACE PROCEDURE first
IS
  v_string VARCHAR2(3);
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  v_string := 'ABCDEF';
END first;
/
CREATE OR REPLACE PROCEDURE second
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure SECOND');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  first;
END second;
/
CREATE OR REPLACE PROCEDURE third
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  second;
END third;
/
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('anonymous block');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  third;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE (CHR(10)||'Backtrace Stack: '||CHR(10)||RPAD('-', 15, '-'));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Depth: '||TO_CHAR(UTL_CALL_STACK.BACKTRACE_DEPTH));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Line: ' ||
      TO_CHAR(UTL_CALL_STACK.BACKTRACE_LINE(UTL_CALL_STACK.BACKTRACE_DEPTH)));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Unit: ' ||
      UTL_CALL_STACK.BACKTRACE_UNIT(UTL_CALL_STACK.BACKTRACE_DEPTH));
END;
```

```
UTL_CALL_STACK.ERROR_DEPTH
RETURN PLS_INTEGER;

UTL_CALL_STACK.ERROR_MSG (error_depth IN PLS_INTEGER)
RETURN VARCHAR2;

UTL_CALL_STACK.ERROR_NUMBER (error_depth IN PLS_INTEGER)
RETURN VARCHAR2;
```

```

CREATE OR REPLACE PROCEDURE first
IS
  v_string VARCHAR2(3);
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure FIRST');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  v_string := 'ABCDEF';
END first;
/

CREATE OR REPLACE PROCEDURE second
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure SCOND');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  first;
END second;
/

CREATE OR REPLACE PROCEDURE third
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE ('procedure THIRD');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  second;
END third;
/


BEGIN
  DBMS_OUTPUT.PUT_LINE ('anonymous block');
  DBMS_OUTPUT.PUT_LINE ('dynamic depth: '||TO_CHAR(UTL_CALL_STACK.DYNAMIC_DEPTH));
  third;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE (CHR(10)||'Backtrace Stack: '||CHR(10)||RPAD('-', 15, '-'));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Depth: '||TO_CHAR(UTL_CALL_STACK.BACKTRACE_DEPTH));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Line: '||TO_CHAR(UTL_CALL_STACK.BACKTRACE_LINE(UTL_CALL_STACK.BACKTRACE_DEPTH)));
    DBMS_OUTPUT.PUT_LINE ('Backtrace Unit: '||UTL_CALL_STACK.BACKTRACE_UNIT(UTL_CALL_STACK.BACKTRACE_DEPTH));
    DBMS_OUTPUT.PUT_LINE (CHR(10)||'Error Info: '||CHR(10)||RPAD('-', 15, '-'));
    DBMS_OUTPUT.PUT_LINE ('Error Depth: '||TO_CHAR(UTL_CALL_STACK.ERROR_DEPTH));
    DBMS_OUTPUT.PUT_LINE ('Error Number: '||TO_CHAR(UTL_CALL_STACK.ERROR_NUMBER(UTL_CALL_STACK.ERROR_DEPTH)));
    DBMS_OUTPUT.PUT_LINE ('Error Message: '||UTL_CALL_STACK.ERROR_MSG(UTL_CALL_STACK.ERROR_DEPTH));
END;

```

```
anonymous block
dynamic depth: 1
procedure THIRD
dynamic depth: 2
procedure SECOND
dynamic depth: 3
procedure FIRST
dynamic depth: 4

Backtrace Stack:
-----
Backtrace Depth: 4
Backtrace Line: 7
Backtrace Unit: STUDENT.FIRST

Error Info:
-----
Error Depth: 1
Error Number: 6502
Error Message: PL/SQL: numeric or value error: character string buffer too small
```

```
ALTER SESSION SET PLSQL_DEBUG = TRUE;
```

```
ALTER [PROCEDURE/FUNCTION/PACKAGE] ROUTINE_NAME  
COMPILE DEBUG [BODY (applicable to packages only)]
```

```
ALTER SESSION SET PLSQL_DEBUG = TRUE;

