



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

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Oracle[®] PL/SQL by Example

Fifth Edition

This page intentionally left blank



Oracle® PL/SQL by Example

Fifth Edition

Benjamin Rosenzweig
Elena Rakhimov



PRENTICE
HALL

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*

This page intentionally left blank



Contents

Preface	xvii
Acknowledgments	xxi
About the Authors	xxiii
Introduction to PL/SQL New Features in Oracle 12c	xxv
Invoker's Rights Functions Can Be Result-Cached	xxvi
More PL/SQL-Only Data Types Can Cross the PL/ SQL-to-SQL Interface Clause	xxvii
ACCESSIBLE BY Clause	xxvii
FETCH FIRST Clause	xxviii
Roles Can Be Granted to PL/SQL Packages and Stand-Alone Subprograms	xxix
More Data Types Have the Same Maximum Size in SQL and PL/SQL	xxx
Database Triggers on Pluggable Databases	xxx
LIBRARY Can Be Defined as a DIRECTORY Object and with a CREDENTIAL Clause	xxx
Implicit Statement Results	xxxi
BEQUEATH CURRENT_USER Views	xxxii

	INHERIT PRIVILEGES and INHERIT ANY PRIVILEGES Privileges	xxxii
	Invisible Columns	xxxiii
	Objects, Not Types, Are Editioned or Noneditioned	xxxiv
	PL/SQL Functions That Run Faster in SQL	xxxiv
	Predefined Inquiry Directives \$\$PLSQL_UNIT_OWNER and \$\$PLSQL_UNIT_TYPE	xxxvi
	Compilation Parameter PLSQL_DEBUG Is Deprecated	xxxvii
Chapter 1	PL/SQL Concepts	1
	Lab 1.1: PL/SQL Architecture	2
	PL/SQL Architecture	2
	PL/SQL Block Structure	5
	How PL/SQL Gets Executed	8
	Lab 1.2: PL/SQL Development Environment	9
	Getting Started with SQL Developer	10
	Getting Started with SQL*Plus	11
	Executing PL/SQL Scripts	14
	Lab 1.3: PL/SQL: The Basics	18
	DBMS_OUTPUT.PUT_LINE Statement	18
	Substitution Variable Feature	19
	Summary	25
Chapter 2	PL/SQL Language Fundamentals	27
	Lab 2.1: PL/SQL Programming Fundamentals	28
	PL/SQL Language Components	28
	PL/SQL Variables	29
	PL/SQL Reserved Words	32
	Identifiers in PL/SQL	33
	Anchored Data Types	34
	Declare and Initialize Variables	36
	Scope of a Block, Nested Blocks, and Labels	39
	Summary	41

Chapter 3	SQL in PL/SQL	43
	Lab 3.1: DML Statements in PL/SQL	44
	Initialize Variables with <code>SELECT INTO</code>	44
	Using the <code>SELECT INTO</code> Syntax for Variable Initialization	45
	Using DML in a PL/SQL Block	47
	Using a Sequence in a PL/SQL Block	48
	Lab 3.2: Transaction Control in PL/SQL	49
	Using <code>COMMIT</code> , <code>ROLLBACK</code> , and <code>SAVEPOINT</code>	49
	Putting Together DML and Transaction Control	53
	Summary	55
Chapter 4	Conditional Control: <code>IF</code> Statements	57
	Lab 4.1: <code>IF</code> Statements	58
	<code>IF-THEN</code> Statements	58
	<code>IF-THEN-ELSE</code> Statement	60
	Lab 4.2: <code>ELSIF</code> Statements	63
	Lab 4.3: Nested <code>IF</code> Statements	67
	Summary	70
Chapter 5	Conditional Control: <code>CASE</code> Statements	71
	Lab 5.1: <code>CASE</code> Statements	71
	<code>CASE</code> Statements	72
	Searched <code>CASE</code> Statements	74
	Lab 5.2: <code>CASE</code> Expressions	80
	Lab 5.3: <code>NULLIF</code> and <code>COALESCE</code> Functions	84
	<code>NULLIF</code> Function	84
	<code>COALESCE</code> Function	87
	Summary	89
Chapter 6	Iterative Control: Part I	91
	Lab 6.1: Simple Loops	92
	<code>EXIT</code> Statement	93
	<code>EXIT WHEN</code> Statement	97

Lab 6.2: WHILE Loops	98
Using WHILE Loops	98
Premature Termination of the WHILE Loop	101
Lab 6.3: Numeric FOR Loops	104
Using the IN Option in the Loop	105
Using the REVERSE Option in the Loop	107
Premature Termination of the Numeric FOR Loop	108
Summary	109
Chapter 7 Iterative Control: Part II	111
Lab 7.1: CONTINUE Statement	111
Using CONTINUE Statement	112
CONTINUE WHEN Statement	115
Lab 7.2: Nested Loops	118
Using Nested Loops	118
Using Loop Labels	120
Summary	122
Chapter 8 Error Handling and Built-in Exceptions	123
Lab 8.1: Handling Errors	124
Lab 8.2: Built-in Exceptions	126
Summary	132
Chapter 9 Exceptions	133
Lab 9.1: Exception Scope	133
Lab 9.2: User-Defined Exceptions	137
Lab 9.3: Exception Propagation	141
Re-raising Exceptions	146
Summary	147
Chapter 10 Exceptions: Advanced Concepts	149
Lab 10.1: RAISE_APPLICATION_ERROR	149
Lab 10.2: EXCEPTION_INIT Pragma	153
Lab 10.3: SQLCODE and SQLERRM	155
Summary	158

Chapter 11	Introduction to Cursors	159
	Lab 11.1: Types of Cursors	159
	Making Use of an Implicit Cursor	160
	Making Use of an Explicit Cursor	161
	Lab 11.2: Cursor Loop	165
	Processing an Explicit Cursor	165
	Making Use of a User-Defined Record	168
	Making Use of Cursor Attributes	170
	Lab 11.3: Cursor FOR LOOPS	175
	Making Use of Cursor FOR LOOPS	175
	Lab 11.4: Nested Cursors	177
	Processing Nested Cursors	177
	Summary	181
Chapter 12	Advanced Cursors	183
	Lab 12.1: Parameterized Cursors	183
	Cursors with Parameters	184
	Lab 12.2: Complex Nested Cursors	185
	Lab 12.3: FOR UPDATE and WHERE CURRENT Cursors	187
	FOR UPDATE Cursor	187
	FOR UPDATE OF in a Cursor	189
	WHERE CURRENT OF in a Cursor	189
	Summary	190
Chapter 13	Triggers	191
	Lab 13.1: What Triggers Are	191
	Database Trigger	192
	BEFORE Triggers	195
	AFTER Triggers	201
	Autonomous Transaction	203
	Lab 13.2: Types of Triggers	205
	Row and Statement Triggers	205
	INSTEAD OF Triggers	206
	Summary	211

Chapter 14	Mutating Tables and Compound Triggers	213
	Lab 14.1: Mutating Tables	213
	What Is a Mutating Table?	214
	Resolving Mutating Table Issues	215
	Lab 14.2: Compound Triggers	217
	What Is a Compound Trigger?	218
	Resolving Mutating Table Issues with Compound Triggers	220
	Summary	223
Chapter 15	Collections	225
	Lab 15.1: PL/SQL Tables	226
	Associative Arrays	226
	Nested Tables	229
	Collection Methods	232
	Lab 15.2: Varrays	235
	Lab 15.3: Multilevel Collections	240
	Summary	242
Chapter 16	Records	243
	Lab 16.1: Record Types	243
	Table-Based and Cursor-Based Records	244
	User-Defined Records	246
	Record Compatibility	248
	Lab 16.2: Nested Records	250
	Lab 16.3: Collections of Records	253
	Summary	257
Chapter 17	Native Dynamic SQL	259
	Lab 17.1: EXECUTE IMMEDIATE Statements	260
	Using the EXECUTE IMMEDIATE Statement	261
	How to Avoid Common ORA Errors When Using EXECUTE IMMEDIATE	262
	Lab 17.2: OPEN-FOR, FETCH, and CLOSE Statements	271
	Opening Cursor	272