-- Create test procedure to be traced
CREATE OR REPLACE PROCEDURE TEST_TRACE
AS
  v_num1 NUMBER;
  v_num2 NUMBER;
  v_num3 NUMBER;
  v_date DATE;
BEGIN
  FOR i IN 1..10
  LOOP
    v_num1 := 1;
    v_num2 := i + i/2 + sqrt(i);
    v_num3 := v_num1 + v_num2;

    SELECT sysdate
      INTO v_date
     FROM DUAL;
  END LOOP;
END TEST_TRACE;
/

-- Trace TEST_TRACE procedure
BEGIN
  DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_CALLS);
  TEST_TRACE;
  DBMS_TRACE.CLEAR_PLSQL_TRACE;
END;
```

```

SELECT r.runid
  ,e.event_seq
  ,e.event_unit_owner
  ,e.event_unit
  ,e.event_unit_kind
  ,e.proc_line
  ,e.event_comment
  FROM plsql_trace_runs r
  ,plsql_trace_events e
 WHERE r.runid = 1 -- this value must change based on the number of traces run
   AND r.runid = e.runid
 ORDER BY r.runid, e.event_seq;

```

RUNID	EVENT_SEQ	EVENT_UNIT_OWNER	EVENT_UNIT	EVENT_UNIT_KIND	PROC_LINE	EVENT_COMMENT
-------	-----------	------------------	------------	-----------------	-----------	---------------

1	1					PL/SQL Trace Tool started
1	2					Trace flags changed
1	3	SYS	DBMS_TRACE	PACKAGE BODY	75	Return from procedure call
1	4	SYS	DBMS_TRACE	PACKAGE BODY	81	Return from procedure call
1	5	SYS	DBMS_TRACE	PACKAGE BODY	3	Return from procedure call
1	6		<anonymous>	ANONYMOUS BLOCK	1	Procedure Call
1	7	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	8	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	9	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	10	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	11	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	12	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	13	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	14	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	15	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	16	STUDENT	TEST_TRACE	PROCEDURE		PL/SQL Internal Call
1	17	STUDENT	TEST_TRACE	PROCEDURE	4	Return from procedure call
1	18		<anonymous>	ANONYMOUS BLOCK	92	Procedure Call
1	19	SYS	DBMS_TRACE	PACKAGE BODY	69	Procedure Call
1	20	SYS	DBMS_TRACE	PACKAGE BODY	64	Procedure Call
1	21	SYS	DBMS_TRACE	PACKAGE BODY	12	Procedure Call
1	22	SYS	DBMS_TRACE	PACKAGE BODY	66	Return from procedure call
1	23	SYS	DBMS_TRACE	PACKAGE BODY	72	Return from procedure call
1	24	SYS	DBMS_TRACE	PACKAGE BODY	21	Procedure Call
1	25					PL/SQL trace stopped

```
CREATE DIRECTORY PLSHPOF_DIR AS '/plshprof/results';
GRANT READ, WRITE ON DIRECTORY PLSHPOF_DIR TO STUDENT;
```

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 0;

1 DECLARE
2   v_num1  NUMBER;
3   v_num2  NUMBER;
4   v_num3  NUMBER;
5   v_run_id BINARY_INTEGER; -- run ID generated by the profiler
6 BEGIN
7   DBMS_PROFILER.START_PROFILER ('Optimizer level at 0');
8
9   FOR i IN 1..1000000
10  LOOP
11    v_num1 := 1;
12    v_num2 := i + i/2 + sqrt(i);
13    v_num3 := v_num1 + v_num2;
14  END LOOP;
15
16 DBMS_PROFILER.STOP_PROFILER();
17
18 SELECT runid
19   INTO v_run_id
20   FROM plsql_profiler_runs
21  WHERE run_comment = 'Optimizer level at 0';
22
23 DBMS_OUTPUT.PUT_LINE ('Optimizer level at 0, run ID - '||v_run_id);
24 END;
```