Fetching from a Cursor	272
Closing a Cursor	273
Summary	280
Chapter 18 Bulk SQL	281
Lab 18.1: FORALL Statements	282
Using FORALL Statements	282
SAVE EXCEPTIONS Option	285
INDICES OF Option	288
VALUES OF Option	289
Lab 18.2: The BULK COLLECT Clause	291
Lab 18.3: Binding Collections in SQL Statements	299
Binding Collections with EXECUTE IMMEDIATE Statements	299
Binding Collections with OPEN-FOR, FETCH, and CLOSE Statements	306
Summary	309
Chapter 19 Procedures	311
Benefits of Modular Code	312
Block Structure	312
Anonymous Blocks	312
Lab 19.1: Creating Procedures	312
Putting Procedure Creation Syntax into Practice	313
Querying the Data Dictionary for Information on Procedures	314
Lab 19.2: Passing Parameters IN and OUT of Procedures	315
Using IN and OUT Parameters with Procedures	316
Summary	319
Chapter 20 Functions	321
Lab 20.1: Creating Functions	321
Creating Stored Functions	322
Making Use of Functions	325

Lab 20.2: Using Functions in SQL Statements	327
Invoking Functions in SQL Statements	327
Writing Complex Functions	328
Lab 20.3: Optimizing Function Execution in SQL	329
Defining a Function Using the WITH Clause	329
Creating a Function with the UDF Pragma	330
Summary	331
Chapter 21 Packages	333
Lab 21.1: Creating Packages	334
Creating Package Specifications	335
Creating Package Bodies	337
Calling Stored Packages	339
Creating Private Objects	341
Lab 21.2: Cursor Variables	344
Lab 21.3: Extending the Package	353
Extending the Package with Additional Procedures	353
Lab 21.4: Package Instantiation and Initialization	366
Creating Package Variables During Initialization	367
Lab 21.5: <code>SERALLY_REUSABLE</code> Packages	368
Using the <code>SERALLY_REUSABLE</code> Pragma	368
Summary	371
Chapter 22 Stored Code	373
Lab 22.1: Gathering Information about Stored Code	373
Getting Stored Code Information from the Data Dictionary	374
Overloading Modules	378
Summary	382
Chapter 23 Object Types in Oracle	385
Lab 23.1: Object Types	386
Creating Object Types	386
Using Object Types with Collections	391

Lab 23.2: Object Type Methods	394
Constructor Methods	395
Member Methods	398
Static Methods	398
Comparing Objects	399
Summary	404
Chapter 24 Oracle-Supplied Packages	405
Lab 24.1: Extending Functionality with Oracle-Supplied Packages	406
Accessing Files within PL/SQL with UTL_FILE	406
Scheduling Jobs with DBMS_JOB	410
Generating an Explain Plan with DBMS_XPLAN	414
Generating Implicit Statement Results with DBMS_SQL	417
Lab 24.2: Error Reporting with Oracle-Supplied Packages	419
Using the DBMS_UTILITY Package for Error Reporting	419
Using the UTL_CALL_STACK Package for Error Reporting	424
Summary	429
Chapter 25 Optimizing PL/SQL	431
Lab 25.1: PL/SQL Tuning Tools	432
PL/SQL Profiler API	432
Trace API	433
PL/SQL Hierarchical Profiler	436
Lab 25.2: PL/SQL Optimization Levels	438
Lab 25.3: Subprogram Inlining	444
Summary	453
Appendix A PL/SQL Formatting Guide	455
Case	455
White Space	455
Naming Conventions	456
Comments	457
Other Suggestions	457

Appendix B	Student Database Schema	461
	Table and Column Descriptions	461
Index		469



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.

This page intentionally left blank



About the Authors

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

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

This page intentionally left blank



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*

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

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

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

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

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

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

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

ACCESSIBLE BY Clause

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

For Example *Procedure Created with the ACCESSIBLE BY Clause*

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

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

```

-- Execute TEST_PROC2
BEGIN
  test_proc2;
END;
/

TEST_PROC2
TEST_PROCI1

-- Execute TEST_PROCI1 directly
BEGIN
  test_proci1;
END;
/

ORA-06550: line 2, column 4:
PLS-00904: insufficient privilege to access object TEST_PROCI1
ORA-06550: line 2, column 4:
PL/SQL: Statement ignored

```

In this example, there are two procedures, `test_proci1` and `test_proc2`, and `test_proci1` is created with the `ACCESSIBLE BY` clause. As a consequence, `test_proci1` 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_proci1` directly and, as a result, causes an error.

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

FETCH FIRST Clause

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

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

```

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

STUDENT_ID
-----
        102
        102
        103
        104
        105

```

```

106
106
107
108
109
109
110
110
...

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

```

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

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

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

```

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

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

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

```

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

Starting with Oracle 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*

```
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*

```
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*

```
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:

```
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*

```
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

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

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

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

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

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

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

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

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

Objects, Not Types, Are Editioned or Noneditioned

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

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

PL/SQL Functions That Run Faster in SQL

Starting with Oracle 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*

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

```

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

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

```

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

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

```

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

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

```

Predefined Inquiry Directives **\$\$PLSQL_UNIT_OWNER** and **\$\$PLSQL_UNIT_TYPE**

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

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

The following example demonstrates how directives may be used.

For Example *Using Predefined Inquiry Directives*

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

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

```

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

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

Compilation Parameter `PLSQL_DEBUG` Is Deprecated

Starting with Oracle release 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.

This page intentionally left blank



SQL in PL/SQL

In this chapter, you will learn about

- DML Statements in PL/SQL Page 44
- Transaction Control in PL/SQL Page 49

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

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

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

Lab 3.1: DML Statements in PL/SQL

After this lab, you will be able to

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

Initialize Variables with `SELECT INTO`

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

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

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

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

For Example *ch03_1a.sql*

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

```

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

```

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

```

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

```

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

```

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

```

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

Using the SELECT INTO Syntax for Variable Initialization

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

For Example *ch03_1a.sql*

```

SET SERVEROUTPUT ON
DECLARE
  v_average_cost VARCHAR2(10);

```

```

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

```

You will then see the following result:

```

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

```

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

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

For Example *ch03_2a.sql*

```

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

```

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

For Example *ch03_3a.sql*

```

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

```

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

By the Way

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

Using DML in a PL/SQL Block

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

For Example *ch03_4a.sql*

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

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

Using an Oracle Sequence

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

Accessing and Incrementing Sequence Values

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

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

The following example creates the sequence `eseq`.

For Example

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

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

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

Drawing Numbers from a Sequence

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

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

For Example *ch03_5a.sql*

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

Using a Sequence in a PL/SQL Block

In this example, a PL/SQL block is used to insert a new student in the `student` table. The PL/SQL code makes use of two variables, `USER` and `SYSDATE`, that are

used in the select statement. The existing `student_id_seq` sequence is used to generate a unique ID for the new student.

For Example *ch03_6a.sql*

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

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

Lab 3.2: Transaction Control in PL/SQL

After this lab, you will be able to

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

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

Transactions are a means to break programming code into manageable units. Grouping transactions into smaller elements is a standard practice that ensures an application will save only correct data. Initially, any application will have to connect to the database to access the data. When a user is issuing DML statements in an application, however, these changes are not visible to other users until a `COMMIT` or `ROLLBACK` has been issued. The Oracle platform guarantees a read-consistent view of the data. Until that point, all data that have been inserted or updated will be held

in memory and will be available only to the current user. The rows that have been changed will be locked by the current user and will not be available for updating to other users until the locks have been released. A `COMMIT` or `ROLLBACK` statement will release these locks. Transactions can be controlled more readily by marking points of the transaction with the `SAVEPOINT` command.

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

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

By the Way

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

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

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

For Example *ch03_7a.sql*

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

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

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

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

COMMIT

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

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

A `COMMIT` statement has the following syntax:

```
COMMIT [WORK];
```

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

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

For Example *ch03_8a.sql*

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



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

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

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

Then User A issues the following command:

```
COMMIT;
```

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

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

ROLLBACK

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

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

A `ROLLBACK` statement has the following syntax:

```
ROLLBACK [WORK];
```

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

SAVEPOINT

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

```
SAVEPOINT name;
```

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

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

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

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

Putting Together DML and Transaction Control

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

For Example *ch03_9a.sql*

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

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

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

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

The result would be “no rows selected.”

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

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

```
ROLLBACK to SAVEPOINT A;
```

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

By the Way

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

Did You Know?

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

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

For Example *ch03_10a.sql*

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

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

Summary

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

The last section of the chapter covered transactional control in PL/SQL by explaining what it means to commit data as well as how `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.

This page intentionally left blank



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.

- Architecture (*continued*)
 - 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–xxxii
- 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
 - fetching, 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

- Databases (*continued*)
 - 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–xxxii
- 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* 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
- FOLLOWS** 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, xxx1–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, xxxi–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–xxxii
- 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
- NONEDITIONABLE
 - 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
- NVARCHAR2 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

- Passing parameters to
 - procedures (*continued*)
 - 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.
 - 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, xxx1–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_REUSABLE 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, xxxi–xxxii
 - multirow, 271–280
 - new for Oracle 12c, xxxi–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