Elapsed: 00:00:03.317
Optimizer level at 0, run ID - 1

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

1 DECLARE
2   v_num1  NUMBER;
3   v_num2  NUMBER;
4   v_num3  NUMBER;
5   v_run_id BINARY_INTEGER; -- run ID generated by the profiler
6 BEGIN
7   DBMS_PROFILER.START_PROFILER ('Optimizer level at 1');
8
9   FOR i IN 1.. 1000000
10  LOOP
11    v_num1 := 1;
12    v_num2 := i + i/2 + sqrt(i);
13    v_num3 := v_num1 + v_num2;
14  END LOOP;
15
16 DBMS_PROFILER.STOP_PROFILER();
17
18 SELECT runid
19   INTO v_run_id
20   FROM plsql_profiler_runs
21  WHERE run_comment = 'Optimizer level at 1';
22
23 DBMS_OUTPUT.PUT_LINE ('Optimizer level at 1, run ID - '||v_run_id);
24 END;
```

Elapsed: 00:00:03.103
Optimizer level at 1, run ID - 2

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1 DECLARE
2   v_num1  NUMBER;
3   v_num2  NUMBER;
4   v_num3  NUMBER;
5   v_run_id BINARY_INTEGER; -- run ID generated by the profiler
6 BEGIN
7   DBMS_PROFILER.START_PROFILER ('Optimizer level at 2');
8
9   FOR i IN 1.. 1000000
10  LOOP
11    v_num1 := 1;
12    v_num2 := i + i/2 + sqrt(i);
13    v_num3 := v_num1 + v_num2;
14  END LOOP;
15
16 DBMS_PROFILER.STOP_PROFILER();
17
18 SELECT runid
19   INTO v_run_id
20   FROM plsql_profiler_runs
21 WHERE run_comment = 'Optimizer level at 2';
22
23 DBMS_OUTPUT.PUT_LINE ('Optimizer level at 2, run ID - '||v_run_id);
24 END;
```

Elapsed: 00:00:02.562
Optimizer level at 2, run ID - 3

```

SELECT r.runid, r.run_comment, d.line#, d.total_occur, d.total_time
  FROM plsql_profiler_runs r
       ,plsql_profiler_data d
       ,plsql_profiler_units u
 WHERE r.runid = d.runid
   AND d.runid = u.runid
   AND d.unit_number = u.unit_number
   AND d.total_occur > 0
ORDER BY d.runid, d.line#;

```

RUNID	RUN_COMMENT	LINE#	TOTAL_OCCUR	TOTAL_TIME
1	Optimizer level at 0	9	1000001	128784207
1	Optimizer level at 0	11	1000000	204515328
1	Optimizer level at 0	12	1000000	1928730621
1	Optimizer level at 0	13	1000000	235407659
1	Optimizer level at 0	14	1	0
1	Optimizer level at 0	16	1	13032
2	Optimizer level at 1	9	1000001	122832257
2	Optimizer level at 1	11	1000000	143920674
2	Optimizer level at 1	12	1000000	1820567385
2	Optimizer level at 1	13	1000000	181771219
2	Optimizer level at 1	14	1	0
2	Optimizer level at 1	16	1	12000
3	Optimizer level at 2	9	1000001	134848732
3	Optimizer level at 2	11	1000000	0
3	Optimizer level at 2	12	1000000	1583962321
3	Optimizer level at 2	13	1000000	188121402
3	Optimizer level at 2	16	1	10015

```
13    v_num3 := v_num1 + v_num2;
```

```
SET TIMING ON;

-- Create test table
CREATE TABLE TEST_TAB
  (col1 NUMBER);
/
-- Populate newly created table with random data
INSERT INTO TEST_TAB
SELECT ROUND(DBMS_RANDOM.VALUE (1, 99999999), 0)
  FROM dual
CONNECT by level < 100001;
COMMIT;

-- Collect statistics
EXEC DBMS_STATS.GATHER_TABLE_STATS (user, 'TEST_TAB');

-- Run the same code sample with different optimization levels
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

BEGIN
  FOR rec IN (SELECT col1 FROM test_tab)
  LOOP
    null; -- do nothing
  END LOOP;
END;
/

ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;
BEGIN
  FOR REC IN (SELECT col1 FROM test_tab)
  LOOP
    NULL; -- do nothing
  END LOOP;
END;
/
```

```
SET TIMING ON;

-- Create test table
CREATE TABLE test_tab1 (col1 NUMBER);
/
-- Run the same code sample with different optimization levels
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 1;

BEGIN
  FOR rec IN (SELECT col1 FROM test_tab)
  LOOP
    INSERT INTO TEST_TAB1 VALUES (rec.col1); -- populate newly created table
  END LOOP;
END;
/

ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

BEGIN
  FOR REC IN (SELECT col1 FROM test_tab)
  LOOP
    INSERT INTO TEST_TAB1 VALUES (rec.col1); -- populate newly created table
  END LOOP;
END;
/
```

```
PRAGMA INLINE (subprogram_name, 'YES');
```

```
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;
```

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1 DECLARE
2   v_num      PLS_INTEGER;
3   v_run_id BINARY_INTEGER; -- run ID generated by the profiler
4
5   FUNCTION test_func (num1 IN PLS_INTEGER
6                      ,num2 IN PLS_INTEGER)
7   RETURN PLS_INTEGER
8   IS
9   BEGIN
10      RETURN (num1 + num2);
11   END test_func;
12
13 BEGIN
14   DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15   FOR i IN 1..100000
16   LOOP
17     v_num := test_func (i-1, i);
18   END LOOP;
19   DBMS_HPROF.STOP_PROFILING;
20
21   -- Analyze profiler output and display its run ID
22   v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
23   DBMS_OUTPUT.PUT_LINE ('Inline pragma is not enabled, run ID - '||v_run_id);
24 END;
```

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is not enabled, run ID - 1  
Elapsed: 00:00:00.602
```

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2    v_num      PLS_INTEGER;
3    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
4
5    FUNCTION test_func (num1 IN PLS_INTEGER
6                        ,num2 IN PLS_INTEGER)
7    RETURN PLS_INTEGER
8    IS
9    BEGIN
10       RETURN (num1 + num2);
11    END test_func;
12
13 BEGIN
14   DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15   FOR i IN 1..100000
16   LOOP
17     -- Inline pragma is enabled for each function call
18     PRAGMA INLINE (test_func, 'YES');
19     v_num := test_func (i-1, i);
20   END LOOP;
21   DBMS_HPROF.STOP_PROFILING;
22
23   -- Analyze profiler output and display its run ID
24   v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
25   DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID - '||v_run_id);
26 END;
```

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is enabled, run ID - 2  
Elapsed: 00:00:00.073
```

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2    v_num   PLS_INTEGER;
3    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
4
5    FUNCTION test_func (num1 IN PLS_INTEGER
6                        ,num2 IN PLS_INTEGER)
7    RETURN PLS_INTEGER
8    IS
9    BEGIN
10      RETURN (num1 + num2);
11    END test_func;
12
13 BEGIN
14   DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15
16   -- Inline pragma is moved outside the loop
17   PRAGMA INLINE (test_func, 'YES');
18   FOR i IN 1..100000
19   LOOP
20     v_num := test_func (i-1, i);
21   END LOOP;
22   DBMS_HPROF.STOP_PROFILING;
23
24   -- Analyze profiler output and display its run ID
25   v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
26   DBMS_OUTPUT.PUT_LINE
27   ('Inline pragma is enabled for a single call, run ID - '||v_run_id);
28 END;
```

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is enabled for a single call, run ID - 3  
Elapsed: 00:00:00.490
```

```
SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info;
```

RUNID	FUNCTION	LINE#	CALLS	S_E_T	F_E_T
1	__anonymous_block.TEST_FUNC	5	100000	17461	17461
1	STOP_PROFILING	63	1	0	0
2	STOP_PROFILING	63	1	0	0
3	__anonymous_block.TEST_FUNC	5	100000	18327	18327
3	STOP_PROFILING	63	1	0	0

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2    v_num      PLS_INTEGER;
3    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
4
5  BEGIN
6    DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
7    FOR i IN 1..100000
8    LOOP
9      v_num := i-1 + i; -- there is no reference to test_func
10   END LOOP;
11   DBMS_HPROF.STOP_PROFILING;
12
13  -- Analyze profiler output and display its run ID
14  v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
15  DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID - '||v_run_id);
16 END;
```

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;

1  DECLARE
2    v_num      PLS_INTEGER;
3    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
4
5    FUNCTION test_func (num1 IN PLS_INTEGER
6                        ,num2 IN PLS_INTEGER)
7    RETURN PLS_INTEGER
8    IS
9    BEGIN
10       RETURN (num1 + num2);
11    END test_func;
12
13 BEGIN
14   DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
15
16   FOR i IN 1..100000
17   LOOP
18     v_num := test_func (i-1, i);
19   END LOOP;
20   DBMS_HPROF.STOP_PROFILING;
21
22   -- Analyze profiler output and display its run ID
23   v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
24   DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled implicitly, run ID - '||v_run_id);
25 END;
```

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is enabled implicitly, run ID - 4  
Elapsed: 00:00:00.065
```

```
SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
   FROM dbmshp_function_info
 WHERE runid = 4;
```

RUNID	FUNCTION	LINE#	CALLS	S_E_T	F_E_T
4	STOP_PROFILING	63	1	0	0

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 2;

1  DECLARE
2    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
3
4    FUNCTION f1 (num1 IN PLS_INTEGER
5                  ,num2 IN PLS_INTEGER)
6    RETURN PLS_INTEGER
7    IS
8    BEGIN
9      RETURN (num1 + num2);
10   END f1;
11
12  FUNCTION f2 (str1 IN VARCHAR2
13                  ,str2 IN VARCHAR2)
14  RETURN VARCHAR2
15  IS
16  BEGIN
17    RETURN (str1||' '||str2);
18  END f2;
19

20 PROCEDURE p1 (num1 IN PLS_INTEGER
21                  ,num2 IN PLS_INTEGER
22                  ,str1 IN VARCHAR2
23                  ,str2 IN VARCHAR2)
24  IS
25    v_num NUMBER;
26    v_str VARCHAR2(100);
27  BEGIN
28    v_num := f1(num1, num2);
29    v_str := f2(str1, str2);
30  END p1;
31
32 BEGIN
33  DBMS_HPROF.START_PROFILING ('PLSHPROF_DIR', 'test.txt');
34  FOR i in 1..100000
35  LOOP
36    -- Inline pragma is enabled for each procedure call
37    PRAGMA INLINE (p1, 'YES');
38    p1 (i-1, i, to_char(i-1), to_char(i));
39  END LOOP;
40  DBMS_HPROF.STOP_PROFILING;
41
42  -- Analyze profiler output
43  v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROF_DIR', 'test.txt');
44  DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID - '||v_run_id);
45 END;
```

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is enabled, run ID - 5  
Elapsed: 00:00:01.179
```

```
SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 5;
```

RUNID	FUNCTION	LINE#	CALLS	S_E_T	F_E_T
5	anonymous_block.F1	4	100000	20595	20595
5	anonymous_block.F2	12	100000	42010	42010
5	STOP_PROFILING	63	1	0	0

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is enabled, run ID - 6  
Elapsed: 00:00:00.042
```

```
SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 6;
```

RUNID	FUNCTION	LINE#	CALLS	S_E_T	F_E_T
6	STOP_PROFILING	63	1	0	0

```
SET TIMING ON;
ALTER SESSION SET PLSQL_OPTIMIZE_LEVEL = 3;

1  DECLARE
2    v_run_id BINARY_INTEGER; -- run ID generated by the profiler
3
4    FUNCTION f1 (num1 IN PLS_INTEGER
5                  ,num2 IN PLS_INTEGER)
6    RETURN PLS_INTEGER
7    IS
8      v_num PLS_INTEGER;
9    BEGIN
10      SELECT num1 + num2
11        INTO v_num
12        FROM dual;
13      RETURN v_num;
14    END f1;
15
16    FUNCTION f2 (str1 IN VARCHAR2
17                  ,str2 IN VARCHAR2)
18    RETURN VARCHAR2
19    IS
20      v_str VARCHAR2(50);
21    BEGIN
22      SELECT str1||' '||str2
23        INTO v_str
24        FROM dual;
25      RETURN (v_str);
26    END f2;
27
28    PROCEDURE p1 (num1 IN PLS_INTEGER
29                  ,num2 IN PLS_INTEGER
30                  ,str1 IN VARCHAR2
31                  ,str2 IN VARCHAR2)
32    IS
33      v_num NUMBER;
34      v_str VARCHAR2(100);
35    BEGIN
36      v_num := f1(num1, num2);
37      v_str := f2(str1, str2);
38    END p1;
39
40  BEGIN
41    DBMS_HPROF.START_PROFILING ('PLSHPROP_DIR', 'test.txt');
42    FOR i in 1..100000
43    LOOP
44      p1 (i-1, i, to_char(i-1), to_char(i));
45    END LOOP;
46    DBMS_HPROF.STOP_PROFILING;
47
48    -- Analyze profiler output
49    v_run_id := DBMS_HPROF.ANALYZE ('PLSHPROP_DIR', 'test.txt');
50    DBMS_OUTPUT.PUT_LINE ('Inline pragma is enabled, run ID - '||v_run_id);
51 END;
```

```
session SET altered.  
Elapsed: 00:00:00.001  
Inline pragma is enabled, run ID - 7  
Elapsed: 00:00:06.156
```

```
SELECT runid, function, line#, calls, subtree_elapsed_time s_e_t
      ,function_elapsed_time f_e_t
  FROM dbmshp_function_info
 WHERE runid = 7;
```

RUNID	FUNCTION	LINE#	CALLS	S_E_T	F_E_T
7	static_sql_exec_line	44	200000	4382821	4382821
7	STOP_PROFILING	63	1	0	0

```
PACKAGE student_admin
-- admin suffix may be used for administration.

PROCEDURE remove_student (i_student_id IN student.studid%TYPE);

FUNCTION student_enroll_count (i_student_id student.studid%TYPE)
RETURN INTEGER;
```

```
REM ****
REM * filename: coursediscount01.sql      version: 1
REM * purpose: To give discounts to courses that have at
REM *           least one section with an enrollment of more
REM *           than 10 students.
REM * args:    none
REM *
REM * created by: s.tashi      date: January 1, 2000
REM * modified by: y.sonam     date: February 1, 2000
REM * description: Fixed cursor, added indentation and
REM *                 comments.
REM ****
DECLARE
  -- C_DISCOUNT_COURSE finds a list of courses that have
  -- at least one section with an enrollment of at least 10
  -- students.
  CURSOR c_discount_course IS
    SELECT DISTINCT course_no
      FROM section sect
     WHERE 10 <= (SELECT COUNT(*)
                   FROM enrollment enr
                  WHERE enr.section_id = sect.section_id
                );
  -- discount rate for courses that cost more than $2000.00
  con_discount_2000 CONSTANT NUMBER := .90;
  -- discount rate for courses that cost between $1001.00
  -- and $2000.00
  con_discount_other CONSTANT NUMBER := .95;
  v_current_course_cost course.cost%TYPE;
  v_discount_all NUMBER;
  e_update_is_problematic EXCEPTION;
BEGIN
  -- For courses to be discounted, determine the current
  -- and new cost values
  FOR r_discount_course IN c_discount_course LOOP
    SELECT cost
      INTO v_current_course_cost
      FROM course
     WHERE course_no = r_discount_course.course_no;
```

```
IF v_current_course_cost > 2000 THEN
    v_discount_all := con_discount_2000;
ELSE
    IF v_current_course_cost > 1000 THEN
        v_discount_all := con_discount_other;
    ELSE
        v_discount_all := 1;
    END IF;
END IF;

BEGIN
    UPDATE course
        SET cost = cost * v_discount_all
    WHERE course_no = r_discount_course.course_no;
EXCEPTION
    WHEN OTHERS THEN
        RAISE e_update_is_problematic;
END; -- end of sub-block to update record
END LOOP; -- end of main LOOP

COMMIT;
EXCEPTION
    WHEN e_update_is_problematic THEN
        -- Undo all transactions in this run of the program
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE
            ('There was a problem updating a course cost.');
    WHEN OTHERS THEN
        NULL;
END;
/
```